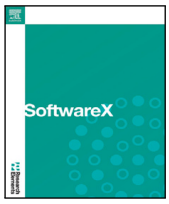
SoftwareX 31 (2025) 102312

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx



Original software publication

MininetGym: A modular SDN-based simulation environment for reinforcement learning in cybersecurity

Salvo Finistrella *, Stefano Mariani, Franco Zambonelli

Department of Sciences and Methods for Engineering, University of Modena and Reggio Emilia, Reggio Emilia, Italy

ARTICLE INFO

Keywords:
Reinforcement Learning
Cybersecurity
Software-Defined Networking
Mininet
Gymnasium
DoS attack
Intrusion Detection System
Simulator

ABSTRACT

Cybersecurity demands increasingly adaptive techniques to detect and mitigate sophisticated threats. This paper presents *MininetGym*, a Software Defined Network (SDN)-based simulation framework with a modular architecture based on Mininet, offering high configurability for experiments aimed at evaluating Reinforcement Learning (RL) strategies in network traffic classification and attack detection tasks. Three use cases are implemented: (i) traffic classification, (ii) detection of Denial of Service attacks in real-time, and (iii) the extension of the simulator with custom environments. The framework supports tabular and deep RL agents and includes modular components for network emulation, traffic generation, and agent interaction. The framework generates evaluation metrics and visualizations, enabling users to analyze agent performance and understand their behavior in dynamic network conditions.

Code metadata

Current code version	v1.0
Permanent link to code/repository	[https://github.com/ElsevierSoftwareX/SOFTX-D-25-00432 [1]]
Legal Code License	MIT
Code versioning system used	Git
Software code languages	Python, Bash
Compilation requirements	Python 3.11, Mininet, Gymnasium, Stable-Baselines3, OpenDaylight (optional)
Computing platforms/Operating Systems	Linux, Windows, MacOS
Link to documentation	https://github.com/dipi-unimore/mininet-gym/blob/main/readme.md
Support email for questions	salvo.finistrella@unimore.it

1. Motivation and significance

RL has shown increasing promise in addressing cybersecurity challenges [2], particularly in dynamic and complex environments like Software-Defined Networks (SDN) [3]. Despite numerous theoretical advancements, the practical deployment and experimental evaluation of RL-based cybersecurity solutions still face significant barriers. These include the need for reproducible testing environments, customizable traffic generation, and support for real-time interaction between agents and networks.

Our framework uniquely combines SDN-based traffic monitoring with support for various RL algorithms. It enables flexible attack classification experiments, which we define as the ability to (i) configure and execute types of attacks (e.g., DoS) with varying intensity and timing; (ii) schedule traffic patterns dynamically at runtime; and (iii) evaluate agent performance under different adversarial strategies and network loads.

Unlike older simulation tools that do not support SDN controllers [4,5], and existing datasets based on fixed, pre-recorded traffic that cannot easily be adapted to new scenarios [6,7], our framework can use OpenDaylight [8] to dynamically manage network traffic and datasets tailored to different environments.

Additionally, most existing simulators emphasize static, discrete attack-response logic, exemplified by platforms such as the Session-level Adversary Intent-Driven Cyberattack Simulator [9] or DEVS-based cyber-attack simulators [10], which often rely on pre-defined attack scenarios or state machine descriptions of attack stages. Our framework

* Corresponding author.
E-mail addresses: salvo.finistrell@unimore.it (S. Finistrella), stefano.mariani@unimore.it (S. Mariani), franco.zambonelli@unimore.it (F. Zambonelli).

https://doi.org/10.1016/j.softx.2025.102312
Received 25 June 2025; Received in revised form 29 July 2025; Accepted 11 August 2025
Available online 3 September 2025

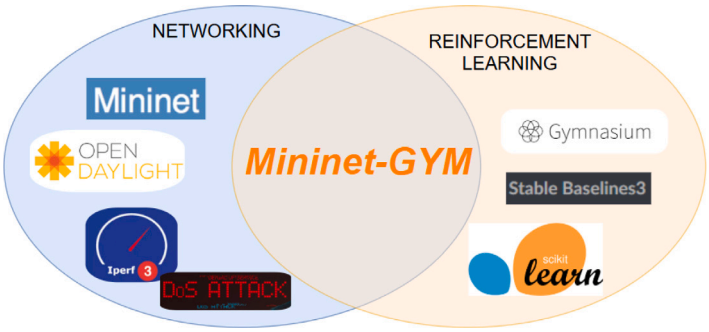


Fig. 1. Simulator key components.

fills this gap by allowing experiments that emulate realistic defender workloads and evolving attacker behaviors, enabling more practical and transferable RL training in cybersecurity contexts.

The framework integrates key technologies (Fig. 1) to provide a RL testing environment:

- **Mininet** [11] a network emulator that creates a realistic virtual network on a single computer, running real Linux kernel network code. It is commonly used for rapidly prototyping, testing, and developing SDN applications and other network protocols.
- **OpenDaylight** and **Open vSwitch** [12] provide SDN control and access to flow-level statistics via REST APIs.
- **Gymnasium** [13] ensures interoperability with RL libraries through a standard environment API.
- **Stable-Baselines3** [14] is used to train deep RL agents, while custom tabular agents (Q-learning, SARSA) support lightweight training and testing.

