NASimJax: A GPU-Accelerated Policy Learning Framework for Penetration Testing

Raphael Simon
Royal Military Academy,
Vrije Universiteit Brussel
Brussels, Belgium
r.simon@cylab.be

José Carrasquel
Royal Military Academy
Brussels, Belgium

Wim Mees
Royal Military Academy
Brussels, Belgium

Pieter Libin
Vrije Universiteit Brussel
Brussels, Belgium

ABSTRACT

Penetration testing—the practice of simulating cyberattacks to identify vulnerabilities—is a complex sequential decision-making task that is inherently partially observable and features large, growing action spaces. Training reinforcement learning (RL) agents for this domain faces a fundamental bottleneck: existing simulators are too slow to train on realistic network scenarios at scale, resulting in policies that fail to generalize. We present NASimJax, a complete JAX-based reimplementation of the Network Attack Simulator (NASim), achieving training speeds of up to 1.6M steps/s—a 100x improvement over the original. By running the entire training pipeline on hardware accelerators, NASimJax enables experimentation on larger networks under fixed compute budgets that were previously infeasible. We formulate automated penetration testing as a Contextual POMDP and introduce a redesigned network generation pipeline that produces structurally diverse, realistic, and guaranteed-solvable scenarios, providing a principled basis for studying zero-shot policy generalization. To address the challenge of rapidly growing action spaces, we propose a two-stage action selection (2SAS) variant of PPO that decomposes decisions into host selection followed by per-host action selection, and show that it outperforms flat action masking as network size increases. We further evaluate Domain Randomization and Prioritized Level Replay for zero-shot transfer across network topologies of varying density, revealing that training context has a significant impact on out-of-distribution performance. NASimJax thus provides a fast, flexible, and realistic platform for advancing research on RL-based penetration testing.

KEYWORDS

Reinforcement Learning, Cyber-Security, Generalization

1 INTRODUCTION

Cybersecurity has become one of the most critical domains in modern society, underpinning healthcare, banking, and energy infrastructure. Among the tools used to proactively defend these systems is penetration testing — the authorised practice of simulating attacks to identify vulnerabilities and misconfigurations in IT networks [24]. While effective, penetration testing is laborious and expensive, requiring seasoned security experts to reason over long decision horizons, weigh competing attack vectors, and adapt to the specifics of each target network. Automating this process via reinforcement learning (RL) has only recently gained traction [20], motivated by the success of RL in other complex sequential decision-making domains such as games [23, 37**?** , 38] and robotics [26]. The penetration testing problem maps naturally onto a Partially Observable Markov Decision Process (POMDP) [29, 30]: an agent cannot directly observe the full network topology or its vulnerabilities, and must actively acquire information before acting. Beyond this inherent partial observability, training effective RL policies for penetration testing faces challenges common to real-world RL applications [6] — most notably the sim-to-real gap and generalization to unseen scenarios [20]. Real-world penetration testing involves long decision horizons, sparse and delayed rewards, combinatorial action spaces, and strong dependence on network-specific structure. Policies that overfit to narrow training scenarios will fail to generalize, and closing this gap requires simulators that expose agents to structural diversity and decision-relevant complexity rather than merely increasing surface-level realism. Existing penetration testing simulators [7, 31, 34] fall short on two fronts. First, they are primarily fixed to single or narrowly parameterized scenarios, providing insufficient diversity to train generalizable policies. Second, they rely on CPU-bound Python implementations, which creates a fundamental throughput bottleneck: RL requires millions of environment interactions due to sample inefficiency, yet these simulators cannot generate experience fast enough to fully utilize modern accelerators. This bottleneck is especially costly in penetration testing, where RL algorithms are not yet well understood and thorough hyperparameter searches — crucial for reliable results in any new domain, given the known sensitivity of RL to hyperparameter choices — are simply intractable at the speeds current tools offer. To address these challenges, we present NASimJAX, a full JAX-based reimplementation and extension of the Python-native Network Attack Simulator (NASim) [31]. NASimJAX preserves the core task abstraction of NASim — modelling penetration testing as a partially observable sequential decision process — but substantially reworks the environment design to support distributional training, curriculum learning, and context-dependent generalization. By running the entire training pipeline on hardware accelerators, NASimJAX enables vectorized training across thousands of parallel environment instances, eliminating the CPU-GPU communication

Proc. of the Adaptive and Learning Agents Workshop (ALA 2026), Aydeniz, Delgrange, Mohammedalamen, Yang (eds.), May 25 – 26, 2026, Paphos, Cyprus, https://alaworkshop2026.github.io/. 2026.

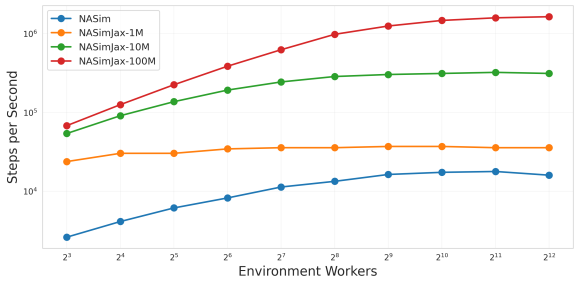


Figure 1: Training speed comparison between NASim with 10M steps of training budget against NASimJAX with 1M, 10M and 100M steps. Number of environment workers are doubled every time. Results show the impact of JAX's JIT-compilation on total runtime. Details of the speed test are available in Section 6.1. The full results are in Appendix A.

bottleneck and achieving up to 100× speed-up over the original. This throughput unlocks experimentation on larger and more complex networks under fixed compute budgets, and makes systematic hyperparameter search feasible on a single entry-level GPU.

Beyond computational improvements, NASimJAX introduces several conceptual advances. We formulate automated penetration testing as a Contextual POMDP [8], where each episode is conditioned on a context describing the underlying network instance. This provides a principled framework for training policies across a distribution of environments and directly facilitates zero-shot policy generalization [16] to previously unseen networks. Realizing this formulation required a fundamental redesign of the network generation process: NASimJAX introduces a new generation function that produces structurally diverse, realistic, and guaranteed-solvable scenarios, enabling fine-grained control over topology and vulnerability density. Building on this setup, we experimentally evaluate Domain Randomization (DR) [36] and Prioritized Level Replay (PLR) [14] for zero-shot policy transfer across network topologies, and show that training context has a significant impact on out-of-distribution generalization.

2 PRELIMINARIES

2.1 Contextual Reinforcement Learning Setting

We formalize our setting as a contextual reinforcement learning problem. We begin with the standard MDP definition and progressively extend it to partially observable and contextual settings.

Markov Decision Process. An MDP [28] is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \rho_0, \gamma \rangle$ where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P}(s'|s,a)$, is the transition probability distribution over next states, conditioned on the current state and action; $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, a reward function; $\rho_0 \in \Delta(\mathcal{S})$, the initial state distribution, from which an initial state can be sampled: $s_0 \sim \rho_0$; and $\gamma \in [0,1]$ a discount factor, hat determines the relative weighting of future rewards. The objective is to learn a policy $\pi(a|s)$, a probability distribution over actions conditioned on the current state, that maximizes the expected cumulative discounted reward $\mathbb{E}_\pi[G_0]$ where $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ is called the return. r_t is a random variable that represents the reward obtained at time step t. In finite-horizon tasks the sum is clipped to T, the maximum number of allowed steps.

Partially Observable Markov Decision Process. A POMDP [33] extends the MDP framework to a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \Omega, \mathcal{O}, \rho_0, \gamma \rangle$, where Ω represents the observation space and $\mathcal{O}(o|s',a)$ defines the observation function. The agent does not directly observe the underlying state. Only parts of it can be observed at a given time through $\mathcal{O}$. The optimal policy now depends on the history of observations or belief state $b_t \in \Delta(\mathcal{S})$.

Contextual Markov Decision Process. Following [9] and the formulation adopted in [16], a Contextual Markov Decision Process (CMDP) is a special class of POMDP with a context variable $c \in C$, where C denotes the context space and c is sampled at the beginning of each episode from a distribution $p(c)$. Conditioned on c, the environment induces a POMDP with context-dependent dynamics and rewards. Formally, a contextual POMDP is defined by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}_c, \mathcal{R}_c, \Omega, \mathcal{O}_c, \rho_{0,c}, \gamma, C, p(c) \rangle$, where $\mathcal{P}_c(s' \mid s,a)$ and $\mathcal{R}_c(s,a)$ denote the transition and reward functions parameterized by c, $\mathcal{O}_c(o \mid s',a)$ is the observation function, and $\rho_{0,c}$ is the initial state distribution conditioned on c. The joint distribution over context and initial state factorizes as $p(c,s_0) = p(c)\,\rho_{0,c}(s_0)$. The context remains fixed throughout an episode but varies across episodes according to $p(c)$, thereby inducing a distribution over tasks. The objective is to learn a policy that maximizes expected return under the context distribution, i.e., $\mathbb{E}_{c\sim p(c)}\,[\mathbb{E}_\pi\,[G_0 \mid c]]$.

