

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

Raphael Simon^{1,2}
r.simon@cylab.be

José Carrasquel¹

Elli Makdis Antoun¹

Wim Mees¹

Pieter Libin²

¹CISS Department, Royal Military Academy, Belgium

²AI Lab, Vrije Universiteit Brussel, 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 action spaces. Existing RL simulators for this domain are CPU-bound and fixed to narrow scenarios, making it infeasible to train policies that generalize across networks. We present NASimJax, a JAX-native framework that formulates penetration testing as a Contextual POMDP and introduces a network generation pipeline producing structurally diverse, guaranteed-solvable scenarios. The framework reaches up to $80\times$ higher environment throughput than previous simulators, enabling experiments on larger networks and tractable hyperparameter searches. We provide PPO and PQN baselines and conduct the first systematic evaluation of unsupervised environment design for penetration testing. We find that Prioritized Level Replay and ACCEL handle dense training distributions better than Domain Randomization, and that training on sparser topologies yields an implicit curriculum that improves generalization – even to topologies denser than those seen during training. A recurrent PPO variant confirms that these distributional findings are not an artifact of feed-forward policies. The code is available at: <https://github.com/raphsimon/NASimJax>.

1 Introduction

Cybersecurity has become one of the most critical domains in modern society, underpinning health-care, 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 [NIST, 2008]. 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 [Liu et al., 2025], motivated by the success of RL in other complex sequential decision-making domains [Mnih et al., 2015, OpenAI et al., 2019, Vinyals et al., 2019]. The penetration testing problem maps naturally onto a Partially Observable Markov Decision Process (POMDP) [Sarraute et al., 2012, 2013]. An agent cannot directly observe the full network topology or its vulnerabilities, and must actively acquire information before acting.

Beyond partial observability, the domain compounds many of the difficulties common to real-world RL [Dulac-Arnold et al., 2021] and specifically catalogued for pentesting by long decision horizons, sparse and delayed rewards, combinatorial action spaces, and strong dependence on network-specific structure [Liu et al., 2025]. Together these properties make generalization the central deployment

obstacle – policies trained on narrow scenarios fail to transfer, and closing this gap requires simulators that expose agents to structural diversity and decision-relevant complexity rather than merely surface-level realism.

Existing penetration testing simulators [Schwartz and Kurniawatti, 2019, Team., 2021, Emerson et al., 2024] 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 [Patterson et al., 2024, Adkins et al., 2024] – are intractable at the speeds current tools offer.

To address these challenges, we present NASimJax, a JAX-native research framework for studying generalization in automated penetration testing. NASimJax adopts NASim’s framing of the task as partially observable sequential information gathering [Schwartz and Kurniawatti, 2019], but redesigns the state representation, network generation, and context interface to support distributional training and unsupervised environment design (UED). Our contributions are:

- **Distributional formulation and generator.** We formulate penetration testing as a Contextual POMDP [Hallak et al., 2015, Kirk et al., 2023], where each episode is conditioned on a network instance sampled from a configurable distribution. This turns a historically fixed-scenario problem into a distributional one suited to studying zero-shot transfer. A parametric generation pipeline provides this distribution, producing structurally diverse, guaranteed-solvable networks with fine-grained control over topology and vulnerability density.
- **Tractable large-scale experimentation.** An end-to-end JIT-compiled and vectorized design reaches up to $80\times$ the throughput of the most widely adopted existing simulator (Figure 2). This makes previously impractical experiments feasible: large hyperparameter sweeps, billion-step training on 40-host networks, and regret-based UED through existing JAX libraries.
- **Baselines and the first UED study for penetration testing.** We provide PPO and PQN baselines on 16-, 26-, and 40-host networks. Using PPO and PPO-LSTM, we evaluate Domain Randomization (DR) [Tobin et al., 2017], Prioritized Level Replay (PLR) [Jiang et al., 2021a], and ACCEL [Parker-Holder et al., 2022] for zero-shot transfer. Regret-based methods are more robust than DR under dense training distributions, and sparse-topology training produces an implicit curriculum that improves generalization, even to topologies denser than those seen during training.

2 Contextual Reinforcement Learning

A Markov Decision Process (MDP) [Puterman, 1990] is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \rho_0, \gamma \rangle$ with state space $\mathcal{S}$, action space $\mathcal{A}$, transition kernel $\mathcal{P}(s' | s, a)$, reward function $\mathcal{R}$, initial state distribution ρ_0 , and discount factor $\gamma \in [0, 1]$. The goal is to learn a policy $\pi(a | s)$ maximizing the expected return $\mathbb{E}_\pi[G_0]$, with $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$. A Partially Observable MDP (POMDP) [Sutton and Barto, 2018] extends this with an observation space Ω and observation function $\mathcal{O}(o | s', a)$: the agent perceives only o_t and must act on the observation history, or equivalently on a belief state $b_t \in \Delta(\mathcal{S})$.

Following Hallak et al. [2015] and Kirk et al. [2023], a *Contextual POMDP* augments this with a context $c \in \mathcal{C}$, sampled once per episode from $p(c)$, that parameterizes the dynamics, rewards, observations, and initial state: $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}_c, \mathcal{R}_c, \Omega, \mathcal{O}_c, \rho_{0,c}, \gamma, \mathcal{C}, p(c) \rangle$. The context is fixed within an episode but varies across episodes, inducing a task distribution. The objective is $\mathbb{E}_{c \sim p(c)} [\mathbb{E}_\pi[G_0 | c]]$.

3 Related Work

JAX and Hardware-Accelerated RL. JAX [Bradbury et al., 2018] enables JIT-compiled, functionally pure programs to run end-to-end on accelerators, and its `vmap` transformation allows environment `step` and `reset` functions to be vectorized across thousands of parallel instances. Together these eliminate the CPU–GPU bottleneck that limits conventional Python-based RL pipelines [Rutherford et al., 2024]. A growing ecosystem of JAX-native RL environments exploits this, including Gymnax [Lange, 2022], Craftax [Matthews et al., 2024], and Jumanji [Bonnet et al., 2024].

Penetration Testing Simulators. Among RL-oriented simulators, NASim [Schwartz and Kurniawatti, 2019] is the most widely adopted and underpins much subsequent work, including the emulation-backed extensions NASimEmu [Janisch et al., 2023] and PenGym [Nguyen et al., 2025], which bridge simulated training with virtualized targets. StochNASim [Simon et al., 2025] adds stochastic resets and variable network sizes on top of NASim. Beyond the NASim lineage, CyberBattleSim [Team., 2021] models lateral movement, CybORG [Standen et al., 2021] couples offensive and defensive agents within a shared network, and C-CyberBattleSim [Terranova et al., 2025] grounds scenarios in real-world vulnerability databases and scanner output. All are CPU-bound Python implementations, fundamentally limiting the scale at which distributional training can be conducted.

Generalization in Automated Penetration Testing. Generalization is a recognized open problem for RL-based penetration testing [Liu et al., 2025], and prior work has approached it along two axes. *Architecturally*, permutation- and size-invariant networks have been proposed to make policies transferable across networks of varying size and host ordering [Janisch et al., 2023, Terranova et al., 2025]. *Distributionally*, agents have been exposed to diverse training environments through multi-scenario training [Janisch et al., 2023], stochastic resets with varying network sizes [Simon et al., 2025], and Domain Randomization [Zhou et al., 2024]; the recent survey of Liu et al. [Liu et al., 2025] identifies DR as the only distributional-based technique applied in this domain to date. To our knowledge, no prior work has applied regret-based Unsupervised Environment Design methods such as PLR or ACCEL, leaving open whether explicit regret-based curricula can outperform DR in this setting. We provide the first empirical evidence on this question.

4 NASimJax

NASimJax preserves NASim’s core task abstraction – penetration testing as a sequential information-gathering problem under partial observability – but the state representation, network generation pipeline, and context framework are redesigned from the ground up to support CMDP-style training, JIT compilation, and structural diversity. The environment is designed as a configurable research abstraction rather than a fixed benchmark scenario: its primary purpose is to support controlled variation over network instances so that policy robustness and zero-shot generalization can be studied systematically. Through a stochastic 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. The environment adheres to the Gymnax API [Lange, 2022], enabling integration with JAX-based RL algorithms and direct comparison of learning methods within a standardized environment.

Modeling scope. NASimJax abstracts penetration testing at the level of attack-path discovery in segmented enterprise networks. The agent must chain reconnaissance, exploitation, and privilege escalation across hosts whose operating systems, services, and processes are initially hidden. Vulnerabilities are modeled as $\text{OS} \times \text{service}$ and $\text{OS} \times \text{process}$ pairs that admit specific exploits; hosts may also be correctly configured and run no exploitable software at all, capturing the uneven security posture of real deployments. Intentionally abstracted away are CVE-level semantics, exploit timing, detection and defender response, persistence mechanisms, and data exfiltration. This scope preserves the combinatorial and information-theoretic structure that makes penetration testing hard for RL (i.e., long horizons, large action spaces, partial observability, and strong dependence on network-specific structure) while remaining tractable enough for systematic study. The framework is suited to investigating attack-path discovery, action prioritization under uncertainty, and policy generalization across network topologies; it is not intended for questions about low-level exploit mechanics or active defender dynamics.

Task. An episode proceeds as follows. An agent interacts with a network by scanning hosts, enumerating services, exploiting vulnerable software to obtain an initial foothold, and escalating privileges through vulnerable processes. Information about topology, running services, and vulnerabilities must be actively acquired through interaction. Progress 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 (scenario), represented by a concrete network instance comprising hosts partitioned into subnets, connectivity rules defining inter-subnet communication,

and host-specific configurations (running services, processes, and associated exploitability). While the policy is shared across episodes, the underlying context varies, providing a principled mechanism to expose the agent to diverse network structures during training. This naturally supports viewing the environment as a Contextual POMDP, where generalization is achieved by learning policies that perform well across a distribution of network contexts rather than memorizing individual attack paths.

4.1 Task specification

We briefly describe the state, action and observation spaces, and reward function. The core structure follows NASim [Schwartz and Kurniawatti, 2019] and subsequent work [Janisch et al., 2023, Simon et al., 2025]; the design contributions of NASimJax – the state encoding, generation pipeline, context framework, and reward scaling – are addressed in Sections 4.2–4.3 and 5. Full semantics are provided in Appendix B.1.

State. Network state comprises an adjacency matrix encoding inter-subnet traffic and firewall rules, together with per-host properties (subnet address, reachability, discovery status, access level, OS, vulnerable services, and vulnerable processes). Host properties are batched into a single $\{0, 1\}^{N_h \times F}$ tensor using one-hot and boolean encodings (cf. Figure 1). This encoding reduces transition logic to bitwise operations and keeps every host property in a single byte, reducing the overall memory footprint. The full state is used internally for transition dynamics but is not exposed to the agent, preserving partial observability.

Actions. The action space contains six discrete action types grouped into reconnaissance (OS, process, service, and subnet scans) and exploitation (exploit, privilege escalation). Each action targets a specific host, so the total number of environment actions is

$$|\mathcal{A}| = N_h \times (|\mathcal{A}_{\text{scan}}| + |\text{OS}| \times (|V_{\text{svc}}| + |V_{\text{proc}}|)), \quad (1)$$

where N_h is the number of hosts, $|\mathcal{A}_{\text{scan}}|$ the number of scan actions, $|\text{OS}|$ the number of operating systems, and $|V_{\text{svc}}|, |V_{\text{proc}}|$ the numbers of vulnerable services and processes.

Observations. Observations are transient and only contain the outcome of the executed action, concatenated with an encoding of the action type and four mutually exclusive status flags (success, connection error, permission error, undefined error).

Reward. Each action incurs a cost, and the reward r_t is defined as $-\text{cost}(a_t)$ plus any additional bonus earned. For a successful subnet scan, a discovery bonus V_d is added for each newly discovered host n_{dh} , yielding $r_t = -\text{cost}(a_t) + (V_d \times n_{dh})$. For a successful privilege escalation, a host value bonus V_h is added, giving $r_t = -\text{cost}(a_t) + V_h$. In all other cases, the reward is simply $r_t = -\text{cost}(a_t)$.

