

# Deep reinforcement learning for autonomous and continual network intrusion detection

Hani Sami <sup>a,\*</sup>, Chamseddine Talhi <sup>a</sup>, Hakima Ould-Slimane <sup>b</sup>, Azzam Mourad <sup>c,d</sup>, Hadi Otrok <sup>e</sup>, Jamal Bentahar <sup>c,f</sup>

<sup>a</sup> Department of Software and IT Engineering, École de technologie supérieure, Montréal, QC, Canada

<sup>b</sup> Department of Mathematics and Computer Science, Université du Québec à Trois-Rivières, Trois-Rivières, QC, Canada

<sup>c</sup> KU 6G Research Center, Department of Computer Science, Khalifa University, Abu Dhabi, UAE

<sup>d</sup> Artificial Intelligence & Cyber Systems Research Center, Department of CSM, Lebanese American University, Beirut, Lebanon

<sup>e</sup> Center of Cyber-Physical Systems (C2PS), Department of Computer Science, Khalifa University, Abu Dhabi, UAE

<sup>f</sup> Concordia Institute for Information Systems Engineering, Concordia University, Montréal, QC, Canada

## ARTICLE INFO

### Keywords:

Network intrusion detection  
Continual learning  
Catastrophic forgetting  
Generative adversarial network  
Deep reinforcement learning  
Multi-head autoencoder

## ABSTRACT

With the rapid evolution of cyber threats, traditional Network Intrusion Detection Systems (NIDS) face major challenges in effectively identifying both novel attack and benign traffic patterns. Machine Learning (ML)-based NIDS often struggle to adapt to evolving threats due to insufficient volumes of unseen data and are prone to catastrophic forgetting, where updates to the model lead to the loss of previously acquired knowledge. To address these issues, prior research has explored the use of Generative Adversarial Networks (GANs) to augment data for training on new patterns, and Reinforcement Learning (RL) to support adaptive detection and updating. However, GAN-based approaches typically require manual threat labeling and repeated fine-tuning, while RL-based systems often suffer from long adaptation times. Consequently, both approaches fall short of enabling autonomous continual adaptation to evolving network traffic, including benign behavior and zero-day attacks. To overcome these limitations, we propose GARDIAN, a novel, autonomous, end-to-end NIDS architecture that performs continual learning (CL) over NID classifiers, enabling ongoing fine-tuning for traffic detection and classification. At its core, GARDIAN integrates a deep RL agent that guides the ML-based NID classifier through the CL process by deciding when to adapt and how much retraining data to use from different sources. Supporting this process, the first layer acts as an emerging-pattern detection and labeling stage, where clustering is used to group and label novel traffic patterns. The second layer employs a conditional GAN (CGAN) to generate CL-related data for the detected patterns, helping support balanced retraining without catastrophic forgetting. Extensive experiments on four widely used NIDS datasets, including ablation studies, demonstrate that GARDIAN improves detection accuracy in the presence of evolving and previously unseen traffic patterns and outperforms existing solutions.

The source code of our implementation is available on Github [https://github.com/hanisami/nids\\_continual\\_learning](https://github.com/hanisami/nids_continual_learning)

## 1. Introduction

Network traffic attacks are deliberate attempts to disrupt, damage, or gain unauthorized access to computer networks and data. These attacks can take various forms and continue to evolve as cybercriminals develop new techniques and adapt to security measures [1]. With the rapid proliferation of the internet and connected devices, the scale and complexity of network attacks have grown dramatically. According to [2], cybercrime is projected to cost approximately 10.5 trillion dollars in 2025, with around 2200 attacks each day, equivalent to one attack

every 39 . Moreover, statistics from [3] reveal that over 98 of web applications are vulnerable to attacks. These threats have evolved into complex patterns, each with unique characteristics and sub-patterns, which pose distinct detection challenges due to their varying behaviors and execution methods. As the number of network attacks increases and new variations are introduced daily, there is an urgent demand for powerful, proactive, and versatile cybersecurity solutions that integrate advanced technologies—this demand forms the central motivation for this article.

Network Intrusion Detection Systems (NIDS) are key cybersecurity tools designed to detect malicious activity over the network,

\* Corresponding author.

E-mail addresses: [hani.sami@etsmtl.ca](mailto:hani.sami@etsmtl.ca) (H. Sami), [chamseddine.talhi@etsmtl.ca](mailto:chamseddine.talhi@etsmtl.ca) (C. Talhi), [hakima.ould-slimane@uqtr.ca](mailto:hakima.ould-slimane@uqtr.ca) (H. Ould-Slimane), [azzam.mourad@ku.ac.ae](mailto:azzam.mourad@ku.ac.ae) (A. Mourad), [hadi.otrok@ku.ac.ae](mailto:hadi.otrok@ku.ac.ae) (H. Otrok), [jamal.bentahar@ku.ac.ae](mailto:jamal.bentahar@ku.ac.ae) (J. Bentahar).

<https://doi.org/10.1016/j.jisa.2026.104594>

Available online 11 August 2026

2214-2126/© 2026 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

ideally adapting to continuously evolving attack patterns [4]. In NIDS, signature-based methods work well for known traffic patterns but fail to generalize to unseen behaviors [5]. In contrast, Machine Learning (ML)-based NIDS have gained popularity for their ability to generalize across diverse attack patterns and automatically extract features from traffic samples via deep learning layers [6,7]. However, ML-based NIDS still face several notable limitations.

*First*, when new traffic patterns emerge, the model may fail to detect them without additional fine-tuning. Fine-tuning is often impractical in real-world deployments, as it requires collecting new traffic samples and labeling them, tasks that must be automated because network traffic evolves rapidly and delays from human intervention can critically degrade detection performance. To address this, Alalmaie et al. [8] proposed an AutoEncoder-based system for detecting zero-day attacks, while [9] used an unsupervised KMeans clustering method to group related traffic patterns without large labeled datasets. While such approaches partially improve detection of unknown patterns, they do not offer a fully autonomous solution for timely model adaptation.

*Second*, even when new patterns are identified, fine-tuning requires sufficient training samples, which may not always be available. This has motivated research into synthetic data generation to overcome data scarcity. In particular, Ding et al. [10], Huang and Lei [11] developed Generative Adversarial Networks (GANs) to create additional samples for underrepresented classes, improving deep learning classifier robustness. Other works [12,13] applied GANs directly to intrusion detection by training the discriminator on normal traffic so that deviations could be flagged as anomalies. However, these models still require periodic fine-tuning and cannot autonomously determine when or how to update.

*Third*, frequent fine-tuning can cause catastrophic forgetting, where the model loses previously learned knowledge and fails to generalize [14]. Some studies have addressed this with continual learning (CL) mechanisms, often using memory replay buffers to reintroduce past samples during training [15–17]. While this reduces forgetting, these solutions remain limited as they do not autonomously trigger updates or decide the optimal balance between past and new data.

*Finally*, in a CL setting, maintaining an up-to-date model requires automating the fine-tuning process [18]. Reinforcement Learning (RL) has been explored for this purpose, with works such as [19,20] employing Deep RL (DRL) for anomaly detection, leveraging DRL's online learning capabilities for self-improvement. These approaches show promise in anomaly detection and classification but overlook DRL's long convergence time, which makes it unsuitable as a primary standalone NID solution.

In summary, none of the existing solutions provide a fully autonomous continual learning process for NIDS classifiers that minimizes reliance on human intervention for time-critical updates. A true solution would need to autonomously determine the optimal timing for updates and the appropriate type and volume of retraining data, while adapting dynamically to both malicious and benign traffic changes.

To address these gaps, we propose GARDIAN, an automated CL framework designed for detecting zero-day attacks and accurately classifying network traffic while overcoming the limitations of current solutions. GARDIAN employs a DRL agent to autonomously and intelligently manage the CL and fine-tuning process of the main NID classifier. At the first layer, GARDIAN includes an emerging pattern detection and labeling stage, in which a Multi-Head Autoencoder (MAE) identifies traffic samples that deviate from the reconstruction profile of previously seen benign and attack families. These candidate emerging patterns are then processed by the DBSCAN clustering algorithm [21] to group them into coherent clusters, after which a semi-supervised labeling mechanism assigns final labels to the clusters. This first layer therefore acts as an early-warning and preparation stage for continual adaptation under evolving traffic conditions, rather than as a formal concept drift detector.

Once the newly labeled patterns are identified, GARDIAN's second layer addresses the challenge of limited training samples. This layer integrates

a Conditional GAN (CGAN) [22] to generate synthetic samples of the new patterns, ensuring adequate data for fine-tuning. The DRL agent (operating within this second layer) monitors classifier performance and determines the optimal amount of synthetic and replayed data to use for effective CL. To prevent catastrophic forgetting, a memory buffer stores representative benign and malicious samples for replay during training.

Therefore, GARDIAN is an end-to-end, fully autonomous CL-based NIDS during runtime operation after the initial setup/calibration phase, and is composed of two tightly integrated layers: (1) emerging pattern detection and labeling via MAE, DBSCAN, and semi-supervised learning, and (2) classification and autonomous continual learning powered by CGAN, DRL, memory buffer, and the main NID classifier. The contributions of this work are as follows:

- We propose GARDIAN, a two-layer end-to-end NID framework that combines emerging pattern detection, labeling, classification, and autonomous continual learning.
- We integrate MAE, DBSCAN clustering, and semi-supervised learning for zero-day attack detection and emerging benign traffic recognition.
- We develop a CGAN for data generation to support DRL-driven fine-tuning without catastrophic forgetting.

Extensive simulations, including ablation studies, evaluate the effectiveness of each component in GARDIAN. Performance comparisons using the NetFlow v2 dataset [23], which aggregates four widely used NIDS benchmark datasets, show that GARDIAN achieves outstanding zero-day attack detection and fully automated continual learning, outperforming state-of-the-art NIDS solutions.

## 2. Related work

In this section, we study the existing literature for behavior-based NID, spanning the following areas: Autoencoders, GANs, RL, and continual learning for NID.

### 2.1. AutoEncoders for NID

The AutoEncoder architecture has been widely used for feature extraction as a preparation for a classifier training, which yields improved model performance [24]. The authors in [8] propose the use of an AutoEncoder to extract relevant information from the network traffic to be passed for classification, ensuring zero-day attack detection. The classifier is a combination of a CNN, multi-head attention, and bi-directional LSTM, ensuring the extraction of spatial and temporal dependencies from the stream of network traffic. Furthermore, in [25] the authors propose a stacked AutoEncoder architecture for reducing the feature space, which is later sent to a deep neural network architecture (DNN).

Besides, the AutoEncoder model has been used to perform the NID by studying the pattern of normal/benign behavior and detecting outliers by measuring the difference between the input and output [26,27]. This approach has shown promising results on existing benchmark datasets; however, there is no continual learning mechanism to support the evolution of network behaviors, addressing the need for fine-tuning.

In this work, we build a Multi-head AutoEncoder (MAE), which benefits from the original idea of an AutoEncoder in the context of NID to identify outliers. However, what GARDIAN's MAE does as well is not only train on normal data, but on all the traffic, including attack categories. Therefore, the developed MAE aims to increase the detection accuracy by building a separate head for each of the major attack classes, including benign traffic.

### 2.2. Generative adversarial networks for data generation

In this section, we focus on literature that uses GAN-based models for *data generation* in the context of network intrusion detection, since this is the role played by GAN in our proposed GARDIAN framework.

In [10], the authors propose a hybrid framework combining K-Nearest Neighbors (KNN) and GAN to address the class-imbalance problem in intrusion detection. KNN is used to undersample normal traffic and reduce redundancy, while GAN is used to oversample minority attack classes. A neural-network classifier is then trained on the balanced data. Experiments on KDD'99, UNSW-NB15, and CIC-IDS2017 show improved detection performance under imbalanced conditions.

Similarly, Huang and Lei [11] proposes IGAN-IDS, in which GAN is used to generate additional samples for underrepresented traffic classes before training a deep neural network classifier. Their architecture combines feature extraction, GAN-based data generation, and anomaly classification, and reports improved detection performance on benchmark intrusion-detection datasets.

In [28], GAN is used to augment the IDS training dataset with synthetic attack samples. The generated data is accepted only if it improves the trained model, which provides an explicit quality-control step for augmentation. In [29], GAN is combined with an AutoEncoder so that generated data is first transformed into a more compact feature representation before being passed to a neural-network classifier.