A curated list of RL-based cybersecurity environments and tools is available at [15]. However, only a few platforms, specifically CybORG, CyberBattleSim, and GNS3+RL/NS3-Gym, are worth comparing against due to their relevance to SDN-based experimentation, agent training, and integration with standard RL libraries (summary in Table 1):

- **CybORG** [16]: a cybersecurity research and agent development platform, integrating RL and the MITRE ATT&CK framework. While recent extensions allow integration with Mininet to enhance realism, the default simulation remains highly abstract, limiting its effectiveness for training agents in realistic and dynamic network environments.
- **CyberBattleSim** by Microsoft [17]: an environment for modeling attacker–defender dynamics in enterprise networks. However, it operates at a high level of abstraction and does not emulate real packet flows or use SDN controllers, which makes it less suitable for flow-based traffic analysis and low-level agent interaction.
- **GNS3+RL** extensions and NS3-Gym [18]: simulation-based platforms that allow interfacing RL agents with realistic network simulators. These tools provide packet-level precision but often involve complex configurations, slower runtime performance, and limited support for real-time dynamic control and extensible Gym-like agent integration compared to our simulator.

As detailed in Table 1, *MininetGym* distinctively addresses these gaps by offering SDN integration, exemplified by its compatibility with OpenDaylight, alongside robust support for dynamic, real-time traffic generation and customizable benchmarking. This feature set, including its support for various RL algorithms, positions MininetGym as a uniquely flexible and realistic platform for evaluating RL strategies in cybersecurity scenarios, surpassing the limitations observed in other solutions.

Structure of the Paper. The remainder of this paper is organized as follows: Section 2 provides an overview of the simulator architecture and its main modules. Section 3 demonstrates three illustrative experiments: traffic classification, DoS attack detection, and adding a new environment. Section 4 discusses the broader impact and use cases of the tool. Finally, Section 5 concludes the paper and highlights potential future extensions.

Table 1
Comparison with similar RL simulation environments.

MininetGym Features	CyberBattle Sim	NS3-Gym	CybORG (extended)
Gymnasium API	✓	✓	✓
SDN integration	×	×	Limited
Mininet traffic generation	×	×	×
Custom attack simulation	✓	×	✓
Deep and tabular RL	×	✓	✓

2. Software description

MininetGym is a Python-based simulation framework designed to support developing and evaluating RL agents for cybersecurity tasks in realistic, SDN environments. It enables training on real-time traffic patterns and interaction with actual flow-level data, allowing the study of network intrusion detection, traffic classification, and dynamic response strategies using RL techniques. It is publicly available at our GitHub page [1], and open to external contributions.

2.1. Software architecture

The proposed simulator comprises five modules (labeled in blue and inserted in colored blocks) that interact to provide a flexible RL-based cybersecurity testing platform, each responsible for a core aspect, as depicted in Fig. 2.

User Interface. The simulator is configurable via a YAML file, and currently, outputs progress through the terminal. A web-based frontend is under development to enable intuitive experiment configuration and real-time visualization of training results. It interacts with the backend via Flask APIs and WebSockets for live updates.

Network Environment. A custom RL environment compatible with the Gymnasium API, wrapping a Mininet-based SDN topology managed by an OpenDaylight controller. The environment is initialized through Gymnasium, which structures the RL loop over a network simulation created with Mininet. This module exposes the network state, specifically packet and byte flow statistics extracted from Open vSwitch, to the RL agents. The network topology is configurable: users can specify the number of hosts, IoT devices, switches, and their interconnections. This flexibility enables testing of RL algorithms across a wide range of

