Scalable and Generalizable RL Agents for Attack Path Discovery via Continuous Invariant Spaces

Franco Terranova, Abdelkader Lahmadi, Isabelle Chrisment

Scalable and Generalizable RL Agents for Attack Path Discovery via Continuous Invariant Spaces

Franco Terranova
Université de Lorraine, CNRS, Inria, LORIA
Nancy, France
franco.terranova@inria.fr

Abdelkader Lahmadi
Université de Lorraine, CNRS, Inria, LORIA
Nancy, France
abdelkader.lahmadi@loria.fr

Isabelle Chrisment
Université de Lorraine, CNRS, Inria, LORIA
Nancy, France
isabelle.chrisment@loria.fr

***Abstract*—Identifying critical attack paths in a network—sequences of vulnerabilities an attacker can chain to achieve a specific threat model—is crucial for pinpointing vulnerable areas where defensive measures should be focused. Recently, Reinforcement Learning (RL) has gained traction for training agents in identifying these critical paths. However, current solutions typically train RL agents tailored to a specific environment—defined by a fixed network structure and vulnerability set—requiring costly retraining whenever either changes. This limitation arises from optimizing the agent to map between discrete input and output spaces, treating network nodes and vulnerabilities as atomic discrete elements. In this paper, we propose a method for constructing continuous and invariant input and output spaces for RL agents, enabling them to learn transferable policies that generalize across diverse network configurations and vulnerability sets. We also release *Continuous CyberBattleSim* (*C-CyberBattleSim*), an enhanced version of Microsoft CyberBattleSim designed to train agents with the novel continuous spaces. The tool is further extended to integrate real-world vulnerability data and a new scenario generation pipeline to improve the realism of training and testing environments. Agents trained in continuous spaces are assessed in 800 scenarios with varying sizes and various allocations of 829 real-world vulnerabilities, demonstrating an average improvement of 9.3x in scalability against agents trained in discrete spaces, as well as an average generalization score of 89% to more complex scenarios when trained in simpler scenarios. A final study evaluates whether continuous agents trained in simulation can adapt to real-world and emulated scans. On average, agents achieve 75% of the score they would have if trained directly on the scans, demonstrating effective knowledge transfer.**

***Index Terms*—Deep Reinforcement Learning, Attack Path Discovery, Invariant Agents, Continuous Spaces, Simulation Environments**

I. Introduction

Attackers typically infiltrate networks in a systematic manner, exploiting chains of vulnerabilities across multiple hosts to accomplish specific malicious objectives [16]. These vulnerabilities are inherent in network device services, configurations, and operating systems, often existing in large numbers. Consequently, prioritization is often necessary to allocate limited defensive resources effectively to the most critical areas, such as prioritizing vulnerabilities for patching [37]. Specifically, understanding how vulnerabilities interact within networks to create attack paths and how these interactions enable attackers to achieve their goals remains a significant challenge. The process of identifying such attack paths is also known as multistage attack prediction [24], and can help organizations strengthen their security position. In particular, attack detection systems that simulate and anticipate potential threats are classified as proactive methods [24]. Historically, these methods have relied on expert-driven manual penetration testing (PT) [49]. Although effective for simple network scenarios, PT struggles to scale to large dynamic scenarios, where the exponential growth of vulnerabilities and nodes makes it impractical to manually evaluate all potential paths. For this reason, automation has become essential to complement human expertise. Early solutions developed attack graph generation tools [50] as an automated approach to enumerate all potential attack paths based on network configurations. They subsequently require prioritization algorithms to identify the most critical paths among the many possibilities. However, pre-enumerating all paths in advance reintroduced scalability challenges [11]. More recent research is shifting to the use of machine learning (ML) as a proactive approach aimed at directly identifying the most critical attack paths based on network configurations in input, addressing scalability issues by recognizing that the most critical paths are those of primary interest. Reinforcement Learning (RL) [53] has emerged as the key ML paradigm for this task, as it enables models to implement sequential decision-making strategies that align with attackers' behavior when selecting vulnerability sequences. More precisely, the RL paradigm trains an agent to take optimal actions across various observations of an environment. In our context, actions represent the selection of critical nodes and vulnerabilities within evolving network configurations, with the evolution driven by previous vulnerability selections. The set of all possible observations and actions forms the observation and action spaces, with RL agents trained to learn mappings within these spaces. Training such agents to find valuable mappings is typically conducted in simulation environments that should closely approximate real-world scenarios, as direct interaction with actual systems or their emulations is often cost-prohibitive [58]. In general, the effectiveness of RL agents is significantly influenced by the realism of training environments and the selection of appropriate observation and action spaces. Enhancing these aspects is crucial for developing a scalable and generalizable model that is trained once and can

generalize to diverse and new scenarios without retraining. This aligns with the principles of artificial narrow intelligence (ANI) [41], which seeks to create models capable of handling all scenarios for a specific task. This paper proposes two contributions to overcome the limitations posing a barrier to reaching these goals:

Environment: The lack of realism in existing simulation environments hinders agents' ability to learn meaningful real-world behaviors, highlighting a *sim-to-real gap* [35], driven by two key issues: (1) simulated training scenarios are often generated with a limited set of randomly assigned vulnerabilities, leading to unrealistic scenarios, and (2) simplistic simulation of vulnerability exploitation distorts the concept of optimality, causing RL agents to learn the selection of suboptimal actions.

Solution: To address these challenges, we introduce an enhanced simulation environment, *Continuous CyberBattleSim* (*C-CyberBattleSim*), built upon Microsoft's *CyberBattleSim* tool [54]. This enhanced environment: (1) allows querying of Cyber-Threat Intelligence (CTI) tools [9] and vulnerability databases to gather real-world statistics via custom queries, and uses these statistics to allocate vulnerabilities in the generated scenarios; and (2) introduces a new pipeline that automates the prediction of vulnerability outcomes based on vulnerability metadata, shifting the responsibility of redefining the simulation away from humans. The paper provides a link to the *C-CyberBattleSim* environment repository[1], enabling researchers to replicate the methodology and the experiments. In addition, it gives access to the data, generated scenarios, and trained models used in the study[2].

RL Agent: Previous studies in this field have relied on discrete observation and action spaces for RL agents, where network nodes and vulnerabilities are represented as discrete elements. Agents are then trained to find the link between these elements in a specific order to identify the most critical attack paths, but only relative to the training vulnerability set and network structure. As a result, within the constraints of the RL framework for discrete spaces, modifying the spaces requires retraining the RL agent. This limits, by construction, the *generalization* of RL agents to new scenarios, since in the cyber-attack path prediction task, the network structure and vulnerability set typically differ across application scenarios. Additionally, agents struggle with *dynamicity* in the environment, as changes in the network nodes or vulnerabilities necessitate modifications to the spaces. Moreover, as the number of nodes or vulnerabilities increases, the discrete spaces expand, resulting in *scalability* issues since the complexity of the mappings to learn grows.

Solution: To address this, we propose a novel approach to achieve scenario independence by redefining the spaces of the RL agent from discrete to continuous ones. In our approach, actions and observations are represented as fixed-dimensional vectors that capture their semantics, with agents trained to find mappings between these semantic spaces. To create these spaces, we design a *world model* [20] that maps the discrete elements of the specific application scenarios as points in the continuous observation and action spaces. This intermediate world model is designed to ensure that the agent can operate within the same continuous spaces, even when the network environment and vulnerabilities change, and ensure that semantics are present, allowing the generalization of RL agents trained on top. To build these semantically meaningful and continuous spaces, we leverage intelligent models, and in particular Graph Neural Networks (GNNs) [59] to represent nodes and the network graph in these spaces, and Language Models (LMs) [61] to represent vulnerabilities based on their textual descriptions.

The paper is organized as follows: Section 2 covers background on ML solutions and cybersecurity data sources. Section 3 reviews related work and its limitations. Section 4 presents our end-to-end methodology, followed by experiments in Section 5. Finally, Section 6 explores the potential real-world applicability, while Section 7 concludes with suggestions for future research directions.

II. Background

This section introduces RL and explores its application to both discrete and continuous spaces. We then discuss how GNNs and LMs can represent environment elements, such as network graphs, nodes, and vulnerabilities, as embeddings in continuous spaces. Additionally, we explore the cybersecurity data sources used to create environments for agent training.

A. Reinforcement Learning

RL is a key ML paradigm designed to develop autonomous agents that learn optimal behaviors within a given environment. The paradigm involves an agent operating in a state s_t, taking action a_t, and receiving feedback as a new state s_{t+1} and reward r_{t+1} from the environment. The environment is typically modeled using a Markov Decision Process (MDP) [43] defined across its elements by a state space *S* and an action space *A*. Partially Observable MDPs (POMDPs) [34] extend MDPs to scenarios with incomplete information, utilizing an observation space O instead of the state space S, enabling learning in partially visible environments. They are particularly valuable in environments designed for progressive discovery, as is often the case in graph-based exploration tasks. The primary goal of the RL framework is to learn an optimal policy π (Equation 1), which maps observations (or states) to actions, guiding the agent's behavior within the environment.

$$\pi : O \rightarrow A \quad \text{such that} \quad \max_{\pi} \mathbb{E}_{\tau \sim \pi} \left[R(\tau) \right] \tag{1}$$

An episode denoted as τ, typically consists of sequences of states, actions, and rewards, $\tau = (s_0, a_0, r_1, s_1, a_1, \dots)$. The optimization is achieved by maximizing the expected return $R(\tau)$, defined as $R(\tau) = \sum_{k=0}^{N} \gamma^k r_{t+k+1}$, where N

[1]C-CyberBattleSim: https://github.com/terranovafr/C-CyberBattleSim
[2]Data: https://doi.org/10.5281/zenodo.14604651

represents the episode length, and γ determines the trade-off between immediate and future rewards, ensuring that future outcomes influence the agent's decisions and that the optimization accounts for the long-term impact of actions. Policy learning in RL occurs through trial and error, refining actions based on rewards, with algorithms differing in their optimization approaches. Deep RL (DRL) enhances RL by employing Neural Networks (NNs) [57] as function approximators for the policy, with input and output layers designed to match the observation and action spaces. For this reason, the representation of observation and action spaces significantly impacts the generalization and scalability of the trained agent. These spaces can be either discrete or continuous, each shaping the policy structure, as well as how the policy learns mappings.