Overall, these works use GAN primarily for data generation to compensate for limited or imbalanced attack data and improve classifier performance. Inspired by this direction, we use a conditional GAN in GARDIAN to generate additional samples for newly detected traffic patterns. Unlike prior augmentation-based approaches, our framework integrates this generation module into an autonomous continual-learning loop supported by DRL and a replay buffer. We also note that related GAN variants have been explored in adjacent settings, including PackCGAN for encrypted traffic classification [30] and ACGAN-based resampling for class imbalance in NIDS [31]. However, these works are only partially aligned with our scope, since our focus is not encrypted traffic classification or standalone imbalance handling, but the use of conditional generation within an autonomous continual-learning framework for evolving network intrusion detection.

### 2.3. Reinforcement learning for NID

There exist several efforts that utilize RL to build network intrusion detection solutions. We investigate the most recent efforts in this section.

The authors in [32] propose a novel approach to intrusion detection, aiming for fast and robust algorithms capable of identifying and classifying dangerous network traffic and evolving threats. The proposed algorithm combines a simple neural network classifier with an RL model, wherein the environment's behavior is actively adjusted during the learning process. Unlike traditional supervised learning paradigms, this approach integrates RL with a simulated environment, mimicking live interaction. The simulated environment generates samples from a training dataset and adjusts rewards based on classifier predictions. Additionally, an adversarial strategy is employed to address training bias and mitigate classification errors for underrepresented classes. Furthermore, in [33], the authors propose a system that employs a context-adaptive IDS utilizing multiple independent deep reinforcement learning agents distributed across the network. These agents aim to accurately detect and classify new and complex attacks while maintaining a balance between accuracy and false positive rates (FPR). Extensive experimentation using benchmark datasets, including NSL-KDD, UNSW-NB15, and AWID, demonstrate superior accuracy and lower FPR compared to existing systems.

The existing RL solutions for intrusion detection classification suffer from several problems. The main one is the long time required to adapt to environmental changes, such as the introduction of new attack patterns. Therefore, our proposed framework uses RL as the main enabler of CL, where it controls the fine-tuning process of an NID classifier. Most importantly, RL does not have to perform crucial update to its models, because its task is support the CL process rather than perform NID.

### 2.4. Continual learning for NID

Catastrophic forgetting occurs when the model forgets previously learned information as it learns new data. This is problematic especially in the context of NID where retaining all or even a small part of data simultaneously is impractical. To overcome this issue, there exist several related work that propose Continual Learning (CL) approaches combined with Data Shift detector to continually improve the detection model.

The work in [34] investigates the challenge of catastrophic forgetting in Deep Learning-based IDSS, where the model forgets previously learned information when trained on newer data points, driven by covariate shift in large-scale systems. To address this, an eight-stage framework is proposed for analyzing and quantifying data distribution drift. The study explores CL models as a solution, focusing on learning without forgetting, experience replay, and dark experience replay approaches. Through experiments on NSL-KDD and CICIDS 2017 datasets, the CL models demonstrate improved accuracy and lower false positive rates compared to traditional methods, effectively adapting to evolving data patterns while retaining previous knowledge. The limitations of conventional statistical and non-gradient-based machine learning approaches in handling covariate shift are also discussed. The work in [35] extends a CL approach to mitigate catastrophic forgetting in Variational AutoEncoder (VAE)-based anomaly detection. Specifically, the method involves continually expanding the definition of normal data without requiring all previously seen data. Instead, it efficiently generates a set of replay data using a generator, allowing the VAE to adapt to new data while retaining knowledge of past normal patterns.

Henceforth, the CL process requires autonomy, which is the main objective of our proposed GARDIAN framework. To the best of our knowledge, no existing work fully addresses the challenge of building a fully autonomous and self-adaptive CL process to support the existing NID classifiers.

## 3. Proposed GARDIAN architecture

In this section, we present the proposed GARDIAN architecture. The objective of the proposed architecture is to offer an intelligent self-adaptive system that enables autonomous fine tuning of the NID classifier. GARDIAN is composed of two detection layers: Layer 1 (presented in Fig. 1) performs emerging pattern detection and labeling; Layer 2 (presented in Fig. 2) offers autonomous continual learning powered by DRL. The first layer also acts as an assurance layer that monitors incoming traffic and flags samples whose reconstruction behavior deviates from that of previously seen traffic families. Importantly, this detection and screening mechanism remains active while the DRL agent is refining the retraining policy of the second layer, so emerging or zero-day-like traffic can still be flagged during the learning phase. Once detected, the next step in the first layer is to assign labels to the emerging patterns. These patterns are then batched to form a fine-tuning round for the NID classifier. Once this batch is ready, the DRL agent triggers the CL process to fine-tune the classifier adequately. The second layer in GARDIAN is mainly composed of DRL, but also utilizes a CGAN model, which generates enough samples from the data of the new batch, making sure that the fine-tuning process is effective. More details about each of these layers and their components are presented in the remainder of this section.

### 3.1. Layer 1: Emerging traffic detection and labeling

The first layer in GARDIAN for NID is presented in Fig. 1. The purpose of this layer is to detect and label emerging traffic patterns, including previously unseen attack or benign behaviors, and to prepare these samples for the autonomous continual learning (CL) process of the second layer. This layer is formed by a Multi-Head AutoEncoder (MAE) and a semi-supervised clustering mechanism. The MAE is initially trained on known traffic, including both benign and

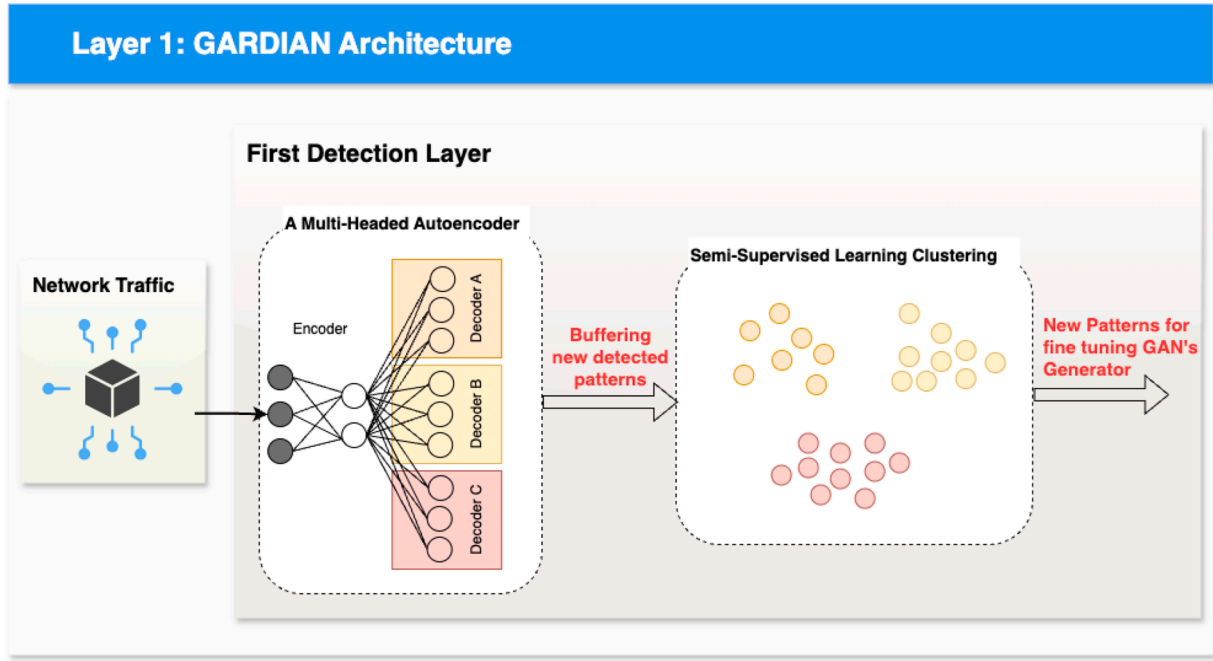


Fig. 1. The first layer of GARDIAN: emerging pattern detection and labeling. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

attack samples. Once traffic patterns that deviate from the learned reconstruction behavior are observed, the MAE flags them as candidate emerging patterns through reconstruction error analysis. Afterward, the semi-supervised clustering mechanism helps in labeling these patterns. Once identified, the new samples are sent to the second layer of the GARDIAN architecture to support fine tuning of the main classifier for CL.

It is important to distinguish emerging pattern detection from formal concept drift detection. In general, concept drift refers to changes over time in the underlying data distribution and/or in the relationship between features and labels. In contrast, the proposed MAE does not explicitly estimate temporal drift or identify which statistical component has changed. Instead, it detects samples that are poorly reconstructed relative to the traffic families observed during training. Therefore, Layer 1 should be interpreted as an early-warning and preparation stage for adaptation under evolving traffic conditions, rather than as a standalone concept drift detector.

In this section, we detail the mechanism behind the MAE and semi-supervised clustering mechanism.

### 3.1.1. Multi-head AutoEncoder (MAE)

The MAE is a specific neural network architecture that has the same size/number of neurons for the input and output layers. An MAE has one input and multiple output layers, where the input layer has all the features corresponding to the network packets. The number of hidden layers is fine tuned to get the best performance. An example of the MAE is shown in Fig. 1 under "A Multi-Headed AutoEncoder" component name. As the figure shows, the MAE has one encoder and multiple decoders. The number of decoders resembles the number of attack families covered by the GARDIAN solution, in addition to one class for normal traffic. As an example, GARDIAN can be responsible for all the variations of the Denial of Service (DoS), infiltration, botnet, web and brute-force attacks. In this case, the MAE requires six decoders, one for normal traffic and the rest for each of the five attack families. Having different decoders makes the MAE more robust to identifying new patterns, as each decoder is focused on its traffic family.

In our implementation of the MAE, we have utilized the Mean Squared Error (MSE). In this case, the total loss is computed as:

$$\text{Loss}_i = \frac{1}{n} \sum_{j=1}^n (x_j - \hat{x}_{ij})^2 \quad \forall i \in \{1, \dots, k\} \quad (1)$$

where  $x_j$  is the  $j^{\text{th}}$  element of the input vector  $x$ . In addition,  $\hat{x}_{ij}$  represent the  $j^{\text{th}}$  element of the vector  $\hat{x}_i$ .

When new traffic comes in, the MAE performs predictions on these samples to obtain  $\hat{x}_i$  for every decoder. To identify emerging patterns relative to previously seen traffic, the reconstruction error between the original input and the decoded output is computed. The reconstruction error for each decoder measures how well the MAE reconstructs the input. MSE is used as the reconstruction error  $RE$ :

$$RE_i = (MSE)_i = \frac{1}{n} \sum_{j=1}^n (x_j - \hat{x}_{ij})^2 \quad (2)$$

Hence, a new pattern is identified if  $RE \geq \text{th}$ , where:

$$RE = \frac{1}{k} \sum_{i=1}^k \text{Loss}_i \quad (3)$$

In the example of three decoders, the combined reconstruction error  $RE$  is compared against a preset threshold  $\text{th}$ . This threshold is determined empirically during the initial calibration phase based on the reconstruction errors of known patterns. Once set, it is applied automatically during deployment as part of the emerging-pattern detection pipeline. If the averaged reconstruction error for the different decoders is greater than or equal to the threshold, the network traffic is classified as anomaly. Otherwise, the combined/averaged reconstruction error for all decoders would be less than the threshold, then this pattern would be identified as known. A pseudo-code for building and training the MAE with multiple decoders for new pattern detection is presented in Algorithm 1.

The emerging patterns are collected for a specified period of time or traffic volume and are then forwarded automatically to the semi-supervised clustering algorithm. An administrator may optionally inter-

---

**Algorithm 1** Autoencoder with multiple decoders for new pattern detection.

---

**Step 1: Define the model architecture**

- 1: Create an input layer for the data.
  - 2: Build a shared encoder:
  - 3: Add layers that reduce the input data's dimensions to create a compressed representation.
  - Step 2: Create separate decoders for each data type:**
  - 4: **for** each decoder (corresponding to normal data and different attack types) **do**
  - 5: Add layers that reconstruct the input data from the compressed representation.
  - 6: **end for**
  - 7: Combine the input layer, shared encoder, and decoders into an MAE model.
  - 8: Compile the MAE model using an appropriate optimizer and loss function.
  - Step 3: Train the MAE model**
  - 9: **for** each type of data (normal and each attack type) **do**
  - 10: Train the model where only the corresponding decoder learns to reconstruct its specific data type.
  - 11: Provide dummy data to other decoders to ensure they do not learn.
  - 12: **end for**
  - 13: After training, the model is ready to detect new patterns based on reconstruction errors.
- 

vene for supervisory purposes, but such intervention is not required in the standard operating flow of GARDIAN.