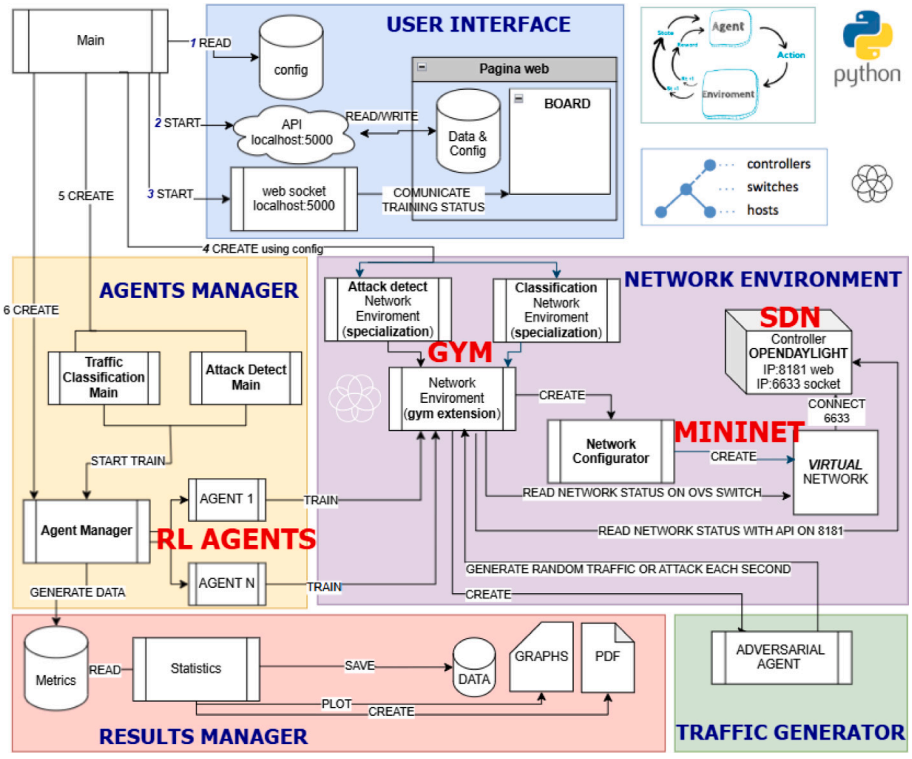


Fig. 2. Software architecture of the simulator, composed of UI, environment, agent manager, traffic generator, and results manager. In a typical workflow, the user configures an experiment directly on the *config.yaml*, which triggers the network environment setup and RL agent training. The adversarial agent continuously generates traffic, and the RL agent receives flow statistics from OpenDaylight to take actions. Results are logged.

scenarios, from simple star topologies to more complex multi-switch environments.

Agents Manager. Manages the instantiation, configuration, training, and evaluation of RL agents, both custom implementations and from the Stable-baselines3 library.

Traffic Generator. This module uses an *adversarial agent* to dynamically generate traffic across the network. In the real-time version, every second, each host is assigned a traffic type, either normal (e.g., ping, UDP, TCP) or malicious (short-term or long-term DoS attack), based on a configurable probability distribution. Normal traffic is generated using standard utilities like `ping` and `iperf`, while DoS attacks are simulated using `iperf3` to produce high-throughput, long-lasting connections. For instance, the probability of generating an attack may be low at the beginning of training (e.g., 2%) and then increase over time to ensure a gracefully increasing distribution of attack samples throughout training.

Results Manager. Stores all raw and processed experimental data to ensure reproducibility and detailed post-hoc analysis. Key performance metrics such as accuracy, precision, recall, F1-score, and confusion matrices for each evaluation episode are saved in JSON format, providing structured access to the agent's performance logs. For in-depth traffic analysis, raw network flow data, including packet and byte counts, is exported to CSV/JSON files. Visualizations of training progression and classification performance, generated during or after experiments, are saved as PNG image files. Further details on the output file structure is provided in the respective evaluation sections for Use Case 1 (Section 3.1) and Use Case 2 (Section 3.2).

2.2. Details of RL components

The simulator is organized around the typical RL loop: the agent observes the current state, takes an action, receives a reward, and updates its policy accordingly. This cycle is implemented using the Gymnasium API, with environments inheriting from a shared base class and customized for specific tasks. Two environments are ready to use, each designed for a specific use case, whose execution requires proper configuration through a dedicated `yaml` file, but it is possible to create a custom environment as described in Section 3.3.

At the core of the simulator is the abstract base class `NetworkEnv` (Listing 1), which encapsulates functions to create **RL environments**, such as: initialization of the Mininet topology; state reset and update through Open vSwitch statistics; support for reward calculation and action execution.

```
class NetworkEnv(gym.Env):
    def __init__(self, params):
        # Network creation
        self.net = create_network(param .... #
            Network creation
        self.action_space = spaces.Discrete(self.
            num_actions)
    def reset(self): ...
    def step(self, action): ...
    @abstractmethod
    def calculate_reward(self, action):  pass
```

Listing 1: Key structure of NetworkEnv class

Concrete environments inherit from this base class and implement task-specific logic. For instance, class `NetworkEnvClassification` handles traffic classification among four types: `none`, `ping`, `udp`, `tcp`. The reward is symmetric (+1 / –1). Instead, class `NetworkEnvAttackDetect` is designed for binary DoS attack detection. It penalizes missed detections more heavily (reward from –1 to +1). The two included environments are summarized in Table 2.

Learning agents are managed by the `AgentManager` class, which: initializes tabular (Q-learning, SARSA) or deep (DQN, PPO, A2C) agents, loads or trains models as per the YAML config, and handles evaluation, metric collection, and logging. Tabular algorithms are implemented natively, while deep RL is handled via Stable-baselines3 [14].

Table 2
Comparison of the two RL environments implemented in the simulator.

	Classification	Attack detection
Action Space	`none`, `ping`, `udp`, `tcp`	`normal`, `attack`
Observation Space	4-dimensional: packets received, packets sent, bytes received, bytes sent	4-dimensional: packets, Δp, bytes, Δb
State Preprocessing	Discretization into bins based on traffic magnitude for tabular agents, normalization to a [0, 1] range to stabilize training for deep agents	Same
Reward	Reward of +1 for correct traffic type, −1 otherwise	Reward of +1 (attack detected), +0.5 (correct normal), −0.5 (false positive), "−1" (missed attack)
Used For	Traffic classification task	Real-time DoS attack detection task

All agents' logic support evaluation over multiple episodes with metrics such as accuracy, precision, and F1-score.