4.2 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 1 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. Vulnerable 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 . This mechanism allows the generator to model varying levels of organizational security maturity, including misconfigurations or users not following security best practices.

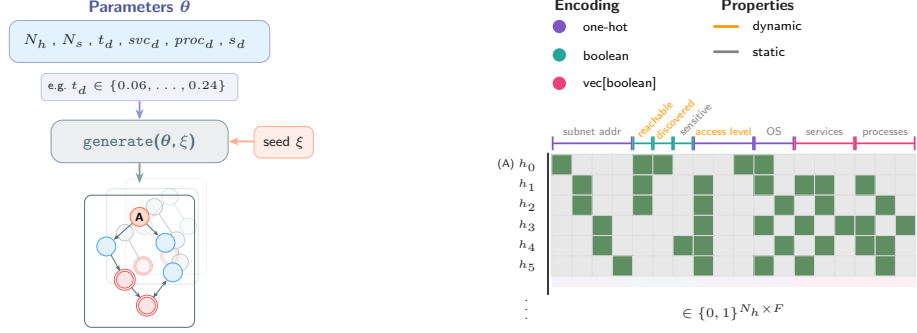


Figure 1: **Left:** Number of hosts (N_h), and subnets (N_s), topology (t_d), service (svc_d), process ($proc_d$) and sensitive host density (s_d) are all parameters that influence the generated networks. The blue nodes represent normal hosts, the red node labelled A represents the attacker’s position, and the double circled red nodes represent sensitive hosts. **Right:** Illustration of the batched state representation. F is the dimension of the host features.

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. They also ensure that solve rates are comparable across t_d and N_h .

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 topology matrix is intentionally not required to be symmetric, so subnets may be unreachable from the attacker’s starting position and effectively become inactive for an episode. This induces variation in active host counts and effective horizon length even at fixed N_h , providing a natural difficulty curriculum 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.

4.3 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 [Cobbe et al., 2019]. 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., 2023]. 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. [2023].

5 Methodology

We use PPO as our primary algorithm, the de facto choice for RL-based penetration testing [Li et al., 2023, Terranova et al., 2024, Simon et al., 2025]. We additionally evaluate PQN [Gallici et al., 2025], a parallelized Q-learning variant whose GPU-native design aligns with our throughput-oriented framework. Both build on pure JAX implementations [Lu et al., 2022, Gallici et al., 2025]. To test whether our findings depend on the feed-forward policy class, we additionally evaluate a recurrent variant, PPO-LSTM, in the zero-shot transfer experiment of Section 6.4.

Training setup. To cope with partial observability, we follow Simon et al. [2025] and append the accumulated history of past observations to the current one – mirroring how human testers track prior scan results. To avoid wasting budget on invalid actions, we apply action masking [Huang and Ontañón, 2022]: we mask actions targeting undiscovered and unreachable hosts, and exploits/privilege escalations incompatible with a host’s OS–service or OS–process combination. We deliberately do *not* mask actions invalid due to missing privilege levels, as this signal is recoverable from the observations (Section 4.1). The recurrent variant keeps this augmentation and adds an LSTM layer on top of it, so the two mechanisms for handling partial observability are combined rather than substituted.

Reward scaling. Since networks are procedurally generated, the number of sensitive hosts (N_{sh}) varies across contexts, causing high variance in episodic returns that can destabilize value estimation [Van Hasselt et al., 2016, Cobbe et al., 2020]. Ensuring guaranteed solvability of scenarios (Section 4.2) allows us to scale rewards by the theoretical maximum return: $\hat{r}_t = r_t / (N_{sh} \cdot V_h)$, where V_h is the sensitive-host reward minus the action cost to get there. This makes the learning signal reflect structural difficulty (vulnerability sparsity, sensitive-host count) rather than network size, yielding a comparable signal across the context distribution. This scaling is also a prerequisite for PLR: without it, regret estimates would be biased toward larger networks that accumulate higher raw returns, conflating size with learning potential.

Unsupervised Environment Design Methods. Prioritized Level Replay (PLR) [Jiang et al., 2021a] 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 with respect to its current policy. PLR performs gradient updates on the random levels, whereas robust PLR (denoted as $\text{PLR}^\perp$) only updates the agent on the levels sampled from the buffer [Jiang et al., 2021b]. ACCEL Parker-Holder et al. [2022] is an evolutionary method based on PLR. It incrementally alters the complexity of levels in the buffer through small mutations. We provide five mutations covering services, processes, OS, sensitive-host placement, and topology edges; details and post-mutation solvability checks are in Appendix B.2. We compare the different UED methods against Domain Randomization [Tobin et al., 2017], where at each end of an episode, a new network is generated from the parameterized distribution (cf. Section 4.2). We use the implementation provided by the JaxUED library [Coward et al., 2024].

6 Experiments

Our experiments investigate three aspects of NASimJax: (1) its throughput advantage and memory footprint, (2) how standard RL algorithms fare on this benchmark, and (3) how training distribution affects zero-shot generalization. Appendix D provides sweep ranges, selection protocol, and final hyperparameters. Appendix G lists the computational resources.

6.1 Throughput and Memory Scaling

We compare throughput between NASim’s Python implementation and NASimJax by training PPO policies with increasing numbers of parallel environment workers (8 to 4096, doubling). StochNASim and NASimEmu are CPU-bound Python implementations inheriting NASim’s throughput profile; we therefore compare against NASim directly as the shared substrate. We train actual policies rather than comparing random rollouts, to reflect end-to-end usage. For NASim and NASimJax, we use the PPO implementations from `cleanRL` [Huang et al., 2022] and `PureJaxRL` [Lu et al., 2022], respectively, with matched minibatch and rollout sizes. Figure 2 (left) shows that NASimJax reaches 4M steps/s

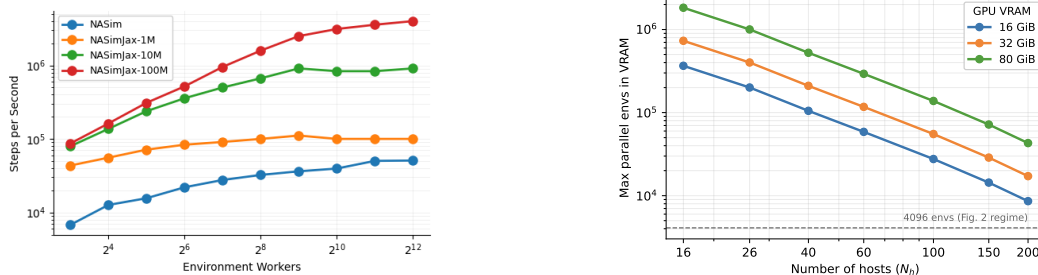


Figure 2: **Left:** Training speed comparison between NASim with 10M steps of training budget against NASimJax with 1M, 10M and 100M steps. Number of environment workers are doubled every time. Results show the impact of JAX’s JIT-compilation on total runtime. Full results are in Appendix C. **Right:** NASimJax’s memory footprint, given an increased number of hosts.

at 4096 workers with a 100M-step budget – an $80\times$ speed-up over NASim. The gap widens with training budget because JIT compilation overhead amortizes over longer runs, making NASimJax especially suited to large-scale experiments. CyberBattleSim does not support parallel workers out of the box and was excluded. Figure 2 (right) reports the memory frontier: at 4096 parallel environments (the throughput regime above), networks of up to 200 hosts fit comfortably on 16GB of VRAM, and larger networks remain feasible at reduced parallelism. Together, the two panels show what scale of experiment is achievable with NASimJax under typical GPU budgets.

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 (t_d), total host count (N_h), service (svc_d) and process ($proc_d$) density, sensitive host density (s_d) (cf. Section 4.2). These allow researchers to instantiate environments aligned with their specific research interests. To ground the experimental investigation that follows, we describe three reference configurations of increasing complexity that we use throughout this work in Table 1. We emphasize that these are illustrative rather than normative. These scales exceed those typically used in prior simulation-based work (e.g. up to 8 hosts in StochNASim and 16 in PenGym and NASimEmu) though these environments differ in scope and formulation.

Table 1: Network configuration and topology densities per network size. All configurations use 3 processes, 3 services, 2 operating systems, and $svc_d = proc_d = 0.7$. Training densities are **bold**.

Hosts	N_s	s_d	Steps	$ \mathcal{A} $	Topology Density t_d				
16	7	0.20	300	256	0.06	0.115	0.15	0.195	0.24
26	10	0.15	300	416	0.04	0.08	0.12	0.16	0.20
40	16	0.15	500	640	0.03	0.06	0.09	0.12	0.15

Across all configurations, action costs are fixed at 1 for scanning and 3 for exploit and privilege escalation actions. This 3:1 ratio aims to discourage brute-force policies that skip information gathering. The sensitive host reward is $V_h = 50$ and we provide a host discovery reward of $V_d = 1$. Compromising a sensitive host strongly dominates the action cost signal, while the small discovery reward provides a mild incentive for exploration. Since rewards are scaled by $N_s \cdot V_h$ during training (cf. Section 5), the absolute magnitude of V_h does not affect the normalized learning signal – only the relative ratios between these values matter.

Parameters such as t_d and s_d were carefully chosen to enable the generation of networks with a varied number of *active* hosts, as illustrated in Figure 3 (see Appendix E for complete distributions). The t_d ranges are scaled inversely with network size because larger networks require lower connectivity probabilities to produce meaningful variation in active host counts. At higher densities, most subnets become reachable and the distribution concentrates near the maximum.

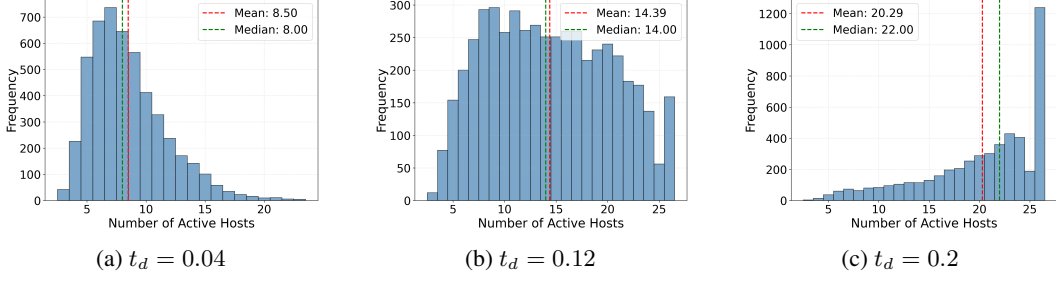


Figure 3: Visualization of t_d 's effect on active host counts within generated networks of 26 hosts.

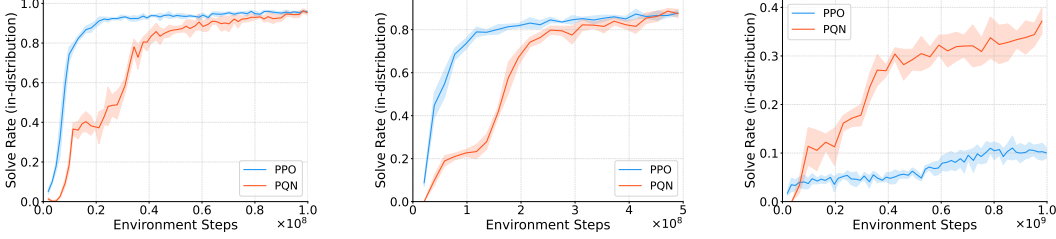


Figure 4: Comparison PPO and PQN on networks with **(Left)** 16 Hosts, $t_d = 0.15$; **(Middle)** 26 Hosts, $t_d = 0.12$; and **(Right)** 40 Hosts, $t_d = 0.09$. Both algorithms use DR and evaluate in-distribution performance on 50 networks. Results are reported over 5 seeds with 95% CI.