### 3.1.2. DBscan semi-supervised learning

The detected emerging traffic samples  $x_{new}$  are encoded by the encoder layer of the MAE (the layer before the decoders) and then passed for clustering, performed using the DBSCAN algorithm. Therefore,  $m_{new} = f_{\theta}(x_{new})$ . The number of clusters is determined automatically by DBSCAN after fine-tuning the hyper-parameters of epsilon and the minimum number of samples.

Once clustering is done, each of the  $m_{new,j}$  (the original representation of each pattern detected) is assigned to a cluster. Afterward, labeling each cluster is necessary to identify to which class (normal or attack type) the sample belongs to. For this reason, a semi-supervised learning mechanism is used. First, the encoded-labeled samples  $m$  from the MAE is obtained. These samples are the originally labeled ones used for training the MAE. Second, the nearest labeled neighbor is identified for each of the new patterns to get them labeled.

The label is assigned for the new pattern after identifying their nearest labeled-encoded samples. The label is decided based on the highest number of nearby labeled samples with the same label. In [Algorithm 2](#), the clustering mechanism for detecting and labeling new patterns is presented.

### 3.2. Layer 2: Autonomous continual learning

The purpose of this layer is to offer autonomous CL mechanism, managed by an intelligent DRL agent and supported by other components, including a conditional GAN (CGAN) for synthetic data generation, and a replay buffer to address the problem of catastrophic forgetting. The architecture of this layer is depicted in [Fig. 2](#).

Per one of the main objectives of the GARDIAN framework, continual learning is achieved autonomously. As per [Fig. 2](#), the NID classifier receives commands from the DRL agent to perform fine-tuning. Initially, the classifier is trained using a labeled dataset that represents known patterns (both normal and attack classes). This is achievable using a typical supervised learning approach, optimizing for classification accuracy and generalization.

---

**Algorithm 2** Clustering mechanism for detecting and labeling new patterns.

---

- 1: **Input:** MAE model, Labeled data  $x, y$ , Unlabeled data  $x_{new}$ , Clustering threshold  $th$ , Number of clusters  $k$
  - 2: **Output:** Labeled clusters for new patterns
  - Step 1: Identify high-error samples**
  - 3: Pass  $x_{new}$  through the MAE to obtain reconstructed outputs  $\hat{x}_{new}$
  - 4: Calculate reconstruction errors  $RE$  for each sample
  - 5: Select samples with reconstruction error above  $th$ , storing them as  $x_{new}$
  - Step 2: Encode high-error samples**
  - 6: Pass  $x_{new}$  through the encoder part of the MAE to obtain encoded representations  $m_{new}$
  - Step 3: Encode labeled data**
  - 7: Pass  $x$  through the encoder to obtain encoded representations  $m$
  - Step 4: Apply DBscan clustering**
  - 8: Combine  $m_{new}$  and  $m$  into a single dataset  $m_{combined}$
  - 9: Apply DBscan clustering on  $m_{combined}$
  - Step 5: Label clusters based on existing data**
  - 10: For each cluster, assign a label by the highest number of known labels  $y$  of  $m$  in the cluster
  - 11: Use the cluster labels to label  $x_{new}$  with their corresponding cluster label
  - 12: **Return:** Labeled clusters for new patterns
- 

As new patterns are detected by the MAE and clustering components, they are passed to the CGAN for fine-tuning. At the same time, the buffer maintains a representative set of old patterns, ensuring that the model does not forget previously learned information while learning new patterns. The data generated by CGAN's generator is combined with enough samples from the memory buffer to properly fine-tune the classifier. The DRL model decides on the amount of data to send to the classifier per class from both the CGAN generator and the buffer. Afterward, the NID classifier is triggered for fine tuning. By combining old valuable data from the memory buffer, catastrophic forgetting is avoided, while fine-tuning is performed autonomously through DRL.

#### 3.2.1. Conditional generative adversarial network

To improve the base classifier for NID, it is important to prepare a dataset containing enough examples of new patterns. The source of these new patterns is DBscan clustering from the first layer of detection. The GARDIAN framework accepts and processes traffic in real-time. Since not enough data would be available for fine-tuning the base classifier, we leverage a conditional Generative Adversarial Network (CGAN) to cover this gap and generate synthetic samples similar to the newly detected patterns.

A CGAN consists of two neural networks, a discriminator and a generator. Initially, the discriminator receives real data, in this case traffic samples from the real flow. After the initial training, the discriminator considers the data detected from the first layer of GARDIAN, combined with real traffic flow to perform fine-tuning of the CGAN. In addition, synthetic data generated by the generator model is also sent to the discriminator as depicted in [Fig. 2](#). The task of the discriminator is to assess and predict if the sample received is real or fake. On the other hand, the generator takes a noise vector and a label, and attempts to generate synthetic data for that label. Besides, the generator attempts to get the generated data as close as possible to the real ones by trying to confuse the discriminator that this data is real. To this end, the discriminator penalizes the generator for producing implausible results using the loss function, which gets back-propagated into the generator to update its behavior.

A known challenge of GAN-based generation is mode collapse, where the generator produces only a limited subset of the target distribution rather than capturing its full variability. In addition, adversarial training can be computationally expensive, and loss curves alone are insufficient

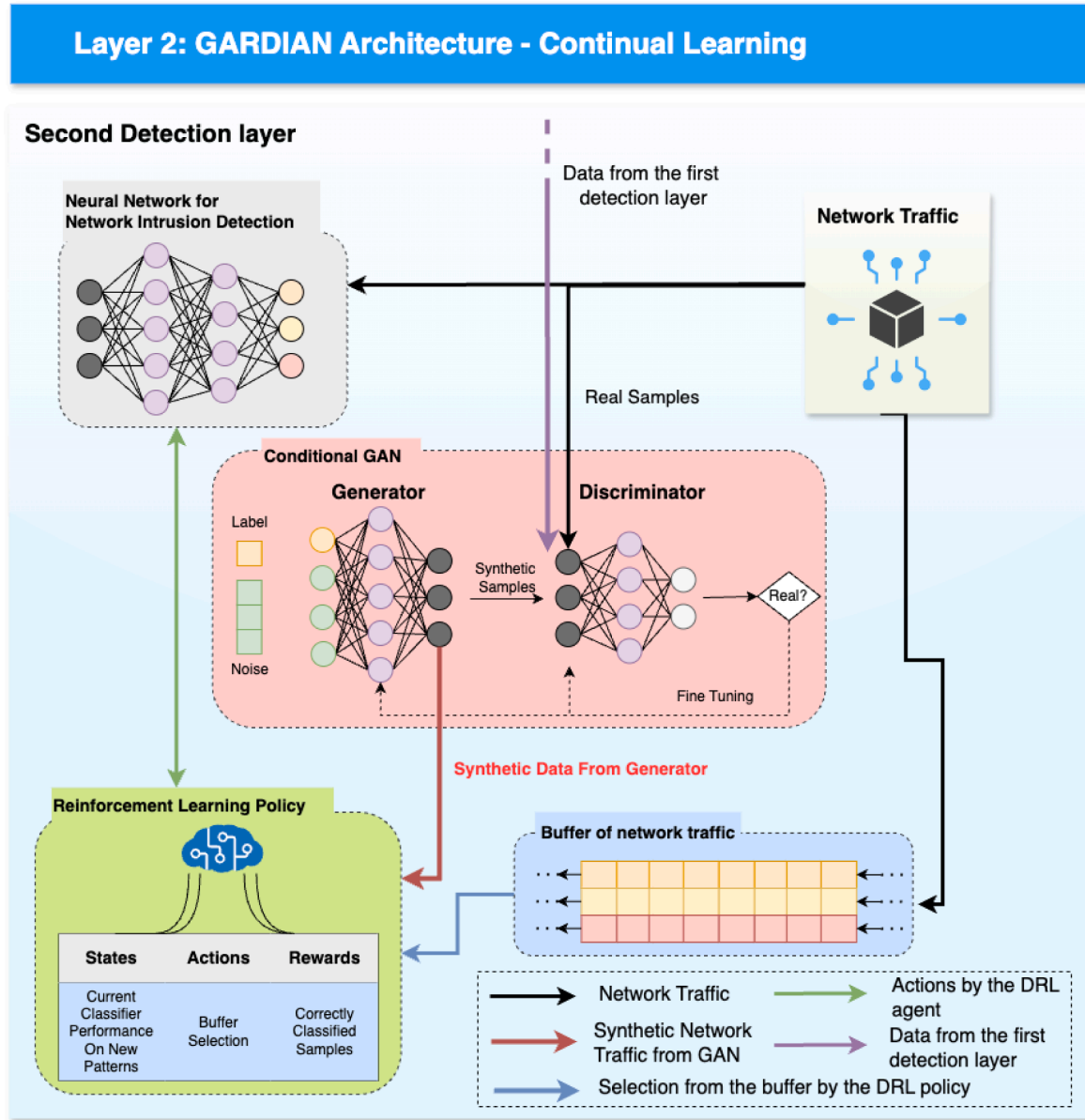


Fig. 2. The second layer of GARDIAN: automated continual learning.

to assess the quality of the generated samples. For these reasons, the CGAN in GARDIAN is not assumed to be inherently stable or immune to collapse. Instead, we explicitly evaluate the diversity, realism, and runtime cost of the generated samples before drawing conclusions about the usefulness of CGAN for continual learning. In particular, the evaluation considers a diversity proxy based on pairwise distances, generated-to-real nearest-neighbor distance, feature-statistics matching, and the measured time required for CGAN fine-tuning and sample generation.

Ultimately, we use the generator's data as an auxiliary source of synthetic samples to support fine-tuning of the NID classifier, provided that the generated data exhibits acceptable diversity and similarity to the real target-class samples. More details about the inner workings of CGAN can be found in the original paper [22].

### 3.2.2. Memory buffer

Passing only new patterns of data to the base classifier for fine tuning (using data from CGAN) would result in the catastrophic forgetting issue. To avoid this problem, we are proposing the use of a memory buffer, which stores a representative subset of old traffic from different fami-

lies selected using random sampling. Thus, the memory buffer consists of a two-dimensional queue, where each row represents a traffic class of fixed size for simplicity. During the classifier fine tuning, samples from the old patterns stored in the memory buffer is combined with the new ones generated by the CGAN. This procedure helps retain knowledge of old patterns while learning new ones.

Since a two-dimensional queue structure is used to represent the memory buffer, once new items arrive for a certain traffic class, old ones are removed. The size of this buffer is a hyper-parameter, which can be adjusted based on the GARDIAN performance. Identifying old patterns and updating the replay buffer is done by monitoring the performance of the classifier. If the reconstruction error is low and the accuracy confidence is high, then the base model has learned this pattern in its inner structure. Duplicates are avoided in the Memory Buffer to ensure that there is enough space for representing different old patterns.

### 3.2.3. Deep reinforcement learning (DRL) agent

The first layer of detection in GARDIAN identifies emerging traffic patterns using the MAE followed by clustering. In this work, autonomy

is defined operationally as the ability of GARDIAN to perform retraining decisions online without human intervention after the initial model setup/calibration phase, including deciding when to adapt and how to compose the retraining data from CGAN and replay-buffer sources. Importantly, the DRL agent is not responsible for immediate traffic screening; emerging or zero-day-like traffic continues to be flagged by the first detection layer and processed by the current classifier while the DRL policy is being refined. Accordingly, the quantitative evaluation of the DRL component later reports the number of automated policy decisions, the runtime cost of PPO-based adaptation, and the retraining amount and source allocation selected by the policy over time. In the second layer, a CGAN is fine-tuned and used to generate additional data for fine-tuning the NID classifier. The challenge is to determine when the classifier's performance degrades, especially when new patterns belonging to an existing family are detected, and then decide the optimal number of samples needed to fine-tune the classifier effectively, without compromising the performance on older patterns. This is achieved through our novel formulation of the RL environment. The following are the objectives of the RL agent: (1) decide when to initiate fine-tuning based on the classifier's performance over a certain period, (2) determine the amount of retraining data needed from the memory buffer and newly generated CGAN samples, and (3) allocate retraining samples across traffic families so that the classifier adapts to new patterns without forgetting old ones.