2.2 Generalization in RL

Generalization in deep reinforcement learning refers to an agent's ability to perform well on environments or states not seen during training, and it presents significant challenges distinct from supervised learning. Unlike standard machine learning where training and test distributions are typically well-defined, deep RL agents can overfit in subtle ways—achieving optimal rewards during training while exhibiting drastically different test performance [4, 39]. This overfitting can occur "robustly", meaning that commonly used stochasticity techniques do not necessarily prevent or detect it [39]. The challenges stem from multiple sources: the lack of systematic evaluation protocols comparable to cross-validation in supervised learning, the difficulty of separating representation learning from long-term planning and exploration, and the fact that theoretical generalization bounds scale poorly with large state and action spaces [17, 39]. A fundamental insight from recent work is that generalization in RL can be framed as an implicit partial observability problem—when an agent is uncertain about which environment it is in, this creates an "epistemic POMDP" that makes optimal generalization inherently difficult [8]. Furthermore, procedurally generated benchmarks like Procgen have revealed that policies trained on limited training levels often fail dramatically on held-out test levels, even when achieving near-optimal training performance [3, 4]. As deep RL is increasingly applied to critical real-world domains like healthcare, finance, and autonomous systems, understanding and addressing these generalization gaps has become crucial for safe deployment [6, 17, 39].

3 RELATED WORK

3.1 JAX and Hardware-Accelerated RL

JAX [2] is a Python library for accelerator-oriented array computation and program transformation, designed for high-performance numerical computing and large-scale machine learning. Its core features — just-in-time (JIT) compilation and functional transformations such as `vmap` — make it particularly well suited for RL workloads, where tight coupling between environment simulation and policy optimization is critical. By requiring adherence to functional programming principles, JAX enables the entire training loop, including both environment stepping and policy updates, to be compiled and executed on accelerators. This eliminates the CPU-GPU communication overhead that plagues conventional Python-based RL frameworks, where the environment typically runs on the CPU while the policy trains on the GPU. Beyond compilation gains, `vmap` enables vectorization of the environment's `step` and `reset` functions, allowing large batches of experience to be collected and processed in parallel. The adoption of JAX-based RL environments has only recently begun, with examples including the Gymnax suite [18], Craftax [22], Jumanji [1].

3.2 Penetration Testing Simulators

There exist several different penetration testing environments targeted at learning RL policies. NASim [31] has been one of the first and therefore featured in several subsequent works. NASimEmu [12] and PenGym [25], are extensions to NASim that bring forward an emulation component, allowing to learn policies in small simulated networks and transfer them to an emulated network consisting of virtual machines. StochNASim [32] features stochastic environment resets and networks of varying sizes to learn robust penetration testing policies. `CyberBattleSim` [34] aims at simulating the phase of lateral movement (moving between hosts) in a network. `CybORG++` [7] simulates both offensive and defensive agents and their dynamics within the same network. `C-CyberBattleSim` [35], concentrates on the generation of more realistic scenarios through real-world information sources, such as vulnerability databases and security scanners–elements that are added to the network nodes as additional information.

4 NASIMJAX

We now describe the RL environment provided by NASimJAX. We first outline the underlying task and abstraction, before detailing the state and observation spaces, action space, and reward function, and then describing how learning contexts are defined.

The environment is designed as a research abstraction rather than a fixed benchmark scenario. Its primary goal is to support controlled variation over network instances, enabling research on policy robustness and generalization beyond fixed scenarios. Through a versatile network generation process, the environment can be scaled in complexity and difficulty, allowing it to remain challenging as RL algorithms improve and preventing overfitting to narrowly defined scenarios.

NASimJAX models automated penetration testing as a sequential decision-making problem under partial observability. An agent interacts with a network by performing actions such as scanning hosts, enumerating services, exploiting vulnerable software to obtain an initial foothold, and escalating privileges through vulnerable processes. Information about the network topology, running services, and vulnerabilities must be actively acquired through interaction. Progress through the network is inherently sequential and long-horizon, as successful exploitation of certain hosts may be required to reach others. An episode terminates once the agent has obtained the required privileges on a predefined set of sensitive hosts.

Each episode is conditioned on a learning context, which we also call scenario, represented by a concrete network instance. A context consists of a set of hosts partitioned into subnets, connectivity rules defining how subnets may communicate, and host-specific configurations such as running services, processes, and associated exploitability. While the agent's policy is shared across episodes, the underlying context varies, providing a principled mechanism to expose the agent to diverse network structures during training. This formulation naturally supports viewing the environment as a CMDP, where generalization is achieved by learning policies that perform well across a distribution of network contexts rather than memorizing individual attack paths.

The environment adheres to the Gymnax API [18]. This design choice enables seamless integration with a wide range of JAX-based RL algorithms, facilitating large-scale experimentation, reproducibility, and direct comparison of learning methods within a standardized environment.

4.1 State

The state of a network is defined by two main elements: Traffic rules, describing which subnets can talk to one another, and the collective properties of all hosts. Since IT networks naturally follow a graph-structure, we represent this via an adjacency matrix. We add another dimension to the adjacency matrix to accommodate firewall rules, that determine which services are allowed to pass between subnets. The networks' host contain the following properties: `subnet address`, `reachable`, `discovered` `access level`, `OS`, `services`, `processes`.

The `services` and `processes` attributes encode only those services and processes that are vulnerable and therefore exploitable by the agent. They do not represent the complete set of software running on a host, but rather a filtered abstraction capturing actionable attack surfaces. As a result, hosts may have zero vulnerable services or processes, modelling systems that are correctly configured or hardened.

To make use of JAX's parallel computation capabilities to it's fullest, the host properties are batched together into one data structure. All the host properties are either one-hot encoded or represented via boolean flags. This has two advantages. First, for most operations in the state transition logic we can solely rely on boolean operations which, have a very small computational overhead. Second, state properties can be stored using unsigned integers that only use one byte in memory, reducing the overall memory footprint of the environment, therefore allowing for more parallel environments during training.

When an agent interacts with the environment, only three host values can change: *reachable*, *discovered*, and *access level*. All other

values remain static after the scenario is generated. While the full state is used internally for transition dynamics, it is intentionally not exposed to the agent, reinforcing partial observability and preventing shortcut learning.

4.2 Action Space

The action space is discrete and consists of six action types, grouped into reconnaissance and exploitation actions. The four reconnaissance actions are `OS Scan`, `Process Scan`, `Service Scan`, and `Subnet Scan`. The first three retrieve host-specific information, while the subnet scan enables the discovery of additional hosts and subnets once sufficient privileges have been obtained on a target host. The two exploitation actions are `Exploit` and `Privilege Escalation`. Exploits are remote actions targeting vulnerable services to gain initial access to a host, whereas privilege escalation actions leverage vulnerable processes to elevate access privileges after a foothold has been established.

The concrete action set is generated based on the set of possible operating systems and vulnerable services and processes defined for the environment. Exploitation actions are constructed from all combinations of operating systems and services, while privilege escalation actions are constructed from all operating system and process combinations. Actions are instantiated per target host, ensuring that a sufficiently expressive set of actions exists to solve any generated network. Consequently, the size of the action space is given by $|\mathcal{A}| = |\mathcal{H}| \times (|\mathcal{A}_{\text{scan}}| + |O| \times (|\mathcal{V}_{\text{svc}}| + |\mathcal{V}_{\text{proc}}|))$ where $\mathcal{H}$ denotes the set of hosts, $\mathcal{A}_{\text{scan}}$ the set of scan actions, O the set of possible operating systems, and $\mathcal{V}_{\text{svc}}$ and $\mathcal{V}_{\text{proc}}$ the set of vulnerable services and processes. As network complexity increases, this construction leads to a rapidly growing action space, significantly increasing exploration difficulty.

4.3 Observation Space

The observations perceived by the agent contain only the outcome of the executed action. The outcome depends on the action type and on whether execution was successful. Successful `OS Scans`, `Service Scans`, and `Process Scans` retrieve the corresponding properties, in addition to the host and subnet address, whereas a successful `Subnet Scan` reveals all hosts within a subnet and any connected subnets. Further, `Exploit` and `Privilege Escalation` reveal a host's OS and service/process, and whether it is sensitive or not. Additionally, the following information is appended to the flattened observation: the encoding of the action type that was executed, and four mutually exclusive flags regarding the outcome of that action (successful, connection error, permission error, and undefined error).

4.4 Reward Function

Each type of action in the environment has a cost associated to them. `Exploit` and `Privilege Escalation` actions incur a higher cost than scanning for instance. The reward function is defined by Equation 1. The value of hosts is denoted by V_h. By scanning the subnet, new hosts can be discovered. The value of discovering a host is denoted by V_d. The reward is multiplied by the number of *newly* discovered hosts, n_{dh}. Hosts may only be discovered once. In all other scenarios, the returned reward is the cost of the action a_t.

$$r_t = \begin{cases} -cost(a_t) + V_d n_{dh} & \text{for successful subnet scans,} \\ -cost(a_t) + V_h & \text{for successful priv. esc.,} \\ -cost(a_t) & \text{else.} \end{cases} \tag{1}$$

4.5 Network Generation

Network generation proceeds in several stages and is designed to balance realism, diversity, and guaranteed solvability of the resulting scenarios. Each generated network represents a distinct learning context while adhering to structural constraints that ensure meaningful penetration testing tasks. Figure 2 provides an illustration for this process.