6.3 Baselines

To establish baseline performances, we train and evaluate PPO and PQN on the configurations established in Section 6.2 using different training budgets: 100M steps for 16-host networks, 500M for 26 hosts, and 1B for 40 hosts. Each algorithm is trained on five seeds to account for variance. We periodically evaluate the policies in-distribution over 50 networks (using the same t_d), as our aim here is to provide baseline performances, not measure out-of-distribution generalization. While the final solve rate is similar for 16 and 26 hosts, the results in Figure 4 show a large difference for 40 hosts. PQN achieves an average solve rate of $\sim 37\%$, compared to PPO's $\sim 15\%$. Although this difference is noticeable, we'd also like to highlight that, given the low solve rate, neither of them constitutes a meaningful policy. This showcases that further efforts are still required in this domain.

6.4 Zero-Shot Policy Transfer

The CMDP formulation allows us to vary the training distribution over active hosts via the t_d parameter. This experiment tests how the choice of training distribution affects generalisation to unseen active host distributions at evaluation time. We use PPO and PPO-LSTM for this experiment as PLR and ACCEL are designed around on-policy regret estimation. For every network described in Table 1, we select five topology densities per network size to create varying distributions (cf. Figure 3). The UED algorithms used are DR, PLR, $\text{PLR}^\perp$ and ACCEL (cf. Section 5).

For each algorithm and network size, we train three policies, each on one of the three training densities highlighted in Table 1. The training budget remains the same as in our previous experiment. Each policy is then evaluated on all five densities, using one set of 50 evaluation environments per t_d . As a result, each policy is evaluated on four densities it did not train on, two of which (the non-bold entries in Table 1) are held out entirely and never used for training at that network size.

Figure 5 shows the aggregated results; For 16-host networks, neither the algorithm nor the training distribution makes a meaningful difference at that scale. For 26- and 40-host networks, training on the lowest t_d consistently yields the best evaluation performance across all densities, including those denser than the training distribution, and this holds for every algorithm under both architectures. Training at high t_d degrades all methods, but most severely DR, which averages only $\sim 29\%$ (PPO) and $\sim 19\%$ (PPO-LSTM) across evaluation densities at $t_d = 0.2$ on 26-host networks, where PLR, $\text{PLR}^\perp$ and ACCEL all stay above 90% under recurrence. On 40-host networks, ACCEL leads most

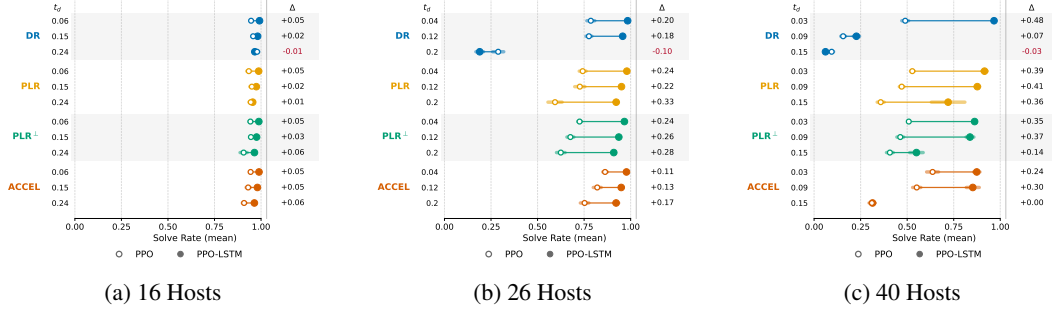


Figure 5: ZSPT across topology densities t_d for 16-, 26- and 40-host networks, PPO (open markers) vs. PPO-LSTM (filled). Each algorithm is trained at three values of t_d (low, mid, high) and evaluated on all t_d . Markers give the solve rate averaged over the evaluation densities and over five seeds. The connector length is the recurrent gain, repeated numerically as Δ (red where recurrence does not help); shaded bands span ± 1 s.d. across seeds.

settings under PPO but falls behind PLR and $\text{PLR}^\perp$ at $t_d = 0.15$, suggesting that mutation-based curriculum struggles when most encountered networks are already difficult; under PPO-LSTM that deficit widens from ~ 5 to ~ 40 points ($\sim 31\%$ vs. $\sim 72\%$ and $\sim 55\%$).

Adding an LSTM block raises solve rates at almost every setting, without changing the picture this experiment paints. The exception is the densest training density, where recurrence brings no gain (40 hosts, $t_d = 0.15$) or a small loss (26 hosts, $t_d = 0.2$), confirming that DR’s collapse under dense training is a property of the training distribution rather than a limitation of the policy class. Among the stronger methods the ordering moves by a few points: PLR edges ahead of ACCEL on 40-host networks, and the top three sit within roughly a point of each other at the sparser 26-host densities. The two effects the experiment surfaces remain unchanged: sparse-topology training generalises best at every size and under both architectures, and regret-based replay is what keeps a method robust when training is dense.

7 Discussion & Limitations

Algorithmic gap at scale. PQN’s advantage over PPO emerges only at 40 hosts (Section 6.3). This crossover is consistent with PQN’s distinguishing property: LayerNorm and ℓ_2 -regularised TD admits formal stability guarantees in the long-horizon, off-policy regime that PPO’s policy-gradient critic does not [Gallici et al., 2025], and its λ -returns plausibly aid credit assignment as effective horizons and action spaces grow with network size. Whether stability, multi-step credit assignment, or representational capacity dominates remains an empirical question that NASimJax is well-suited to investigate.

Training context and zero-shot transfer. Curriculum formation is the dominant factor in our results. Sparse-topology training produces an implicit curriculum. The agents meet shorter-horizon tasks early and build competence before facing complex attack paths. Regret-based replay produces an explicit version of the same effect, which is why PLR-family methods stay competitive at high t_d where DR collapses. ACCEL’s sensitivity to initial level difficulty is an expected consequence of its evolutionary design [Parker-Holder et al., 2022]: small mutations extend the curriculum outward from achievable levels, but cannot manufacture them when the generator predominantly produces hard instances, as at $t_d = 0.15$ on 40 hosts. Practically, where the deployment distribution is dense, training on cheaper sparse networks is preferable to training on-distribution.

Evaluative Scope & Limitations. Beyond the modelling abstractions outlined in Section 4, several empirical and interpretive limitations constrain our claims. First, algorithmic coverage spans PPO (feed-forward and recurrent) and PQN; transformer-based, permutation-invariant and off-policy actor-critic designs remain unexplored. Repeating the zero-shot transfer experiment with a recurrent policy (Section 6.4) raises the absolute solve rates throughout reproduces and the mechanisms we report: DR’s collapse when training is dense, and ACCEL’s difficulty when the training distribution is already hard. The exact ordering among the strongest methods shifts by a few points in places, but overall findings remain intact. Second, our zero-shot transfer study varies only topology density t_d ,

holding svc_d , $proc_d$, s_d , and N_h fixed within each configuration, so generalization claims apply along the topology axis rather than to arbitrary distributional shift. Third, we assess policies via solve rate on fixed held-out network sets per density, which captures task completion but not action efficiency, stealth, or attack-path quality. Finally, NASimJax supports evaluative claims about RL behaviour on procedurally generated networks under our generator: improvements should be interpreted as evidence about scalable distributional training in this abstraction. Transfer to real enterprise networks, to emulation backends such as NASimEmu or PenGym, and to defender-aware settings is left as a separate question that this framework does not by itself adjudicate.

8 Conclusion

We introduced NASimJax, a JAX-native Contextual POMDP framework and configurable benchmark for penetration testing that resolves the throughput bottleneck of existing simulators (up to $80\times$) and exposes a controllable distribution over network instances along topology, size, and vulnerability density. Together with the limitations articulated above, our baseline and UED findings position NASimJax as a configurable, guaranteed-solvable platform for advancing RL in this domain, with capacity to support future work on action-space scaling, topology effects, architectural design, and curriculum methods at scales that prior simulators could not reach.

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 Hicham Azmani, Bram Silue and all other lab members that have provided feedback throughout the creation of this work.

References

- NIST. Technical guide to information security testing and assessment. Technical Report SP 800-115, NIST, September 2008. URL <https://doi.org/10.6028/NIST.SP.800-115>.
- Jingju Liu, Yue Zhang, Shicheng Zhou, Jiahai Yang, Yuliang Lu, and Xiaofeng Zhong. Autonomous penetration testing using reinforcement learning: A review and perspectives. *Expert Systems with Applications*, page 130219, 2025.
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.
- OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. 2019. URL <https://arxiv.org/abs/1912.06680>.
- Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *nature*, 575(7782):350–354, 2019.
- Carlos Sarraute, Olivier Buffet, and Jörg Hoffmann. POMDPs Make Better Hackers: Accounting for Uncertainty in Penetration Testing. *Proceedings of the AAAI Conference on Artificial Intelligence*, 26(1):1816–1824, 2012. ISSN 2374-3468. Number: 1.
- Carlos Sarraute, Olivier Buffet, and Joerg Hoffmann. Penetration Testing == POMDP Solving?, June 2013. arXiv:1306.4714 [cs].

- Gabriel Dulac-Arnold, Nir Levine, Daniel J Mankowitz, Jerry Li, Cosmin Paduraru, Sven Gowal, and Todd Hester. Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. *Machine Learning*, 110(9):2419–2468, 2021.
- Jonathon Schwartz and Hanna Kurniawatti. Nasim: Network attack simulator. <https://networkattacksimulator.readthedocs.io/>, 2019.
- Microsoft Defender Research Team. Cyberbattlesim. <https://github.com/microsoft/cyberbattlesim>, 2021. 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.
- Harry Emerson, Liz Bates, Chris Hicks, and Vasilios Mavroudis. Cyborg++: An enhanced gym for the development of autonomous cyber agents. *arXiv preprint arXiv:2410.16324*, 2024.
- Andrew Patterson, Samuel Neumann, Martha White, and Adam White. Empirical design in reinforcement learning. *Journal of Machine Learning Research*, 25(318):1–63, 2024.
- Jacob Adkins, Michael Bowling, and Adam White. A method for evaluating hyperparameter sensitivity in reinforcement learning. *Advances in Neural Information Processing Systems*, 37: 124820–124842, 2024.
- Assaf Hallak, Dotan Di Castro, and Shie Mannor. Contextual markov decision processes. *arXiv preprint arXiv:1502.02259*, 2015.
- Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. A Survey of Zero-shot Generalisation in Deep Reinforcement Learning. *jair*, 76:201–264, January 2023. ISSN 1076-9757. doi: 10.1613/jair.1.14174. URL <http://jair.org/index.php/jair/article/view/14174>.
- Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. 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)*, pages 23–30, 2017. doi: 10.1109/IROS.2017.8202133.
- Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay. In *International Conference on Machine Learning*, pages 4940–4950. PMLR, 2021a.
- Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving curricula with regret-based environment design. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 17473–17498. PMLR, 17–23 Jul 2022. URL <https://proceedings.mlr.press/v162/parker-holder22a.html>.
- Martin L. Puterman. Chapter 8 markov decision processes. In *Stochastic Models*, volume 2 of *Handbooks in Operations Research and Management Science*, pages 331–434. Elsevier, 1990.
- Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0262039249.
- James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Nectra, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL <http://github.com/google/jax>.
- Alexander Rutherford, Benjamin Ellis, Matteo Gallici, Jonathan Cook, Andrei Lupu, Garð ar Ingvarsson, Timon Willi, Ravi Hammond, Akbir Khan, Christian Schroeder de Witt, Alexandra Souly, Saptarashmi Bandyopadhyay, Mikayel Samvelyan, Minqi Jiang, Robert Lange, Shimon Whiteson, Bruno Lacerda, Nick Hawes, Tim Rocktäschel, Chris Lu, and Jakob Foerster. Jaxmarl: Multi-agent rl environments and algorithms in jax. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 50925–50951. Curran Associates, Inc., 2024. doi: 10.52202/079017-1612. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5aee125f052c90e326dcf6f380df94f6-Paper-Datasets_and_Benchmarks_Track.pdf.