#### Markov decision process (MDP) formulation

A DRL agent algorithm requires a predefined MDP environment, which requires defining the state space  $S$ , action space  $A$ , and reward function  $R$ . Starting with the state representation,  $S$  should capture the classifier's current performance, the status of the memory buffer, and the characteristics of the detected patterns. Specifically:

**Buffer status  $S_B$ :** A matrix or vector where each entry corresponds to a traffic type (class). Each row represents a class (e.g., normal traffic, attack type A, attack type B, etc.), and the value in each entry indicates the number of samples stored in the buffer for that class. Studying the buffer status is important to keep track of how balanced the buffer is in terms of the representation of different traffic types. It helps in deciding how many samples to draw from the buffer during fine-tuning.

**Classifier performance  $S_P$ :** A vector or set of values that capture the classifier's current performance metrics on the same data clustered in the first detection layer. These include: Accuracy, F1-Score, Confusion Matrix. Studying the classifier performance is important to inform the RL agent of the classifier's current state, helping it decide whether more samples are needed and how to adjust  $NA_{min}$  and  $NA_{max}$ . These two numbers represent the interval of number of samples to be extracted from (1) the data generated by CGAN and (2) the new patterns of clusters combined.

**New pattern statistics  $S_N$ :** Metrics related to the newly detected patterns, composed of the *number of new Patterns detected* and the *initial classification performance*. Representing these information in the state space is vital to determining the novelty and challenge posed by the new patterns. This helps the RL agent decide how aggressively to adjust sample sizes and focus on these new patterns during fine-tuning.

**Current  $NA_{min}$  and  $NA_{max}$ :** Two natural numbers which determine the exact number samples are being drawn from both the CGAN and buffer for each class. We present these numbers in each state to give the RL agent awareness of its previous actions, allowing it to decide whether to increase, decrease, or maintain these values based on the resulting classifier performance.

Therefore, the state space is represented as:

$$S = [S_B, S_P, S_N, NA_{min}, NA_{max}] \quad (4)$$

The action space  $A$  is a set of possible actions that the RL agent can take to adjust the system. These actions primarily involve adjusting the number of samples generated for each class and deciding on the

selection of samples between the CGAN and the buffer. The action space is composed of  $A_S$  and  $A_A$ :

**Adjust samples sizes  $A_S$ :** A vector of two elements to decrease or increase  $NA_{min}$  and  $NA_{max}$ . Therefore  $A_S = [\Delta NA_{min}, \Delta NA_{max}]$ , where  $\Delta NA$  represents the change in sample size.  $NA_{min}$  can start from zero, while  $NA_{max}$  is constraint by the buffer size.

**Allocate samples per class  $A_A$ :** A vector containing  $2 \times k$ , where  $k$  resembles the number of clusters. For each element in this vector, the action of the agent indicates how many samples is either generated from CGAN or selected from the Memory Buffer, while respecting the  $NA_{min}$  and  $NA_{max}$  ranges. Hence,  $A_A = [\Delta NA_{CGAN, normal}, \Delta NA_{Buffer, normal}, \dots, \Delta NA_{CGAN, attack B}, \Delta NA_{Buffer, attack B}]$

Therefore the action space is represented as:

$$A_t = [A_S, A_A] \quad (5)$$

The reward function  $R_t$  of this MDP environment aims to optimize the selection of data and fine-tuning of the base classifier. This reward is composed of two objective functions, including the improvement of the classifier performance, while making sure that the number of samples selected from the Memory Buffer and CGAN is efficient.

The objective of the **Classifier improvement reward  $R_P$**  is to measure the improvement in the classifier's performance after fine-tuning with the newly generated samples. The goal is to reward actions that lead to better performance in terms of accuracy. Formally, this objective is represented as:

$$R_P = \sum_{i=1}^k \lambda_i \Delta Acc_i \quad (6)$$

where  $k$  represents the number of classes including normal and attack types. Furthermore,  $\lambda_i$  is a weight that determines the importance of improving the performance of class  $i$ , giving more focus on improving the detection of some families. In addition,  $\Delta Acc_i$  corresponds to the accuracy score for a given class instance before and after the fine-tuning. Hence,  $\Delta Acc_i = Acc_i^{post-finetune} - Acc_i^{pre-finetune}$

The second objective corresponds to optimizing the **Sample efficiency reward  $R_E$** , which penalizes generating too many unnecessary samples. Optimizing the number of instances to be used for fine-tuning makes sure that fine-tuning finishes faster, which is crucial in NID. Therefore,  $R_E$  is computed as follows:

$$R_E = -\beta \sum_{i=1}^k (NA_{CGAN,i} + NA_{buff,i}) \times (1 - \Delta Acc_i) \quad (7)$$

where  $NA_{CGAN,i}$  is the normalized number of samples generated by CGAN for class  $i$ . Additionally,  $NA_{buff,i}$  is the normalized number of samples selected from the memory buffer for class  $i$ . Moreover,  $\beta$  is a penalty factor that increases as more unnecessary samples are generated. Besides,  $1 - \Delta Acc_i$  ensures that penalties are higher if the generated samples do not lead to significant performance improvements.

Therefore, the reward function  $R_t$  is represented as:

$$R_t = R_P + R_E \quad (8)$$

In summary, the RL agent is rewarded for improving performance accuracy and sample generation by penalizing for inefficiency, which are key drivers for effective and continual learning in this context. Since the reward depends on the class-weighting vector  $\lambda_i$  and the efficiency penalty factor  $\beta$ , we later evaluate how sensitive the learned continual-learning behavior is to these parameters under a representative adaptation scenario.

#### DRL algorithm

In this section, we present a pseudo-code of the Proximal Policy Optimization (PPO) algorithm [36] used for building the Deep RL (DRL) agent of GARDIAN. We are resorting to build a DRL solution because the state and action spaces are too large, requiring a non-linear approximation for the policy. PPO is selected among other DRL solutions because

it offers a stable balance between exploration and exploitation, making it well-suited for GARDIAN's continual learning needs in dynamic network environments where other algorithms may suffer from instability or poor sample efficiency. A copy of the pseudocode is presented in Algorithm [Algorithm 3](#).

**Algorithm 3** PPO for continual learning with CGAN and classifier fine-tuning.

**Require:** Initialize policy parameters  $\theta$ , value function parameters  $\phi$   
**Require:** Set clipping parameter  $\epsilon$ , learning rate  $\alpha$ , discount factor  $\gamma$ , and number of epochs  $K$

**Require:** Initialize experience buffer  $D = \emptyset$

```

1: Initialize environment state  $s_0$  using clustering results and classifier
   performance metrics.
2: for iteration = 1, 2, ..., M do
3:   for t = 1, 2, ..., T do
4:     Sample action  $a_t \sim \pi_\theta(a_t|s_t)$  based on current policy.
5:     Execute action  $a_t$  in the environment, receive next state  $s_{t+1}$  and
       reward  $r_t$ .
6:     Store transition  $(s_t, a_t, r_t, s_{t+1})$  in buffer  $D$ .
7:     Update environment: adjust memory buffer and CGAN model
       data according to action  $a_t$ .
8:   end for
9:   for epoch = 1, 2, ..., K do
10:    Shuffle the buffer  $D$ .
11:    for each minibatch  $B \subseteq D$  do
12:      Compute advantage estimates  $\hat{A}_t$  using generalized advantage
       estimation (GAE):

```

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l}$$

```

13:      Compute the clipped surrogate objective:

```

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[ \min \left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t, \right. \right. \\ \left. \left. \text{clip} \left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1 - \epsilon, 1 + \epsilon \right) \hat{A}_t \right) \right]$$

```

14:      Update policy parameters  $\theta$  by maximizing  $L^{CLIP}(\theta)$  using
       stochastic gradient ascent.

```

```

15:      Update value function parameters  $\phi$  by minimizing the value
       loss:

```

$$L^{VF}(\phi) = \mathbb{E}_t \left[ (V_\phi(s_t) - R_t)^2 \right]$$

```

16:      end for

```

```

17:      end for

```

```

18:      Reset experience buffer  $D = \emptyset$ .

```

```

19:      Adjust the clipping parameter  $\epsilon$  or learning rates  $\alpha$  if needed.

```

```

20: end for

```

### 3.2.4. NID classifier

The NID classifier operates as a deep neural network (DNN) designed to classify network traffic into multiple classes (e.g., normal and various attack types/families). Based on the utilized DNN architecture, the model is capable of reflecting both old and new patterns while avoiding catastrophic forgetting through GARDIAN's autonomous continual learning mechanism. The emerging pattern detection decisions of the first layer's MAE and the classifier should be jointly considered to make sure that the traffic is benign or malicious. In case the classifier detects traffic as malicious while the MAE identifies it as normal, the traffic is labeled as malicious. On the other hand, if the MAE detects the traffic as malicious while the classifier detects it as normal, the traffic is labeled as malicious and is passed for clustering. To this end, the classifier is

**Table 1**

Distribution of benign, DoS, DDoS, and reconnaissance samples in NetFlow-based datasets.

| Dataset            | Benign     | DoS (D)   | DDoS (DD) | Reconnaissance (R) |
|--------------------|------------|-----------|-----------|--------------------|
| NF-UNSW-NB15-v2    | 2,237,731  | 5980      | –         | 17,074             |
| NF-ToN-IoT-v3      | 16,792,214 | 203,456   | 4,141,256 | 1,358,977          |
| NF-BoT-IoT         | 51,989     | 8,034,190 | 7,150,882 | 1,695,132          |
| NF-CSE-CIC-IDS2018 | 17,514,626 | 302,966   | 1,324,350 | –                  |

the main point of detection over the network, since it is able to handle multi-class classification and is flexible to handle continual learning.

## 4. Experimental study

In this section, we present the details about the conducted experimental study. First, we start with a description of the datasets, including the different traffic types we utilize and their distributions. Second, we present the experimental setup, development environment, and characteristics of the different components of GARDIAN. Next, we showcase an ablation while mapping some of the combination of components to existing state-of-the-art literature, showcasing GARDIAN's advantages. Finally, we show the advantage of using GARDIAN to achieve a fully autonomous continual learning system for network intrusion detection by conducting an studies over domain incremental and task incremental learning using four different datasets.

### 4.1. Datasets

In this study, we conduct our experiments using the NetFlow v2 datasets [23], which offers a unified data representation of four different NIDS datasets, making it easier to test GARDIAN under different continual learning scenarios. The four datasets include: CSE-CIC-IDS2018-v2 [37], BoT-IoT-v2 [38], ToN-IoT-v2 [39], and UNSW-NB15 [40].

The NetFlow-based version of multiple cybersecurity datasets have been developed to enhance attack detection and classification. The NF-UNSW-NB15-v2 dataset consists of 2,390,275 data flows, with 95,053 (3.98%) labeled as attacks. Among them, 5980 represent Denial of Service (DoS) attempts, while 17,074 are reconnaissance activities. NF-ToN-IoT-v3 is a larger dataset containing 27,520,260 flows, with 10,728,046 (38.98%) attacks. It includes 203,456 DoS samples, 4,141,256 Distributed Denial of Service (DDoS) attempts, and 1,358,977 reconnaissance cases. NF-BoT-IoT is an IoT-specific dataset with 16,993,808 flows, where 16,881,819 (99.7%) are attacks. DoS and DDoS attacks are heavily represented, with 8,034,190 and 7,150,882 samples, respectively, while reconnaissance activity accounts for 1,695,132 flows. Finally, NF-CSE-CIC-IDS2018 is built from CSE-CIC-IDS2018 pcap files, comprising 20,115,529 flows, with 2,600,903 (12.93%) attacks. This dataset contains 302,966 DoS, 1,324,350 DDoS, and 17,514,626 benign instances. These datasets collectively provide a diverse and comprehensive foundation for evaluating network intrusion detection techniques.

In these experiments, we are focusing on Benign, DoS, and Reconnaissance classes across all datasets, which are sufficient for the scope of this paper to illustrate the advantage of GARDIAN. [Table 1](#) summarizes the distribution of each class across the four datasets:

### 4.2. Setup and configurations

To properly evaluate the proposed GARDIAN framework, we have developed each of the components presented in [Section 3](#). The different components are developed using existing Machine Learning frameworks, including Scikit-Learn and TensorFlow.

#### Multi-head AutoEncoder (MAE)

**Encoder:** The encoder takes an input of a specified shape (specified traffic features), applies a dropout layer (with a rate of 0.5) to introduce

regularization, and then passes the data through a dense layer with 64 neurons using ReLU activation. This layer includes an L2 regularization term to prevent overfitting. A second dropout layer is applied, followed by another dense layer with 48 neurons and ReLU activation. This final layer outputs a shared encoded data.