We begin by randomly distributing a fixed number of hosts N_h across a fixed number of subnets N_s. Among these subnets, two are assigned special semantic roles that remain fixed across all generated scenarios. One subnet represents the Internet and contains a single host corresponding to the attacker's machine. A second subnet represents a demilitarized zone (DMZ), which serves as a buffer between external and internal infrastructure. All remaining subnets are treated as arbitrary internal subnets.

Host-level properties are then assigned. Each host is randomly assigned an operating system according to a fixed distribution. Services and processes are distributed independently across hosts using the service density svc_d and process density $proc_d$ parameters, respectively. These parameters control the average number of exploitable services and processes present in the network.

Host sensitivity is assigned independently with probability s_d, i.e., Pr(host is sensitive) = s_d. This mechanism allows the generator to model varying levels of organizational security maturity, including misconfigurations or users not following security best practices.

To ensure that every generated scenario is solvable and supports meaningful agent interaction, we enforce several feasibility constraints after the initial random assignment. First, every subnet is guaranteed to contain at least one host running at least one service, ensuring that the agent can pivot into every subnet that becomes reachable. Second, every sensitive host is guaranteed to run at least one service and at least one process, which ensures that privilege escalation is possible on all sensitive machines. If any of these conditions are violated, the minimal number of required services or processes is added randomly to the affected hosts. These post-generation checks guarantee that successful episodes are achievable without constraining the overall diversity of generated networks.

After host and subnet properties are fixed, we generate the network topology. Connectivity between subnets is represented by an adjacency matrix of size $N_s \times N_s$, where entries are sampled according to the topology density parameter t_d, which controls the probability that two subnets are connected. To reflect basic network security restrictions, we impose that the Internet subnet may only communicate with the DMZ. To ensure that hosts within the same subnet can always communicate, the diagonal of the adjacency matrix is populated with ones.

The resulting topology matrix is intentionally not required to be symmetric, meaning that connections between subnets are directed.

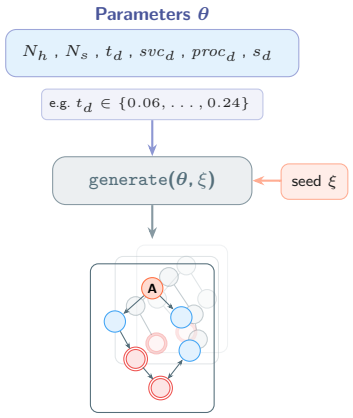


Figure 2: Illustration of the network generation process. Number of hosts (N_h), and subnets (N_s), topology (t_d), service (s_d), process (p_d) and sensitive host density (s_d) are all parameters that influence the generated networks. The blue nodes represent normal hosts, the red node labelled A represents the attacker's position, and the double circled red nodes represent sensitive hosts.

Combined with the random sampling process, this asymmetry can lead to subnets that are unreachable from the attacker's starting position. Such subnets effectively become inactive for a given episode. While this may reduce the number of reachable hosts, it serves an important role in shaping the learning problem. Networks with fewer reachable subnets induce shorter horizons and smaller effective state spaces, whereas fully connected networks result in more complex attack paths. This mechanism provides a natural way to generate a curriculum of environments with varying difficulty levels without explicitly staging scenarios.

Although the total number of hosts N_h is fixed in order to allocate static memory layouts required for JIT compilation, the number of active hosts—those belonging to reachable subnets—varies across generated networks. As a result, a distribution over effective problem sizes is obtained even when N_h is held constant. This property enables curriculum learning by exposing agents to scenarios of increasing complexity during training. We illustrate this variability empirically in Section 6.2.

In addition to fully random host configurations, we support the generation of homogeneous subnets to better reflect real-world network administration practices, where hosts within the same subnet are often configured with similar software stacks. In this mode, services and processes are sampled jointly for all hosts in a subnet from Beta distributions parameterized by svc_d and $proc_d$. This approach allows controlled variation in intra-subnet diversity while preserving the overall density of services and processes.

4.6 Context Sets

The design of the environment allows for several context sources. One approach is to simply use a seed, which then influences the random number generators used during the network generation phase. Train and evaluation phases can as such be seeded differently, akin to Cobbe et al. [4]. Further, generation parameters may be changed, to create denser or more sparse networks. The process and service density in hosts, describing how many of them are vulnerable, can also be adjusted. Changing these parameters provides the opportunity to evaluate on out-of-distribution networks, and assess zero-shot transfer capabilities of policies as discussed in Kirk et al. [17]. Importantly, the context c is not included in the agent's observations; the agent must act under uncertainty about the current network configuration, inferring relevant information through interaction. This corresponds to the unobserved-context case of the CMDP as described by Kirk et al. [17].

5 METHODOLOGY

5.1 Algorithms

We now present the selected algorithms and methods. Our implementation is based on the pure JAX PPO implementation from Lu et al. [21]. PPO has been widely used and shown successes in most areas of RL. This field is no exception, and many other works have used it to learn policies for penetration testing [19, 32]. As the environment is both partially observable and features a growing action space as the number of hosts increases, we discuss the choices that have been made to help learn with these challenges.

5.1.1 Partial Observability. To handle the challenge of partial observability, we implement a wrapper around the environment that maintains a record of information discovered throughout an episode. This follows the methodology described by Simon et al. [32], where the most recent observation is kept and a condensed history of past observations is appended. This approach is analogous to how human penetration testers operate—keeping track of scan results to decide which actions to take next on the target hosts. While using both the latest observation and the aggregated history increases memory consumption, it indirectly alleviates the credit assignment problem by providing richer temporal context, allowing the agent to better associate actions with delayed outcomes.

5.1.2 Large Action Spaces. To learn in large action spaces, we test two distinct methods. The first is using normal action masking, to help with exploration by disallowing actions that are invalid given the current state. The second is more sophisticated, whereby we condition the action to be executed on the selected host.

Action Masking. Lots of training budget in RL can be spent on actions that are invalid given the current state. Action masking has been widely used in the literature to address this issue [15, 37**?**]. For a discrete action space, action masking modifies the categorical distribution by giving invalid actions an infinitesimal value before applying the softmax function. This results in actions whose sampling probability becomes virtually zero. Huang et al. [11] have further analysed this method, and shown that it produces valid policy gradients. In our setting, we mask all actions for hosts that have both not been discovered yet, and are not reachable. Additionally, we mask all exploits and privilege escalation actions that are invalid due to the host not running the right combination of OS and service or OS and process respectively. We chose to not mask actions that are invalid due to missing privilege levels, as

this is a learnable feature contained within the environment the observations (cf. Section 4.3).

Two-Stage Action Selection (2SAS). In network attack simulations the flat action space grows linearly with the number of hosts, which dilutes exploration and slows learning. Inspired by the factored action decomposition used in DeepNash for Stratego—where the agent first selects a piece and then a legal move for that piece [27]—we decompose each decision into two stages: *host selection* followed by *action selection* on the chosen host. Concretely, the actor-critic network shares a common feature trunk and branches into two policy heads. The first head outputs a distribution over hosts, masked to exclude unreachable or undiscovered targets. Given the sampled host, a learned host embedding is concatenated with the trunk representation and passed to the second head, which outputs a distribution over per-host actions, again masked to remove invalid choices. Because each head operates over at most $\max(H, A/H)$ categories rather than the full product $|\mathcal{A}| = H \cdot (A/H)$, the effective decision complexity at each stage is substantially reduced. During training we compute separate importance ratios for the two stages and multiply them to form the combined ratio used in the PPO clipping objective, with independent entropy bonuses for each head. We refer to this scheme as *two-stage action selection* (*2SAS*).

5.1.3 Reward Scaling. Since networks are procedurally generated, the number of sensitive hosts N_s varies across learning contexts. This leads to high variance in the cumulative reward per episode, which can destabilize value function estimation and advantage calculation. To mitigate this, we scale the rewards such that the maximum potential return is approximately invariant to the network size. Specifically, for a given context C, the reward r_t at each step is scaled by the theoretical maximum reward obtainable:

$$\hat{r}_t = \frac{r_t}{N_s \cdot V_h} \tag{2}$$

where V_h is the reward value for compromising a sensitive host. This transformation ensures that completing a scenario—regardless of its host density—yields a normalized return on a consistent scale, thereby providing a more stable learning signal across the diverse distribution of network topologies.