To **configure experiments** comfortably, all training parameters are externalized into a YAML configuration file. This includes: network topology (hosts, switches, IoT nodes), traffic generation parameters, and RL agent hyperparameters and episode length. Researchers can switch tasks or agents simply by editing the configuration, without touching the core code.

The behavior of the **traffic generator** can differ by environment. In the `classification` scenario, each episode triggers the generation of a single, isolated traffic type per step (one-shot). In the `attack detection` scenario, traffic generation is continuous and asynchronous. DoS attacks of varying durations (short and long) are injected with increasing probability if no attacks are detected for several steps. This design ensures realistic variability and avoids uniform traffic phases.

2.3. Key features

The simulator offers a flexible platform for RL in network security. Designed for usability and extensibility, it brings together real-time experimentation, fine-grained configurability, and modular design. In summary, its core features include:

- **Diverse Environments:** Based on the unified Gymnasium interface, the simulator supports two predefined tasks and simplifies the creation of new ones through subclassing and YAML-based configuration.
- **Agents Flexibility:** Agents using tabular and deep RL algorithms can be enabled in parallel with distinct settings, facilitating comparative studies across learning paradigms without code modifications.
- **Easy Hyperparameter Tuning:** Multiple variants of the same algorithm can be instantiated with different parameters, allowing efficient experimentation with learning strategies. This design facilitates comparative studies.
- **Real-Time Flow Monitoring:** Integration with Open vSwitch and using `ovs-dpctl` enables real-time access to traffic statistics. The `ovs-dpctl` tool allows direct querying of Open vSwitch to extract packets and bytes, which are used to build the observation space.
- **Adversarial Traffic Simulation:** A traffic generator emulates diverse traffic conditions, delivering a wide range of training data and controlled exposure to threats.

These features, built atop well-established tools and a modular architecture, make the simulator suitable not only for evaluating RL strategies but also for experimentation, education, and prototyping of cybersecurity defenses.

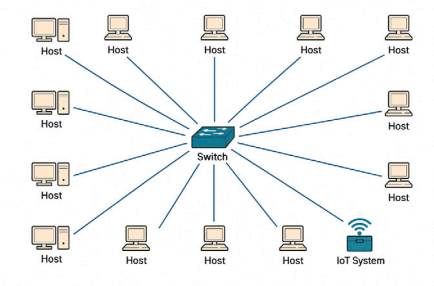


Fig. 3. Network topology used in use cases Traffic Classification and Dos Attack Detection.

3. Illustrative examples

To demonstrate the versatility of the simulator, we present three illustrative usage examples:

1. *Traffic Classification* – the agent identifies whether the ongoing traffic is `None`, `Ping`, `UDP`, or `TCP`. This environment is already implemented and can be used by simply adjusting the configuration in the YAML file.
2. *DoS Attack Detection* – the agent determines whether traffic in the network is normal or indicative of a DoS attack. This use case is also pre-implemented and requires only a YAML-based configuration to run.
3. *Adding a New Environment* – this example illustrates how to define a new Gymnasium-compatible environment by extending the simulator's base classes before configuring the tasks via a YAML file and running it.

The first two use cases share a **common base configuration** to ensure comparability of results and reproducibility of testing. Specifically, the experiments were run on a simple star topology composed of (see Fig. 3): 10 hosts simulating user or service traffic, 1 IoT device as a lightweight communication node, and 1 switch (Open vSwitch) controlled by an OpenDaylight SDN controller. This topology is configured in the `config.yaml` file under the `env_params` section as reported in Listing 2:

```
env_params:
    net_params:
      num_hosts: 10
      num_switches: 1
      num_iot: 1
      controller
          ip: 127.0.0.1
          port: 6653
          usr: xxx
          pwd: xxx
```

Listing 2: Excerpt of common environment configuration covering network parameters

The **environment** also includes task-specific configurations such as: `n_bins`, as the number of bins used to discretize continuous observation features for tabular agents; `steps_min_percentage` and `accuracy_min`, so that if the agent reaches at least `accuracy_min` after completing `steps_min_percentage` of the episode steps, the episode ends early; and `max_steps` the maximum number of steps in a episodes.

The **broader simulation behavior** is controlled by the additional parameters at the root level of the YAML shown in Listing 3.

```
training_directory: "_training"        #
    Output path
```

```
server_user: "ubuntu"                    #
    Linux user running the main
    application. It is required because
    launching Mininet involves root
    privileges, while the rest of the
    pipeline should run under a standard
    user account
random_seed: 42                          #
    For reproducibility
test_episodes: 20                        #
    Evaluation episodes after training
```

Listing 3: General simulation settings

Each use case utilizes a consistent structure for defining **RL agents** through the `agents` section of the configuration file. Each entry defines a single agent, its algorithm, and associated hyperparameters. Some parameters are shared across all agent types: name, algorithm (e.g., `Q-learning`, `Sarsa`, `DQN`, `PPO`, `A2C`), enabled, skip_learn, load, save, episodes.