- Robert Tjarko Lange. gymmax: A JAX-based reinforcement learning environment library, 2022. URL <http://github.com/RobertTLange/gymmax>.
- Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Jackson, Samuel Coward, and Jakob Foerster. Craftax: A lightning-fast benchmark for open-ended reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2024.
- 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, Arnau Pretorius, and Alexandre Laterre. Jumanji: a diverse suite of scalable reinforcement learning environments in jax, 2024. URL <https://arxiv.org/abs/2306.09884>.
- Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. NASimEmu: Network Attack Simulator & Emulator for Training Agents Generalizing to Novel Scenarios, August 2023. arXiv:2305.17246 [cs].
- Huynh Phuong Thanh Nguyen, Kento Hasegawa, Kazuhide Fukushima, and Razvan Beuran. Pengym: Realistic training environment for reinforcement learning pentesting agents. *Computers & Security*, 148:104140, 2025. ISSN 0167-4048. doi: <https://doi.org/10.1016/j.cose.2024.104140>. URL <https://www.sciencedirect.com/science/article/pii/S0167404824004450>.
- Raphael Simon, Pieter Libin, and Wim Mees. Learning robust penetration-testing policies under partial observability: A systematic evaluation, 2025. URL <https://arxiv.org/abs/2509.20008>.
- Maxwell Standen, Martin Lucas, David Bowman, Toby J. Richer, Junae Kim, and Damian Marriott. CyBORG: A Gym for the Development of Autonomous Cyber Agents, August 2021. arXiv:2108.09118 [cs].
- Franco Terranova, Abdelkader Lahmadi, and Isabelle Chrisment. 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)*, page 18, 2025.
- Shicheng Zhou, Jingju Liu, Yuliang Lu, Jiahai Yang, Yue Zhang, and Jie Chen. Towards generalizable autonomous penetration testing via domain randomization and meta-reinforcement learning. *ArXiv abs/2412.04078*, 2024.
- Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. In *International conference on machine learning*, pages 1282–1289. PMLR, 2019.
- Zegang Li, Qian Zhang, and Guangwen Yang. EPPTA: Efficient partially observable reinforcement learning agent for penetration testing applications. n/a:e12818, 2023. ISSN 2577-8196. _eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/eng2.12818>.
- Franco Terranova, Abdelkader Lahmadi, and Isabelle Chrisment. 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*, pages 1–16, 2024.
- Matteo Gallici, Mattie Fellows, Benjamin Ellis, Bartomeu Pou, Ivan Masmitja, Jakob Nicolaus Foerster, and Mario Martin. Simplifying deep temporal difference learning. *The International Conference on Learning Representations (ICLR)*, 2025. URL <https://arxiv.org/abs/2407.04811>.
- Chris Lu, Jakub Kuba, Alistair Letcher, Luke Metz, Christian Schroeder de Witt, and Jakob Foerster. Discovered policy optimisation. *Advances in Neural Information Processing Systems*, 35:16455–16468, 2022.
- Shengyi Huang and Santiago Ontañón. A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. *The International FLAIRS Conference Proceedings*, 35, May 2022. ISSN 2334-0762. arXiv:2006.14171 [cs, stat].

- Hado P Van Hasselt, Arthur Guez, Matteo Hessel, Volodymyr Mnih, and David Silver. Learning values across many orders of magnitude. *Advances in neural information processing systems*, 29, 2016.
- Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning*, ICML’20. JMLR.org, 2020.
- Minqi Jiang, Michael Dennis, Jack Parker-Holder, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Replay-guided adversarial environment design. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 1884–1897. Curran Associates, Inc., 2021b. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/0e915db6326b6fb6a3c56546980a8c93-Paper.pdf.
- Samuel Coward, Michael Beukman, and Jakob Foerster. Jaxued: A simple and useable ued library in jax. *arXiv preprint*, 2024.
- Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

A Contextual Reinforcement Learning Setting

To readers more familiar with Cyber-Security than Reinforcement Learning, we provide an extended description of the notations presented in Section 2.

A.1 Markov Decision Process

An MDP [Puterman, 1990] 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; $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$, 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, that 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.

A.2 Partially Observable Markov Decision Process

A POMDP [Sutton and Barto, 2018] 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 $b_t \in \Delta(\mathcal{S})$.

A.3 Contextual Markov Decision Process

Following [Hallak et al., 2015] and the formulation adopted in [Kirk et al., 2023], a Contextual Markov Decision Process (CMDP) is a special class of POMDP with a context variable $c \in \mathcal{C}$, where $\mathcal{C}$ 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 $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}_c, \mathcal{R}_c, \Omega, \mathcal{O}_c, \rho_{0,c}, \gamma, \mathcal{C}, p(c) \rangle$, where $\mathcal{P}_c(s' | s, a)$ and $\mathcal{R}_c(s, a)$ denote the transition and reward functions parameterized by c , $\mathcal{O}_c(o | s', a)$ 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., $\mathbb{E}_{c \sim p(c)} [\mathbb{E}_\pi [G_0 | c]]$.

B Implementation Details

A note about software licenses. NASimJax is released under the MIT license. Algorithmic baselines adapted from JaxUED [Coward et al., 2024] and PureJaxQL [Gallici et al., 2025] retain their upstream Apache-2.0 license; full per-file attribution is provided in `THIRD_PARTY_NOTICES.md`. The remainder of this appendix provides the full semantics of host properties and encoding details elided from Section 4.1.

B.1 State Representation

Host properties. Each host carries the following attributes:

- `subnet address`: identifies the subnet the host belongs to (one-hot encoded).
- `reachable`: boolean flag, true once the host is network-reachable from the attacker.
- `discovered`: boolean flag, true once the host has been observed via subnet scan.
- `access level`: current privilege level held by the agent on the host (one-hot encoded over `{none, user, root}`).
- `OS`: the host’s operating system (one-hot encoded).
- `services`: vector of boolean flags indicating which vulnerable services are running.

- **processes:** vector of boolean flags indicating which vulnerable processes are running.

Static vs. dynamic properties. During episode execution, only three host values can change: `reachable`, `discovered`, and `access_level`. All other properties are fixed during network generation and remain static throughout the episode. This separation allows the transition function to be implemented as targeted bitwise updates on a small subset of the state tensor, which is one of the factors contributing to the throughput gains reported in Section 6.1.

Services and processes as filtered abstractions. 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 software inventory of a host, but rather a filtered abstraction capturing actionable attack surfaces. As a consequence, hosts may have zero vulnerable services or processes, modeling systems that are correctly configured or hardened. This design keeps the state representation compact while preserving the combinatorial structure of vulnerability-driven exploitation, which is the principal driver of action-space growth in Equation 1.

Adjacency matrix and firewall rules. Inter-subnet connectivity is represented by an $N_s \times N_s$ adjacency matrix with an additional dimension accommodating per-service firewall rules, which determine whether a given service is permitted to traverse a subnet boundary. The diagonal is populated with ones so that intra-subnet communication is always permitted. The matrix is not required to be symmetric, which allows the generator to produce directional connectivity patterns (see Section 4.2).

Encoding rationale and throughput impact. Consolidating host properties into a single $\{0, 1\}^{N_h \times F}$ tensor (Figure 1) with unsigned-integer storage (one byte per property) has two advantages. First, the majority of transition logic reduces to bitwise operations, which carry very small computational overhead under JIT compilation. 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.

Homogeneous Subnets. 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 suc_d and $proc_d$. This approach allows controlled variation in intra-subnet diversity while preserving the overall density of services and processes.

B.2 Mutations

We provide five mutations for generated networks:

1. **Toggle service.** Adds or removes a service from a random host with equal probability.
2. **Toggle process.** Adds or removes a process from a random host with equal probability.
3. **Change OS.** Performs an OS change to a random host.
4. **Move sensitive.** Moves the sensitive flag from one host to another. This changes the goal condition of the generated network.
5. **Flip network edge.** Flips the edge between two subnets.

After mutations, we perform checks whether the solvability is still guaranteed and apply patches if necessary. This is especially crucial for the last mutation, where the sole subnet with sensitive host(s) can get disconnected. Therefore, to still provide guaranteed solvable networks, a new sensitive host that is reachable needs to be chosen that is not the starting host and within the DMZ.

C Speed Comparison Details

Table 2 showcases the full details of our experiment in Section 6.1.

Table 2: Performance Comparison of NASim and NASimJax Implementations on small environment

Num Envs	NASim		NASimJax-1M		NASimJax-10M		NASimJax-100M	
	Time (s)	Steps/s	Time (s)	Steps/s	Time (s)	Steps/s	Time (s)	Steps/s
8	1487	6,724	23	43,478	126	79,365	1154	86,655
16	789	12,674	18	55,556	73	136,986	617	162,075
32	642	15,576	14	71,429	42	238,095	323	309,598
64	457	21,881	12	83,333	28	357,143	193	518,135
128	358	27,932	11	90,909	20	500,000	105	952,381
256	305	32,786	10	100,000	15	666,667	63	1,587,302
512	272	36,764	9	111,111	11	909,091	40	2,500,000
1024	250	40,000	10	100,000	12	833,333	32	3,125,000
2048	195	51,282	10	100,000	12	833,333	28	3,571,429
4096	197	51,546	10	100,000	11	909,091	25	4,000,000

Table 3: Hyperparameter Search Ranges: PPO 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}
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}

D Hyperparameters

We perform a hyperparameter search using Bayesian-sampling for every discussed algorithm in Section 5, with a budget of at least 275 trials. The number of trials for every algorithm is listed in Table 19. Each sampled hyperparameter set is trained on 26 host networks. 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 sweep, to select the final parameter sets, we evaluate the top five most promising hyperparameters sets using five control seeds and pick the best performing and most stable ones. This selection process is repeated for every network size.

D.1 Hyperparameter Ranges

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 PPO with DR. Then based on these results, narrowed down 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. [Jiang et al., 2021a,b]. We run two sweeps for both PLR and $\text{PLR}^\perp$, each using a different prioritization, to further limit the search space. This is because top_k prioritization additionally needs to sample the k , which rank -based prioritization does not. For UED methods using PPO-LSTM we use reduced search spaces based on relevant hyperparameters found through the previously executed PPO sweeps. We only thus only present the selected values for PPO-LSTM.

D.2 Experiment Hyperparameters

The following hyperparameters were held constant across all algorithms and network sizes: number of environments (1024), activation function ($\tanh$), discount factor (0.995), and number of update epochs (4). In addition, we fix the sizes of the minibatches to 4096. For PLR and $\text{PLR}^\perp$ variants, the

Table 4: Hyperparameter Search Ranges: PQN with DR

Hyperparameter	Considered Values
Learning Rate (LR)	{1e-4, 2e-4, 3e-4}
# Environments	{1024}
# Steps	{32, 64, 128}
Layer Size	{256, 512}
Activation Function	{tanh}
Discount Factor (γ)	{0.975, 0.99, 0.995}
GAE Lambda (λ)	{0.3, 0.5, 0.65, 0.8, 0.9}
Epsilon Start (ϵ)	{1.0}
Epsilon Finish (ϵ)	{0.01, 0.03, 0.05, 0.08, 0.1, 0.12}
Epsilon Decay (ϵ)	{0.8, 0.5, 0.3}
# Minibatches	{4, 8, 16}
Update Epochs	{4}
Reward Scale	{1.0}

Table 5: Hyperparameter Search Ranges: PLR and PLR⁺

Hyperparameter	Considered Values
Learning Rate (LR)	{2e-4, 3e-4, 5e-4}
# Steps	{32, 64, 128}
Layer Size	{256, 512}
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 6: Hyperparameter Search Ranges: ACCEL

Hyperparameter	Considered Values
Learning Rate (LR)	{2e-4, 3e-4, 5e-4}
# Steps	{64, 128}
GAE Lambda (λ)	{0.8, 0.95}
Clip Epsilon	{0.1, 0.2, 0.3}
Entropy Coef.	{0.005, 0.01}
Replay Probability (ρ)	{0.5, 0.7, 0.8, 0.9}
Staleness Coefficient (ω)	{0.1, 0.3, 0.5, 0.7}
Temperature (β)	{0.1, 0.3, 0.5, 1.0, 2.0}
Level Buffer Capacity	{1000, 4000, 10000}
Score Function	{MaxMC, PVL}
Prioritization	{Rank}
# Edits	{3, 5, 10}

Table 7: Hyperparameters for PPO DR across 16, 26, and 40 host networks.