1) Discrete Spaces: In discrete optimization tasks, the RL environment is generally composed of a set of discrete choices (e.g., grid cells, graph nodes) on top of which the agent can act. The typical approach to solving these problems with the RL framework is to leverage discrete observation and action spaces. The observation vector provided by the environment is typically designed to represent the set of discrete choices available, with its size proportional to their number. Let N be the number of discrete elements and f be the number of features per element. The observation vector can be expressed as $\mathbf{o} \in \mathbb{R}^{N \times f}$. The agent is then trained to output a vector that assigns a score or quality value to each possible action, with the vector size typically determined by the number of discrete choices. Let $|A|$ denote the number of possible discrete actions per element. The policy maps observations to an action space, with both input and output dimensions depending on the task. This mapping can be abstractly written as in Equation 2.

$$\pi : \mathbb{R}^{N \times f} \rightarrow \mathbb{R}^{f(N,|A|)} \tag{2}$$

The function $f(N, |A|)$ defines the dimensionality of the action space; for example, if the task involves selecting a single element and an action, then $f(N, |A|) = N \times |A|$, while if it involves selecting pairs of elements and an action on each pair, then $f(N, |A|) = N \times N \times |A|$. In this context, the observation and action spaces scale with the number of discrete choices, and in DRL, the NN's input and output layers must match these dimensions—resulting in models specialized for fixed-size vectors and learning order-dependent patterns. In some discrete optimization tasks, the set of discrete choices remains unchanged across all scenarios, making RL within discrete spaces an ideal framework. For example, in chess [51], the board size (observation space) and possible moves (action space) are often fixed. However, in tasks where the number and order of discrete choices vary—such as graph-based optimization problems—discrete RL presents limitations. A common approach used is *padding*, which involves resizing the observation and action vectors to match the largest set of discrete choices across all scenarios. However, this approach still preserves order-dependent patterns and results in overly large vectors when the maximum scenario size is significantly larger than most instances, hindering the learning process.

2) Continuous Spaces: Recent studies in RL research [13] [17] propose solving discrete optimization problems using continuous embedding observation and action spaces created with a world model. The goal is to enable the agent to process a fixed-dimensional input vector and map it to a fixed-dimensional output vector, regardless of the number of discrete choices in a given scenario. This should be achieved while maintaining order invariance, ensuring that the vector remains the same regardless of how the order of discrete elements is rearranged. When using DRL, this also makes the NN representing the policy scenario-agnostic, allowing the same model to be applied across different problem settings. It is important to note that the fixed-dimensionality problem can also be addressed using a compressed, discrete representation. However, continuous embeddings are more commonly used, as discrete spaces remain less explored due to the difficulties they present for gradient-based optimization [38]. To handle discrete optimization problems in fixed-dimensional continuous spaces, all discrete elements must be mapped to these spaces accordingly. To better align with the context of our study, one possible formulation of a world model comprises the following three key components:

- **Observation mapping function**: Transforms the environment's observation (state) into a fixed-dimensional vector, serving as the agent's input.
- **Action mapping function**: Represents all discrete actions as points in a fixed-dimensional continuous action space. The agent is subsequently tasked to generate continuous action points within this space.
- **Proximity-based selection strategy**: Uses proximity methods (e.g., *nearest-neighbor*) to map the agent's output vector to the closest discrete action point [13] [22].

For this approach to be effective, the continuous spaces must have semantic meaning—similar observations and actions should be positioned close together. Therefore, the mapping functions must ensure this semantic structure. However, a major challenge remains: world models and their mapping functions to build embedding representations must be tailored to the specific application, as their design depends on the structure and semantics of both the environment observation space and the discrete action set. For example, [62] introduces an embedding-based mapping specifically for the single-host penetration testing task, while [13] presents a solution applied to the classic CartPole environment.

B. Representation Learning

Emerging advancements in ML are pushing the boundaries of representation learning, enabling models to encode diverse data into semantically meaningful and continuous vectors (i.e., embeddings) that can be used in various tasks, including serving as the mapping functions of world models.

1) Graph Neural Networks: In recent years, GNNs have emerged as a transformative approach for analyzing data structured as graphs [26]. They excel due to their invariance to node permutations and graph sizes, allowing them to adapt to various graph structures. By leveraging message-passing

techniques, GNNs are tasked to derive fixed-dimensional node embeddings based not only on the features of the nodes themselves but also on their local graph structure, capturing information from neighboring nodes (Equation 3).

$$\mathbf{h}_v^{(k+1)} = \text{UPDATE}\left(\mathbf{h}_v^{(k)}, \text{JOIN}\left(\{\mathbf{h}_{v_i}^{(k)} \mid v_i \in \mathcal{N}(v)\}\right)\right) \quad (3)$$

At the k-th iteration, $\mathbf{h}_v^{(k)}$ denotes the embedding of node v, and $\mathcal{N}(v)$ represents the set of its neighboring nodes. The initial embedding $\mathbf{h}_v^{(0)}$ is typically set to the node's input feature vector $\mathbf{f}_v$. The function JOIN aggregates information from neighboring nodes, while UPDATE combines the aggregated message with the node's current embedding to produce the new embedding. Both functions can be parameterized and learned during training. Subsequently, a fixed-dimensional graph embedding can be obtained by applying a pooling operation [18] on the node embeddings, resulting in a compact, fixed-dimensional representation of the entire graph, regardless of its size (Equation 4).

$$\mathbf{h}_G = \text{POOL}\left(\{\mathbf{h}_v \mid v \in \mathcal{V}\}\right) \quad (4)$$

If the pooling operation is order-invariant (e.g., *mean*), the embedding also loses ordering features.

2) Language Models: LMs have revolutionized the field of natural language processing by demonstrating an exceptional ability to understand natural language. Pretrained on a vast corpus of text from diverse contexts, these models have learned to map text into semantically rich embeddings that can achieve tasks with remarkable accuracy. These general linguistic models can be reused across various applications while maintaining a semantic understanding of language. Vulnerabilities, in particular, are typically accompanied by a description in free-text format, and mapping vulnerability descriptions to semantically meaningful embeddings, which can be used for various tasks, is where LMs are showing promising results [31].

C. Cyber-Security Data Sources

To support the generation of diverse scenarios, this study combines multiple sources of cybersecurity data:

- **Cyber threat intelligence** (CTI) tools like Shodan[3] allow gathering statistics and identifying patterns across Internet-connected devices through custom queries, revealing real-world information about services, operating systems, and other components. As these components are often linked to vulnerabilities, they help in understanding their distribution and prevalence. For example, Shodan can answer the query: *What are the most common services (hence vulnerabilities) in a specific geographic region?*
- **Vulnerability databases**, such as the National Vulnerability Database (NVD) [8], offer detailed metadata on vulnerabilities, including textual descriptions and impact metrics. Across these metrics, the Common Vulnerability Scoring System (CVSS) [32] vector helps assess severity through multiple components (e.g., *attack complexity*).
- The **MITRE ATT&CK Framework** [52] is a knowledge base of adversarial tactics and techniques used in cyber attacks. It defines classes of attack behaviors to which vulnerabilities can be mapped.

[3]Shodan: https://www.shodan.io/

III. Related Work

In this section, we explain the motivation for transitioning from attack graphs to raw network environments for agent training and discuss how simulation can be leveraged to model these environments. Finally, we explore existing encoding strategies for observation and action spaces, which play a crucial role in shaping the agent's learning process and the effectiveness of its decision-making.

A. From Attack Graphs to POMDP-based Environments

One of the first automation approaches in proactive cybersecurity has been the generation of attack graphs, which identify all potential attack paths within a network for a given goal. These tools, such as MulVAL [39], require detailed input on the whole network representation, including the complete list of node services, vulnerabilities, and security policies, to produce their output. The resulting attack graph consists of nodes representing system states, while edges depict the possible transitions between these states based on exploitability conditions. However, in some cases, more than the full set of attack paths, the most critical ones for a given threat model may be of interest. To this end, prioritization algorithms, including RL-based approaches [23] [15], have been using attack graphs as MDPs for attack path prioritization, presenting, however, several challenges [11]. New environments like *CyberBattleSim* address these limitations by directly using raw networks represented as POMDPs. This approach allows the agent to interact directly with the real network representation rather than an attack graph overlay, offering distinct advantages:

1) **Complete Knowledge**
   - *Attack graphs* require full knowledge of the network topology for their generation, which is often unrealistic for large, complex scenarios.
   - In contrast, *raw network POMDPs* allow RL agents to operate without complete network knowledge. The agent starts with a partially discovered network and progressively expands its visibility through its actions.
2) **Dynamicity Issues**
   - *Attack graphs* must be regenerated whenever there are network modifications, such as host disconnections or service shutdowns. These changes result in a change of the whole attack graph, where paths should be prioritized, requiring its regeneration and re-execution of path prioritization algorithms. This presents particular challenges in highly dynamic networks, such as those found in Internet of Things (IoT) ecosystems.
   - In contrast, *raw network POMDPs* inherently support dynamic environments, as changes in nodes or their

features do not alter the inherent structure of the environment but only require modification of the number of nodes or their feature vector.

3) **Scalability Issues**
   - *Attack graph* generation tools face challenges as the number of nodes or vulnerabilities increases, as these additions result in a significantly larger number of new states and transitions, leading to an exponential increase in the size of the attack graph. This makes them impractical for large-scale networks [40].
   - In contrast, *raw network POMDPs* reduce scalability issues, as adding a new node or vulnerability introduces only a single new element in the network graph.

Building on the challenges mentioned, this work aligns with recent studies [55] [11], that advocate for using the network itself as a training environment and represent it as a POMDP. This approach allows the agent to learn the prioritization strategy by directly interacting with an evolving network representation, eliminating the need for additional overlays and their associated drawbacks. These network environments are typically modeled through simulation—such as in *CyberBattleSim*—as it enables faster action execution compared to emulation or real-world setups, which is essential given the high number of iterations required for RL algorithms to converge. Moreover, simulation supports the generation of hundreds of highly customizable scenarios, making it well-suited for both large-scale training and systematic validation. However, they often lack realism because of simplified vulnerability allocation and weak approximations of vulnerability exploitation [58] [36], which can lead agents to learn from unrealistic scenarios. The current version of *CyberBattleSim*, for example, requires the manual definition of the network environment, the placement of vulnerabilities, and the definition of transitions when vulnerabilities are selected. Our study aims to address these challenges by proposing an enhanced version of the current tool, focused on improving these aspects.

B. Discrete RL Approaches for Attack Path Prioritization

The effectiveness of using POMDPs for attack path prioritization depends on selecting appropriate observation and action spaces for the agent. In DRL, this choice dictates the number and semantics of input neurons for processing observations and output neurons for generating actions, shaping the NN and the patterns it learns. By eliminating scenario dependence in these spaces, NN can be applied to scenarios with structural characteristics different from those seen during training. This adaptability serves as an initial step toward developing an ANI for the task. In the studies focusing on POMDPs for the task, the agent (Equation 5) is generally designed to use a graph representation of the network discovered at a given timestep t and should learn to produce in output the optimal tuple *(source node, target node, vulnerability)*.

$$\begin{aligned} \pi_{\text{general}} : \quad & \textit{graph encoding} \rightarrow \textit{(source, target, vuln)} \\ \text{such that} \quad & \max_{\pi} \mathbb{E}_{\tau \sim \pi} \left[\textit{Impact}(\text{Path } \tau \mid \text{Threat Model}) \right] \end{aligned} \tag{5}$$

Optimal selection here refers to maximizing the expected impact of a path based on the desired threat model, which is incorporated as the reward function. All existing studies used a discrete form of this input and output. Most approaches (e.g., [5], [19], [60]) adopted a global view (Equation 6 or similar).

$$\pi_{\text{global}} : \mathbb{R}^{N \times f} \rightarrow \mathbb{R}^{N \times N \times |V|} \tag{6}$$

In approaches using a global view, the observation vector represents the discovered graph as an array that concatenates the feature vectors of all discovered nodes, each of size f. Because the number of discovered nodes at each timestep in the POMDP framework can vary, the array is typically padded to the maximum graph size N. The discrete action space includes all possible discrete source-target-vulnerability cases, leading to a space proportional to N and the size of the vulnerability set V, requiring the agent to output a score for each of these tuples. These observation and action spaces present scalability challenges as nodes and vulnerabilities increase and cause the agent to learn patterns that are sensitive to the order of these elements. This approach also limits the agent's ability to operate on larger network scenarios or handle vulnerabilities beyond those predefined during training.