**Decoder:** There are three separate decoders in our implementation for this dataset, one for each class, which take the shared encoded representation as input and attempt to reconstruct the input data for their respective class. Each decoder consists of a dense layer of 64 neurons and ReLU activation, followed by another dense layer that reconstructs the output with the same shape as the input.

The MAE is compiled with the Adam optimizer and a mean squared error loss function, indicating a focus on minimizing the reconstruction error. The training data is prepared for each class by isolating samples corresponding to specific classes and fitting the model using its decoders.

#### DBscan clustering

DBscan clustering is applied to the high-error samples (or samples with high reconstruction error) identified by the MAE. The number of clusters is identified automatically by the DBscan algorithm. After multiple trials, we identify the following parameters: epsilon = 0.9 and number of samples = 5.

#### Conditional generative adversarial network

The CGAN is designed with a specific architecture for both the generator and discriminator. The **generator** starts with an input layer that combines random noise (a latent vector of size 100) and a conditional label. This is followed by three hidden layers, each with 128 neurons, using the SPOCU activation function. Each hidden layer is fully connected and includes batch normalization to stabilize the learning process and improve convergence. The optimizer for the generator is Adam with a learning rate of 0.0002. The output layer of the generator uses a linear activation function to produce synthetic data samples conditioned on the labels.

The **discriminator** consists of an input layer that takes both real and generated samples along with their labels. It has three hidden layers with 256, 128, and 64 neurons, respectively, each using LeakyReLU activation with a slope of 0.2 to handle negative inputs. The discriminator incorporates a dropout layer after each hidden layer with a ratio of 0.3. The optimizer for the discriminator is also Adam, with a learning rate of 0.0002. The output layer of the discriminator has a single neuron with a sigmoid activation to output the probability of the input being real or fake.

We also evaluate the stability of CGAN training using explicit diagnostics rather than relying only on adversarial losses. In particular, we assess: (1) a diversity proxy based on the ratio between the average pairwise distance of generated samples and that of real target-class samples, (2) sample realism using feature-statistics matching and generated-to-real nearest-neighbor distance, and (3) computational overhead using the CGAN fine-tuning time and generation time. These diagnostics are used to determine whether a given training configuration is prone to collapse and whether the synthetic samples are suitable for supporting continual learning. They also allow us to quantify how sensitive CGAN behavior is to the training configuration, rather than assuming stable generation by design.

#### DRL agent

The MDP environment presented in Section 3.2.3 is implemented using the OpenAI Gym library [41], which makes it straightforward to implement and test various RL and DRL algorithms. The PPO algorithm is implemented and tested on the developed Gym environment using the Stable-baselines library [42]. The following parameters are used for PPO:  $\gamma = 0.99$ , learning rate of 0.0003 and 0.001 for actor and critic respectively, and a clip ratio of 0.2.

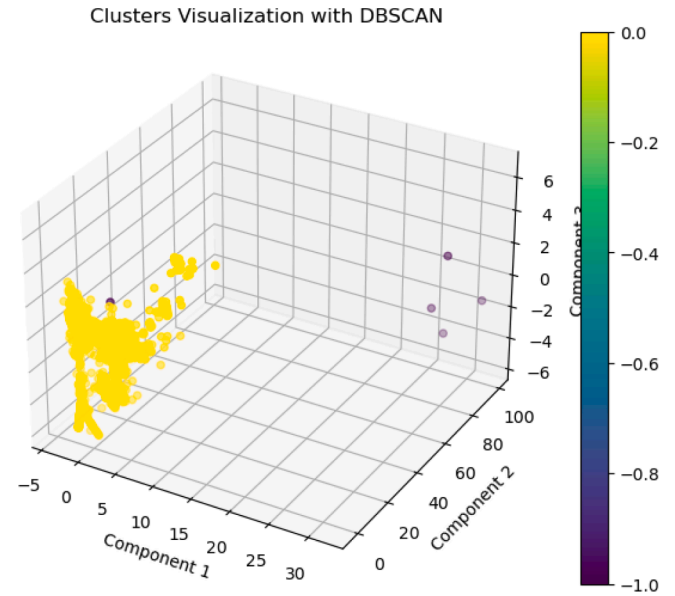


Fig. 3. DBscan clustering on the testing data.

#### NID classifier

The NID classifier is a multi-layer perceptron implemented using the Tensorflow library. The network comprises of four hidden layers of the following number of neurons: 64, 128, 128, 64 with ReLu activation. The output layer comprises three neurons with a softmax activation, each representing the traffic family type, either attack type or benign. The Adam optimizer is used along with the sparse categorical crossentropy loss.

#### 4.3. Experimental evaluation

We divide this section into four parts: (1) Modular Evaluation, (2) CGAN Diagnostics, (3) RL Diagnostics and Literature Comparison, and (4) Class & Incremental Learning Evaluation.

After testing the components of GARDIAN's first layer separately, now we introduce the unseen DDoS traffic pattern to this layer. The test data comprises the unseen pattern in addition to data from the first two days of the dataset. The MAE is able to detect the new patterns with an accuracy of 99%, corresponding to the samples that are correctly identified as new patterns from the test set. The remaining 1% are misclassified as noise by DBscan. This data is passed to the encoder layer to obtain the encoded representation, since the encoder is able to extract important features out of these patterns to compute the reconstruction error. These patterns are then sent to DBscan clustering for separation. DBscan is able to obtain the optimal number of clusters. In our case, we are able to obtain one cluster in addition to the noise. The obtained clusters from the testing data are shown in Fig. 3. A PCA with three dimensions is applied on the clustered data for plotting. Furthermore, the nearest neighbor method is applied to obtain the true label of the cluster from the labeled data.

As shown in this figure, the new encoded patterns by the MAE are classified in one cluster representing the DDoS. As a result, we can conclude the success of the first layer of GARDIAN in correctly identifying and classifying the new patterns, thereby detecting emerging traffic behavior relative to the patterns observed during training. The next step is to evaluate components' performance on the second layer of GARDIAN, including the CGAN, and NID classifier. Since the proposed DRL is at the core of GARDIAN, we evaluate its performance in Section 4.3.4.

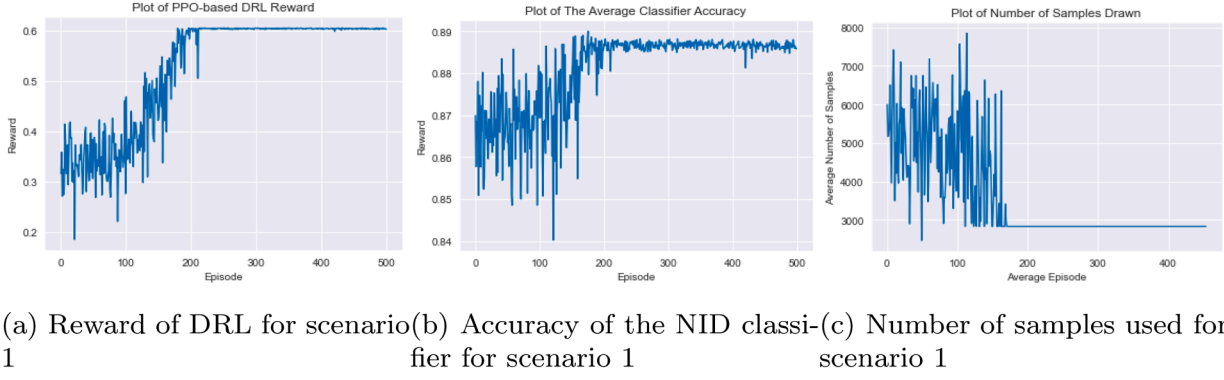


Fig. 4. Scenario 1: the proposed GARDIAN framework.

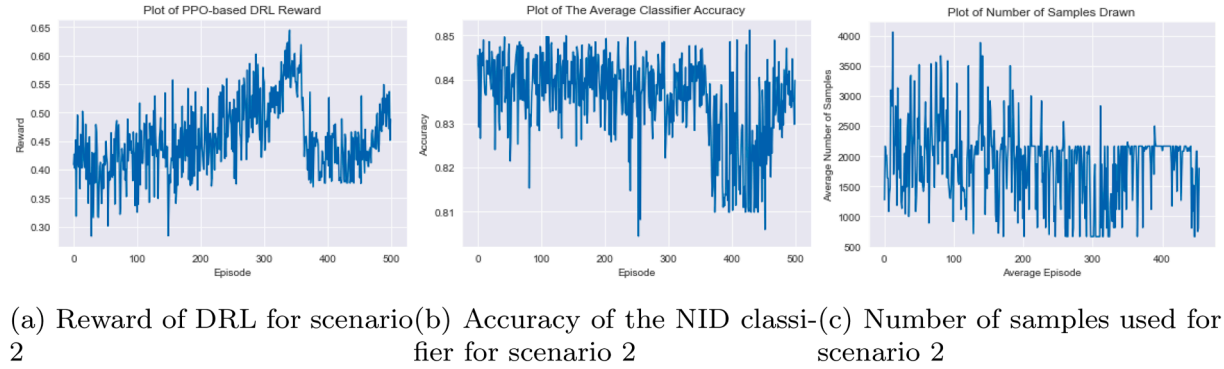


Fig. 5. Scenario 2 [11]: a framework with DRL and CGAN.

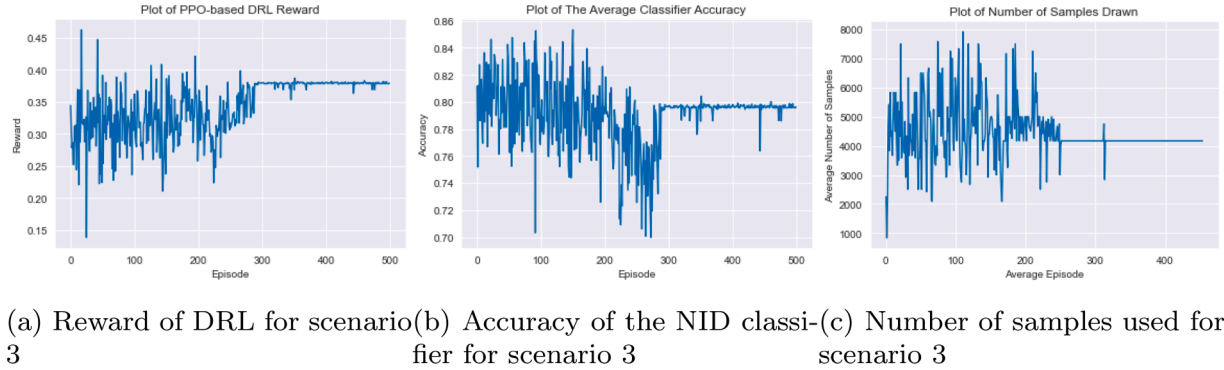


Fig. 6. Scenario 3 [34]: a framework with DRL and memory buffer.

#### 4.3.1. CGAN diagnostics: Diversity, sample quality, and runtime cost

Since adversarial losses alone are insufficient to assess sample quality, we added a focused diagnostics experiment for the CGAN on the target unseen class. The objective of this experiment is to explicitly study three aspects: mode collapse, generated sample quality, and computational cost. To this end, we compare two CGAN configurations: a collapse-prone aggressive setting and a lighter early-stopped setting.

To quantify potential mode collapse, we use a diversity proxy defined as the ratio between the average pairwise Euclidean distance of generated samples and that of real samples from the same target class. A value close to zero indicates severe collapse. To assess sample quality, we measure the average absolute difference in feature-wise means and standard deviations between real and generated samples, in addition to the average nearest-neighbor distance from generated samples to real target-class samples. Finally, to quantify computational overhead, we record the CGAN fine-tuning time and generation time.