Hyperparameter	PPO DR		
	16h	26h	40h
Learning Rate	3e-4	3e-4	3e-4
# Steps	32	128	64
Layer Size	512	512	512
GAE Lambda (λ)	0.8	0.8	0.8
Clip Epsilon	0.1	0.2	0.1
Entropy Coef.	0.005	0.005	0.005
Max Grad Norm	1.0	0.5	0.5
Value Coef.	1.0	0.5	0.25

Table 8: Hyperparameters for PQN DR across 16, 26, and 40 host networks.

Hyperparameter	PQN DR		
	16h	26h	40h
Learning Rate	2e-4	2e-4	2e-4
# Steps	32	128	128
Layer Size	512	512	512
Discount Factor (γ)	0.975	0.99	0.99
GAE Lambda (λ)	0.3	0.65	0.65
Epsilon Start (ϵ)	1.0	1.0	1.0
Epsilon Finish (ϵ)	0.01	0.01	0.01
Epsilon Decay (ϵ)	0.3	0.3	0.3
# Minibatches	8	8	8
Max Grad Norm	0.5	0.5	0.5
Reward Scale	1.0	1.0	1.0

level buffer capacity (10,000), minimum fill ratio (0.5), prioritization strategy (rank-based), and top- k (disabled) were additionally held constant.

For PQN, we fixed the reward scale at 1.0. We also empirically tested whether greedy evaluation is better than stochastic evaluation and found that greedy evaluation works better. We ran this test because of the stochastic nature of the environment, and on the basis that generally, stochastic policies perform better in partially observable environments.

For PPO-LSTM we held constant: number of environments (1024), activation function ($\tanh$), discount factor (0.995), update epochs (4), trunk layer size (512), LSTM layers (1), and LR annealing. For the PLR family: rank prioritization, top- k disabled, level buffer capacity (10,000), and minimum fill ratio (0.5, except ACCEL at 16 hosts). *Exploratory grad updates* (True for PLR and ACCEL, False for PLR[⊥]) is the structural PLR/PLR[⊥] distinction, not a tuned value.

E Active Host Distributions

Here we present the full set of distributions for the t_d -values established in Table 1. 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, 7 and 8. These clearly showcase the shift in active hosts given t_d .

F 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 12, 13 14 display more detailed results, where we plot for every algorithm and t_d -train value the performance against the evaluation t_d -values.

Table 9: Hyperparameters for PLR across 16, 26, and 40 host networks.

Hyperparameter	PPO PLR		
	16h	26h	40h
Learning Rate	3e-4	3e-4	3e-4
# Steps	128	64	64
Layer Size	512	512	512
GAE Lambda (λ)	0.8	0.8	0.8
Clip Epsilon	0.3	0.3	0.3
Entropy Coef.	0.01	0.005	0.01
Max Grad Norm	0.5	0.5	0.5
Value Coef.	0.5	0.5	0.5
Replay Prob. (ρ)	0.5	0.5	0.5
Staleness (ω)	0.7	0.3	0.5
Temperature (β)	0.1	2.0	2.0
Score Function	PVL	MaxMC	MaxMC

Table 10: Hyperparameters for PLR[⊥] across 16, 26, and 40 host networks.

Hyperparameter	PPO PLR [⊥]		
	16h	26h	40h
Learning Rate	3e-4	3e-4	3e-4
# Steps	128	128	128
Layer Size	512	512	512
GAE Lambda (λ)	0.95	0.95	0.95
Clip Epsilon	0.3	0.3	0.3
Entropy Coef.	0.005	0.005	0.005
Max Grad Norm	0.5	0.5	0.5
Value Coef.	0.5	0.5	0.5
Replay Prob. (ρ)	0.9	0.9	0.9
Staleness (ω)	0.1	0.5	0.1
Temperature (β)	2.0	2.0	2.0
Score Function	MaxMC	MaxMC	MaxMC

Table 11: Hyperparameters for ACCEL across 16, 26, and 40 host networks.

Hyperparameter	PPO ACCEL		
	16h	26h	40h
Learning Rate	2e-4	3e-4	2e-4
# Steps	128	128	128
Discount Factor (γ)	0.99	0.99	0.99
Layer Size	512	512	512
GAE Lambda (λ)	0.95	0.8	0.95
Clip Epsilon	0.2	0.2	0.2
Entropy Coef.	0.01	0.005	0.01
# Edits	10	5	5
Max Grad Norm	0.5	0.5	0.5
Value Coef.	0.5	0.5	0.5
Replay Prob. (ρ)	0.5	0.5	0.5
Staleness (ω)	0.7	0.3	0.1
Temperature (β)	2.0	0.3	1.0
Score Function	PVL	PVL	PVL
Min. Fill Ratio	0.3	0.5	0.5
Level Buffer Capacity	4000	10000	10000

Table 12: Hyperparameters for PPO-LSTM DR across 16, 26, and 40 host networks.

Hyperparameter	PPO-LSTM DR		
	16h	26h	40h
Learning Rate	3e-4	3e-4	3e-4
# Steps	64	32	64
LSTM Hidden Size	512	512	512
GAE Lambda (λ)	0.8	0.8	0.8
Clip Epsilon	0.1	0.1	0.1
Entropy Coef.	0.005	0.005	0.005
Max Grad Norm	1.0	1.0	0.5
Value Coef.	0.5	0.5	0.5

Table 13: Hyperparameters for PPO-LSTM PLR across 16, 26, and 40 host networks.

Hyperparameter	PPO-LSTM PLR		
	16h	26h	40h
Learning Rate	2e-4	2e-4	3e-4
# Steps	128	128	128
LSTM Hidden Size	256	256	512
GAE Lambda (λ)	0.8	0.8	0.8
Clip Epsilon	0.1	0.1	0.1
Entropy Coef.	0.01	0.01	0.005
Max Grad Norm	0.5	0.5	0.5
Value Coef.	0.5	0.5	0.5
Replay Prob. (ρ)	0.5	0.5	0.5
Staleness (ω)	0.7	0.7	0.5
Temperature (β)	2.0	2.0	2.0
Score Function	MaxMC	MaxMC	MaxMC

Table 14: Hyperparameters for PPO-LSTM PLR[⊥] across 16, 26, and 40 host networks.

Hyperparameter	PPO-LSTM PLR [⊥]		
	16h	26h	40h
Learning Rate	3e-4	2e-4	2e-4
# Steps	128	128	128
LSTM Hidden Size	256	256	256
GAE Lambda (λ)	0.95	0.95	0.95
Clip Epsilon	0.2	0.2	0.2
Entropy Coef.	0.01	0.01	0.01
Max Grad Norm	0.5	0.5	0.5
Value Coef.	0.5	0.5	0.5
Replay Prob. (ρ)	0.9	0.9	0.9
Staleness (ω)	0.3	0.3	0.3
Temperature (β)	1.4	2.0	2.0
Score Function	MaxMC	MaxMC	MaxMC

Table 15: Hyperparameters for PPO-LSTM ACCEL across 16, 26, and 40 host networks.

Hyperparameter	PPO-LSTM ACCEL		
	16h	26h	40h
Learning Rate	3e-4	3e-4	3e-4
# Steps	128	128	128
LSTM Hidden Size	512	512	512
GAE Lambda (λ)	0.8	0.8	0.8
Clip Epsilon	0.3	0.1	0.1
Entropy Coef.	0.01	0.005	0.005
Max Grad Norm	0.5	0.5	0.5
Value Coef.	0.5	0.5	0.5
Replay Prob. (ρ)	0.5	0.5	0.5
Staleness (ω)	0.7	0.1	0.5
Temperature (β)	0.1	0.3	0.1
Score Function	MaxMC	MaxMC	PVL
# Edits	10	10	10
Min. Fill Ratio	0.3	0.5	0.5

Table 16: Composition and hardware configuration of the cluster compute nodes partially used for experiments.

Cluster	GPU #	CPU	GPU	VRAM	RAM
Cluster 1	Node 1	64 core CPU	4xNVIDIA H100	372 GB	251 GB
	Node 2	64 core CPU	4xNVIDIA A100	320 GB	377 GB
Cluster 2	Node 1	2xAMD EPYC 9275F (24C)	2xNVIDIA H200	280 GB	754 GB

G Computational resources

In this Section we present the computational resources used to produce the results presented in the paper. Table 16 presents the hardware configurations of the compute clusters we had access to. Table 17 presents the hardware configurations of the compute workstations & servers. Table 18 discloses the total wall-clock time necessary to reproduce the results of the experiments. Table 19 reports the total wall-clock time and the number of configurations evaluated for each hyperparameter sweep.

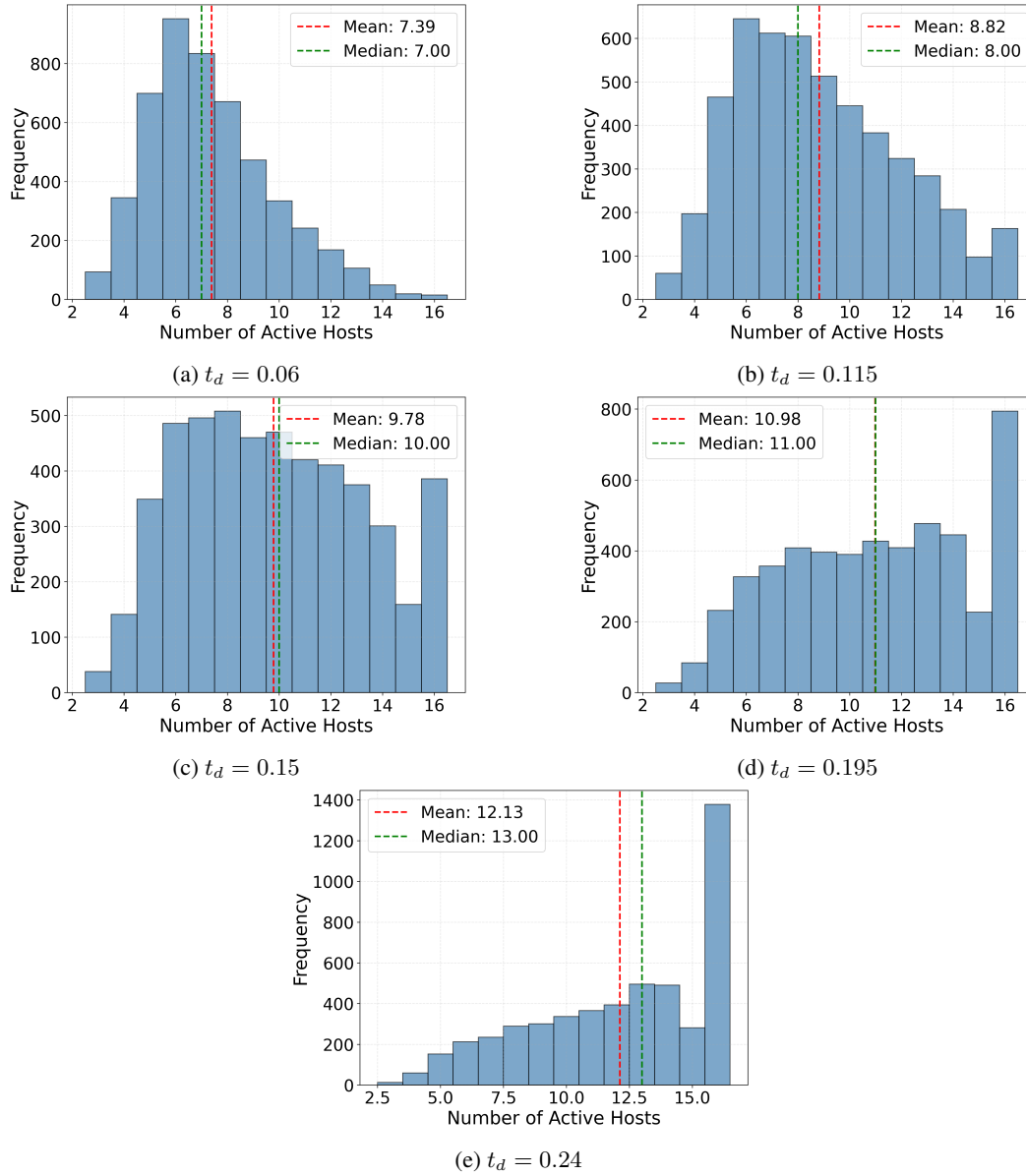


Figure 6: Distributions of active hosts within 16 host networks and how the topology density (t_d) parameter influences them.