Another approach proposed a local view [55] (Equation 7), which restricts the ability of the agent to work on a subset of nodes at each timestep to solve some of these issues.

$$\pi_{\text{local}} : \mathbb{R}^{2 \times f + g} \rightarrow \mathbb{R}^{|V| + |S|} \tag{7}$$

This choice led to a smaller, localized observation space, focusing only on a single source and target node, resulting in an observation vector of size $2 \times f$. Additionally, it can be optionally concatenated with a graph-invariant feature vector g, which provides condensed information about the graph's structure (e.g., average importance of other nodes). The discrete action space, in this case, requires producing a score for each possible vulnerability on the current source-target pair, and for a set of switching actions S (e.g., *switch source*, *switch target*), forming an action space of size $|V| + |S|$. Although this discrete local view approach removes the dependency on the number of nodes N, it may result in sub-optimal choices due to limited visibility, as another node pair may be more critical at a given timestep, and requires one or more switching actions to find it. Moreover, it still presents scalability issues due to the dependence on the vulnerability set. In summary, variations in graph structure (*global* view) or vulnerability sets (*global* and *local* views) alter the spaces and require retraining policies, a costly and time-consuming process. As a result, most studies focus on scenario-specific agents rather than aiming for an ANI. However, highly dynamic scenarios (e.g., in IoT environments) and evolving vulnerabilities (e.g., resulting from software updates) pose significant challenges to these approaches. Our approach aims to lose these dependencies, from both the graph structure and vulnerability set, creating an agent that does not need redefinition and retraining in new scenarios by redefining the observation and action spaces.

IV. Methodology

This section highlights the key contributions of this paper: improving the realism of the simulation environment and designing observation and action spaces that remain consistent across various network environments and vulnerability sets. The section subsequently explores the reward function and the episode structure used for training.

A. Towards a More Realistic Simulation Environment

This section presents the pipeline integrated into *C-CyberBattleSim* to enhance realism and ensure automation in the simulation. Specifically, our approach proposes:

1) **Automated vulnerability selection** – Eliminating the need for manually defining a set of vulnerabilities to be incorporated in the scenarios.
2) **Improved scenario generation** – Implementing a more strategic and realistic placement of vulnerabilities and cyber-terrain features [15] within the environment.
3) **Scalable outcome approximation** – Replacing the need to hard-code environment transitions for each vulnerability by introducing a scalable, automated method that dynamically determines these transitions without human intervention.

1) Automated vulnerability selection: The first step in the scenario generation process tackles the challenge of selecting real-world vulnerabilities for populating synthetic graphs by leveraging trusted data sources. To identify this information, a CTI tool is used to collect anonymous statistics on vulnerability distributions across internet-connected devices based on a custom query (Step 1, Figure 1). For instance, when aiming to load service-level vulnerabilities, *Shodan* can be queried to retrieve the most common services and their versions within a specific geographical area, along with their prevalence (i.e., *device counts*). Once these services and versions are identified, databases like the NVD can be queried to retrieve the vulnerabilities associated with each version. Together with the list of vulnerability identifiers, their description, and CVSS vector can be stored for later use. All this information is then stored in the environment database in the format: *(service, version, vulnerabilities, device counts)*.

2) Improved scenario generation: The second step in the pipeline focuses on achieving a more realistic placement of vulnerabilities within the generated graphs, based on the custom query used to gather data and a controlled set of parameters (list in Appendix D). Graph-based parameters (e.g., *number of nodes*) are used to create an initial empty graph. Subsequently, service versions (and their associated vulnerabilities) are assigned to graph nodes (Step 2, Figure 1). To ensure a more realistic assignment, we focus on both validity and mirroring real-world conditions.

- **Validity checks** involve simulating specific node types (e.g., *Windows* host) and restricting service assignments to those that are compatible (e.g., *Microsoft Outlook* only for *Windows* hosts).

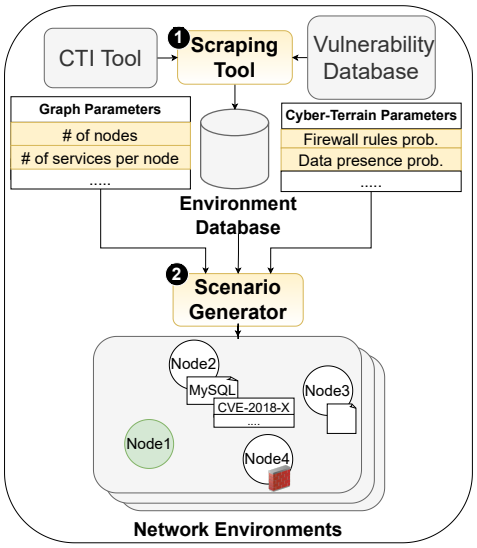

Fig. 1: (1) Services and vulnerabilities are selected using a CTI tool and a vulnerability database, and (2) a generator creates scenarios based on these selections and input parameters.

- **Mirroring real-world conditions** is achieved by weighted sampling when multiple services meet the validity constraints. The sampling priority is determined using a weight $w_i = f(\textit{device count}_i)$, ensuring that services most commonly encountered in the query results are assigned more frequently. This aligns the graph with real-world distributions of the query of interest. The weight function can be adjusted (e.g., $1 + \log(\textit{device count}_i)$) to control the level of real-world mirroring desired.

Additionally, cyber-terrain elements can be specified in the input parameters and incorporated into the simulation, including probabilities for *firewall rules*, *initial service shutdowns*, *privilege escalation* prerequisites for accessing nodes, and *restricted initial visibility*, which necessitates discovery actions to reveal full node details. Optionally, valuable elements such as *data presence* and *node importance* (e.g., to indicate a business-critical asset) are embedded. To prevent the agent from overfitting to a single vulnerability allocation or a fixed set of graph and cyber-terrain parameters, the scenario generator supports *domain randomization* [56]. This feature will create a variety of scenarios by varying the input parameters and the placement of vulnerabilities, all while respecting the constraints and the weighted sampling.

3) Scalable outcome approximation: The third contribution seeks to eliminate the need for the simulation tool developer to manually hard-code environment transitions for each vulnerability in the environment. This limitation has previously constrained the number of vulnerabilities incorporated into each simulator. For example, the original *CyberBattleSim* scenarios, commonly used in the literature, include fewer than 10 unique vulnerabilities each. With dynamic integration of services and vulnerabilities from an external dynamic database based on a custom query, manually encoding outcomes for

each vulnerability is impractical. Our approach leverages vulnerability metadata, sourced from the vulnerability database, to infer potential outcomes in an automated way. In particular, the set of MITRE ATT&CK Tactics associated with each vulnerability can be used as the outcome set and inferred from the vulnerability description using NLP tools (e.g., using LMs fine-tuned for multi-label classification, as discussed in [4]). Once vulnerabilities are mapped to the relevant subset of

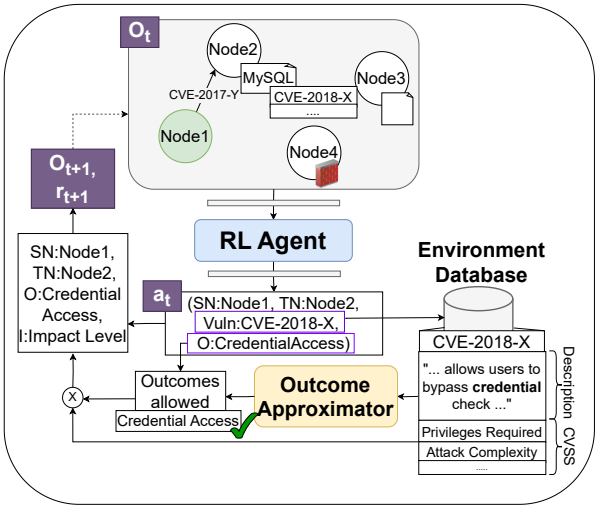

Fig. 2: The requested (source, target, vulnerability, outcome) is validated based on the description and CVSS vector.

Tactics, the integration process is simplified, as environment transitions are defined solely and just once for the limited set of Tactics (14 Tactics in the MITRE ATT&CK framework). Since vulnerabilities may have multiple potential outcomes, we also expand the agent's action space to allow the agent to decide the desired outcome (how it wants to use the vulnerability) in the action space. Previous studies have been limited by assuming a single predefined outcome per vulnerability, which restricts the realism of the agent in simulating attacker behavior. In reality, some vulnerabilities can lead to multiple outcomes (e.g., a vulnerability that enables general code execution), and an attacker could potentially choose which outcome to exploit. Moreover, recognizing that mapping vulnerabilities to a general set of Tactics is overly simplistic, the environment also incorporates the CVSS vector of the targeted vulnerability to better approximate the exploitation process. Specifically, the following elements of the CVSS vector are used to determine whether the exploit is valid and to weigh the outcome:

- **Attack Vector Component**: Determines whether the vulnerability can be exploited locally or remotely.
- **Privileges Required**: Checks the minimum privilege level needed on the node before allowing the exploitation.
- **Confidentiality Impact**: Weighs the impact on confidentiality for vulnerabilities involving information release.
- **Attack Complexity**: Weighs the success rate probabilistically when the agent selects the vulnerability.

Overall, the environment extension processes the agent's selected action tuple a_t at each timestep t through a two-step validation (Figure 2): first, it uses the vulnerability description to confirm if the outcome selected by the agent is permissible (e.g., *Credential Access*). Second, it applies the CVSS vector to ensure additional validity checks and weighs the impact of the outcome using the vulnerability details.

B. RL Agent Design with Continuous Spaces

To overcome challenges with discrete representations (Section III-B), we propose an agent optimizing mappings between continuous observation and action spaces. These spaces are defined to be invariant to specific graph structures and vulnerability sets, enabling a single agent to be trained and then evaluated across diverse scenarios. Moreover, using intelligent models trained for representation learning as mapping functions within the world model, we represent environment elements with semantical placement in the spaces. This enables the agent to learn transferable mappings that generalize effectively across diverse scenarios.

1) Observation Space: As discussed in Section III-B, the observation vector in this task should represent the graph discovered by the agent at a given timestep (see Appendix A for details on its structure and feature vectors). The observation mapping function should encode the discovered graph into a fixed-dimensional vector serving as the agent's observation. To obtain this vector, a pre-trained GNN is used to map nodes into p-dimensional meaningful embeddings of fixed size. An order-invariant pooling operation, POOL (e.g., *mean*), aggregates the node embeddings into a meaningful graph embedding of the same fixed dimension p. As a result, the observation vector is invariant to both graph order and size, providing a fixed-dimensional input for the agent (Figure 3).