5.1.4 UED Methods. Priozitized Level Repaly (PLR) [14] works in conjunction with procedurally generated environments, and aims to form a natural curriculum of levels that are still deemed *interesting* for the learning progression. These can be levels where the agent currently exhibits the highest regret. PLR performs gradient updates on the random levels, whereas robust PLR (denoted as PLR$^{\perp}$) only updates the agent on the levels sampled from the buffer. This is theoretically justified and (in some domains) empirically results in higher performance [13]. We compare the different UED methods against Domain Randomization (DR) [36], where at each end of an episode, a new network is generated from the parameterized distribution (cf. Section 4.5). We use the implementation provided by the JaxUED library [5].

5.2 Hyperparameter Tuning

We perform a hyperparameter search using Bayesian-sampling for every discussed algorithm in Section 5.1, with a budget of 250 trials. Each sampled hyperparameter set is trained on 26 host networks, on three separate seeds to evaluate the robustness of the parameters. JAX allows for these three runs to be made in parallel on the same device. We report the aggregated performance of those runs. The performance of a run is reported as the mean solve rate over the set of evaluation levels during the final evaluation phase. After the search, we evaluate the top ten most promising hyperparamters on a set of five seeds, to select the most robust set of parameters. Further details and all the search spaces can be found in Appendix B.

6 EXPERIMENTS

The aim of our experiments is to investigate three different questions: (1) Does NASimJax indeed bring the expected speed-up compared to the original implementation? (2) Which method to help in large action spaces is more appropriate given our setting (3) Which UED-methods discussed in Section 5.1 perform best on tasks related to ZSPT? These latter experiments are intended to be exploratory and diagnostic rather than definitive, providing interesting insights and further showcasing the capabilities of the introduced framework, NASimJax.

6.1 Performance Comparison

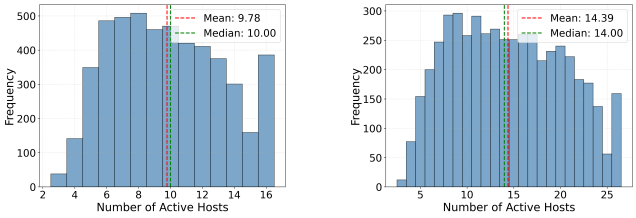
To investigate the performance increase compared to the original implementation, we perform several training runs with an increasing number of environment workers, starting with 8 and doubling until reaching 4096. We highlight that we train an actual policy, instead of comparing performance of random policies against one another, as this provides a more accurate measure of the performance to be expected when using the environment. For NASim, we use `cleanRL`'s [10] implementation of PPO and compare it against NASimJax using the `PureJaxRL`'s [21] PPO implementation. We match the hyperparameters, such that we collect the same amount of experience per rollout. The difference in training speeds is shown by Figure 1. At it's peak NASimJAX trained over 100 million steps achieves a performance of 1.6M steps per second, which amounts to a speed-up of 100 times over the original. The machine we perform these experiments on is equipped with an Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz and an NVIDIA RTX A4000. These results pronounce NASimJax's ability to train policies with a greater budget, as compiling the code takes up a significant amount of time when using small training budgets. We also tried comparing our implementation against CyberBattleSim [34], but the environment does not support running parallel environment workers by default, therefore now allowing us to perform the same experiment.

6.2 Environment Configurations

Rather than prescribing fixed benchmark scenarios, NASimJax is designed as a configurable research framework. The network generation pipeline exposes a set of parameters: topology density, total host count, service and process density, sensitive host density (cf. Section 4.5). These allow researchers to instantiate environments aligned with their specific research interests. To ground the experimental investigation that follows, we describe two reference configurations of increasing complexity that we use throughout this work in Table 1. We emphasize that these are illustrative rather

Table 1: Network Configuration Parameters

	16 Hosts	26 Hosts
Processes	3	3
Services	3	3
Operating Systems	2	2
Subnets	7	10
Topology Density	0.15	0.12
Service Density	0.7	0.7
Process Density	0.7	0.7
Sensitive Density	0.2	0.15
Distribute Homogeneously	True	True
Step Limit	300	300
Action Space Size	256	416



(a) 16 Hosts **(b) 26 Hosts**

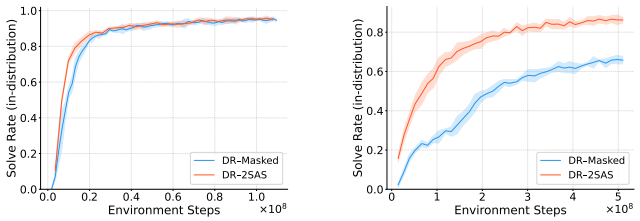
Figure 3: Visualization of active host counts within generated networks of 16 and 26 hosts. The full network parameters are displayed in Table 1. Visualizations for every network size and t_d are available in Appendix C

than normative. A network increases in complexity via two factors: (1) More *active* hosts in the network means a longer horizon, and more unmasked actions. This directly affects the amount of exploration required in the early stages of training before finding good trajectories to bootstrap on. (2) A larger number of hosts, coupled with services, processes and OSes directly affects the size of resulting action space as established in Section 4.2.

Parameters such as the sensitive and topology density were carefully chosen to enable the generation of networks with a varied number of *active* hosts. We illustrate the resulting active host distributions in Figure 3, and the complete set of Figures for each t_d displayed in Table 2 can be found in Appendix C. The topology density allows us to control the number of active hosts in the network. The sparser the topology, the more subnets will be disconnected from the main network, and thus not play a role in training.

6.3 Growing Action Spaces

To compare the performance of our two methods to help learning policies in large action spaces, we train both algorithms on our environment configurations established in Section 6.2. Results can be seen in Figure 4. For 16 host networks, the training budget is 100M steps, while for 26 host networks, it is 500M. Each algorithm is trained on five seeds to also test stability. The periodic evaluation of policies over 50 different evaluation networks is done in-distribution, using the same t_d, as we're interested in the capabilities of handling growing action spaces of the respective methods, not generalization. The results showcase that for smaller action spaces, 2SAS and DR Masked perform equally. For 26 host networks on the other hand, the advantage of decomposing the action selection in two stages is clearly pronounced, with DR-2SAS achieving a solve rate of 82% vs. DR-Masked's 66%. The full hyperparameters we used are displayed in Table 7.



(a) 16 Hosts **(b) 26 Hosts**

Figure 4: Comparison between DR with Action Masking and DR with 2SAS for 16 and 26 host networks.

6.4 Zero-Shot Policy Transfer

Since we implement the environment as a CMDP, this experiment aims to test different learning contexts in the form of active host distributions, influenced by the topology density (t_d) parameter, and how these affect the evaluation performance on distinct active host distributions. For every network described in Table 1, we choose five topology densities per network size to create varying distributions (cf. Section 6.2). The UED-algorithms used in this experiment are DR, PLR and PLR$^{\perp}$ (cf. Section 5.1.4). We combine these with our two methods of handling growing action spaces: Masking and 2SAS, to further investigate their influence on the solve rate on OOD networks.

For each algorithm, we train on the lower t_d, one on the middle t_d, and one on the higher t_d, highlighted in Table 2. During training the policy is periodically evaluated on each of the five total densities. The full hyperparameters we used are displayed in Table 7.

Table 2: Topology densities t_d used per network size. Densities used for training are bold.

Hosts	Topology Density t_d				
16	**0.06**	0.115	**0.15**	0.195	**0.24**
26	**0.04**	0.08	**0.12**	0.16	**0.20**

The results displayed in Figure 5 showcase that for 16 host networks, the impact of evaluating on a different t_d than we trained on is fairly low, and all methods achieve a similar solve rate, with PLR$^{\perp}$ exhibiting some more variance when trained on $t_d = 0.24$ networks. Things change drastically for 26 host networks on the other hand. We're able to see that training on lower t_d networks, which results in less active hosts, leads to the best overall solve rate for the tested methods. With the exception of DR-2SAS, that performs

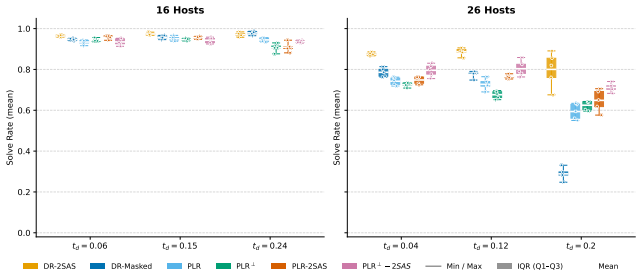


Figure 5: ZSPT across topology densities t_d for 16- and 26-host networks. Each algorithm is trained on three values of t_d (low, mid, high) and evaluated on all other t_d. Bars show the mean solve rate aggregated over five seeds; boxes span the IQR (Q1–Q3), whiskers the full range.

best when trained on $t_d = 0.12$ networks. Training on $t_d = 0.2$, the highest topology density, results in worse evaluation scores for the PLR-based methods, with DR-Masked not even achieving a solve rate of 0.2 across all 50 evaluation levels. The complete plots showing per t_d performance can be found in Appendix D.