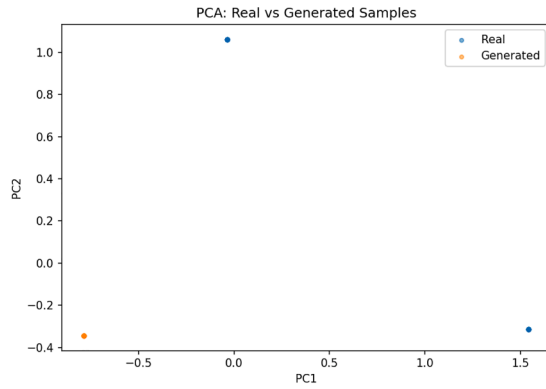
Under the aggressive configuration using target class *DDoS attack-HOIC*,  $\text{unseen\_cap} = 120000$ ,  $\text{gan\_hidden} = 256$ ,  $\text{gan\_lr} = 10^{-3}$ ,  $\text{gan\_batch} = 256$ , and  $\text{gan\_epochs} = 20$ , the generator exhibited complete collapse. In this case, the diversity ratio was 0.0, the generated-to-real nearest-neighbor distance was 1.5939, and the CGAN fine-tuning time reached 380.27 s. This setting is collapse-prone because it combines a relatively large model capacity, a high learning rate, and prolonged adversarial training, which together make the generator-discriminator optimization more unstable and increase the risk that the generator converges to a very limited subset of the target distribution. Fig. 7(a) visually confirms that the generated samples collapse into a highly restricted region compared to the real data distribution.

In contrast, under a lighter early-stopped configuration using target class *DDoS attacks-LOIC-HTTP*,  $\text{unseen\_cap} = 80000$ ,  $\text{gan\_hidden} = 128$ ,  $\text{gan\_lr} = 10^{-4}$ ,  $\text{gan\_batch} = 256$ , and  $\text{gan\_epochs} = 4$ , the collapse behavior was substantially mitigated and the computational overhead

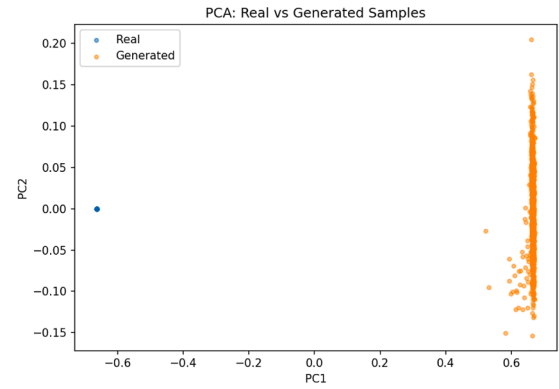
**Table 2**

CGAN diagnostics comparing a collapse-prone aggressive configuration and a lighter early-stopped configuration.

| Metric                               | Aggressive Configuration | Early-Stopped Configuration |
|--------------------------------------|--------------------------|-----------------------------|
| Target class                         | DDoS attack-HOIC         | DDoS attacks-LOIC-HTTP      |
| unseen_cap                           | 120,000                  | 80,000                      |
| gan_hidden                           | 256                      | 128                         |
| gan_lr                               | $10^{-3}$                | $10^{-4}$                   |
| gan_batch                            | 256                      | 256                         |
| gan_epochs                           | 20                       | 4                           |
| Diversity ratio                      | 0.0                      | 25.35                       |
| Feature mean abs. diff.              | –                        | 0.0377                      |
| Feature std. abs. diff.              | –                        | 0.0033                      |
| Generated-to-real NN distance        | 1.5939                   | 1.3307                      |
| CGAN fine-tuning time (s)            | 380.27                   | 35.18                       |
| Generation time for 2000 samples (s) | –                        | 0.0247                      |



(a) Aggressive CGAN setting showing severe collapse.



(b) Lighter early-stopped setting with substantially mitigated collapse behavior.

**Fig. 7.** PCA projection of real and generated target-class samples under two CGAN configurations. In the aggressive setting, the generated samples collapse into a highly restricted region of the feature space. In the lighter early-stopped setting, the generated samples exhibit a much broader spread, indicating that collapse symptoms are substantially mitigated.

was much lower. In this case, the diversity ratio increased to 25.35, the feature mean absolute difference was 0.0377, the feature standard-deviation absolute difference was 0.0033, the generated-to-real nearest-neighbor distance was 1.3307, the CGAN fine-tuning time was reduced to 35.18 s, and the generation time for 2000 samples was 0.0247 s. This revised setting is more conservative: the lower learning rate makes adversarial updates less volatile, the smaller hidden dimension reduces unnecessary model complexity, the shorter training horizon limits prolonged adversarial fitting, and the lower unseen-class cap further reduces adaptation cost. Fig. 7(b) shows a much broader spread of generated samples. Although this setting does not imply that the generated distribution perfectly matches the real one, it demonstrates that early stopping and lighter training can substantially reduce collapse symptoms while lowering the computational overhead.

Table 2 summarizes these diagnostics. Overall, these results show that CGAN training in GARDIAN must be monitored explicitly rather than assumed to be stable by design, and that the adopted lighter configuration offers a more practical trade-off between sample diversity and training cost. At the same time, these diagnostics are reported for a representative unseen-class case and should be interpreted as a focused validation of CGAN behavior in the proposed framework, rather than as a claim of universal robustness across all datasets or deployment scenarios.

#### 4.3.2. RL diagnostics: Autonomy, runtime, and retraining decisions

To evaluate the autonomy of GARDIAN more explicitly, we added a focused RL diagnostics experiment. This analysis is designed to quantify three aspects of the DRL component: the degree of autonomous decision-

making, the runtime cost of PPO-based adaptation, and the retraining amount and source-allocation behavior learned by the policy. In this work, autonomy is defined operationally as the ability of the deployed framework to perform adaptation decisions without human intervention once the initial models are trained. These decisions include whether to trigger retraining, how many samples to use for retraining, and how to allocate retraining data between CGAN-generated samples and replay-buffer samples.

Accordingly, we evaluate the RL agent from three complementary perspectives. First, we measure decision behavior, including the number of retraining rounds triggered and the quantity and source composition of the selected retraining data over time. Second, we measure decision utility through the corresponding post-update improvement in classifier performance. Third, we measure computational overhead using the PPO adaptation time, the mean policy inference time per decision, and the mean retraining-round time during the continual learning process.

The RL diagnostics are summarized in Table 3. In the reported run, the PPO-based adaptation phase required 1.24 s in total, with a mean policy inference time of 25.47 ms, a maximum inference time of 34.39 ms, and a mean retraining-round time of 0.0349 s. These results indicate that, in the current implementation, the runtime overhead of RL is explicitly measurable and modest rather than hidden or prohibitively large.

Taken together with the CGAN runtime diagnostics, these results indicate that the main latency of the adaptation loop is not due to PPO decision-making, but rather to the fine-tuning stage of the generative component. Importantly, this latency affects model updating rather than

**Table 3**  
RL diagnostics for automated continual adaptation.

| Metric                                   | Value   |
|------------------------------------------|---------|
| Total PPO adaptation time (s)            | 1.24    |
| Mean policy inference time (ms)          | 25.47   |
| Max policy inference time (ms)           | 34.39   |
| Mean retraining-round time (s)           | 0.0349  |
| Number of policy decisions               | 20      |
| Policy decisions without manual override | 20      |
| Autonomy rate                            | 1.00    |
| Mean total samples selected per round    | 18.15   |
| Mean CGAN samples selected per round     | 15.60   |
| Mean replay samples selected per round   | 2.55    |
| Final total samples selected             | 23      |
| Final CGAN fraction                      | 0.7826  |
| Mean post-update delta macro-F1          | 0.00052 |
| Best post-update delta macro-F1          | 0.02633 |
| Std. total samples selected per round    | 2.83    |

immediate traffic screening, since the first detection layer and the current classifier remain active while adaptation is being executed.

The same diagnostics also provide a quantitative characterization of autonomy. We compute the autonomy rate as the ratio between the number of policy decisions executed without manual override and the total number of policy decisions taken during adaptation. Formally,

$$\text{Autonomy Rate} = \frac{N_{\text{automated decisions}}}{N_{\text{total policy decisions}}} \quad (9)$$

In the reported run, the policy made 20 retraining decisions, all 20 without manual override, yielding an autonomy rate of 1.00. We emphasize, however, that autonomy is not evaluated only through this rate, since the present deployment loop does not include a manual-override path. For this reason, we also quantify the behavior and utility of those automated decisions through the selected retraining actions and their performance effects.

Regarding what RL actually controls, the current formulation does not optimize arbitrary low-level hyperparameters such as learning rate, GAN epochs, or classifier architecture. Instead, the action space controls the amount of retraining data and its allocation between CGAN-generated and replay-buffer samples, including per-class allocation through the action vector. In the reported run, the policy selected on average 18.15 samples per retraining round, consisting of 15.60 CGAN samples and 2.55 replay-buffer samples per round. In the final retraining round, the policy selected 23 total samples, with a final CGAN fraction of 0.7826. The mean post-update change in macro-F1 was 0.00052, with a best round-wise improvement of 0.02633. Fig. 8 shows that the selected retraining amounts and source allocations vary across rounds rather than remaining fixed, which confirms that the policy is making nontrivial retraining decisions. Taken together, these diagnostics provide an explicit quantitative assessment of autonomy, RL runtime overhead, and retraining-data selection behavior, rather than leaving these properties implicit in the broader continual-learning results.

#### 4.3.3. Reward sensitivity analysis

Because the reward function depends explicitly on the class-weighting vector  $\lambda_i$  and the efficiency penalty factor  $\beta$ , we added a focused reward-sensitivity study to evaluate how these parameters affect both predictive performance and the learned retraining behavior. The study was conducted on a representative continual-adaptation scenario with active classes {Benign, Bot, Infiltration, SSH-Bruteforce}, where Bot was treated as the target unseen class. We considered three weighting modes for  $\lambda_i$ —*uniform*, *attack\_priority*, and *benign\_priority*—and four penalty settings,  $\beta \in \{0.0, 0.005, 0.02, 0.1\}$ .

Fig. 9 summarizes the effect of these reward settings on both the final continual-learning outcome and the learned retraining policy. Fig. 9(a) shows the final macro-F1 obtained under the different combinations of  $\lambda_i$  and  $\beta$ . Across the 12 tested configurations, the final macro-F1 varied

from approximately 0.480 to 0.498, corresponding to an overall range of 0.0185. This indicates that the reward parameters do influence the final continual-learning outcome, but the system remains reasonably stable under moderate reward variation rather than exhibiting excessively brittle behavior.

The same analysis also shows that the reward parameters change the learned retraining policy. Fig. 9(b) reports the late-stage average number of selected retraining samples per round as a function of  $\beta$ . Under the uniform weighting scheme, increasing  $\beta$  from 0.0 to 0.1 reduced the late-stage retraining demand from 10.552 to 7.592 samples per round, corresponding to an approximately 28% decrease, while simultaneously improving the final macro-F1 from 0.4818 to 0.4981. This confirms that  $\beta$  does not merely rescale the reward numerically; it changes the operating point of the learned policy by shifting it toward more sample-efficient adaptation.

The class-weighting configuration  $\lambda_i$  also changes the policy behavior. For example, at  $\beta = 0.005$ , the uniform setting yielded a final macro-F1 of 0.4796 with 7.878 late-stage samples per round, whereas both *attack\_priority* and *benign\_priority* reached 0.4981, but with different retraining budgets, 9.560 and 9.038, respectively. This shows that  $\lambda_i$  affects not only the final score but also how aggressively the policy allocates retraining samples across the adaptation process.

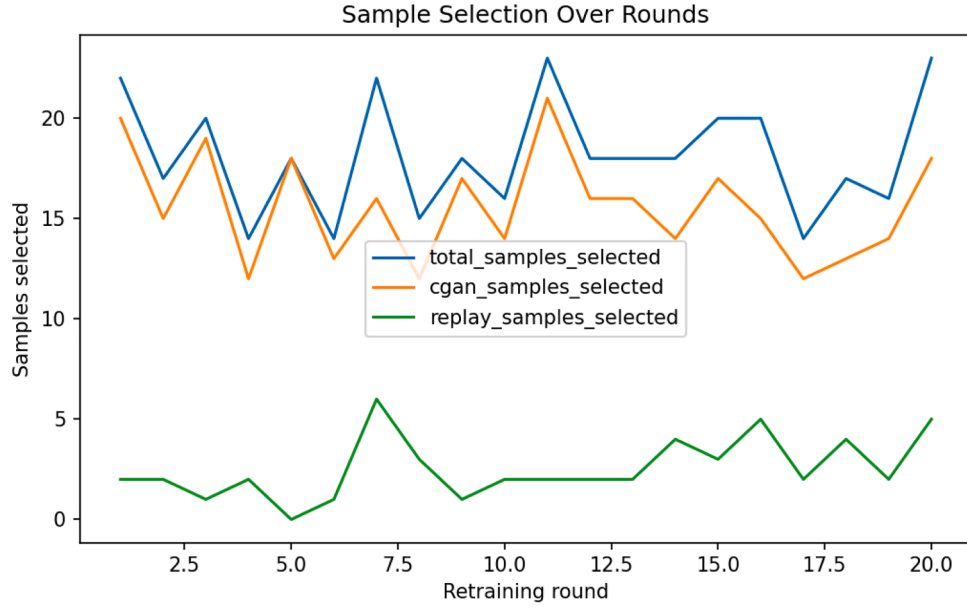
Table 4 summarizes the complete sweep. The best-performing setting in terms of final macro-F1 was *attack\_priority* with  $\beta = 0.005$ , whereas uniform with  $\beta = 0.1$  achieved the same final macro-F1 with a lower late-stage retraining demand, corresponding to a more sample-efficient operating point. Therefore, the manuscript does not treat  $\lambda_i$  and  $\beta$  as unexamined reward constants. Instead, the added analysis shows that these parameters govern a meaningful trade-off between predictive performance and retraining efficiency, and may need to be selected according to the target deployment environment.