Table 17: Hardware configuration of the compute machines used for experiments.

Machine	CPU	GPU	VRAM	RAM
6 Workstations	AMD Ryzen 9 9950x (16C)	NVIDIA RTX 5090	32 GB	64 GB
Server 1	Intel(R) Xeon(R) Gold 6230R (26C)	NVIDIA RTX A4000	16 GB	128 GB
Server 2	AMD EPYC 7413 (8VCPU Cores)	NVIDIA L40S	48 GB	96 GB

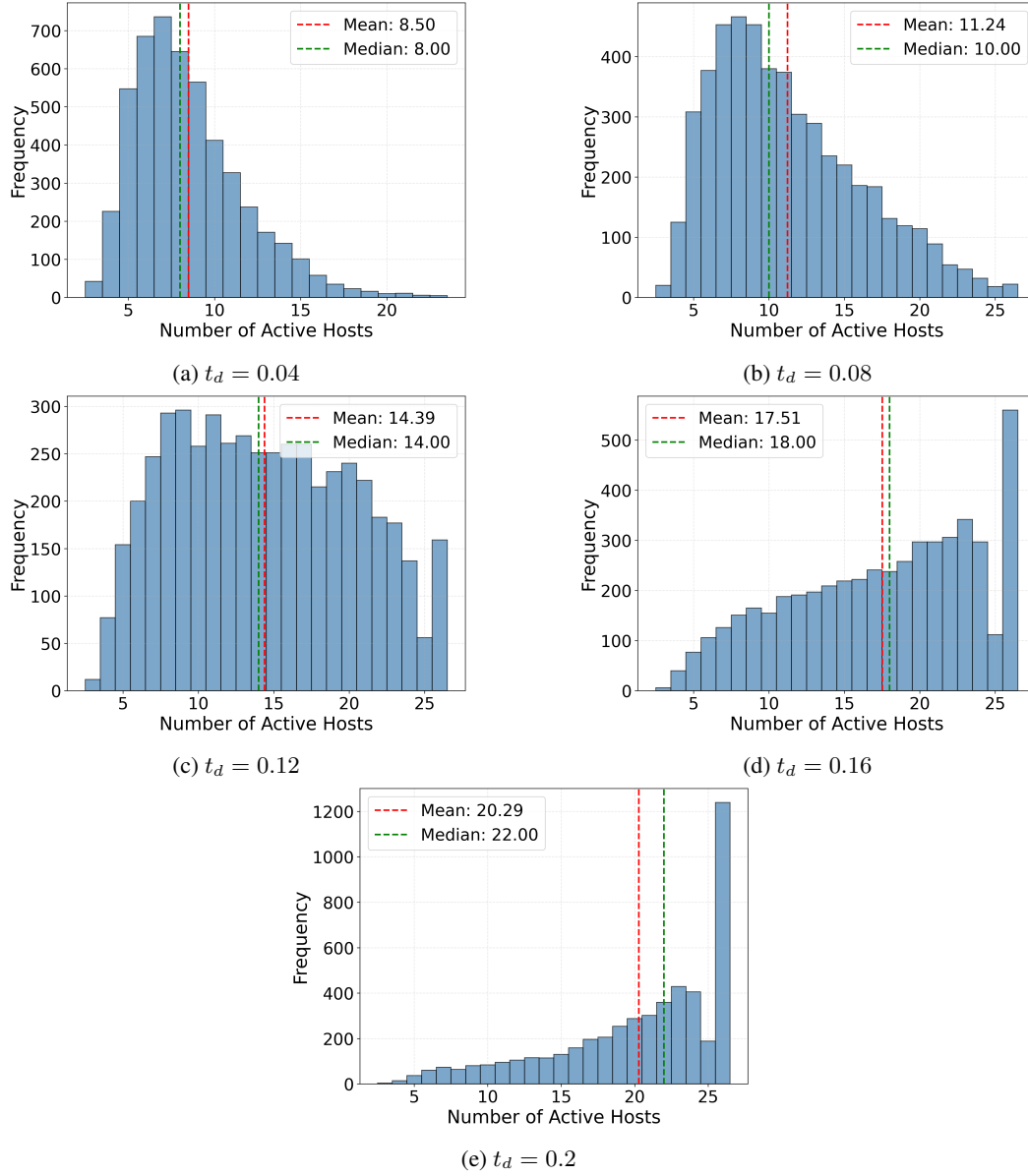


Figure 7: Distributions of active hosts within 26 host networks and how the topology density (t_d) parameter influences them.

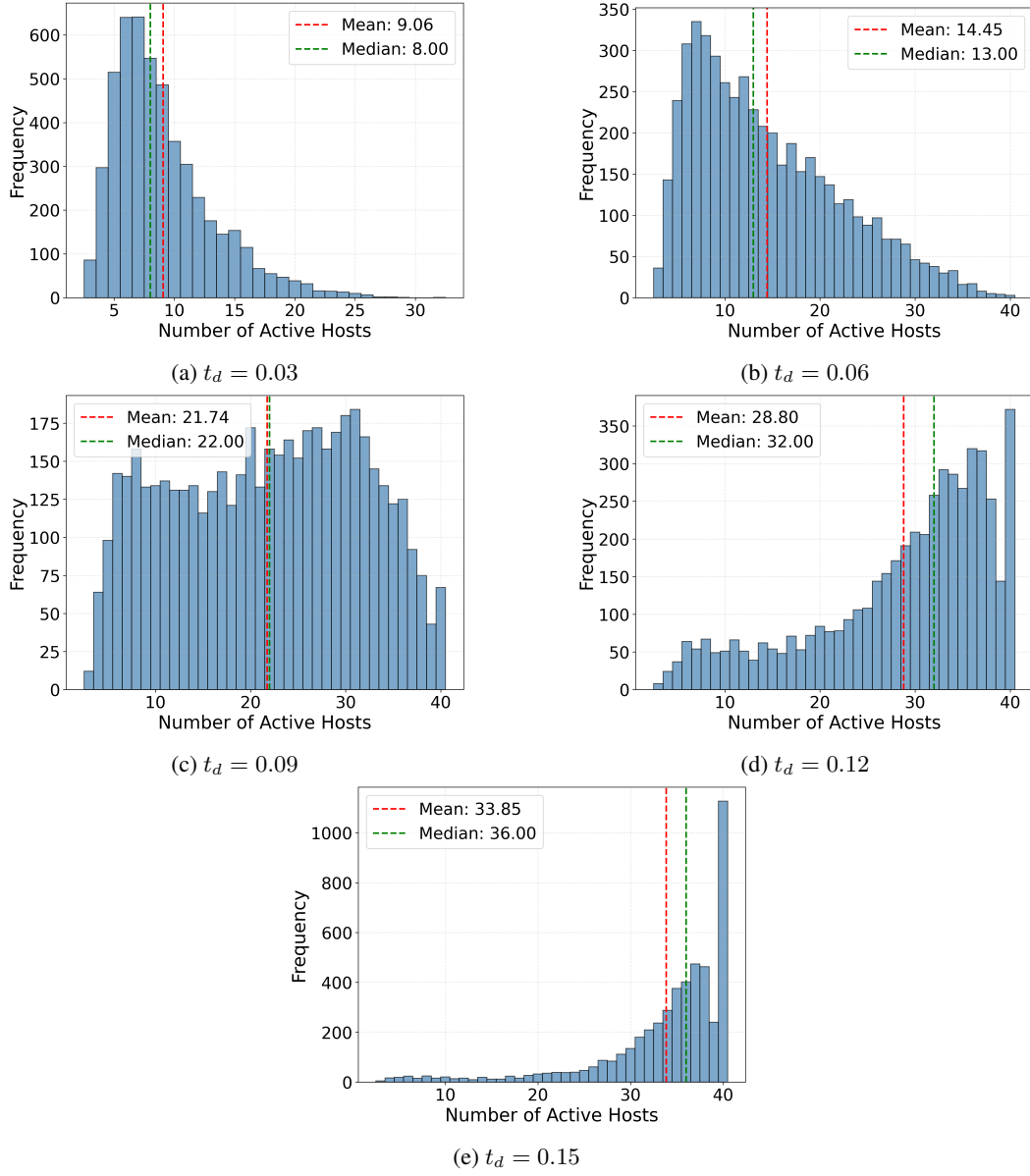


Figure 8: Distributions of active hosts for 40 host networks and how the topology density (t_d) parameter influences them.

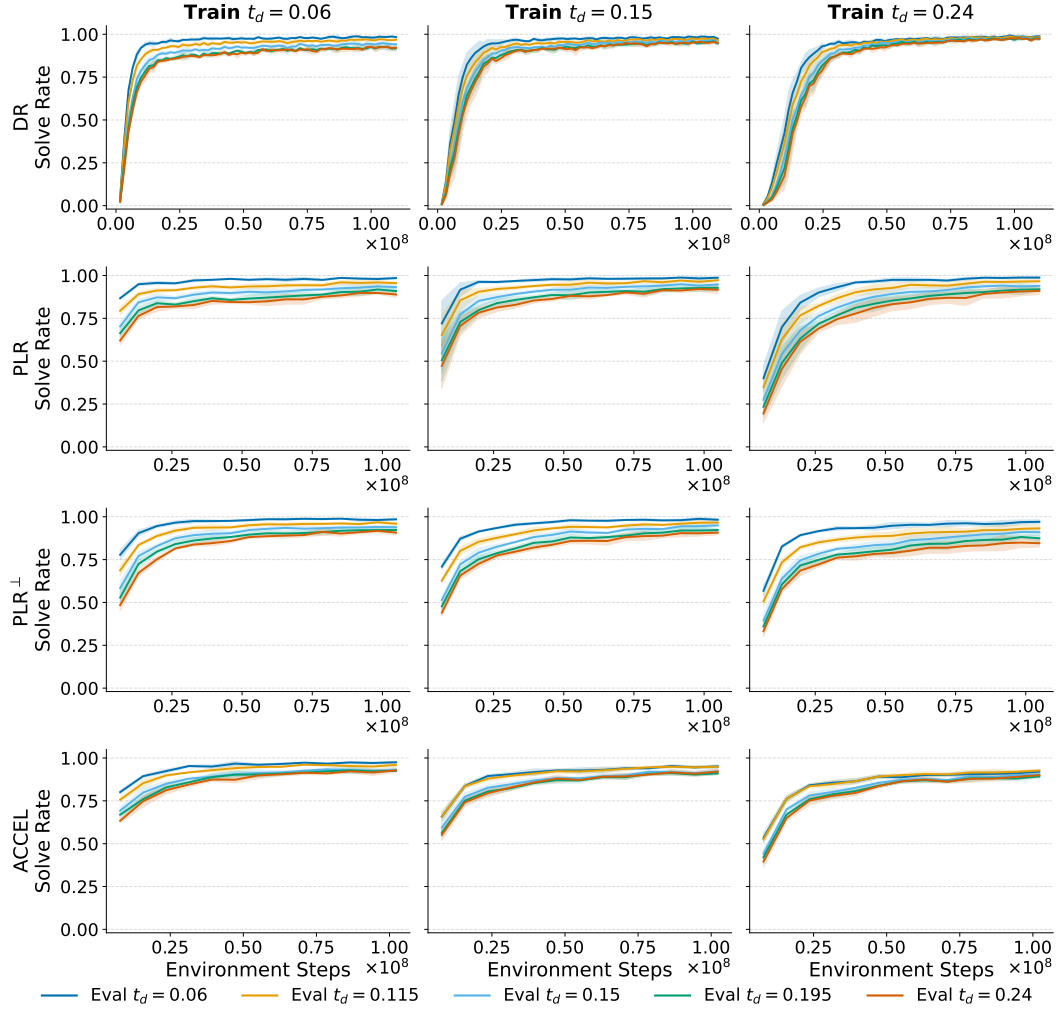


Figure 9: Evaluation curves for 16 host networks. Every column represents the network’s t_d the policy was trained on. The plots show the evaluation performance against all other selected t_d for the given network size, as displayed in Table 1.