7 DISCUSSION

While our experiments focus on ZSPT across network topologies of varying density, the framework is designed to support a broader range of research questions. The number of services and processes can be increased to further exacerbate the large action space problem, and the topology and sensitivity densities can be tuned to systematically vary task difficulty independently of network size. This positions NASimJax not as a fixed benchmark, but as a configurable platform for investigating the breaking points of current methods under controlled conditions.

Training context and zero-shot transfer. The results in Figure 5 reveal that training on lower topology density networks tends to yield better overall generalization across densities. We attribute this to an implicit curriculum effect: sparser topologies produce networks with fewer active hosts on average, exposing the agent to shorter-horizon tasks early in training and allowing it to build competence before encountering more complex attack paths. This effect is less pronounced for DR-2SAS, which benefits from the more structured exploration afforded by two-stage action selection, and performs best when trained at mid-density.

Reward scaling. The reward scaling introduced in Section 5.1.3 is not merely a training stabilization heuristic-it serves two distinct purposes. First, it ensures that the learning signal reflects the structural difficulty of a network, such as the sparsity of vulnerabilities, and number of sensitive hosts, rather than its size alone. A large but sensitive-sparse network may in fact be easier to solve than a smaller but sensitive-dense one; without scaling, this distinction is obscured. Second, reward scaling is a prerequisite for PLR to function correctly in this setting. Without it, regret estimates are biased toward larger networks simply because they accumulate higher raw returns, causing PLR's level selection to conflate network size with learning potential. Scaling makes regret estimates comparable across contexts, allowing PLR to prioritize levels based on genuine learning progress.

PLR in the penetration testing setting. The PLR-based methods perform comparably to DR in most conditions, which is consistent with findings in other domains. DR has been shown to be a highly competitive baseline even in settings where PLR was originally proposed [5, 14]. The largest difference though can be seed for 26 host networks and a high t_d of 0.2, where PLR performs significantly better than DR, due to the ability to replay networks with a lower active host count. Using 2SAS in combination with PLR does result in better performance than PLR with Masking, but its performance compared to DR-2SAS is still inferior. This can be attributed to the high complexity of the hyperparameter search space, as a result of trying to combine two complex methods with one another.

Future extensions. Several natural extensions to the environment remain as future work. The introduction of firewall rules between subnets would add another layer of structural complexity to the network generation process. Moving toward a two-player formulation with both offensive and defensive agents would make the environment dynamic rather than static, and is an important step for studying more realistic adversarial scenarios. Heterogeneous attacker starting positions and time-varying network configurations are further directions that would increase the realism of generated scenarios without sacrificing the controlled variability that makes the framework useful for research.

8 CONCLUSION

In this work, we introduced NASimJax, the first RL environment for learning penetration testing policies written in pure JAX. By implementing the environment as a partially observable Contextual MDP, NASimJax provides a principled framework for investigating ZSPT to unseen network topologies-an important step towards further investigating generalization in this domain. Alongside this formulation, we redesigned the network generation pipeline to produce more structurally diverse and realistic scenarios, enabling the creation of training distributions that better reflect the complexity and diversity of real IT infrastructure. Compared to earlier CPU-bound simulators, NASimJax achieves a 100x speed improvement, reaching 1.6M steps per second on a single entry-level GPU. This performance gain unlocks experimentation at a scale that was previously infeasible, and we leverage it to investigate generalization and ZSPT to out-of-distribution networks. Finally, as the action space grows linearly with the number of hosts, we introduced a two-stage action selection variant of PPO that decomposes each decision into host selection followed by action selection, substantially reducing effective decision complexity and enabling learning on larger networks. Together, these contributions establish NASimJax as a fast, and flexible platform for advancing research on RL-based penetration testing.

ACKNOWLEDGMENTS

This research was funded by the Royal Higher Institute of Defence under the project DAP23/05. Further support comes from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" program. The resources and

services used in this work were, in part, provided by the VSC (Flemish Supercomputer Center), funded by the Research Foundation - Flanders (FWO) and the Flemish Government. Pieter Libin acknowledges support from the Research council of the Vrije Universiteit Brussel (OZR-VUB) via grant number OZR3863BOF. We also thank Elli Makdis Antoun, Hicham Azmani, Bram Silue and all other lab members that have provided feedback throughout the creation of this work.

REFERENCES

[1] Clément Bonnet, Daniel Luo, Donal Byrne, Shikha Surana, Sasha Abramowitz, Paul Duckworth, Vincent Coyette, Laurence I. Midgley, Elshadai Tegegn, Tristan Kalloniatis, Omayma Mahjoub, Matthew Macfarlane, Andries P. Smit, Nathan Grinsztajn, Raphael Boige, Cemlyn N. Waters, Mohamed A. Mimouni, Ulrich A. Mbou Sob, Ruan de Kock, Siddarth Singh, Daniel Furelos-Blanco, Victor Le, Arnu Pretorius, and Alexandre Laterre. 2024. Jumanji: a Diverse Suite of Scalable Reinforcement Learning Environments in JAX. arXiv:2306.09884 [cs.LG] https://arxiv.org/abs/2306.09884

[2] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/jax-ml/jax

[3] Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. 2020. Leveraging procedural generation to benchmark reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. JMLR.org, Article 191, 9 pages.

[4] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. 2019. Quantifying generalization in reinforcement learning. In *International conference on machine learning*. PMLR, 1282–1289.

[5] Samuel Coward, Michael Beukman, and Jakob Foerster. 2024. JaxUED: A simple and useable UED library in Jax. *arXiv preprint* (2024).

[6] Gabriel Dulac-Arnold, Nir Levine, Daniel J Mankowitz, Jerry Li, Cosmin Paduraru, Sven Gowal, and Todd Hester. 2021. Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. *Machine Learning* 110, 9 (2021), 2419–2468.

[7] Harry Emerson, Liz Bates, Chris Hicks, and Vasilios Mavroudis. 2024. Cyborg++: An enhanced gym for the development of autonomous cyber agents. *arXiv preprint arXiv:2410.16324* (2024).

[8] Dibya Ghosh, Jad Rahme, Aviral Kumar, Amy Zhang, Ryan P Adams, and Sergey Levine. 2021. Why generalization in rl is difficult: Epistemic pomdps and implicit partial observability. *Advances in neural information processing systems* 34 (2021), 25502–25515.

[9] Assaf Hallak, Dotan Di Castro, and Shie Mannor. 2015. Contextual markov decision processes. *arXiv preprint arXiv:1502.02259* (2015).

[10] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. 2022. CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms. *Journal of Machine Learning Research* 23, 274 (2022), 1–18.

[11] Shengyi Huang and Santiago Ontañón. 2022. A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. *The International FLAIRS Conference Proceedings* 35 (May 2022). arXiv:2006.14171 [cs, stat].

[12] Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. 2023. NASimEmu: Network Attack Simulator & Emulator for Training Agents Generalizing to Novel Scenarios. arXiv:2305.17246 [cs].

[13] Minqi Jiang, Michael Dennis, Jack Parker-Holder, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. 2021. Replay-Guided Adversarial Environment Design. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 1884–1897. https://proceedings.neurips.cc/paper_files/paper/2021/file/0e915db6326b6fb6a3c56546980a8c93-Paper.pdf

[14] Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. 2021. Prioritized level replay. In *International Conference on Machine Learning*. PMLR, 4940–4950.

[15] Anssi Kanervisto, Christian Scheller, and Ville Hautamäki. 2020. Action space shaping in deep reinforcement learning. In *2020 IEEE conference on games (CoG)*. IEEE, 479–486.

[16] Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. 2023. A survey of zero-shot generalisation in deep reinforcement learning. *Journal of Artificial Intelligence Research* 76 (2023), 201–264.

[17] Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. 2023. A Survey of Zero-shot Generalisation in Deep Reinforcement Learning. *jair* 76 (Jan. 2023), 201–264. doi:10.1613/jair.1.14174

[18] Robert Tjarko Lange. 2022. *gymnax: A JAX-based Reinforcement Learning Environment Library.* http://github.com/RobertTLange/gymnax

[19] Zegang Li, Qian Zhang, and Guangwen Yang. [n. d.]. EPPTA: Efficient partially observable reinforcement learning agent for penetration testing applications. n/a ([n. d.]), e12818. Issue n/a. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/eng2.12818.

[20] Jingju Liu, Yue Zhang, Shicheng Zhou, Jiahai Yang, Yuliang Lu, and Xiaofeng Zhong. 2025. Autonomous Penetration Testing using Reinforcement Learning: A Review and Perspectives. *Expert Systems with Applications* (2025), 130219.

[21] Chris Lu, Jakub Kuba, Alistair Letcher, Luke Metz, Christian Schroeder de Witt, and Jakob Foerster. 2022. Discovered policy optimisation. *Advances in Neural Information Processing Systems* 35 (2022), 16455–16468.