#### 4.3.4. Ablation study and literature comparison

At this stage of evaluation, the first layer of GARDIAN is performing well, as well as CGAN and MLP, tested on a single dataset. In this section, we are evaluating the performance of the DRL Agent by evaluating the fine-tuning actions taken against the NID classifier. Following the prediction results of the classifier on the test data (which now includes DDoS), the average accuracy is nearly 82%. The test data includes a variation of the different classes from the dataset. Furthermore, the replay buffer contains 2000 samples for each of the three labels corresponding to the traffic classes. After obtaining the new patterns and having CGAN ready, we start the DRL training procedure towards improving the performance of the classifier.

To evaluate the importance of each of the components in the second layer of GARDIAN, we have performed an ablation study and trained DRL, thus obtaining three different scenarios: (1) DRL with CGAN and memory buffer; (2) DRL with CGAN; (3) DRL with memory buffer. The second scenario resembles a similar setup to [11], except that a detection of new patterns is not part of their solution and we are using a DRL agent to control the fine-tuning process. Similarly, Prasath et al. [34] corresponds to the third scenario where the authors utilize a NID solution with a memory buffer. Each of these scenarios is evaluated against the average rewards of DRL, the accuracy of the NID classifier, and the number of samples extracted from the memory buffer.

In Figs. 4–6, we display the results of the continual learning procedure for the three different scenarios. By comparing the graphs vertically, we are able to differentiate between the different metrics for each scenario. Starting with the proposed GARDIAN framework of Fig. 4a, we can observe that the model is able to converge towards a reward of 0.6 after 200 episodes, utilizing both CGAN and the memory buffer. This reward trajectory should be interpreted as the improvement of the retraining policy rather than the availability of the detection pipeline itself, since the first detection layer and the classifier continue to operate while the DRL policy is being refined, allowing emerging or zero-day-like traffic to remain subject to detection and screening during this

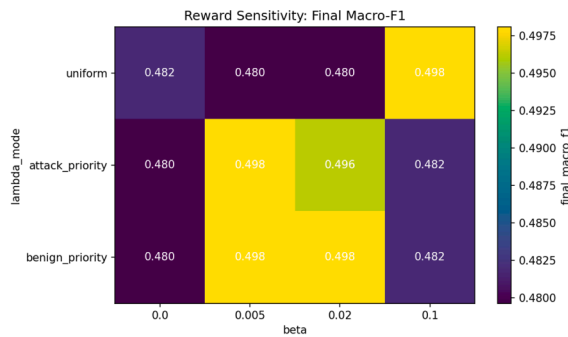


**Fig. 8.** Per-round retraining data selected by the PPO policy, showing total selected samples together with the decomposition into CGAN-generated and replay-buffer samples. This figure visualizes the policy's autonomous control of retraining amount and data source during continual adaptation. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

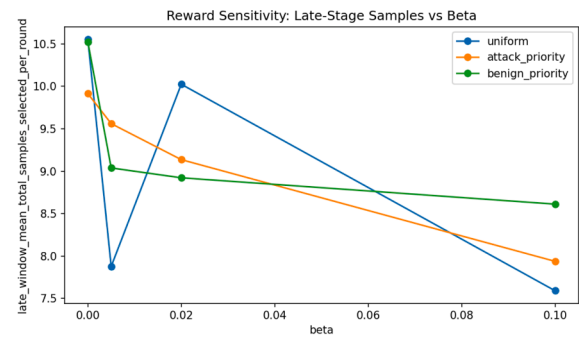
**Table 4**

Reward-sensitivity results under different class-weighting modes  $\lambda_i$  and efficiency penalties  $\beta$ . Late-stage values denote the average number of selected retraining samples per round near the end of adaptation.

| $\lambda$ mode  | $\beta$ | Final macro-F1 | Late-stage total | Late-stage CGAN | Late-stage replay |
|-----------------|---------|----------------|------------------|-----------------|-------------------|
| uniform         | 0.0     | 0.4818         | 10.552           | 8.386           | 2.166             |
| uniform         | 0.005   | 0.4796         | 7.878            | 6.442           | 1.436             |
| uniform         | 0.02    | 0.4796         | 10.024           | 8.408           | 1.616             |
| uniform         | 0.1     | 0.4981         | 7.592            | 5.978           | 1.614             |
| attack_priority | 0.0     | 0.4796         | 9.914            | 8.072           | 1.842             |
| attack_priority | 0.005   | 0.4981         | 9.560            | 7.730           | 1.830             |
| attack_priority | 0.02    | 0.4962         | 9.136            | 7.134           | 2.002             |
| attack_priority | 0.1     | 0.4818         | 7.938            | 6.236           | 1.702             |
| benign_priority | 0.0     | 0.4796         | 10.524           | 8.480           | 2.044             |
| benign_priority | 0.005   | 0.4981         | 9.038            | 7.318           | 1.720             |
| benign_priority | 0.02    | 0.4981         | 8.922            | 7.392           | 1.530             |
| benign_priority | 0.1     | 0.4818         | 8.612            | 6.380           | 2.232             |



(a) Final macro-F1 across  $\lambda_i$  modes and  $\beta$  values.



(b) Late-stage retraining demand versus  $\beta$ .

**Fig. 9.** Reward-sensitivity analysis under different class-weighting modes  $\lambda_i$  and efficiency penalties  $\beta$ . Fig. 9(a) shows that both reward components affect the final macro-F1, while Fig. 9(b) shows that  $\beta$  directly changes the learned retraining policy and its operating point in terms of late-stage selected samples per round. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

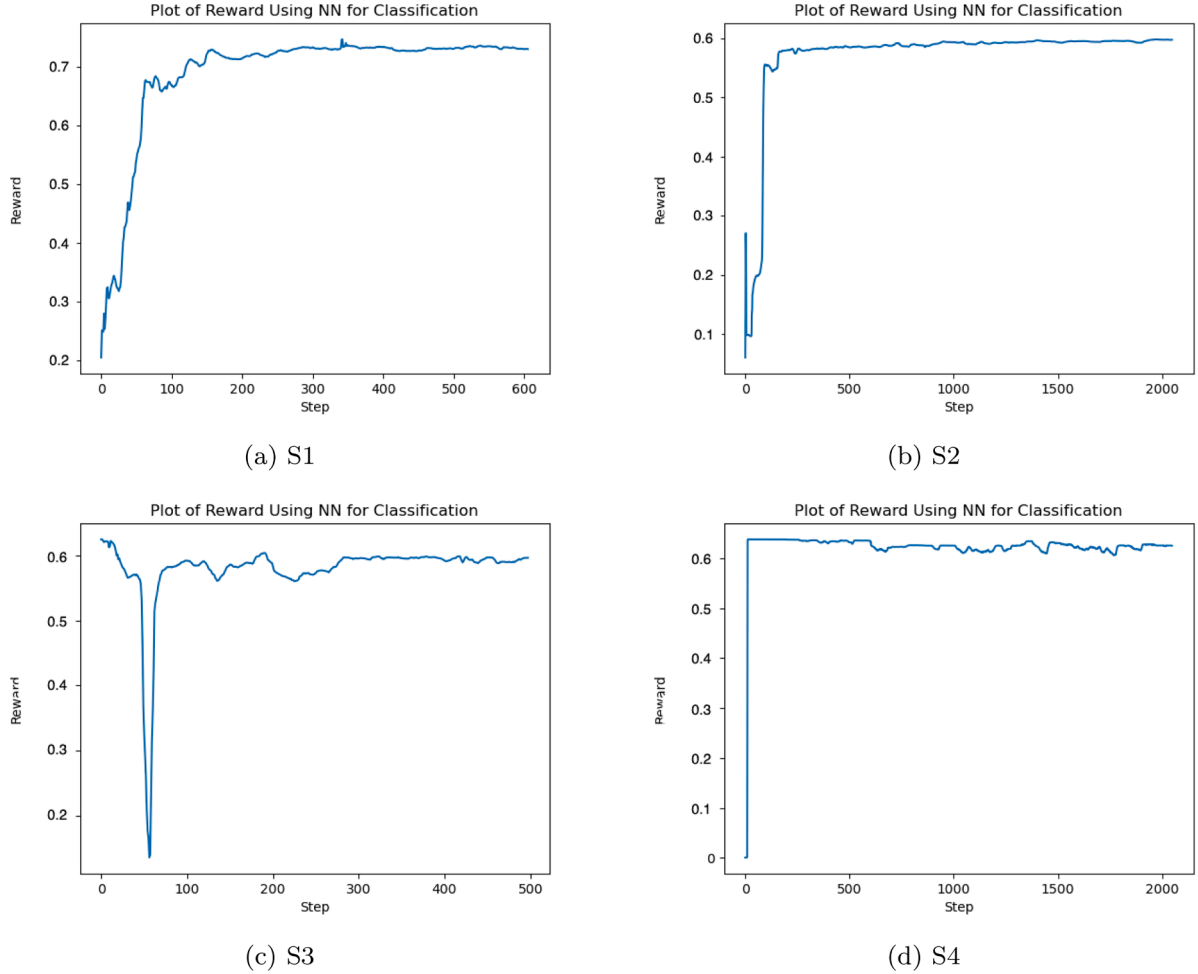


Fig. 10. DRL reward for each of the continual learning scenarios of Table 5.

Table 5

Domain & class incremental learning using NetFlow.

| Use Case   | UNSW-NB15 | ToN-IoT     | BoT-IoT     | CSE-CIC-IDS18 |
|------------|-----------|-------------|-------------|---------------|
| S1 (train) | B, D, R   | B, D, DD, R | –           | –             |
| S1 (test)  | –         | –           | B, D, DD, R | B, D, DD      |
| S2 (train) | –         | B, D, DD, R | B, D, DD, R | –             |
| S2 (test)  | B, D, R   | –           | –           | B, D, DD      |
| S3 (train) | B, R      | –           | –           | DD            |
| S3 (test)  | –         | D           | D           | –             |
| S4 (train) | –         | D, DD, R    | –           | –             |
| S4 (test)  | –         | –           | –           | B             |

phase. The reward for the second scenario is shown in Fig. 5a, where the model struggles to converge to a stable reward over the course of 500 episodes. This is mainly due to the catastrophic forgetting issue, which causes fluctuations in the NID performance because samples from the memory buffer are not available during training. Additionally, in Fig. 6a, we show the reward graph of the DRL agent for the third scenario. We can notice from this figure that the agent is able to converge around 0.38, which is a lower value compared to the proposed GARDIAN. The performance of DRL in the third scenario is due to the lack of enough samples related to the new patterns of attack, which are produced by the CGAN network, which this scenario is missing.

Analyzing the accuracy of the NID classifier across the different scenarios, we can notice that GARDIAN is able to achieve the highest constant accuracy of almost 90% according to Fig. 4b, benefiting from both

data of the buffer and CGAN. In contrast, the accuracy of Scenario 2 in Fig. 5b is not stable nor exceeding 85%, which is a result of the catastrophic forgetting resulted by missing a memory before. Furthermore, we can notice in this figure that the accuracy drops to 81%, illustrating the impact of catastrophic forgetting. In the third scenario of Fig. 5c, we can see that the accuracy drops over time while fine-tuning on data only from the memory buffer. Therefore, the accuracy of the NID classifier converges to a value lower than 80%.