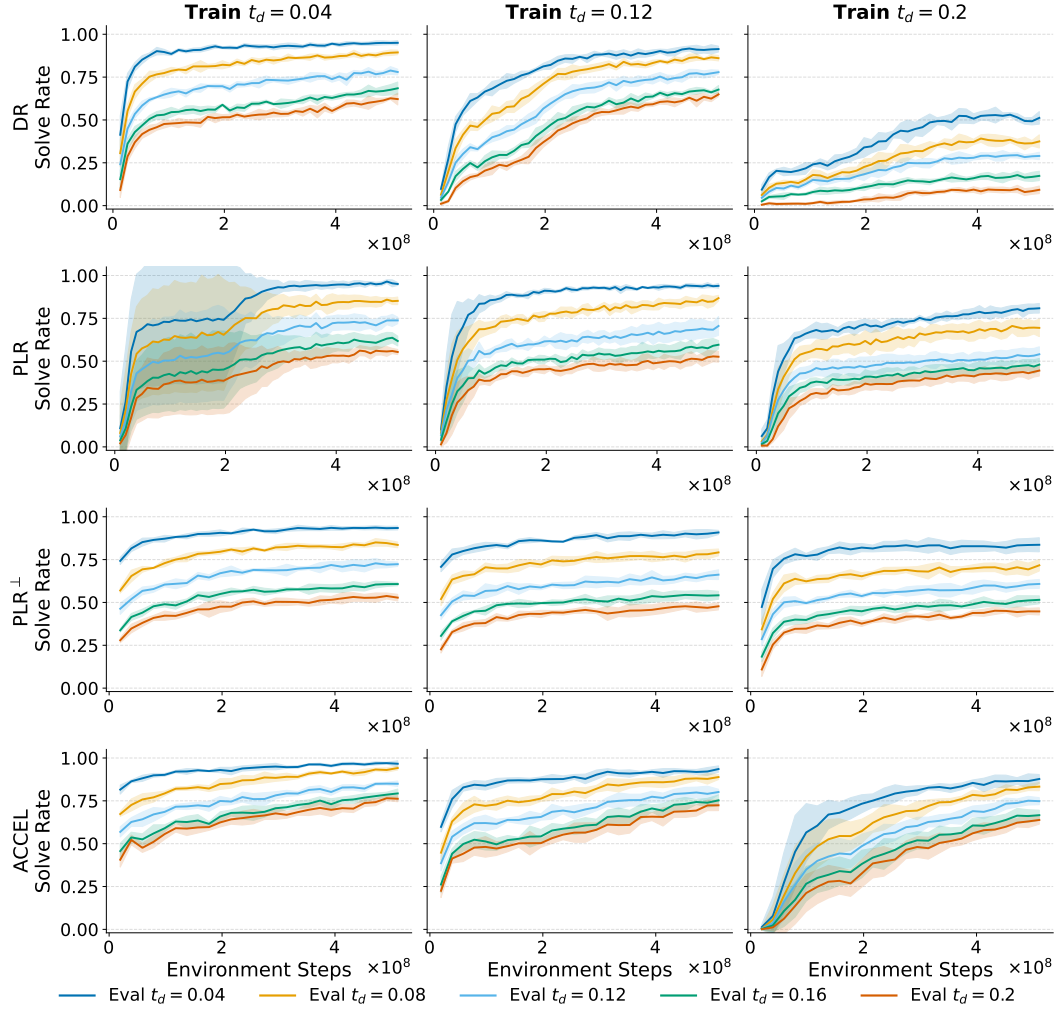


Figure 10: Evaluation curves for 26 host networks. Every column represents the network's t_d the policy was trained on. The plots show the evaluation performance against all other selected t_d for the given network size, as displayed in Table 1.

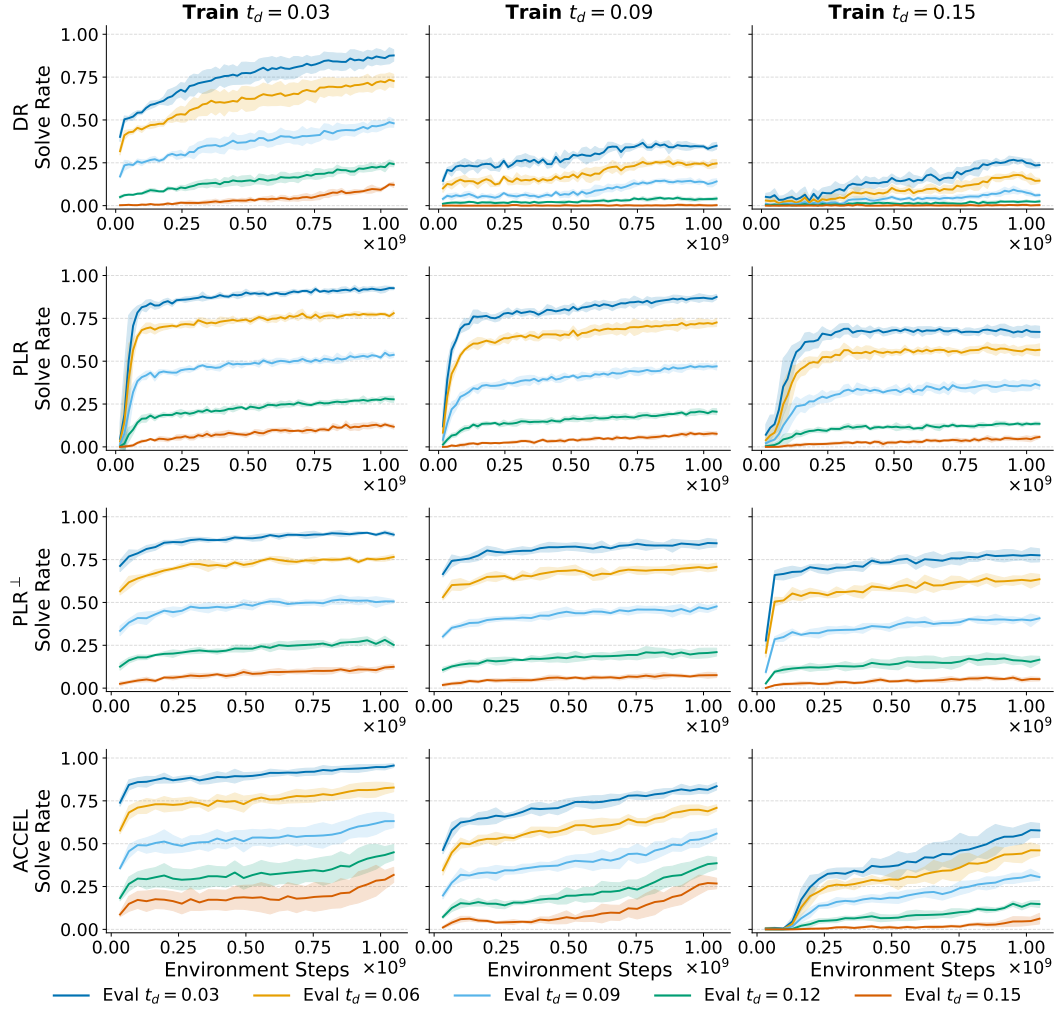


Figure 11: Evaluation curves for 40 host networks. Every column represents the network’s t_d the policy was trained on. The plots show the evaluation performance against all other selected t_d for the given network size, as displayed in Table 1.

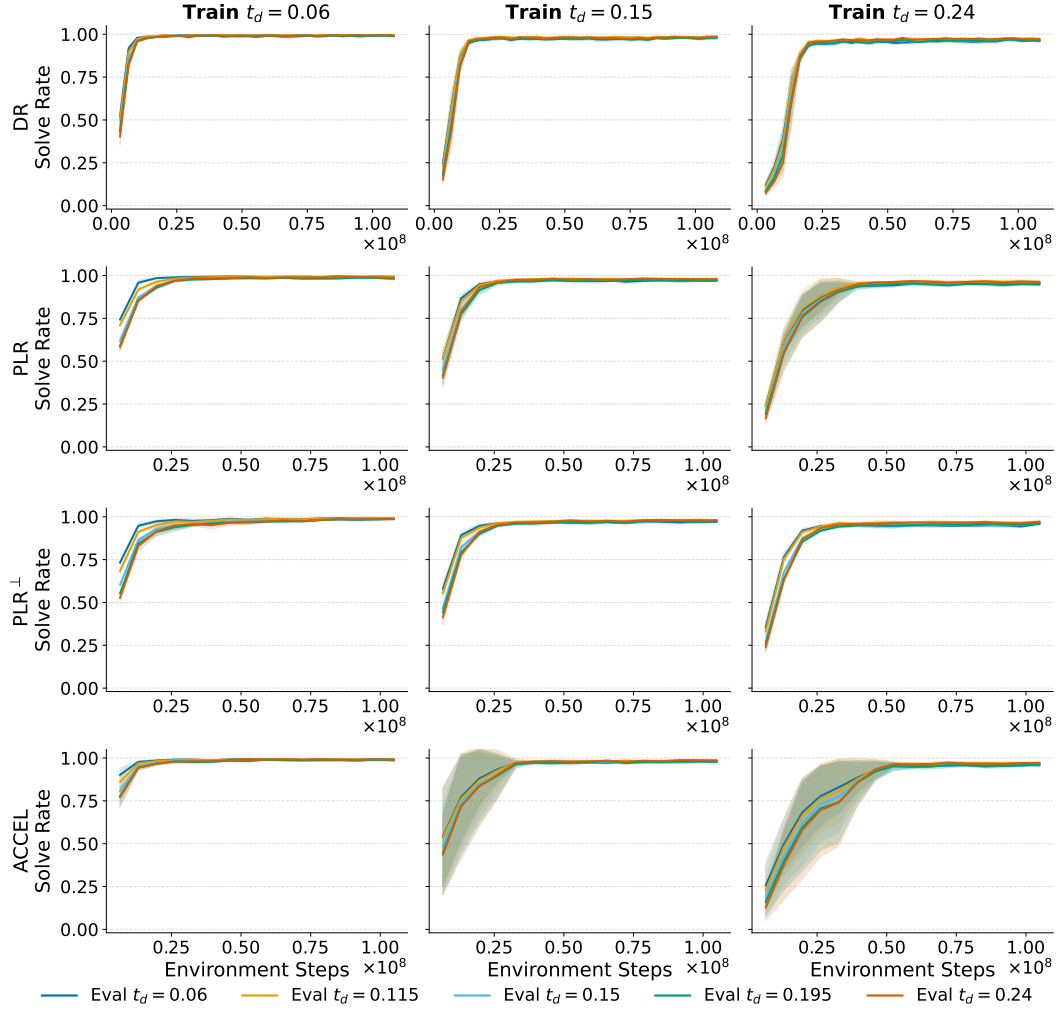


Figure 12: Evaluation curves for 16 host networks (PPO-LSTM). Every column represents the network’s t_d the policy was trained on. The plots show the evaluation performance against all other selected t_d for the given network size, as displayed in Table 1.