[22] Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Jackson, Samuel Coward, and Jakob Foerster. 2024. Craftax: A Lightning-Fast Benchmark for Open-Ended Reinforcement Learning. In *International Conference on Machine Learning (ICML)*.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[24] National Institute of Standards and Technology. 2008. *Technical Guide to Information Security Testing and Assessment.* Technical Report SP 800-115. NIST. https://doi.org/10.6028/NIST.SP.800-115

[25] Huynh Phuong Thanh Nguyen, Kento Hasegawa, Kazuhide Fukushima, and Razvan Beuran. 2025. PenGym: Realistic training environment for reinforcement learning pentesting agents. *Computers & Security* 148 (2025), 104140. doi:10.1016/j.cose.2024.104140

[26] OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. 2019. Solving Rubik's Cube with a Robot Hand. *CoRR* abs/1910.07113 (2019). arXiv:1910.07113 http://arxiv.org/abs/1910.07113

[27] Julien Perolat, Bart De Vylder, Daniel Hennes, Eugene Tarassov, Florian Strub, Vincent de Boer, Paul Muller, Jerome T Connor, Neil Burch, Thomas Anthony, et al. 2022. Mastering the game of Stratego with model-free multiagent reinforcement learning. *Science* 378, 6623 (2022), 990–996.

[28] Martin L. Puterman. 1990. Chapter 8 Markov decision processes. In *Stochastic Models*. Handbooks in Operations Research and Management Science, Vol. 2. Elsevier, 331–434.

[29] Carlos Sarraute, Olivier Buffet, and Jörg Hoffmann. 2012. POMDPs Make Better Hackers: Accounting for Uncertainty in Penetration Testing. *Proceedings of the AAAI Conference on Artificial Intelligence* 26, 1 (2012), 1816–1824. Number: 1.

[30] Carlos Sarraute, Olivier Buffet, and Joerg Hoffmann. 2013. Penetration Testing == POMDP Solving? arXiv:1306.4714 [cs].

[31] Jonathon Schwartz and Hanna Kurniawatti. 2019. NASim: Network Attack Simulator. https://networkattacksimulator.readthedocs.io/.

[32] Raphael Simon, Pieter Libin, and Wim Mees. 2025. Learning Robust Penetration-Testing Policies under Partial Observability: A systematic evaluation. arXiv:2509.20008 [cs.LG] https://arxiv.org/abs/2509.20008

[33] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.

[34] Microsoft Defender Research Team. 2021. CyberBattleSim. https://github.com/microsoft/cyberbattlesim. Created by Christian Seifert, Michael Betser, William Blum, James Bono, Kate Farris, Emily Goren, Justin Grana, Kristian Holsheimer, Brandon Marken, Joshua Neil, Nicole Nichols, Jugal Parikh, Haoran Wei..

[35] Franco Terranova, Abdelkader Lahmadi, and Isabelle Chrisment. 2025. Scalable and Generalizable RL Agents for Attack Path Discovery via Continuous Invariant Spaces. In *2025 28th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 18.

[36] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. 2017. 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)*. 23–30. doi:10.1109/IROS.2017.8202133

[37] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *nature* 575, 7782 (2019), 350–354.

[38] Peter R Wurman, Samuel Barrett, Kenta Kawamoto, James MacGlashan, Kaushik Subramanian, Thomas J Walsh, Roberto Capobianco, Alisa Devlic, Franziska Eckert, Florian Fuchs, et al. 2022. Outracing champion Gran Turismo drivers with deep reinforcement learning. *Nature* 602, 7896 (2022), 223–228.

[39] Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. 2018. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893* (2018).

A SPEED COMPARISON DETAILS

Table 3 showcases the full details of our experiment in Section 6.1.

Table 3: Performance Comparison of NASim and NASimJax Implementations

Num	NASim		NASimJax-1M		NASimJax-10M		NASimJax-100M	
Envs	Time (s)	Steps/s	Time (s)	Steps/s	Time (s)	Steps/s	Time (s)	Steps/s
8	3823	2,616	42	23,810	185	54,054	1471	67,981
16	2420	4,132	33	30,303	110	90,909	796	125,628
32	1627	6,146	33	30,303	73	136,986	445	224,719
64	1215	8,230	29	34,483	52	192,308	259	386,100
128	883	11,325	28	35,714	41	243,902	160	625,000
256	748	13,369	28	35,714	35	285,714	102	980,392
512	612	16,340	27	37,037	33	303,030	80	1,250,000
1024	575	17,391	27	37,037	32	312,500	68	1,470,588
2048	561	17,825	28	35,714	31	322,581	63	1,587,302
4096	625	16,000	28	35,714	32	312,500	61	1,639,344

B HYPERPARAMETERS

This section discusses the search ranges and the exact hyperparameters we've selected for the experiments in Section 6. Some more information regarding the tuning strategy: We first tuned both DR methods, and then based on these results, limited the search space for the PLR-based methods. The ranges selected for the PLR-based methods are based on the original works of Jiang et al. [13, 14]. Since the Top-k prioritization has an additional parameter, the k, which isolates the levels with the k-highest scores to select from.

Table 4: Hyperparameter Search Ranges: Masked PPO with DR

Hyperparameter	Considered Values
Learning Rate (LR)	{3e-5, 5e-5, 1e-4, 3e-4, 5e-4}
# Environments	{1024}
# Steps	{16, 32, 64, 128}
Layer Size	{256, 512}
Activation Function	{tanh}
Discount Factor (γ)	{0.975, 0.99, 0.995}
GAE Lambda (λ)	{0.8, 0.9, 0.95}
Clip Epsilon	{0.1, 0.2, 0.3}
Entropy Coef.	{0.005, 0.01, 0.02, 0.05}
Max Gradient Norm	{0.5, 1.0, 1.5}
Value Function Coef.	{0.25, 0.5, 1.0}
Update Epochs	{4}

C ACTIVE HOST DISTRIBUTIONS

Here we present the full set of distributions for the t_d-values established in Table 2. As we've selected to use five values for each number of total hosts in the network, we present the resulting distributions in Figures 6 and 6. These clearly showcase the shift in active hosts given t_d.

D EXTENDED RESULTS

In this Section we highlight some extended results for the experiments conducted in Section 6.4, which showcases the aggregated evaluation performance given different training t_d-values. Figures 8 and 9 display more detailed results, where we plot for every algorithm and t_d-train value the performance against the evaluation t_d-values.

Table 5: Hyperparameter Search Ranges: PPO-2SAS with DR

Hyperparameter	Considered Values
Learning Rate (LR)	{5e-5, 1e-4, 2e-4, 3e-4, 5e-4}
# Environments	{1024}
# Steps	{16, 32, 64, 128}
Layer Size	{256, 512}
Activation Function	{tanh}
Discount Factor (γ)	{0.975, 0.99, 0.995}
GAE Lambda (λ)	{0.8, 0.9, 0.95}
Clip Epsilon	{0.1, 0.2, 0.3}
Host Entropy Coef.	{0.01, 0.02, 0.05, 0.08}
Action Entropy Coef.	{0.005, 0.01, 0.02, 0.05}
Host Embedding Dim	{16, 32, 64}
Max Gradient Norm	{0.5, 1.0, 1.5}
Value Function Coef.	{0.25, 0.5, 1.0}
Update Epochs	{4}

Table 6: Hyperparameter Search Ranges: PLR and PLR$^{\perp}$

Hyperparameter	Considered Values
Learning Rate (LR)	{2e-4, 3e-4, 5e-4}
# Steps	{32, 64, 128}
GAE Lambda (λ)	{0.8, 0.95}
Clip Epsilon	{0.1, 0.2, 0.3}
Entropy Coef.	{0.005, 0.01, 0.05}
Replay Probability (ρ)	{0.5, 0.7, 0.9}
Staleness Coefficient (ω)	{0.1, 0.3, 0.5, 0.7}
Temperature (β)	{0.1, 0.5, 1.0, 1.4, 2.0}
Level Buffer Capacity	{5000, 10000}
Score Function	{MaxMC, PVL}
Prioritization	{Rank, Top-k}
Top-k (k)	{5 , 25, 50, 100}

Table 7: Selected Hyperparameters across all algorithms (Masked and 2SAS variants) for 16 and 26 host networks.