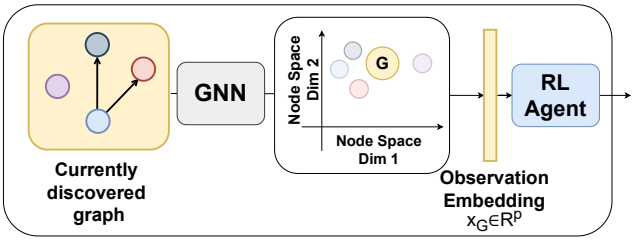

Fig. 3: The discovered graph is processed by a GNN to generate node embeddings, which are pooled into a fixed-dimensional graph embedding used as the agent's observation.

2) Action Space: As the agent is tasked with selecting a tuple *(source, target, vulnerability, outcome)* at each timestep, different models should be used to generate embeddings for each tuple element and form the *action mapping function*:

- For the source and target nodes, p-dimensional embeddings can be generated using a GNN trained for representation learning (e.g., the same GNN used to create the node embeddings in Section IV-B1).
- For vulnerability choices, embeddings can be derived from the textual descriptions using pre-trained LMs. These models, trained to understand textual information,

can generate continuous vector embeddings of size q, capturing the semantics of the vulnerability description.
- Outcome embeddings can be represented as one-hot vectors of size $|O|$, where O denotes the outcome set. Depending on this dimensionality is not critical since this set remains consistent across scenarios.

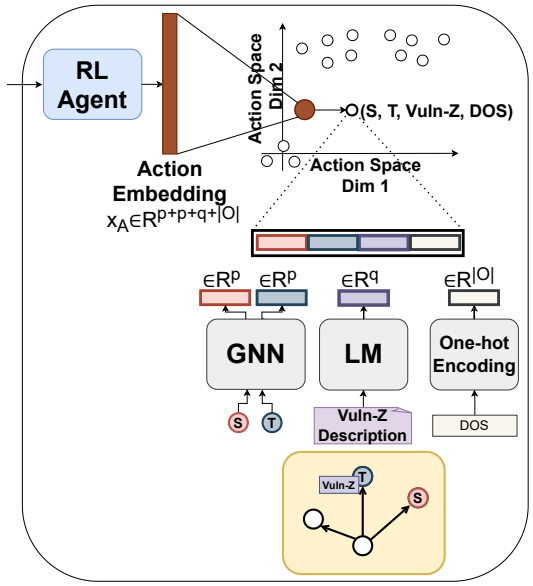

Fig. 4: Action space defined as the set of all possible concatenations of embeddings of discrete elements in the environment, with the agent tasked to navigate this space.

Once embeddings for each element of the tuple have been constructed individually (Figure 4), we need to create a unified space for the entire action space, where points represent tuples *(source, target, vulnerability, outcome)*. To construct the action space, we concatenate all possible combinations of source node embeddings with those of the target nodes, followed by the embeddings of the possible vulnerabilities and outcomes. This results in each action point being represented as a vector of dimensionality $p + p + q + |O|$, with each position semantically meaningful as the vector is derived from the concatenation of meaningful embeddings within their respective spaces. As depicted in Figure 4, the agent can subsequently be trained to generate an action vector within this space, and based on its prediction, a *proximity-based strategy* can be used to select the action vector closest to the agent's prediction. The selected action vector is mapped to the corresponding discrete action tuple in the real action space and processed by the environment.

3) Advantages of Continuous Spaces: As a result of the fixed-dimensional vectors, the agent policy is optimized using an input and output space whose dimensions are independent of the graph structure and vulnerability set, and using an order-invariant pooling for the graph embeddings ensures that the concept of graph order disappears from these spaces. As a result, the agent can be trained or tested across scenarios with varying graph structures and vulnerability sets (Equation 8), as the dimensionality of the input/output spaces now depends solely on the dimensionality of the embeddings.

$$\pi_{continuous}: \quad \mathbb{R}^p \rightarrow \mathbb{R}^{p+p+q+|O|} \tag{8}$$

This approach enables the agent to learn from two intelligent, invariant models—one for graphs and one for text—and navigate their knowledge spaces instead of treating discrete choices as atomic elements. Considering the advantages of this re-formulation:

- Continuous spaces built by intelligent models ensure **generalization** by mapping similar graphs, vulnerabilities, and nodes to adjacent embeddings, enabling knowledge transfer between scenarios with different structures but similar semantics. Unlike discrete approaches, embeddings capture the internal structure of nodes and vulnerabilities. If the agent can then be exposed to various scenarios (for example, with proper *domain randomization*), it may incorporate generalizable knowledge.
- As different studies have demonstrated on other tasks [22] [10], continuous action spaces often outperform discrete ones as the number of discrete elements increases, enhancing **scalability**. In discrete approaches, the size of observation and action vectors increases as more elements are added. In continuous spaces, adding elements leads to a more compressed observation vector and increases the density of the action space, but the vector sizes are kept constant. The scalability challenge shifts from vector dimensionality to managing space density.
- **Dynamicity** is now supported by this model, as it only triggers a change in the elements condensed into the same observation vector and the appearance or disappearance of points in the action space. However, the size of both vectors and their semantics remain consistent, allowing the same NN to process these vectors.

C. Threat Models: Reward Function

To enable the agent to learn different attacker threat models, we define a parameterized reward function with adjustable weights to prioritize specific behaviors. This setup allows us to define the desired prioritization strategy for the agent when selecting actions, using a weighted combination of bonuses and penalties (Equation 9).

$$R(o, a) = \sum_j w_j^B \cdot B_j(o, a) - \sum_k w_k^P \cdot P_k(o, a) \tag{9}$$

Each bonus and penalty depends on the agent's current observation o and selected action a, as they are also influenced by factors such as the number of assets involved and their importance (e.g., number and importance of nodes discovered).

- $B_j(o, a)$: Bonus terms that reward desirable behavior, such as discovering new nodes or escalating privileges.
- $P_k(o, a)$: Penalty terms that capture undesirable effects or costs, such as the cost of hitting firewalls, or to include the cost of exploiting a vulnerability (derived from the *Exploitability score* in the vulnerability metadata).

- w_j^B, w_k^P: Weights that determine the importance of each bonus and penalty component.

This structure enables RL agents to learn policies that align with different attack strategies by optimizing the trade-off between bonuses and penalties. An example of a prioritization strategy for a *control* threat model would be to emphasize large bonuses for outcomes such as credential access, while imposing high penalties for stopping nodes with a denial of service attack. Details on the specific bonus and penalty terms used in the study are provided in Appendix E.

D. Episode Structure

The interaction between the agent and the environment occurs in episodes. Each episode begins with the agent positioned at a randomly selected starter node within a network scenario, with only the starter node initially discovered. As the agent takes actions, it uncovers additional nodes and can exploit their vulnerabilities.

Episode example: The environment selects a scenario and a starter node (e.g., Node1). The agent begins by exploiting local vulnerabilities on Node1 (e.g., outcome *Reconnaissance*) to discover new nodes, moves to Node2 via a remote vulnerability (outcome *Lateral Movement*), escalates privileges (outcome *Privilege Escalation*), collects data (outcome *Collection*), ... The agent's policy should learn these structured behaviors by mapping observations to actions in the redesigned observation and action spaces. The environment processes each action, updates the partially discovered network graph, and returns a new observation and reward, computed according to a specific threat model. Action validity is ensured using vulnerability metadata, as detailed in Section IV-A3. The environment also maintains internal lists to track the status of discovered, owned, and disrupted nodes, updating them dynamically. Each agent episode corresponds to an attack path, with a final score assigned at termination to quantify its criticality, such as the number of owned nodes in a *control* threat model.

V. Experiments

The experimental section presents an empirical comparison of the *scalability* of both discrete and continuous spaces and the agents resulting from each. As continuous spaces allow invariance across scenarios, the study then examines the *generalization* and *dynamicity* of agents trained within them.

A. Experimental Setup

The experimental study uses *C-CyberBattleSim* as the environment and the following setup choices:

- The **environment database**, which is used to create synthetic network environments represented as graphs (as outlined in Section IV-A), was constructed using Shodan data. It identifies 172 service versions and 829 unique vulnerabilities by querying the "50 most frequently used services globally, and their top 5 versions." The specific services targeted by the study are listed in Appendix C.
- The environment database is used in combination with a set of ranges for graph-based and cyber-terrain parameters (details in Appendix D) to generate a total of 800 synthetic scenarios through **domain randomization**. Each scenario results in a distinct graph structure and a corresponding vulnerability subset.
- The **outcome approximator** integrated into the environment is SecureBERT [2] finetuned for multi-label classification of MITRE ATT&CK Tactics from the vulnerability description, with a reported test set F1 score of 76%.

The choices for RL episodes, scoring metrics, and RL libraries are as follows:

- Considering the **episode evolution**, agents undergo each training and testing episode by varying the starter node and scenario from the corresponding set (e.g., training set). The episode length scales with the graph size as $K \times \#nodes$, with K defined as the *episode-length coefficient*, ensuring unbiased scores across varying graph sizes. This finite action budget encourages the agent to prioritize the most critical vulnerability paths and maximize impact within the given action frame.
- We use *control*, *discovery*, and *disruption* goals as **experimental threat models**, conducting multiple experiments by varying the goal in each run. The **goal scores** used for the evaluation of the paths are defined here and used throughout the experiments: percentage of nodes controlled and with maximum privilege (ROOT privilege) for *control*, the amount of information gathered and exfiltrated (including node IDs and their data) for *discovery*, and the percentage of nodes disrupted for *disruption*. These scores are normalized relative to the maximum achievable score starting from the episode's initial node—i.e., the maximum number of other nodes it can control, discover, or disrupt.
- Across the studies, we follow **best practices** for RL evaluation, including hyperparameter optimization, static seeds, and reporting bootstrapped confidence intervals (BCIs) with a confidence level $\alpha = 0.95$ for statistical reliability from a limited number of runs [1]. All simulation and model parameters are documented in the attached files to ensure reproducibility.

The world model for the experiments is constructed as follows:

- For generating **node** and **graph embeddings**, a GNN was trained in an unsupervised manner on the scenario set using a Graph Auto-Encoder (GAE) architecture [25]. The GAE was optimized using a composite reconstruction loss (Equation 10) incorporating terms for node and edge feature vectors, the adjacency matrix, and a soft constraint to encourage embedding diversity:

$$\begin{aligned} \textit{Total Loss} = &\sum_{i \in V} \textit{Loss}_{\text{node}_i} + \sum_{i \in V} \sum_{j \in V} \textit{Loss}_{\text{edge}_{ij}} \\ &+ \textit{Loss}_{\text{adj(G)}} + \textit{Diversity}_{\text{Embeddings(V)}} \end{aligned} \tag{10}$$

  This loss formulation ensures that the learned embeddings capture both structural and semantic information from the

graph while maintaining diversity. The dimensionality of the embedding vector p was set to $p = 64$, with both this dimension, the GAE architecture, and its hyperparameters tuned through hyperparameter optimization. The GAE was trained independently of the RL agent, after which its encoder component was frozen and reused to generate node embeddings for the downstream task. For constructing graph-level embeddings, the node embeddings were aggregated using a combination of *mean*, *min*, and *max* pooling operations. Detailed information on the GAE training and validation process can be found in the Appendix B.