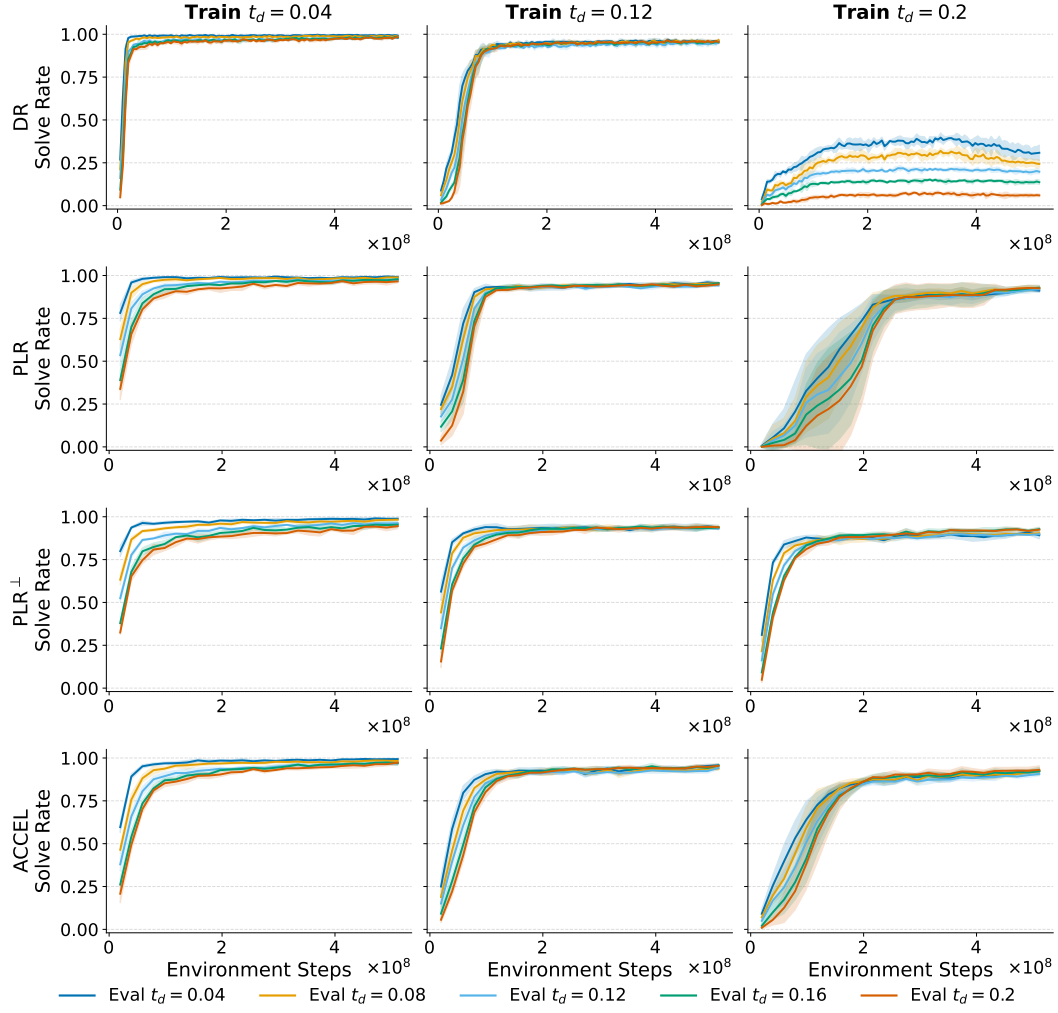


Figure 13: Evaluation curves for 26 host networks (PPO-LSTM). Every column represents the network’s t_d the policy was trained on. The plots show the evaluation performance against all other selected t_d for the given network size, as displayed in Table 1.

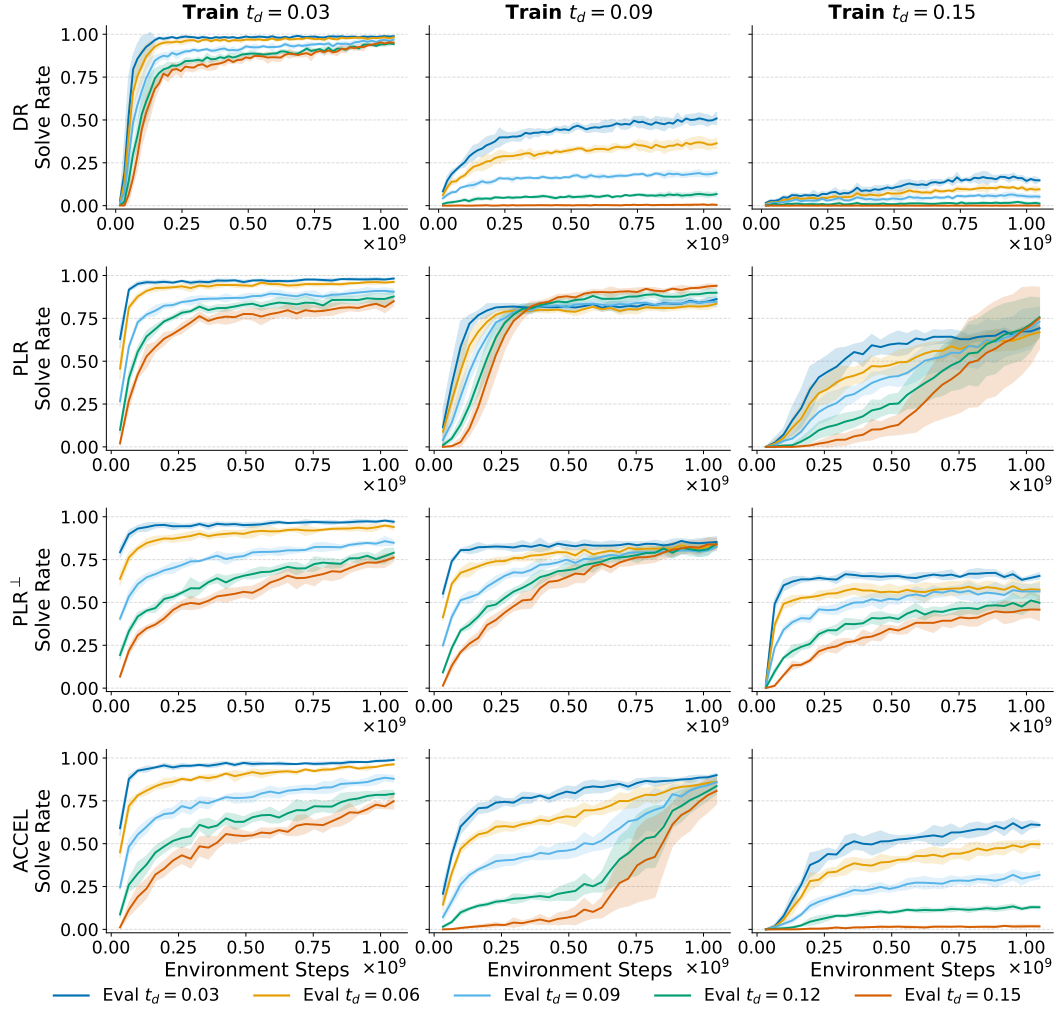


Figure 14: Evaluation curves for 40 host networks (PPO-LSTM). Every column represents the network’s t_d the policy was trained on. The plots show the evaluation performance against all other selected t_d for the given network size, as displayed in Table 1.

Table 18: Compute used to produce the experimental results. “GPU(s)” lists every distinct device that contributed; “Runtime” is the total wall-clock time summed over all seeds reported.

Experiment	GPU(s)	Runtime (all seeds)
Baseline PPO – 16 hosts	RTX 5090	57.0 min
Baseline PQN – 16 hosts	RTX 5090	1.2 h
Baseline PPO – 26 hosts	RTX 5090	3.2 h
Baseline PQN – 26 hosts	RTX 5090	3.2 h
Baseline PPO – 40 hosts	RTX 5090	7.7 h
Baseline PQN – 40 hosts	RTX 5090	7.5 h
Topo-density UED (PPO) – DR – 16 hosts	H100 NVL	2.6 h
Topo-density UED (PPO) – PLR – 16 hosts	H100 NVL	1.1 h
Topo-density UED (PPO) – PLR [⊥] – 16 hosts	H100 NVL	1.3 h
Topo-density UED (PPO) – ACCEL – 16 hosts	RTX 5090, RTX A4000	2.9 h
Topo-density UED (PPO) – DR – 26 hosts	H100 NVL	7.6 h
Topo-density UED (PPO) – PLR – 26 hosts	H100 NVL	6.8 h
Topo-density UED (PPO) – PLR [⊥] – 26 hosts	H100 NVL	5.0 h
Topo-density UED (PPO) – ACCEL – 26 hosts	RTX 5090	5.2 h
Topo-density UED (PPO) – DR – 40 hosts	RTX 5090	41.2 h
Topo-density UED (PPO) – PLR – 40 hosts	RTX 5090	36.7 h
Topo-density UED (PPO) – PLR [⊥] – 40 hosts	A100-PCIE-40GB, RTX 5090	39.5 h
Topo-density UED (PPO) – ACCEL – 40 hosts	L40S, RTX 5090	18.0 h
Topo-density UED (LSTM) – DR – 16 hosts	RTX A4000	11.3 h
Topo-density UED (LSTM) – PLR – 16 hosts	RTX A4000	7.2 h
Topo-density UED (LSTM) – PLR [⊥] – 16 hosts	RTX 5090	3.9 h
Topo-density UED (LSTM) – ACCEL – 16 hosts	RTX 5090	4.2 h
Topo-density UED (LSTM) – DR – 26 hosts	RTX 5090	21.3 h
Topo-density UED (LSTM) – PLR – 26 hosts	RTX 5090, RTX 6000 Ada Generation	16.5 h
Topo-density UED (LSTM) – PLR [⊥] – 26 hosts	RTX 5090, RTX 6000 Ada Generation	16.3 h
Topo-density UED (LSTM) – ACCEL – 26 hosts	H100 NVL, RTX 5090	20.2 h
Topo-density UED (LSTM) – DR – 40 hosts	RTX 5090	40.6 h
Topo-density UED (LSTM) – PLR – 40 hosts	RTX 5090, RTX 6000 Ada Generation	38.8 h
Topo-density UED (LSTM) – PLR [⊥] – 40 hosts	H100 NVL, H200 NVL, RTX 5090	41.4 h
Topo-density UED (LSTM) – ACCEL – 40 hosts	RTX 5090	38.3 h
Total	(437 runs)	18.8 d (451 h)

Table 19: Compute used for the hyperparameter sweeps per algorithm.

Algorithm	GPU(s)	# Runs	Runtime (all runs)
PQN	RTX 5090, RTX 6000 Ada Generation, RTX A4000	383	149 h
ACCEL	A100-SXM4-80GB, H200 NVL, RTX 5090, RTX A4000	275	82.3 h
Robust PLR	A100-SXM4-80GB, H100 NVL, RTX 5090	471	250 h
PLR	A100-SXM4-80GB, H100 NVL, RTX 5090	476	305 h
PPO DR	H100 NVL, RTX 5090	331	129 h
PPO-LSTM DR	RTX 5090, RTX A4000	38	35.3 h
PPO-LSTM PLR	RTX 5090	58	55.3 h
PPO-LSTM Robust PLR	RTX 5090	48	46.0 h
PPO-LSTM ACCEL	H100 NVL, H200 NVL, RTX 5090, RTX 6000 Ada Generation, RTX A4000	258	238 h
Total		2338	1289 h