	DR (Masked)		DR (2SAS)		PLR (Masked)		PLR (2SAS)		PLR$^{\perp}$ (Masked)		PLR$^{\perp}$ (2SAS)	
Hyperparameter	**16h**	**26h**	**16h**	**26h**	**16h**	**26h**	**16h**	**26h**	**16h**	**26h**	**16h**	**26h**
Learning Rate	3e-4	3e-4	2e-4	1e-4	3e-4	3e-4	2e-4	1e-4	3e-4	3e-4	2e-4	2e-4
# Environments	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024
# Steps	32	128	64	128	128	64	128	128	128	128	128	128
Layer Size	512	512	256	512	512	512	512	512	512	512	512	512
Activation Function	tanh	tanh	tanh	tanh	tanh	tanh	tanh	tanh	tanh	tanh	tanh	tanh
Discount Factor	0.995	0.995	0.995	0.995	0.995	0.995	0.995	0.995	0.995	0.995	0.995	0.995
GAE Lambda (λ)	0.8	0.8	0.8	0.9	0.8	0.8	0.9	0.95	0.95	0.95	0.95	0.95
Clip Epsilon	0.1	0.2	0.1	0.2	0.3	0.3	0.2	0.2	0.3	0.3	0.3	0.3
Entropy Coef.	0.005	0.005	–	–	0.01	0.005	–	–	0.005	0.005	–	–
Host Ent. Coef.	–	–	0.01	0.01	–	–	0.005	0.01	–	–	0.005	0.01
Action Ent. Coef.	–	–	0.01	0.01	–	–	0.01	0.01	–	–	0.01	0.005
Host Embedding Dim	–	–	64	64	–	–	32	16	–	–	32	16
Max Grad Norm	1.0	0.5	0.5	0.5	0.5	0.5	1.0	1.5	0.5	0.5	1.5	0.5
Value Coef.	1.0	0.5	0.25	1.0	0.5	0.5	1.0	0.25	0.5	0.5	1.0	0.25
Update Epochs	4	4	4	4	4	4	4	4	4	4	4	4
Replay Prob. (ρ)	–	–	–	–	0.5	0.5	0.5	0.5	0.9	0.9	0.9	0.9
Staleness (ω)	–	–	–	–	0.7	0.3	0.3	0.1	0.1	0.5	0.7	0.3
Temperature (β)	–	–	–	–	0.1	2.0	2.0	2.0	2.0	2.0	2.0	2.0
Level Buffer Capacity	–	–	–	–	10000	10000	10000	10000	10000	10000	10000	10000
Min. Fill Ratio	–	–	–	–	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
Score Function	–	–	–	–	PVL	MaxMC	MaxMC	PVL	MaxMC	MaxMC	MaxMC	MaxMC
Prioritization	–	–	–	–	Rank	Rank	Rank	Rank	Rank	Rank	Rank	Rank
Top-k (k)	–	–	–	–	–	–	–	–	–	–	–	–

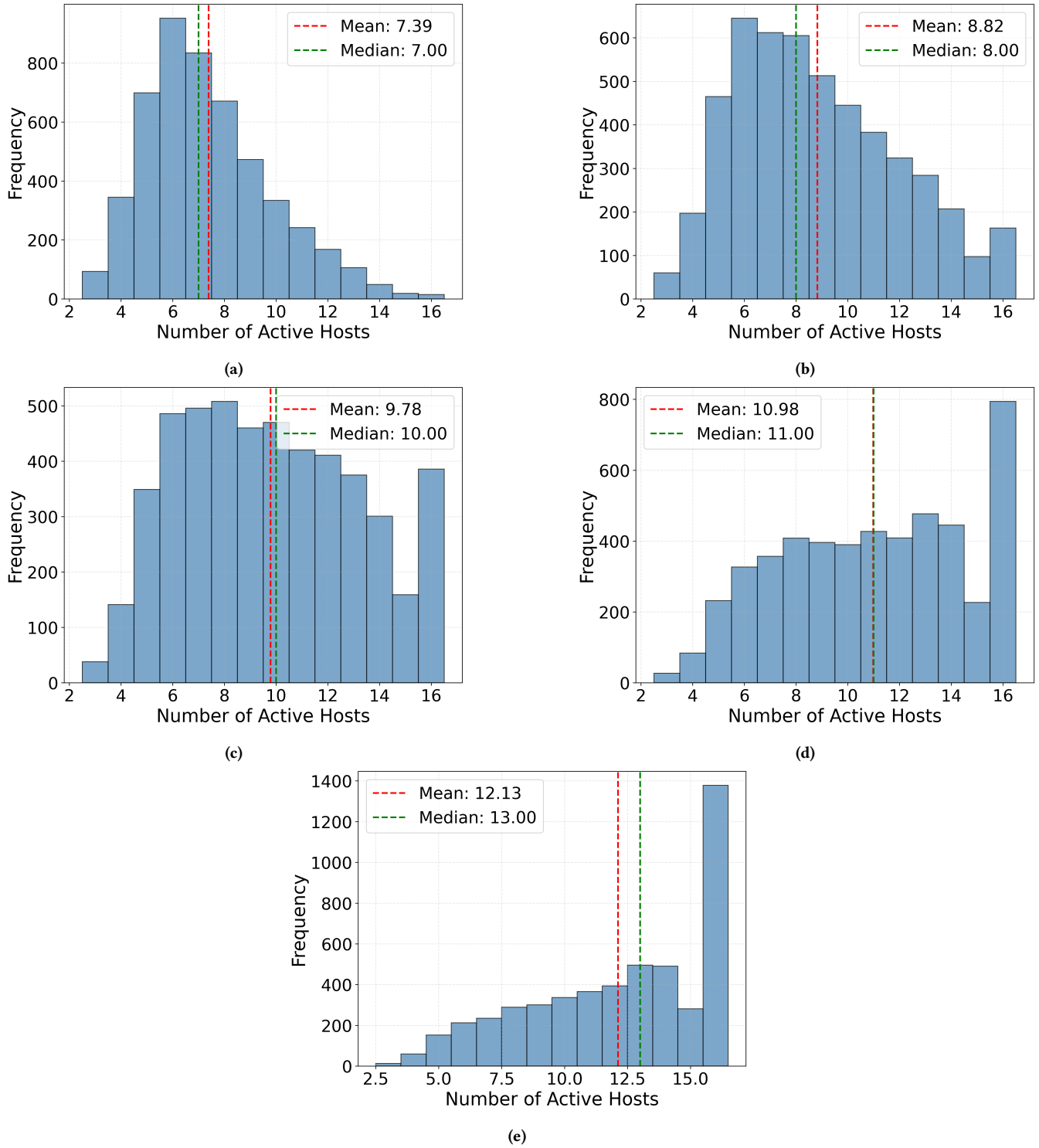


Figure 6: Distributions of active hosts within 16 host networks and how the topology density (t_d parameter influences them.

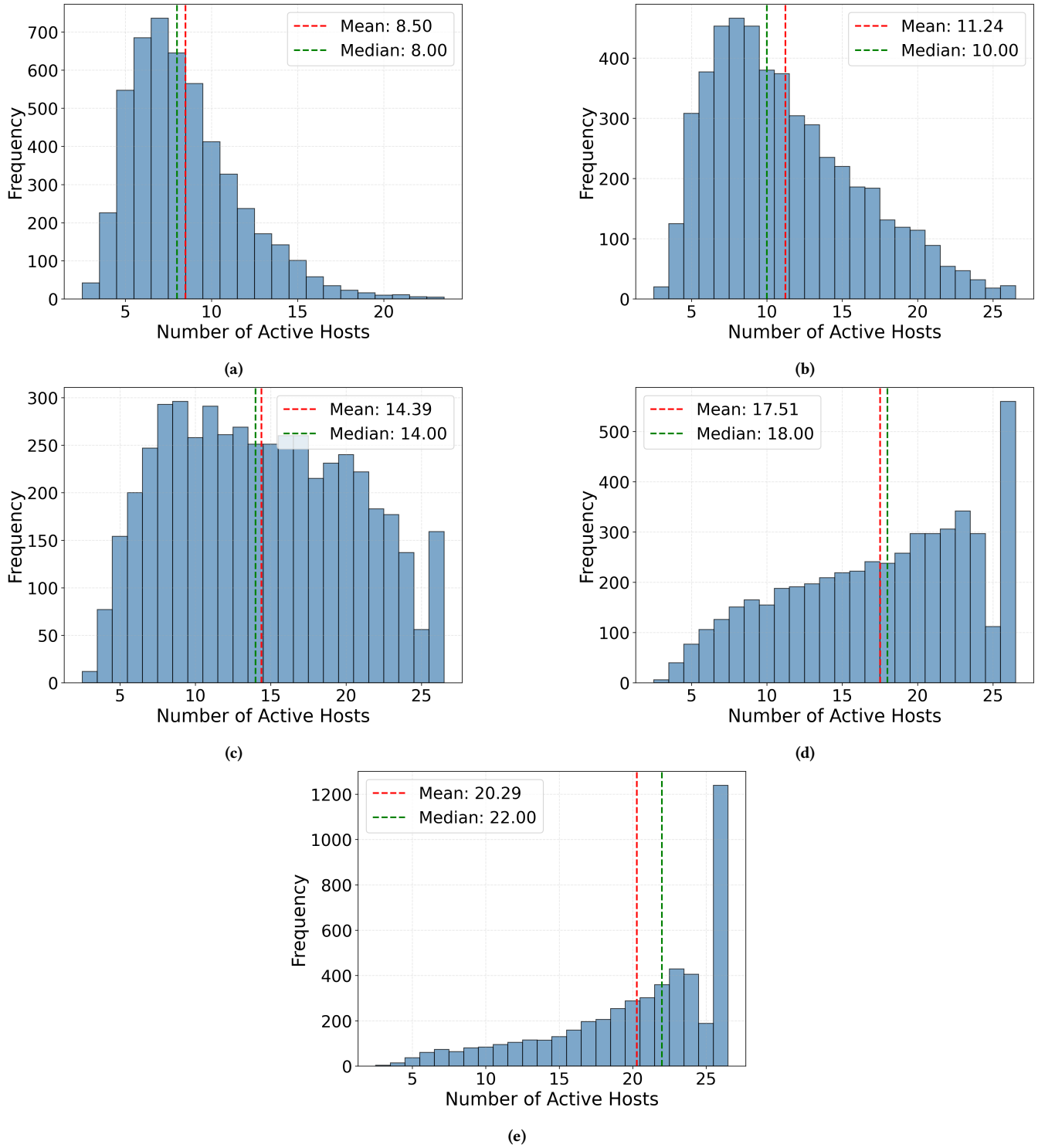


Figure 7: Distributions of active hosts within 26 host networks and how the topology density (t_d parameter influences them.