- For the generation of **vulnerability embeddings**, pre-trained LMs are used and varied between experiments. These include four general-purpose LMs (BERT [12], DistilBERT [46], GPT-2 [44], RoBERTa [30]) and four cybersecurity-specific LMs (CySecBERT [6], SecBERT [27], SecRoBERTa [27], SecureBERT [2]). All of these models have the same output layer size, producing 768-dimensional embeddings, fixing the vector dimensionality to $q = 768$. In each RL agent training run, a different pre-trained and frozen LM is selected to generate embeddings, ensuring that we perform a comparison of agents on multiple mapping functions for vulnerabilities.
- The *nearest-neighbor* selection method is used as the **proximity-based strategy**, employing *cosine distance* to address the curse of dimensionality in the high-dimensional action space.

B. Scalability Study

This study assesses the ability of discrete and continuous spaces to create agents able to learn from scenarios with increasing sizes of nodes and vulnerabilities. Increasing the number of nodes and vulnerabilities expands the agent's available options while complicating the mappings between spaces. In discrete methods, this leads to larger observation/action vectors and necessitates an NN with larger input and output layers. In contrast, continuous spaces maintain fixed input and output vectors and NN layers, with the complexity manifesting instead as a higher density of points within the spaces, as explained in Section IV-B3.

Scenario Set: A total of 300 synthetic scenarios are generated, with three scenarios (one for each threat model) created for each of the 100 unique combinations of node and vulnerability counts, both ranging from 5 to 250.

Comparison: The proposed *continuous* spaces, compared to the discrete *global* and *local* spaces used in the literature until now. Instead of comparing with individual studies using the discrete spaces, we shift the focus to comparing with the best agent for each of these spaces. Each approach used its best-performing DRL algorithm and hyperparameters, ensuring that the spaces were compared with the top model trained on them.

Study Design: Each space is used to train 300 agents over 250k iterations, each corresponding to a unique combination of graph size, vulnerability set size, and goal, with the episode-length coefficient set to $K = 30$.

Evaluation Strategy: Each run trains an agent to achieve a specific goal (using for each goal run the scores explained in Section V). We then calculate the Area Under the Curve (AUC) of the goal score evolution during the training run, normalized with respect to the AUC of the curve of an ideal agent that instantly learns the optimal policy. This metric reflects both the speed and final value of score improvements, offering a single measure of the agent's learning efficiency in each space.

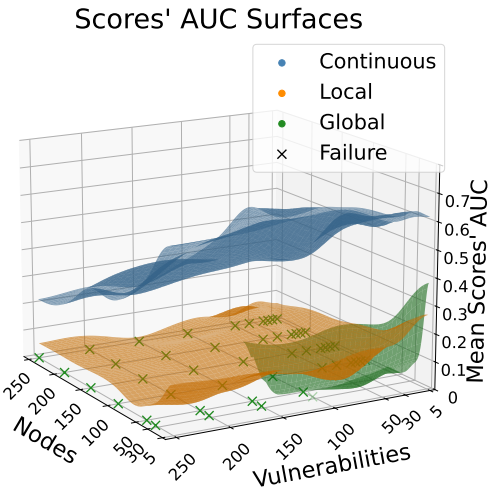

Fig. 5: The normalized average AUC score for agents learning from different spaces, with the curve depicting score progress.

Figure 5 aggregates the scores across goals per scenario size and shows the trend of the training AUC scores when the scenario sizes increase. The results show that discrete agents struggle as the number of nodes and vulnerabilities increases, since the observation and action vector sizes—and consequently the NN input and output layers—scale with these factors. While discrete global agents initially achieve higher scores, they plateau sooner than discrete local agents, and their training experiences failures due to the excessively large resulting NN model needed to process the corresponding vectors, which exceeds the capacity of the benchmark machine used for training[4]. In contrast, the discrete local agents support all experiments but achieve lower scores due to the limited visibility on a subset of nodes. Previous studies using discrete spaces in this task have, in fact, mainly focused on scenarios with either few nodes or vulnerabilities in their experimental section. To the best of our knowledge, the largest studies with POMDP environments used 30 nodes and 7 vulnerabilities for the global approach [60], and 100 nodes and 4 vulnerabilities for the local approach [55]. On the other side, continuous agents demonstrate faster and more efficient learning, with improved scalability and a smaller performance

[4]x86_64 AMD Ryzen 9 5900X 12-Core Processor, 125 GB RAM, Ubuntu 22.04.1

decline, achieving an average relative improvement of 9.3x against discrete methods, with a 7.4x improvement compared to the discrete local approach and an 11.2x improvement compared to the discrete global approach.

C. Generalization Study

The proposed continuous spaces enable agents to learn mappings on a scenario with a fixed graph structure and vulnerability set while allowing the agent to be tested on other scenarios with different graph or vulnerability configurations. Additionally, the agent can be trained across a variety of configurations to ensure it learns generalizable behavior. This capability is not available in discrete approaches, as agents are restricted to scenario-specific mappings that cannot be transferred across scenarios. We conducted generalization experiments to assess how well DRL algorithms train generalizable agents in continuous spaces, particularly in transferring knowledge from simpler to more complex scenarios (e.g., with more firewall rules).

Scenario Set: 500 synthetic scenarios generated using *domain randomization* with varying graph parameters (e.g., number of nodes in [10, 100]), cyber-terrain features (e.g., firewall rule probabilities), and vulnerability allocations. These scenarios are divided into training (60%), validation (20%), and testing (20%) sets, based on an aggregated complexity metric derived from the randomized features of each scenario (details in Appendix D). In this way, scenarios are split by increasing difficulty: training uses simpler cases, validation includes moderate ones, and testing involves the most complex.

Comparison: All the DRL algorithms from Stable-Baselines3 [45] supporting continuous action spaces (PPO [48], TRPO [47], TD3 [14], TQC [28], SAC [21], Recurrent PPO [42], A2C [33]) and a *Random* heuristic selecting at each timestep *random valid* points from the action space.

Study Design: Each algorithm is evaluated using its optimal hyperparameters, determined through hyperparameter optimization (see Appendix F for details), and is used to train 8 agents per goal for 300k iterations with $K = 10$ (limited action budget per episode). A *random switching strategy* is employed to ensure that the simulator switches scenarios and starting nodes at the beginning of each episode, throughout training, validation, and testing. This approach guarantees that the agent encounters a wide range of configurations during training, and is evaluated on multiple configurations during validation and testing.

Evaluation Strategy: The best checkpoint of each training run is determined doing *early stopping* on the validation set (scenarios of moderate complexity) and it is used on the test set (most complex scenarios) to generate 100 episodes, generating overall 24 agents $\times$ 8 algorithms $\times$ 100 episodes = 19,200 trajectories used for comparison, with the goal score used as trajectory metric (e.g., % of owned nodes with ROOT privilege for *control*).

Figure 6 moves beyond traditional binary comparisons, presenting a summary of the test set goal scores with a probability rank matrix indicating the likelihood of each algorithm in achieving a specific rank, averaging across all goal experiments. The results indicate that TRPO outperforms all other algorithms, while TD3 decisively ranks the lowest. It is noteworthy that several of the other algorithms perform at a level comparable to, or marginally better than the *random valid* heuristic, showcasing that not all algorithms are able to learn meaningful policies in continuous spaces. The TRPO algorithm, recognized as the top performer, is used to analyze scores by re-testing its agents across the different set splits, considering that they were only exposed to the training set consisting of simpler scenarios. Table I summarizes the distributional shifts in key scenario features observed in the validation and test sets relative to the training set.

TABLE I: Relative increase or decrease of selected scenario features in validation and test sets compared to training.

Feature	Validation	Test
Avg. number of nodes per scenario	1.21×	1.29×
Unique service combinations across hosts	1.04×	1.17×
Avg. firewall rule probability	1.11×	1.18×
Probability of initial user-level access on first node access	1.09×	1.16×
Probability of partial visibility on first node access	1.15×	1.23×
Probability of data presence on a node	0.87×	0.78×
Probability of service startup shutdown	1.29×	1.42×
Avg. % nodes discoverable across all starter nodes	0.92×	0.88×
Avg. % nodes reachable via lateral movement across all starter nodes	0.88×	0.82×
Avg. % nodes vulnerable to disruption across all starter nodes	0.86×	0.80×

The results in Figure 7 show that scenario complexity impacts the *control* and *disruption* goals, while *discovery* maintains more consistent performance across different scenario sets. To summarize generalization performance, we compute the mean ratio between the test set scores (representing the most complex scenarios) and the training set scores (corresponding to the simplest scenarios and those encountered during training). On average, TRPO agents achieve an 89% test-to-training score ratio, demonstrating effective generalization to more complex scenarios.

D. Dynamicity Study

TRPO has been used to analyze generalization under dynamic conditions, a critical requirement for deployment in online environments [29]. To this aim, we introduce a *random events probabilistic defender* into the test set scenarios, maintaining the same experimental study of Section V-C. The defender agent intervenes on each node with probability p_n after each attacker action, and we visualize the resulting effect on the episode scores for the attacker agent. When intervening, the defender randomly selects one of four actions with equal probability, applied to a randomly chosen service or firewall rule on the node:

- **Starting a service** (potentially beneficial)
- **Stopping a service** (potentially detrimental)

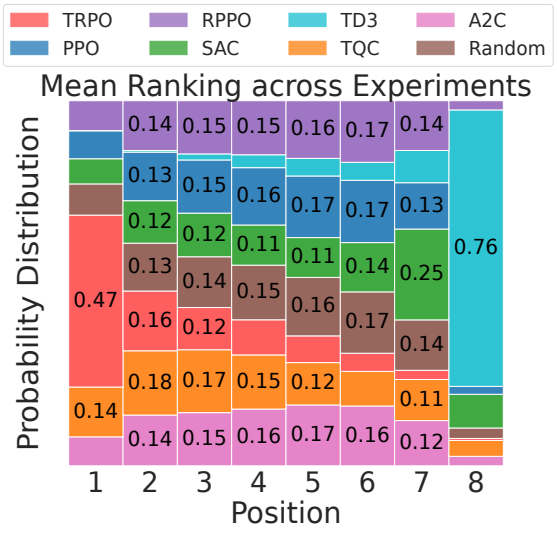

Fig. 6: Rank probability matrix highlighting the likelihood of each DRL algorithm producing agents that achieve a certain rank across all goals. Bars represent non-negligible probabilities greater than 10%.

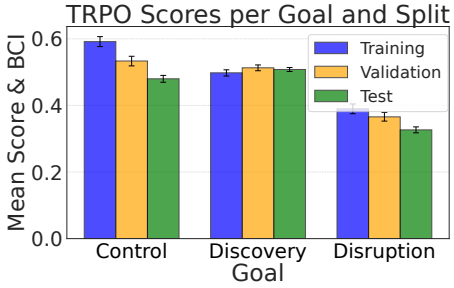

Fig. 7: TRPO agent scores for each goal and scenario set.