Then, only in the *traffic classification* use case, a **supervised agent** is also defined. Unlike RL-based agents, this classifier (currently implemented using a Decision Tree) is trained offline using labeled traffic traces exported from the simulator. These traces are saved in CSV format at the end of a simulation by the RM module (e.g., `traffic.csv`). This agent is not included in the DoS detection scenario.

Tabular methods (e.g. `Q-learning` and `Sarsa`) are implemented from scratch in the simulator, exemplified in Listing 4.

```
  name: "QL_slow"             # Unique identifier
      for the agent
  algorithm: "Q-learning"     # RL algorithm custom
  enabled: true               # Set to false to
      exclude this agent
  skip_learn: false           # If true, skips
      training and only tests
  load: false                 # Load pretrained
      agent
  save: true                  # Save agent after
      training
  learning_rate: 0.05       # How much new info
      replaces old estimates
  discount_factor: 0.5        # Importance of future
       rewards (gamma)
  exploration_rate: 1.0       # Initial probability
      of random action
  exploration_decay: 0.9995   # Decay rate of
      exploration over time
  episodes: 100               # Total training
      episodes
```

Listing 4: Q-learning agent configuration with inline comments

Deep RL agents (e.g. `DQN`, `PPO`, `A2C`) are implemented using the Stable-Baselines3 library [14]. For example, `DQN` can be configured as per Listing 5.

```
  name: "DQN_tuned"               # Identifier for
      the agent
  algorithm: "DQN"                # RL algorithm from
       SB3
  enabled: true                   # Include in
      training
  skip_learn: false               # Set true to only
      evaluate
  load: false                     # Load pretrained
      model
  save: true                      # Save model after
      training
  net_arch: [8, 8]                # Hidden layers
      architecture
  learning_rate: 0.001            # Learning rate for
       optimizer
  gamma: 0.1                      # Discount factor
      for future rewards
  exploration_fraction: 0.2   # Fraction of training
       with exploration
  exploration_initial_eps: 1.0  # Starting epsilon
      for exploration
  exploration_final_eps: 0.05   # Minimum epsilon
      value
  buffer_size: 10000              # Size of
      experience replay buffer
  batch_size: 1                   # Size of training
      batches
  target_update_interval: 80      # Target net update
       frequency
  learning_starts: 20             # Steps before
      training starts
  episodes: 100                   # Number of
      training episodes
```

Listing 5: DQN agent configuration with inline comments

This configuration structure enables fine-grained control over agent behavior and facilitates extensive comparative analysis between tabular and deep learning approaches under identical network conditions.

The next subsections detail the configuration specific to each use case, completing the common base one just presented.

3.1. Use case 1: Configuring a traffic classification experiment

This use case evaluates the ability of various learning algorithms to classify network traffic types based on real-time flow statistics. The RL environment exposes packet and byte metrics observed at the switch, and the agent must learn to classify the traffic correctly.

Environment setup. The classification scenario is activated by setting the following parameter in the configuration:

```
gym_type: classification
```

This setting activates the `NetworkEnvClassification` environment, where a single host is randomly selected at each step to generate one of four traffic types: ICMP (Ping), UDP, TCP, or no traffic. The environment structure and reward function are preconfigured to support traffic classification, with the learning objective focused on maximizing classification accuracy.

Action space. As only one host produces traffic at each timestep, the magnitude of observed statistics (packet count and byte count) is independent of the total number of hosts.

Agent setup. All enabled agents are trained independently. Each agent interacts with the environment over the configured number of episodes. At the end of training, each agent is evaluated on a new sequence of traffic episodes.

A baseline classifier (Decision Tree) is optionally included by enabling the supervised agent in the configuration. It does not learn during the simulation but is trained offline from a CSV dataset generated from prior runs. During evaluation, it predicts based on the current observation, enabling comparison with RL models.

Evaluation. At the end of the test phase, the simulator's Results Manager saves all agent outputs, including: accuracy, precision, recall, F1-score, confusion matrix, episode logs and per-step outputs in *json* format.

This use case demonstrates the simulator's capability to use both RL and supervised models under identical traffic conditions, ensuring fair and reproducible evaluations. While the scenario is straightforward, it provides a solid foundation for more complex extensions involving additional traffic types or patterns.

The system generates a *traffic.csv* file. This dataset captures information for each observed traffic flow, including: packet count, byte count, source host, destination host, and the type of traffic. The CSV file effectively combines the observed state with additional contextual information, providing a rich dataset for further analysis or training supervised baseline models. To provide insights into each experimental run and ensure reproducibility, the simulator organizes its output in a well-defined directory structure, as illustrated in Fig. 4. For every experiment, a dedicated directory is created, labeled with a timestamp indicating the start of the run and details of the network configuration (e.g., number of switches, hosts, and IoT devices). Within this top-level directory, users will find:

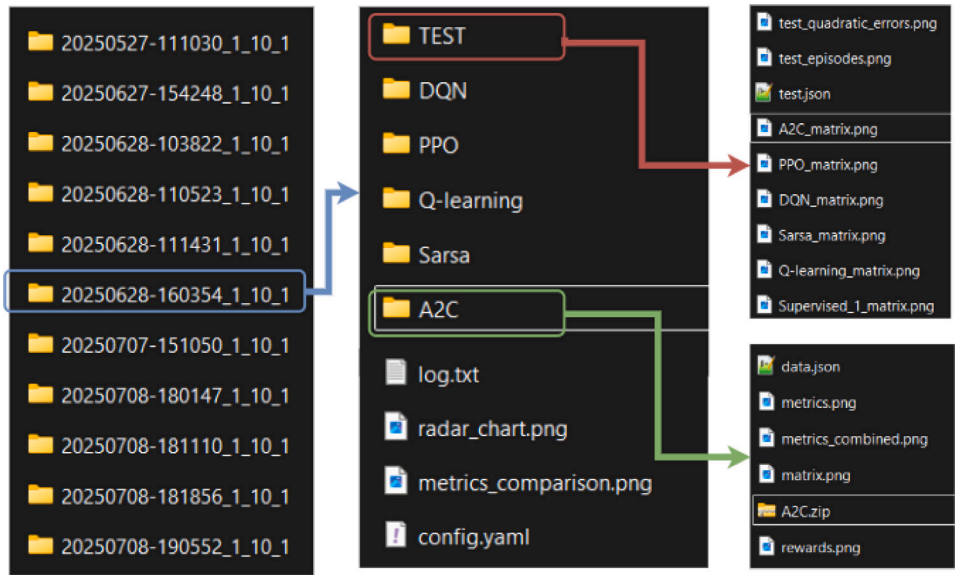


Fig. 4. Structure of the output results directory. Each experiment generates a time-stamped folder containing configuration files, logs, and subdirectories for individual agent performance and overall test results.

- *log.txt*: A log file capturing all events and messages during the entire execution.
- *config.yaml*: A copy of the configuration file used for that specific experiment.
- Two comparative charts: These visualize the training average accuracy, presented as a histogram and a radar plot, offering quick visual summaries of the agents' learning progression.

Furthermore, this main directory contains subdirectories for each agent configured for training and a dedicated *test* directory for overall evaluation results. Within each agent's dedicated directory, all registered training and evaluation data are stored (*data.json*), including detailed metrics, raw training data, and figures such as confusion matrices, plots of cumulative rewards, and two different plots of performance metrics. Additionally, the trained model's file is saved, allowing for its reloading and further analysis or deployment. The test directory contains all data registered during the prediction phase, including a confusion matrix for each evaluated agent, and two plots illustrating the predicted values (one displaying raw predictions and another showing them with an emphasis on quadratic error).

The various charts and plots generated are summarized in Table 3 for quick reference.

Table 3
Summary of plots in experiment output.

Plot name	Location	Purpose
Training Average Accuracy (Histogram)	Main Dir	Visualizes the distribution of average accuracy during training and compares the agents.
Training Average Accuracy (Radar Plot)	Main Dir	Provides a multi-axis visualization of average accuracy during training.
Confusion Matrices	Each Agent Dir	Illustrates the performance of the classification/detection (TP, TN, FP, FN).
Cumulative Rewards Plot	Each Agent Dir	Shows the agent's reward accumulation and learning progression over episodes.
Performance Metrics Plots (e.g., Accuracy)	Each Agent Dir	Visualizes trends of key performance metrics over training or evaluation.
Predicted Values (Raw)	Test Dir	Displays the agent's raw prediction outputs for evaluation.
Predicted Values (with Quadratic Error)	Test Dir	Visualizes predictions, with a focus on highlighting the magnitude of prediction errors.

3.2. Use case 2: Configuring an attack detection experiment

In this scenario, the simulator is used to detect malicious traffic, Denial of Service (DoS) attacks. Unlike the traffic classification case (Use Case 1), here all hosts in the network continuously generate traffic, and one or more may launch short or long duration DoS attacks at any given time. The adversarial traffic generator operates independently from the simulator step count.

Environment setup. The environment is selected via the YAML setting:

```
gym_type: attacks
```

This activates the `NetworkEnvAttackDetect` environment, while the network topology remains the same. Moreover, this case needs the configuration of threshold in net params:

```
packets: 1000   #packets to determine an attack
var_packets: 50 #percentage packet variation to
    determine an init or end attack
bytes: 1000000  #bytes to determine an attack
var_bytes: 30   #percentage byte variation to
    determine an init or end attack
```

We adopted standard parameters commonly found in the literature, chosen to demonstrate the integration and execution capabilities of the framework rather than to fine-tune detection performance. While this paper does not focus on hyperparameter optimization, it is important to note that the framework supports automatic parameter tuning through its modular configuration file system. This allows users to adjust detection thresholds, learning rates, discretization levels, and other relevant parameters to tailor experiments or perform systematic optimization in future studies.