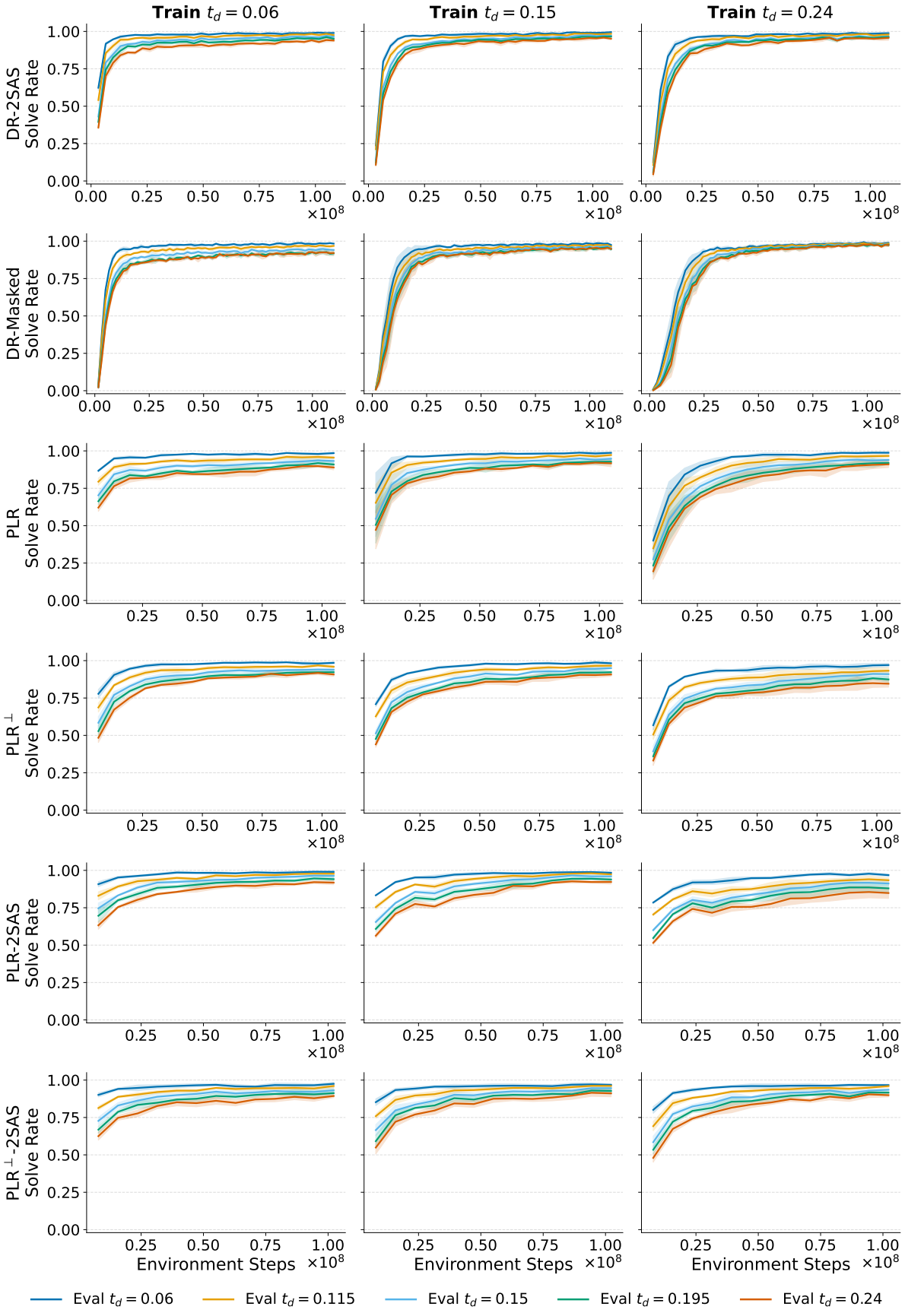


Figure 8: Evaluation curves for 16 host networks. Every column represents the network's t_d the policy was trained on. The plots show the evaluation performance against all other selected t_d for the given network size, as displayed in Table 2.

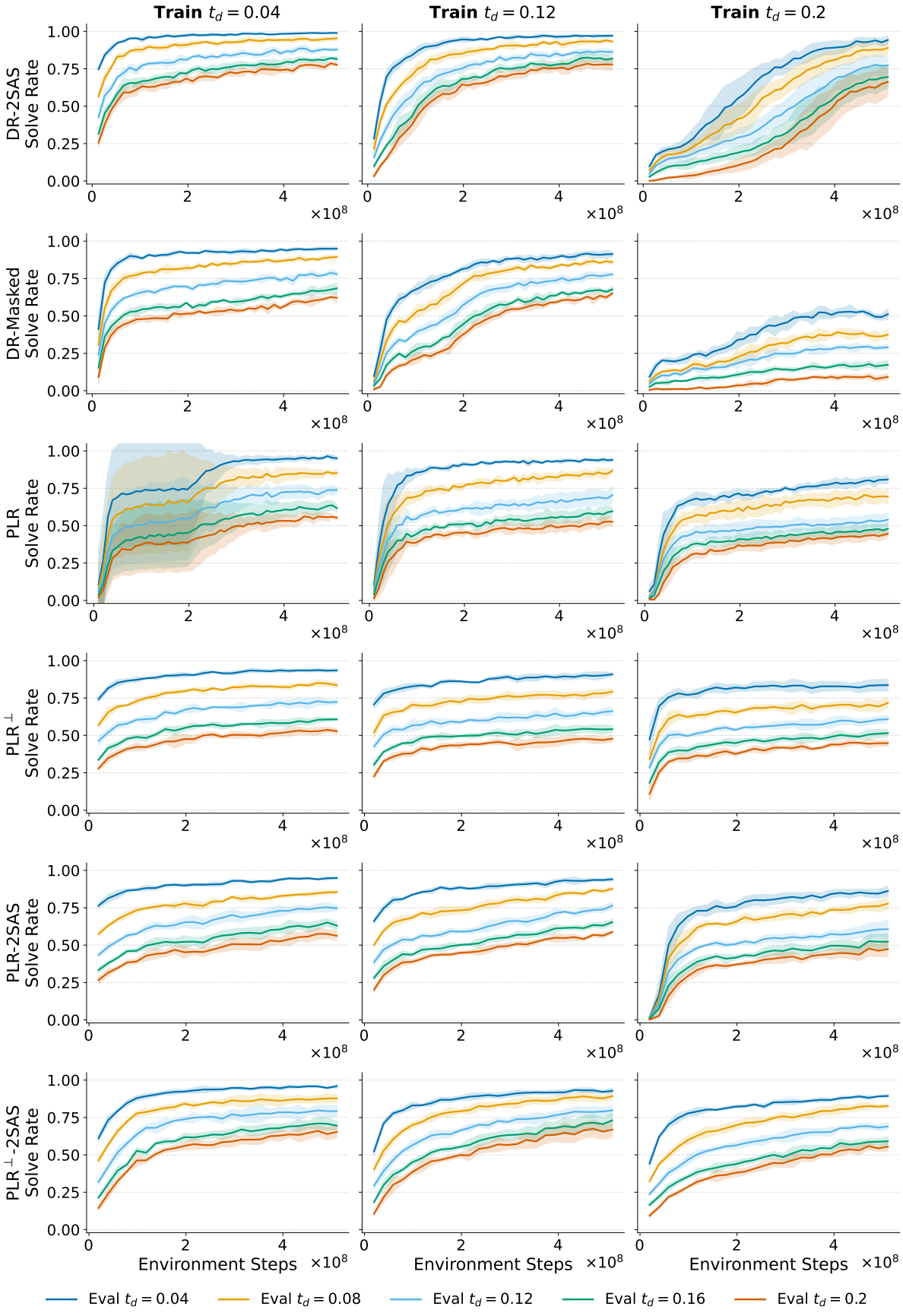


Figure 9: Evaluation curves for 26 host networks. Every column represents the network's t_d the policy was trained on. The plots show the evaluation performance against all other selected t_d for the given network size, as displayed in Table 2.