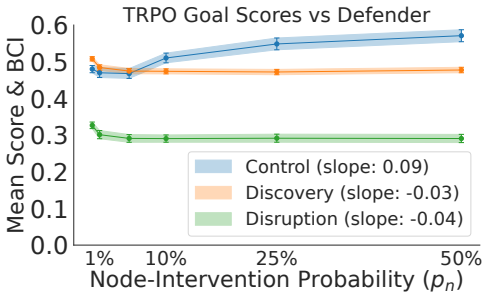

Fig. 8: TRPO agent test set scores vs defender probability.

- **Allowing a firewall rule** (potentially beneficial)
- **Blocking a firewall rule** (potentially detrimental)

These actions alter the network services and firewalls, making certain vulnerabilities exploitable or not. Consequently, the observation vector changes (as node embeddings vary), and the set of valid points in the action space shifts accordingly. Whether the defender's action benefits or harms the attacker depends on which action is randomly selected. The results (Figure 8) indicate that agents trained for *discovery* and *disruption* goals experience a slight performance decline (i.e., a reduction in the average amount of information discovered and nodes disrupted), but still generalize effectively, even with high intervention probabilities (up to 50% of nodes that undergo at least a change after each attacker agent action). In contrast, agents trained for the *control* goal show improved scores by effectively leveraging beneficial changes to gain control over more nodes, while also exhibiting resilience to detrimental changes.

VI. Real-World Applicability

The simulation environment released with this paper not only enables the generation of synthetic scenarios but also allows the creation of scenarios by importing vulnerability scans and representing them as virtual replicas. The deployment pipeline for agents in these replicas follows these steps:

1) A virtual replica, with the same structure, overlays the application scenario, optionally updated with periodic vulnerability scan snapshots.
2) Missing features and information (e.g., data presence) are either provided by the administrator or probabilistically varied to represent various scenarios.
3) Continuous spaces are constructed using a set of mapping functions (GNN and LM of choice) to generate embeddings based on the graph and vulnerability descriptions.
4) Agents can either be retrained on the virtual replicas and then deployed in them for the prediction of paths, or trained on synthetic scenarios and then only deployed to the virtual replicas. The deployment involves running the agents for a configurable number of episodes N, with a budget of actions (episode-length coefficient K) from selected starter nodes, producing critical paths.
5) The generated critical paths—logs detailing sequences of source, target, vulnerability, and outcome—can help administrators allocate defenses based on research questions (e.g., identifying frequently targeted nodes), with their efficacy measured by the associated episode scores.
6) *(Optional)* The pipeline can be re-iterated to evaluate defense strategies by re-deploying the agents after the modifications and tracking any score deterioration.

We validate this pipeline by using virtual replicas derived from real-world and emulated network scans of organizational scenarios. Additionally, we use this experiment to evaluate the research question: *Can agents trained exclusively on synthetic scenarios generalize to realistic scenarios, and is their performance comparable to that of agents trained directly in those realistic environments?*

Scenario Set: Two scenarios of 100 nodes: one derived from a scan of an emulated network (350 vulnerabilities, adapted from [40]) and one from a real-world network (784 vulnerabilities), with distinct vulnerability sets and distributions (as represented in Figure 9).

Comparison: 1) Continuous TRPO agents *retrained* specifi-

cally on each application scenario for 300k iterations and then re-tested in the corresponding scenario seen during training; 2) Continuous TRPO agents trained solely on *synthetic* scenarios using domain randomization (as done in Section V-C) and then tested zero-shot on the unseen application scenarios; 3) A heuristic strategy prioritizing vulnerabilities with the highest *impact scores* (maximize damages); and 4) a heuristic strategy prioritizing *exploitability scores* (maximize ease of exploitation).

Study Design: Each trained agent and heuristic is used to generate $N = 240$ trajectories (episodes) for each of the application scenarios using $K = 30$ (allow longer episodes and monitor near-to-maximum impact).

Evaluation Strategy: Average goal scores from testing episodes are used to assess the criticality of the attack paths and determine whether an agent or heuristic is more effective for generating these paths for a given goal.

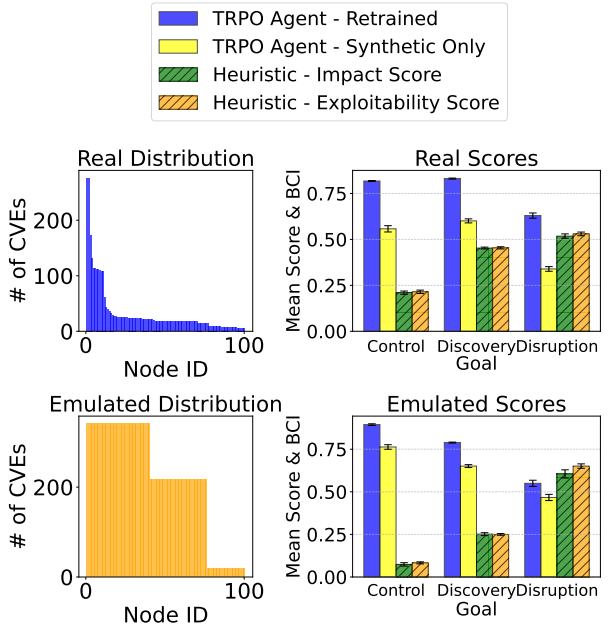

Fig. 9: TRPO agent, retrained or not, tested against heuristics on both emulated and real-world replica environments.

1) Agents vs Heuristics: Results (Figure 9) show that both agents (retrained and trained solely on synthetic scenarios) outperform heuristics on both the real-world and emulated scans for the *control* and *discovery* goals. Heuristics and agents show no clear winner for the *disruption* goal, underscoring the challenge for agents of learning disruption strategies. This is further complicated by having heuristics that are more effective in this case, as vulnerabilities with high exploitability and impact scores typically result in disruption as a potential outcome.

2) Retrained vs. Synthetic-Only Agents: As expected, agents retrained on the specific application scenarios outperform those trained solely on synthetic data. However, computing the average ratio of scores between synthetic-only and retrained agents shows that synthetic-only agents achieve 75% of the scores of retrained agents (ratio of scores)—84% in the emulated scenario and 65% in the real-world scenario. This validates the generalization of continuous agents, showing that those trained on synthetic scenarios can achieve competitive results, with retrained agents as the upper bound. Future research may bridge the sim-to-real gap when synthetic-trained agents match the performance of those trained on specific application scenarios. While retraining on application-specific data ensures high-performance agents, it may not be feasible in privacy-sensitive contexts where vulnerability scans are restricted. In such cases, agents trained on synthetic data can, in fact, provide a viable alternative.

3) Execution Time: The total execution time of the agent on the replica environments depends on several factors, including the episode-length coefficient K and the number of episodes N. For instance, on a benchmark personal computer[5], each iteration (action) of the agent on the replica of the real-world network with 100 nodes and 784 vulnerabilities takes approximately 0.0215 ± 0.0016 seconds. Using $K = 10$ and $N = 10$ episodes, the agent would take approximately 3 minutes and 35 seconds on average for the entire run.

VII. Conclusions & Future Work

In this study, we present two main contributions: (1) an automated and more realistic simulation environment that leverages CTI tools and vulnerability databases for scenario generation, along with an automated strategy for approximating vulnerability exploitation, and (2) the proposal of continuous, generalizable observation and action spaces, which decouple the agent's input and output from specific network structures and vulnerability sets, enabling the agent to avoid re-training when deployed on other scenarios, scale to unprecedented sizes, and generalize effectively across diverse, complex, and realistic scenarios. The *C-CyberBattleSim* environment, provided alongside this paper, enables the re-scraping of data to expand the environment database, generates new scenarios, and supports the training and testing of both discrete and continuous agents. The scenarios used in the experimental section, along with the hyperparameters employed, are attached to ensure reproducibility. Future work will focus on: transitioning from MITRE ATT&CK *Tactics* to *Techniques* for greater detail in vulnerability outcome approximation, expanding the focus to vulnerabilities beyond the service level, and further broadening the heuristic reward function.

Acknowledgment

This work has been partially supported by the French National Research Agency under the France 2030 label (Superviz ANR-22-PECY-0008). The views reflected herein do not necessarily reflect the opinion of the French government.

[5]x86_64 AMD Ryzen 7 PRO 7840U with Radeon 780M Graphics, 30 GB RAM, Ubuntu 22.04.1

References

[1] Agarwal, R., Schwarzer, M., Castro, P.S., Courville, A.C., Bellemare, M.: Deep reinforcement learning at the edge of the statistical precipice. Advances in Neural Information Processing Systems (2021). https://doi.org/10.48550/arXiv.2108.13264

[2] Aghaei, E., Niu, X., Shadid, W., Al-Shaer, E.: Securebert: A domain-specific language model for cybersecurity. In: Security and Privacy in Communication Networks (2023). https://doi.org/10.1007/978-3-031-25538-0_3

[3] Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M.: Optuna: A next-generation hyperparameter optimization framework (2019)

[4] Ampel, B., Samtani, S., Ullman, S., Chen, H.: Linking common vulnerabilities and exposures to the mitre att&ck framework: A self-distillation approach (2021). https://doi.org/10.48550/arXiv.2108.01696

[5] Applebaum, A., Dennler, C., Dwyer, P., Moskowitz, M., Nguyen, H., Nichols, N., Park, N., Rachwalski, P., Rau, F., Webster, A., Wolk, M.: Bridging automated to autonomous cyber defense: Foundational analysis of tabular q-learning. In: Proceedings of the 15th ACM Workshop on Artificial Intelligence and Security. AISec'22 (2022). https://doi.org/10.1145/3560830.3563732

[6] Bayer, M., Kuehn, P., Shanehsaz, R., Reuter, C.: Cysecbert: A domain-adapted language model for the cybersecurity domain. ACM Trans. Priv. Secur. (2024). https://doi.org/10.48550/arXiv.2212.02974

[7] Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for hyper-parameter optimization. In: Advances in neural information processing systems. vol. 24 (2011)

[8] Booth, H., Rike, D., Witte, G.A.: The national vulnerability database (nvd): Overview (2013)

[9] Brown, R., Lee, R.M.: The evolution of cyber threat intelligence (cti): 2019 sans cti survey. Tech. rep., SANS Institute (February 2019)

[10] Chandak, Y., Theocharous, G., Kostas, J., Jordan, S., Thomas, P.S.: Learning action representations for reinforcement learning (2019), https://arxiv.org/abs/1902.00183

[11] Cody, T.: A layered reference model for penetration testing with reinforcement learning and attack graphs. In: 2022 IEEE 29th Annual Software Technology Conference (STC) (2022). https://doi.org/10.1109/STC55697.2022.00015

[12] Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding (2019)

[13] Dulac-Arnold, G., Evans, R., van Hasselt, H., Sunehag, P., Lillicrap, T., Hunt, J., Mann, T., Weber, T., Degris, T., Coppin, B.: Deep reinforcement learning in large discrete action spaces (2016). https://doi.org/10.48550/arXiv.1512.07679

[14] Fujimoto, S., van Hoof, H., Meger, D.: Addressing function approximation error in actor-critic methods (2018). https://doi.org/10.48550/arXiv.1802.09477