Action space. During each step, every host may send randomly or normal traffic (Ping, TCP, UDP) or an attack (DoS) to one host based on a probability. This probability increases over time if no attack occurs and resets after an attack is triggered, ensuring homogeneous attack scenarios across episodes. The RL agent performs binary classification with 0 for **Normal traffic** and 1 for **Attack detected**. The observation space includes **packet count** and **byte count** per second and their **percentage variations** over the previous timestep. Observations are either **Discretized** for tabular agents (e.g., Q-learning, SARSA) or **Normalized** for deep RL agents (e.g., DQN, PPO, A2C). The low and high

values to discretize and normalize are directly related to the number of hosts. The reward function is:

- **TP** +1.0 detection of an attack.
- **TN** +0.5 identifying normal traffic.
- **FP** −0.5 for normal traffic flagged as attack.
- **FN** −1.0 for missed attacks.

Agent setup. The agent receives feedback according to the classification result. As in Use Case 1, each agent is initialized based on the configuration file, allowing parallel experimentation with different algorithms and hyperparameters.

Evaluation. After training, all agents are evaluated as in the other use case. As seen for Use Case 1, the system provides an output structure similar to that illustrated in Section 3.1, with the types of generated charts and plots summarized in Table 3. For the DoS attack detection scenario, the *statuses.json* dataset records specific flow information critical for analysis. This includes: packet count, byte count, variation in packet count (Δp), variation in byte count (Δb), source host, destination host, and the type of traffic (either 'normal' or 'DoS attack'). This detailed logging allows for in-depth post-analysis of attack patterns and agent responses.

This use case is proposed to validate the ability of RL models to detect threats under real-time multi-host network activity, simulating realistic security conditions.

3.3. Use case 3: Adding a new environment

One of the core design goals of *MininetGym* is ease of extensibility. This use case demonstrates how a developer or researcher can add a new custom RL environment to experiment with novel network defense strategies.

Unlike the previous use cases – which are ready to run with simple YAML configuration – this scenario requires a development phase. The user must implement a new environment class, define task-specific behavior (e.g., observation, reward, and action spaces), and extend the configuration file with a new `gym_type` value to make it selectable at runtime.

Steps to extend the simulator. To implement a new scenario (e.g., anomaly scoring, multi-flow correlation), the following steps are required:

1. **Define a new environment class** inheriting from `NetworkEnv` (e.g., `NetworkEnvAnomalyScore`).
2. Override the `calculate_reward()` method and, if necessary, the `step()` and `reset()` methods to adjust the episode logic.
3. Customize the **observation space** and **action space** in the `__init__()` constructor to reflect the new use case.
4. Add a new value under `gym_type` in the YAML config (e.g., `anomaly_scoring`).
5. Create a new launcher script (e.g., `anomaly_main.py`) that loads the environment, agents, and configuration.
6. Define a YAML configuration for the new scenario, reusing the structure from existing tasks.

Example YAML snippet. To activate the new environment, the following must be set in the configuration file:

```
gym_type: anomaly_scoring
```

This setting is parsed by the main script and used to instantiate the appropriate environment class.

Minimal code template. Listing 6 is a simplified structure for the new environment:

```
class NetworkEnvAnomalyScore(NetworkEnv):
    def __init__(self, config):
        super().__init__(config)
        self.action_space = gym.spaces.Discrete(3)
        self.observation_space = gym.spaces.Box(low
            =0, high=1, shape=(6,), dtype=np.
            float32)

    def calculate_reward(self, action, done):
        # Custom scoring logic
        ...
```

Listing 6: Minimal example of a new custom environment

Moreover, the developer can create a new function to generate traffic in the TG class.

This design allows users to quickly prototype and benchmark custom RL environments without modifying core simulator logic. By using the existing structure, new environments can be deployed.

4. Impact

The proposed simulator offers a flexible tool for research at the intersection of RL and cybersecurity. Its modular and extensible design allows researchers, students, and practitioners to experiment with a wide range of learning agents, traffic conditions, and network configurations, all within a controllable simulation environment.

Academic research. The simulator lowers the entry barrier for conducting RL-based experiments in network security, providing a standardized Gymnasium interface and ready-to-use environments. It facilitates benchmarking of RL algorithms in realistic environments, comparative analysis of tabular and deep RL methods, exploration of reward shaping and observation design strategies, and development of new environments for emerging use cases.

Practical testing. Security professionals and system designers can use the simulator to test RL-based intrusion detection and mitigation strategies in a setting, supporting real-time traffic analysis, SDN controllers, and realistic topologies.

Educational use. The simulator can also serve as a hands-on teaching tool in cybersecurity or RL courses. Its open-source nature allows students to understand the RL training pipeline from creation to evaluation, explore how traffic and attack patterns affect agent behavior, and customize and extend the simulator for project-based learning.

5. Conclusions and future work

This paper presented ***MininetGym***, a simulation framework that enables the application and evaluation of RL algorithms in SDN environments.

Researchers can use this platform for experimentation, algorithm comparison, and hyperparameter tuning, while practitioners can prototype and validate real-time security strategies in a controlled environment.

Future developments. We are continuously improving the simulator by extending the platform in several directions:

- **Multi-agent learning:** Incorporating cooperative or competitive RL agents that interact across different parts of the network.
- **Adaptive adversarial agents:** Replacing the current random traffic generator with an RL-based adversarial model that dynamically evolves to evade detection.
- **User interface improvements:** Enhancing the web-based dashboard with live visualization, interactive experiment configuration, and integration with experiment-tracking tools.

Overall, *MininetGym* lays the groundwork for a new class of experimentation tools in cybersecurity research and education.

Reusability and documentation

The *MininetGym* simulator is publicly available at [1]. It is released under an open-source license and includes extensive documentation, including a README file describing installation, configuration, and execution steps.

The modular architecture and adherence to the Gymnasium API make it easy to extend the simulator with custom environments, agents. Contributions are welcomed via GitHub pull requests or issue reports.

CRediT authorship contribution statement

Salvo Finistrella: Writing – review & editing, Writing – original draft, Software, Resources, Project administration, Investigation, Formal analysis, Data curation, Conceptualization. **Stefano Mariani:** Writing – review & editing, Supervision, Methodology, Conceptualization. **Franco Zambonelli:** Writing – review & editing, Supervision, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Finistrella S. Reinforcement learning for cybersecurity. 2025, https://github.com/.

[2] Finistrella S, Mariani S, Zambonelli F. Multi-agent reinforcement learning for cybersecurity: Classification and survey. Intell Syst Appl 2025;26:200495. http://dx.doi.org/10.1016/j.iswa.2025.200495.

[3] Anerousis N, Chemouil P, Lazar AA, Mihai N, Weinstein SB. The origin and evolution of open programmable networks and SDN. IEEE Commun Surv Tutor. 2021;23(3):1956–71.

[4] Issariyakul T, Hossain E. Introduction to network simulator 2 (NS2). In: Introduction to network simulator NS2. Boston, MA: Springer US; 2009, p. 1–18. http://dx.doi.org/10.1007/978-0-387-71760-9_2.

[5] Weingartner E, vom Lehn H, Wehrle K. A performance comparison of recent network simulators. In: 2009 IEEE international conference on communications. 2009, p. 1–5. http://dx.doi.org/10.1109/ICC.2009.5198657.

[6] Khraisat A, Gondal I, Vamplew P, Kamruzzaman J. Survey of intrusion detection systems: techniques, datasets and challenges. Cybersecurity 2019;2(1):1–22.

[7] Issa MM, Aljanabi M, Muhialdeen HM. Systematic literature review on intrusion detection systems: Research trends, algorithms, methods, datasets, and limitations. J Intell Syst 2024;33(1):20230248.

[8] Khattak ZK, Awais M, Iqbal A. Performance evaluation of OpenDaylight SDN controller. In: 2014 20th IEEE international conference on parallel and distributed systems. ICPADS, IEEE; 2014, p. 671–6.

[9] Drašar M, Moskal S, Yang S, Zat'ko P. Session-level adversary intent-driven cyberattack simulator. In: 2020 IEEE/ACM 24th international symposium on distributed simulation and real time applications (DS-rT). 2020, p. 1–9. http://dx.doi.org/10.1109/DS-RT50469.2020.9213690.

[10] Kara S, Hizal S, Zengin A, et al. Design and implementation of a DEVS-based cyber-attack simulator for cyber security. Int J Simul Model 2022;21(1):53–64.

[11] Gupta N, Maashi MS, Tanwar S, Badotra S, Aljebreen M, Bharany S. A comparative study of software defined networking controllers using mininet. Electronics 2022;11(17):2715.

[12] Pfaff B, Pettit J, Koponen T, Jackson E, Zhou A, Rajahalme J, Gross J, Wang A, Stringer J, Shelar P, et al. The design and implementation of open {vswitch}. In: 12th USeNIX symposium on networked systems design and implementation (NSDI 15). 2015, p. 117–30.

[13] Foundation F. Gymnasium. 2023, https://github.com/Farama-Foundation/Gymnasium. (Accessed July 2023).

[14] Raffin A, Hill A, Gleave A, Ernestus M, Dormann N, et al. Stable-Baselines3: Reliable reinforcement learning implementations. 2021, https://stable-baselines3.readthedocs.io/en/master/.(Accessed 16 May 2025).

[15] Limmen R. Awesome reinforcement learning for cybersecurity. 2024, https://github.com/Limmen/awesome-rl-for-cybersecurity. (Accessed 13 May 2025).

[16] Yeh C-T, Neema H, Balasubramanian D. Realistic and lightweight cyber agent training environment using network emulation in mininet. In: 2024 IEEE workshop on design automation for CPS and ioT. DESTION, 2024, p. 28–9. http://dx.doi.org/10.1109/DESTION62938.2024.00011.

[17] Team M. CyberBattleSim. 2021, https://Github.com/Microsoft/Cyberbattlesim.

[18] Samatar AF, Hirsi A, Omar AM, Abdiaziz MJ. Investigating ddos attack mitigation strategies and simulation tools using GNS3. In: 2024 FORTEI-international conference on electrical engineering (FORTEI-ICEE). 2024, p. 142–7. http://dx.doi.org/10.1109/FORTEI-ICEE64706.2024.10824613.