[15] Gangupantulu, R., Cody, T., Park, P., Rahman, A., Eisenbeiser, L., Radke, D., Clark, R.: Using cyber terrain in reinforcement learning for penetration testing (2022). https://doi.org/10.1109/COINS54846.2022.9855011

[16] Garg, U., Sikka, G., Awasthi, L.K.: Empirical analysis of attack graphs for mitigating critical paths and vulnerabilities. Computers & Security **77**, 349–359 (2018)

[17] Gelada, C., Kumar, S., Buckman, J., Nachum, O., Bellemare, M.G.: Deepmdp: Learning continuous latent space models for representation learning (2019), https://arxiv.org/abs/1906.02736

[18] Grattarola, D., Zambon, D., Bianchi, F.M., Alippi, C.: Understanding pooling in graph neural networks. IEEE Transactions on Neural Networks and Learning Systems **35**(2), 2708–2718 (2024). https://doi.org/10.1109/TNNLS.2022.3190922

[19] Guo, X., Ren, J., Zheng, J., Liao, J., Sun, C., Zhu, H., Song, T., Wang, S., Wang, W.: Automated penetration testing with fine-grained control through deep reinforcement learning. Journal of Communications and Information Networks . https://doi.org/10.23919/JCIN.2023.10272349

[20] Ha, D., Schmidhuber, J.: World models (2018). https://doi.org/10.5281/ZENODO.1207631, https://zenodo.org/record/1207631

[21] Haarnoja, T., Zhou, A., Abbeel, P., Levine, S.: Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor (2018). https://doi.org/10.48550/arXiv.1801.01290

[22] van Hasselt, H., Wiering, M.A.: Using continuous action spaces to solve discrete problems. In: 2009 International Joint Conference on Neural Networks (2009). https://doi.org/10.1109/IJCNN.2009.5178745

[23] Hu, Z., Beuran, R., Tan, Y.: Automated penetration testing using deep reinforcement learning. In: 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) (2020). https://doi.org/10.1109/EuroSPW51379.2020.00010

[24] Husák, M., Komárková, J., Bou-Harb, E., Čeleda, P.: Survey of attack projection, prediction, and forecasting in cyber security. IEEE Communications Surveys & Tutorials (2019). https://doi.org/10.1109/COMST.2018.2871866

[25] Kipf, T.N., Welling, M.: Variational graph auto-encoders (2016), https://arxiv.org/abs/1611.07308

[26] Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks (2017), https://arxiv.org/abs/1609.02907

[27] Kun: Secbert: A pretrained language model for cyber security text (2024)

[28] Kuznetsov, A., Shvechikov, P., Grishin, A., Vetrov, D.: Controlling overestimation bias with truncated mixture of continuous distributional quantile critics (2020). https://doi.org/10.48550/arXiv.2005.04269

[29] Li, Q., Wang, R., Li, D., Shi, F., Zhang, M., Chattopadhyay, A., Shen, Y., Li, Y.: Dynpen: Automated penetration testing in dynamic network scenarios using deep reinforcement learning. Trans. Info. For. Sec. (2024). https://doi.org/10.1109/TIFS.2024.3461950

[30] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V.: Roberta: A robustly optimized bert pretraining approach (2019). https://doi.org/10.48550/arXiv.1907.11692

[31] Liu, Z., Shi, J., Buford, J.F.: Cyberbench: A multi-task benchmark for evaluating large language models in cybersecurity (2024)

[32] Mell, P., Scarfone, K., Romanosky, S.: Common vulnerability scoring system. IEEE Security Privacy **4**(6), 85–89 (2006). https://doi.org/10.1109/MSP.2006.145

[33] Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T.P., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning (2016). https://doi.org/10.48550/arXiv.1602.01783

[34] Monahan, G.E.: State of the art—a survey of partially observable markov decision processes: Theory, models, and algorithms. Management Science (1982). https://doi.org/10.1287/mnsc.28.1.1

[35] Muratore, F., Gienger, M., Peters, J.: Assessing transferability from simulation to reality for reinforcement learning. IEEE Transactions on Pattern Analysis and Machine Intelligence **43**(4), 1172–1183 (Apr 2021). https://doi.org/10.1109/tpami.2019.2952353

[36] Nguyen, T., Chen, Z., Hasegawa, K., Fukushima, K., Beuran, R.: Pengym: Pentesting training framework for reinforcement learning agents. In: Proceedings of the 10th International Conference on Information Systems Security and Privacy (2024). https://doi.org/10.1016/j.cose.2024.104140

[37] Olswang, A., Gonda, T., Puzis, R., Shani, G., Shapira, B., Tractinsky, N.: Prioritizing vulnerability patches in large networks. Expert Systems with Applications **193**, 116467 (2022). https://doi.org/https://doi.org/10.1016/j.eswa.2021.116467

[38] van den Oord, A., Vinyals, O., Kavukcuoglu, K.: Neural discrete representation learning (2018), https://arxiv.org/abs/1711.00937

[39] Ou, X., Govindavajhala, S., Appel, A.W.: MulVAL: A logic-based network security analyzer. In: 14th USENIX Security Symposium (USENIX Security 05) (2005)

[40] Palma, A., Angelini, M.: It is time to steer: A scalable framework for analysis-driven attack graph generation. In: Computer Security – ESORICS 2024 (2024). https://doi.org/10.1007/978-3-031-70903-6_12

[41] Pennachin, C., Goertzel, B.: Contemporary approaches to artificial general intelligence. In: Artificial general intelligence, pp. 1–30. Springer (2007)

[42] Pleines, M., Pallasch, M., Zimmer, F., Preuss, M.: Generalization, mayhems and limits in recurrent proximal policy optimization (2022). https://doi.org/10.48550/arXiv.2205.11104

[43] Puterman, M.L.: Markov decision processes: Discrete stochastic dynamic programming (1994)

[44] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.: Language models are unsupervised multitask learners (2019)

[45] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., Dormann, N.: Stable-baselines3: Reliable reinforcement learning implementations. Journal of Machine Learning Research (2021)

[46] Sanh, V., Debut, L., Chaumond, J., Wolf, T.: Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter (2020). https://doi.org/10.48550/arXiv.1910.01108

[47] Schulman, J., Levine, S., Moritz, P., Jordan, M.I., Abbeel, P.: Trust region policy optimization (2017). https://doi.org/10.48550/arXiv.1502.05477

[48] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms (2017). https://doi.org/10.48550/arXiv.1707.06347
[49] Shah, S., Mehtre, B.: An overview of vulnerability assessment and penetration testing techniques. Journal of Computer Virology and Hacking Techniques (2015). https://doi.org/10.1007/s11416-014-0231-x
[50] Sheyner, O., Haines, J., Jha, S., Lippmann, R., Wing, J.: Automated generation and analysis of attack graphs. In: Proceedings 2002 IEEE Symposium on Security and Privacy (2002). https://doi.org/10.1109/SECPRI.2002.1004377
[51] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al.: A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science **362**(6419), 1140–1144 (2018)
[52] Strom, B.E., Applebaum, A., Miller, D.P., Nickels, K.C., Pennington, A.G., Thomas, C.B.: Mitre att&ck: Design and philosophy. In: Technical report. The MITRE Corporation (2018)
[53] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction (2018)
[54] Team., M.D.R.: Cyberbattlesim (2021), created by Seifert C., Betser M., Blum W., Bono J., Farris K., Goren E., Grana J., Holsheimer K., Marken B., Neil J., Nichols N., Parikh J., Wei H.
[55] Terranova, F., Lahmadi, A., Chrisment, I.: Leveraging deep reinforcement learning for cyber-attack paths prediction: Formulation, generalization, and evaluation. In: Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024) (2024). https://doi.org/10.1145/3678890.3678902
[56] Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., Abbeel, P.: Domain randomization for transferring deep neural networks from simulation to the real world. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 23–30 (2017). https://doi.org/10.1109/IROS.2017.8202133
[57] Wang, S.C.: Artificial Neural Network, pp. 81–100. Springer US, Boston, MA (2003). https://doi.org/10.1007/978-1-4615-0377-4_5
[58] Wang, Y., Liu, S., Wang, W., Zhou, C., Zhang, C., Jin, J., Zhu, C.: A unified modeling framework for automated penetration testing (2025), https://arxiv.org/abs/2502.11588
[59] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P.S.: A comprehensive survey on graph neural networks. IEEE Transactions on Neural Networks and Learning Systems (2021). https://doi.org/10.1109/TNNLS.2020.2978386
[60] Yao, Q., Wang, Y., Xiong, X., Li, Y.: Intelligent penetration testing in dynamic defense environment. In: Proceedings of the 2022 International Conference on Cyber Security (2023). https://doi.org/10.1145/3584714.3584716
[61] Zhao, W.X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.Y., Wen, J.R.: A survey of large language models (2023). https://doi.org/10.48550/arXiv.2303.18223
[62] Zhou, S., Liu, J., Lu, Y., Yang, J., Hou, D., Zhang, Y., Hu, S.: April: Towards scalable and transferable autonomous penetration testing in large action space via action embedding. IEEE Transactions on Dependable and Secure Computing **22**(3), 2443–2459 (2025). https://doi.org/10.1109/TDSC.2024.3518500

APPENDIX A
GRAPH REPRESENTATION

The graph representation used in this study to model the environment at timestep t (Figure 10) consists of graph nodes that represent network hosts. Each node's feature vector combines several attributes:

- **firewall_config_array**: An array detailing the node's firewall configuration. Each element corresponds to a port and indicates whether an outgoing or incoming BLOCK rule is present.
- **listening_services_running_array**: An array with binary values indicating whether each service on the node is currently running.

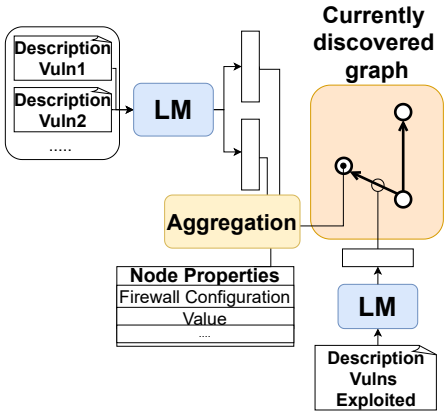

Fig. 10: Graph representation of the environment, illustrating node and edge feature vectors.

- **visible**: A binary value representing the level of visibility of the node—whether it is fully visible or partially visible.
- **persistence**: A binary value indicating whether the agent has achieved persistence on the node.
- **has_data**: A flag indicating whether data is present on the node, treated as a binary value.
- **data_collected**: A binary value indicating whether data has been collected from the node.
- **data_exfiltrated**: A binary value indicating if data has been exfiltrated from the node.
- **defense_evasion**: A binary value reflecting whether the agent has evaded the defense systems on the node (firewall and potential defenders).
- **privilege_level**: An integer denoting the agent's privilege level on the node (*No Access*, *Basic User*, *ROOT*).
- **status**: An integer value representing the current status of the node, such as running, re-imaging, or stopping.
- **value**: Importance value assigned to the node.
- **sla_weight**: The weight associated with the node's service level agreement, reflecting its criticality.
- **vulnerabilities_embedding**: A vector with the pooled embeddings of vulnerabilities, derived with LMs.
- **listening_services_embeddings**: A vector with the pooled embeddings of service names, derived with LMs.

Edges of the graph represent the vulnerabilities exploited between nodes up to the current timestep, with their feature vectors aggregating the embeddings of the exploited vulnerabilities. This structure enables the agent to encapsulate the current state of discovery, the historical context of vulnerabilities exploited up to the current timestep, and the semantics of vulnerabilities within the network.

APPENDIX B
GRAPH AUTO-ENCODER

The GAE employed in this study consists of a GNN-based *Encoder* followed by a simple feed-forward NN *Decoder*.

Feature	Description	Range used	Sampling Strategy	Complexity Categorization
Number of nodes	Number of graph nodes in the scenario	[10, 100]	Random	Ascending
Number of services per node	Number of services sampled per node	[1, 3]	Random	Ascending
Homogeneity value	Real value determining the amount of homogeneity present in the scenario in service assignment	[0.05, 0.33]	Random	Descending
Percentage of nodes per type	Proportion of nodes belonging to each category (e.g., *Windows*) in the generated scenario	[0.05, 0.95]	Random	N/A
Firewall incoming rule probability	Probability that for every service of every node, there is a blocking rule incoming	[0, 0.33]	Random	Ascending
Firewall outgoing rule probability	Probability that for every service of every node, there is a blocking rule outgoing	[0, 0.33]	Random	Ascending
Probability of neighbor information release	Probability applied to each vulnerability that reveals neighboring node IDs, determining how many nodes will be discovered exploiting this vulnerability	[0.2, 0.8]	Determined based on the *Confidentiality Impact* of the vulnerability	Descending
Probability of data presence	Probability that each node has data to collect (applied only if the node has at least one vulnerability allowing "Collection" outcome)	[0.2, 0.8]	Random	Descending
Probability of partial visibility on nodes	Probability that each node will be partially visible once owned (applied only if the node has at least one vulnerability allowing "Discovery" outcome)	[0.2, 0.8]	Random	Ascending
Probability of need for escalation	Probability that the node, when owned, starts with a "basic user" level and not "root" level (applied only if the node has at least one vulnerability allowing "Privilege Escalation" outcome)	[0.2, 0.8]	Random	Ascending
Probability of service shutdown	Probability that every service of every node will start with a shutdown status	[0, 0.2]	Random	Ascending
Success rate	Probability representing the success rate for every vulnerability, used to simulate failure of vulnerability exploitation	[0.6, 1]	Determined based on the *Attack Complexity* of the vulnerability	Descending
Node value	Value assigned to each node, representing its importance in the scenario	[1, 100]	Random	N/A

TABLE II: Description of scenario features, including information about which range has been sampled during scenario generation, with which sampling strategy, and the resulting complexity categorization.

The overall goal is to compress and decompress node feature vectors along with the surrounding graph structure into meaningful p-dimensional continuous embeddings. The model training uses unsupervised learning with the reconstruction loss described in Equation 10. After training, the *Decoder* is discarded, leaving the *Encoder* to generate meaningful node embeddings. The GAE was trained on 500 scenario graphs (the same set of the *Generalization Study* in Section V-C), following hyperparameter optimization, and using early stopping on a validation set to select the most generalizable checkpoint (Figure 11).

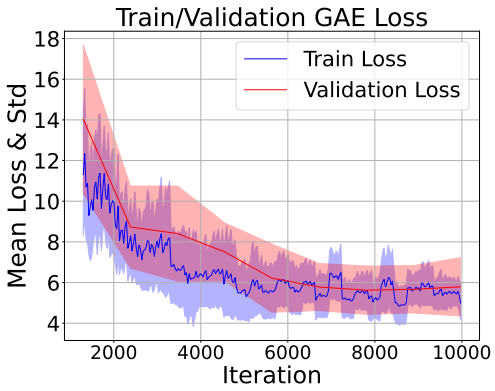

Fig. 11: GAE reconstruction training and validation loss.

Appendix C
Vulnerabilities Data

In the experimental section (Section V), scenarios are constructed using the 50 most vulnerable services identified by Shodan. For each service, up to five of the most vulnerable versions were selected. However, due to data availability constraints, some services had fewer than five versions listed, resulting in a total of 172 service versions. Vulnerabilities associated with these versions were retrieved from the NVD using their Common Platform Enumeration identifiers. This process resulted in 829 unique vulnerabilities, each of which may affect one or more service versions. During the scenario generation process, these services are assigned to nodes representing different device types (e.g., IoT nodes), ensuring that each node type is assigned only services relevant to its category. The following are the services identified from the Shodan query (at the timestep of 18 November 2024) and the node types simulated:

Services: Nginx, Apache httpd, Squid http proxy, Microsoft IIS httpd, Exim smtpd, Jetty, MongoDB, Remote Desktop Protocol, Jenkins, Wildix Collaboration, Apache Tomcat, NET-DK, Metabase, Outlook Web App, Tengine, VMware ESXi, OpenSSH, OpenResty, MySQL, Control Web Panel, Apache Tomcat/Coyote JSP engine, Nextcloud, Elastic, lighttpd, GoAhead Embedded Web Server, Hikvision IP Camera, RabbitMQ, Boa Web

	Learning Rate	Batch Size	γ	N Steps	Ent Coef	Value Coef	Max Grad Norm	Target KL	Tau	Buffer Size	Target Entropy	Target Policy Noise	GA(λ)	Num Layers	NN Channels	Out Channels	Activation
PPO	**0.00001**, 0.0001, 0.001	32, 64, **128**	**0.9**, 0.95, 0.99	512, **1024**, 2048	0.01, **0.1**, 0.2		0.1, **0.3**, 0.5										
TRPO	0.0001, **0.001**, 0.01	**64**, 128, 256	0.9, **0.95**, 0.99	1024, 2048, **4096**				**0.01**, 0.05, 0.1					**0.9**, 0.95, 0.99				
A2C	0.0001, **0.001**, 0.01		0.9, **0.95**, 0.99	5, 10, **20**	**0.01**, 0.1, 0.2	0.5, **1.0**, 1.5	0.3, **0.5**, 0.7										
SAC	**0.00001**, 0.0001, 0.001	32, 64, **128**	0.9, **0.95**, 0.99						0.0005, 0.001, **0.005**	**100000**, 500000, 1000000	-2, -1.5, **-1.0**						
TD3	**0.0001**, 0.001, 0.01	64, 128, **256**	0.9, **0.95**, 0.99						**0.001**, 0.005, 0.01	**100000**, 500000, 1000000		0.1, **0.2**, 0.3					
TQC	**0.0001**, 0.001, 0.01	**64**, 128, 256	0.9, 0.95, **0.99**		0.01, 0.1, **0.2**				0.001, **0.005**, 0.01	100000, **500000**, 1000000							
GAE	0.000001, 0.00001, 0.00005, 0.0001, 0.001, **0.01**	16, **32**, 64, 128												**2**, 3, 4	**16**, 32, 64	16, 32, **64**	LeakyReLU, ReLU, **null**

TABLE III: Hyperparameters and ranges explored for each of the different algorithms undergoing the optimization process. Optimal hyperparameters found are highlighted in bold.

Server, Grafana, micro_httpd, Boa HTTPd, DrayTek Vigor Series (2925, 2862, 2860, 2762, 2912, 2926, 2133, 2927, 2865), ZTE H268A, ZTE ZXHN H168N, Cisco Systems, ZTE F680, Bbox.

Types: Windows Host, Unix Host, IoT node, Industrial Control System (ICS) node, Router.

APPENDIX D
GRAPH AND CYBER-TERRAIN PARAMETERS

Several factors have been incorporated to enhance the realism of the simulated scenarios, categorized into graph parameters and cyber-terrain parameters. Table II provides a detailed description of these features, which collectively define a scenario along with the corresponding service/vulnerability allocations. For each generated scenario, the values of these features are sampled from the specified ranges, as indicated in the table. The sampling process depends on the CVSS vector of specific vulnerabilities (if the feature is tied to a particular vulnerability) or random sampling (if the feature is tied to the whole scenario), as detailed in the proper column of the table. The complexity categorization column outlines how each feature affects the complexity score, determining whether the scenario's complexity increases (ascending feature) or decreases (descending feature) with an increase in the value of this feature. The complexity score used in this study is defined in Equation 11.

$$\textit{Complexity Score} = \sum_{i \in \text{Ascending}} w_i \cdot \text{Norm}_{[0,1]}(f_i) - \sum_{j \in \text{Descending}} w_j \cdot (1 - \text{Norm}_{[0,1]}(f_j)) \tag{11}$$

Each feature range has been normalized using min-max scaling to the range [0, 1] in order to ensure consistent scaling across all features. The sums of components determining the score can be adjusted with appropriate weights to emphasize certain factors over others in determining the overall complexity. In our experiments, all these weights were set to 1, giving equal importance to each feature.

APPENDIX E
REWARD FUNCTION

The components of the reward function are organized into bonuses and penalties. These terms have been carefully selected and adjusted based on the specific goals (*control*, *discovery*, *disruption*) and their semantics. The specific values assigned for each goal are detailed in the configuration files provided with the paper.

Reward Bonuses: Node Owned, Node Disrupted, Node Discovered, Data Collected, Privilege Escalation, Acquired Visibility, Data Exfiltrated, Defense Evaded, Episode Won

Reward Penalties: Vulnerability Cost, No Enough Privileges, Success Rate Failed, No Data to Collect, No Data to Discover, No Data to Exfiltrate, Already Persistent, Machine Already Stopped, Node Already Owned, Node Already Visible, Already Defense Evasion, Scanning Unopen Port, Privilege Escalation in Node Not Owned, Privilege Escalation to Level Already Had, Blocked by Local/Remote Firewall, Episode Lost

APPENDIX F
HYPERPARAMETER OPTIMIZATION

For the hyperparameter optimization process, experiments utilized the Optuna library [3], specifically employing the

Tree-Structured Parzen Estimator [7] optimization algorithm. A total of 25 trials were conducted for each algorithm being optimized (DRL algorithm and GAE). Table III lists the ranges explored for each algorithm. The objective used for the GAE is the minimization of the reconstruction loss on the validation set. For RL algorithms, the objective used is the maximization of the AUC of the validation goal scores (averaging across the three experimental goals). The following hyperparameters are instead kept constant throughout the study for the algorithms:

- **NN Layer Sizes**: For all DRL algorithms, the sizes of the NN layers were set to [256, 128, 64]. For RecurrentPPO, an LSTM layer was added of size [32].
- **Activation Function**: LeakyReLU was used as the activation function for the NN.
- **Optimizer**: The Adam optimizer was used for both DRL algorithms and GAE. A value of 1×10^{-7} was set for the epsilon parameter. The weight decay was fixed at 1×10^{-4}.
- **GAE Architecture**: The GAE consisted of one NNConv layer as the first layer. This ensures that edge feature vectors are used in the computation of the embeddings. Subsequent layers are GCNConv layers.

All other hyperparameters, not explicitly mentioned, were left at their default values as defined by the respective libraries.