Finally, we analyze the amount of samples used from both CGAN and the memory buffer, which we aim to minimize according to the second objective of the MDP formulation for our DRL agent. In Fig. 4c, we can see that the DRL of GARDIAN converges to utilizing around 2900 samples from both CGAN and the buffer to fine-tune and stabilize the classifier. In Fig. 5c, we can identify that the DRL agent is not able to converge to a specific number of samples, nor converging the accuracy of the model. The low number of samples utilized by the agent is due to the fact that the second scenario does not have access to CGAN, thus only the buffer samples are used. With regards to the third scenario, we can see in Fig. 6c that the agent is able to converge in terms of the number of samples used for fine-tuning to a value close to 4100 samples, which is higher than the GARDIAN's DRL.

As a conclusion to our evaluations, we are able to confirm that outstanding performance of the different components of the proposed GARDIAN framework and how they can coordinate together to achieve zero-day detection in the first layer and automated continual learning procedure in the second layer.

#### 4.3.5. Domain & class incremental evaluation

Four different scenarios are built to test the framework on the different datasets of NetFlow. These scenarios cover the cases of class incremental and domain incremental learning across the four datasets. Table 5 refers to the combination of classes for training and testing data used for each scenario. The abbreviations of B, D, DD, and R is defined in Table 1. Based on how GARDIAN is built, the training data is used to build the MLP classifier, MAE, and CGAN. The test data is then passed through the first detection layer to obtain and classify anomalies, in addition to fine-tuning CGAN and starting the DRL mechanism for policy updates. The DRL reward for each of the four scenarios is shown in Fig. 10

For S1, we have the case of domain incremental learning, where the data is trained on two initial datasets, while testing is performed on completely other datasets with the same classes. This is possible due to the NetFlow dataset concatenation procedure. The performance of DRL for S1 shown in Fig. 10a. Based on the graph obtained, we can see that the reward starts close to 0.2 and moves up after around 100 steps to 0.7. This indicates that the MLP classifier is not able to correctly classify samples in the test data. Over-time, we can notice that the MLP classifier becomes capable of classifying test samples through the GARDIAN mechanism with a combination of MAE, DBscan, DRL, CGAN, and the MLP. The same scenario is true with S2 but we swap the datasets for training v.s. testing.

In scenarios S3 and S4, we use the pre-trained DRL for Scenario S1. The evaluation focus is on class incremental learning. In Figs. 10c and 10d we can notice that the DRL agent is able to improve the MLP accuracy on the test data over time. For S3, we can notice a sudden drop in the reward, where the DRL agent is able to manage it quickly and act accordingly restore the MLP performance. This indicates that the model is able to properly classify the D class on the test data. Similarly, in S4, the DRL model is able to properly classify the B classes after a quick recovery with the first few iterations. To this end, we can conclude that the DRL model is able to adapt to domain and class incremental learning scenarios while alternating between different datasets for training and testing.

## 5. Conclusion

With the increasing number of connected devices and applications, network attacks are becoming more diverse and frequent. This evolution poses a significant challenge for Network Intrusion Detection Systems (NIDS) to adapt to new attack patterns. In this study, we examined various Deep Learning-based NID solutions and identified a critical limitation: the lack of automated frameworks for continual learning and zero-day attack detection. Existing solutions often struggle with inadequate training data and the absence of a unified approach for ongoing learning. To address these limitations, we propose GARDIAN, an autonomous framework with a dual-layer system. The first layer aims to detect and label emerging traffic patterns, including zero-day-like behaviors, by employing an MAE and DBSCAN. The second layer performs autonomous continual learning, powered by the novel DRL agent. This layer incorporates a CGAN, a memory buffer, a DRL, and a base classifier to manage automated continual learning, with the DRL agent overseeing the process. Our simulations using the NetFlow dataset demonstrate the effectiveness of these components within the GARDIAN framework. Additionally, an ablation study, that compares GARDIAN with existing literature approaches, highlights the contributions of each component. Finally, we present four scenarios to validate the proposed approach in continual learning settings. In addition, the paper explicitly evaluates both the stability of the CGAN component and the sensitivity of the RL reward design, showing that adversarial training is sensitive to hyperparameter selection and that the reward parameters  $\lambda_i$  and  $\beta$  influence the trade-off between predictive performance and retraining efficiency.

This work is a first step towards achieving a full pipeline for supporting autonomous continual learning for NID. Future directions aims

to address existing flows in the system and expand the capabilities of GARDIAN. In the future, we plan to focus on addressing the existing limitations of GARDIAN. This includes the requirement for fine-tuning of MAE and CGAN. Furthermore, we aim to test our approach on various types of NID classifiers, not only MLP.

## CRedit authorship contribution statement

**Hani Sami:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Formal analysis, Data curation, Conceptualization; **Chamseddine Talhi:** Writing – review & editing, Validation, Supervision, Methodology; **Hakima Ould-Slimane:** Writing – review & editing, Validation, Supervision; **Azzam Mourad:** Writing – review & editing, Validation; **Hadi Otrouk:** Writing – review & editing, Supervision; **Jamal Bentahar:** Writing – review & editing, Supervision.

## 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.

## Data availability

Research Link and source code link Provided in the rebuttal letter and manuscript. In addition, the data is publicly available and cited in the work.

## References

- [1] Chou D, Jiang M. A survey on data-driven network intrusion detection. *ACM Comput Surv* 2021;54(9):1–36.
- [2] Cybersecurity Ventures. *Hackerpocalypse: Cybercrime report 2016*. 2016. Accessed: 2024-09-05; <https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/>.
- [3] Magazine I. Study reveals web apps most common attack vector. 2024. Accessed: 2024-09-05; <https://www.infosecurity-magazine.com/news/study-reveals-web-apps/>.
- [4] Vu L, Nguyen QU, Nguyen DN, Hoang DT, Dutkiewicz E. Deep generative learning models for cloud intrusion detection systems. *IEEE Trans Cybern* 2022;53(1):565–77.
- [5] Abdulganiyu OH, Ait Tchakoucht T, Saheed YK. A systematic literature review for network intrusion detection system (IDS). *Int J Inf Secur* 2023;22(5):1125–62.
- [6] MahdaviFar S, Ghorbani AA. CapsRule: explainable deep learning for classifying network attacks. *IEEE Trans Neural Netw Learn Syst* 2023;35(9):12434–48.
- [7] Zhou X, Wu J, Liang W, Kevin I, Wang K, Yan Z, et al. Reconstructed graph neural network with knowledge distillation for lightweight anomaly detection. *IEEE Trans Neural Netw Learn Syst* 2024;35(9):11817–28.
- [8] Alalmaie A, Nanda P, He X. Zero trust network intrusion detection system (NIDS) using auto encoder for attention-based CNN-biLSTM. In: *Proceedings of the 2023 Australasian computer science week*. 2023, p. 1–9.
- [9] Madhuri A, Jyothi VE, Praveen SP, Sindhura S, Srinivas VS, Kumar D LS. A new multi-level semi-supervised learning approach for network intrusion detection system based on the 'GOA'. *J Interconnect Netw* 2022;24(suppl1):2143047.
- [10] Ding H, Chen L, Dong L, Fu Z, Cui X. Imbalanced data classification: a KNN and generative adversarial networks-based hybrid approach for intrusion detection. *Future Gener Comput Syst* 2022;131:240–54.
- [11] Huang S, Lei K. IGAN-IDS: an imbalanced generative adversarial network towards intrusion detection system in Ad-Hoc networks. *Ad Hoc Netw* 2020;105:102177.
- [12] Iliyasa AS, Deng H. N-GAN: A novel anomaly-based network intrusion detection with generative adversarial networks. *Int J Inf Technol* 2022;14(7):3365–75.
- [13] Wu Y, Nie L, Wang S, Ning Z, Li S. Intelligent intrusion detection for internet of things security: a deep convolutional generative adversarial network-enabled approach. *IEEE Internet Things J* 2021;10(4):3094–106.
- [14] Kemker R, McClure M, Abitino A, Hayes T, Kanan C. Measuring catastrophic forgetting in neural networks. In: *Proceedings of the AAAI conference on artificial intelligence*; vol. 32. 2018,.
- [15] Channappayya S, Tammaru BR, et al. Augmented memory replay-based continual learning approaches for network intrusion detection. *Adv Neural Inf Process Syst* 2024;36:17156–69.
- [16] Oikonomou C, Iliopoulos I, Ioannidis D, Tzovaras D. A multi-class intrusion detection system based on continual learning. In: *2023 IEEE International conference on cyber security and resilience (CSR)*. IEEE; 2023, p. 86–91.
- [17] Zhang Z, Zhang Y, Li H, Liu S, Chen W, Zhang Z, et al. Federated continual representation learning for evolutionary distributed intrusion detection in industrial internet of things. *Eng Appl Artif Intell* 2024;135:108826.

- [18] Wang L, Zhang X, Su H, Zhu J. A comprehensive survey of continual learning: theory, method and application. *IEEE Trans Pattern Anal Mach Intell* 2024;46(8):5362–83.
- [19] Suwannalai E, Polprasert C. Network intrusion detection systems using adversarial reinforcement learning with deep q-network. In: 2020 18th international conference on ICT and knowledge engineering (ICT&KE). IEEE; 2020, p. 1–7.
- [20] Benaddi H, Ibrahim K, Benslimane A, Qadir J. A deep reinforcement learning based intrusion detection system (DRL-IDS) for securing wireless sensor networks and internet of things. In: International wireless internet conference. Springer; 2019, p. 73–87.
- [21] Schubert E, Sander J, Ester M, Kriegel HP, Xu X. DBSCAN revisited, revisited: why and how you should (still) use dbscan. *ACM Trans Database Syst* 2017;42(3):1–21.
- [22] Mirza M, Osindero S. Conditional generative adversarial nets. 2014. arXiv preprint arXiv:1411.1784.
- [23] Sarhan M, Layeghy S, Portmann M. Towards a standard feature set for network intrusion detection system datasets. *Mob Netw Appl* 2022;27(1):357–70.
- [24] Nicolau M, McDermott J, et al. Learning neural representations for network anomaly detection. *IEEE Trans Cybern* 2018;49(8):3074–87.
- [25] Muhammad G, Hossain MS, Garg S. Stacked autoencoder-based intrusion detection system to combat financial fraudulent. *IEEE Internet Things J* 2020;10(3):2071–8.
- [26] Narmadha S, Balaji NV. Improved network anomaly detection system using optimized autoencoder- LSTM. *Expert Syst Appl* 2025;273:126854.
- [27] Lunardi WT, Lopez MA, Giacalone J-P. Arcade: adversarially regularized convolutional autoencoder for network anomaly detection. *IEEE Trans Netw Serv Manage* 2022;20(2):1305–18.
- [28] Shahriar MH, Haque NI, Rahman MA, Alonso M. G-IDS: generative adversarial networks assisted intrusion detection system. In: 2020 IEEE 44th annual computers, software, and applications conference (COMPSAC). IEEE; 2020, p. 376–85.
- [29] Park C, Lee J, Kim Y, Park J-G, Kim H, Hong D. An enhanced AI-based network intrusion detection system using generative adversarial networks. *IEEE Internet Things J* 2022;10(3):2330–45.
- [30] Wang P, Li S, Ye F, Wang Z, Zhang M. PacketCGAN: exploratory study of class imbalance for encrypted traffic classification using CGAN. In: ICC 2020-2020 IEEE International conference on communications (ICC). IEEE; 2020, p. 1–7.
- [31] Yang H, Xu J, Xiao Y, Hu L. SPE-ACGAN: a resampling approach for class imbalance problem in network intrusion detection systems. *Electronics* 2023;12(15):3323.
- [32] Caminero G, Lopez-Martin M, Carro B. Adversarial environment reinforcement learning algorithm for intrusion detection. *Comput Netw* 2019;159:96–109.
- [33] Sethi K, Sai Rupesh E, Kumar R, Bera P, Venu Madhav Y. A context-aware robust intrusion detection system: a reinforcement learning-based approach. *Int J Inf Secur* 2020;19:657–78.
- [34] Prasath S, Sethi K, Mohanty D, Bera P, Samantaray SR. Analysis of continual learning models for intrusion detection system. *IEEE Access* 2022;10:121444–64.
- [35] Wiewel F, Yang B. Continual learning for anomaly detection with variational autoencoder. In: ICASSP 2019-2019 IEEE International conference on acoustics, speech and signal processing (ICASSP). IEEE; 2019, p. 3837–41.
- [36] Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O. Proximal policy optimization algorithms. 2017. arXiv preprint arXiv:1707.06347.
- [37] University of New Brunswick. IDS 2018 dataset. 2018. <https://www.unb.ca/cic/datasets/ids-2018.html>.
- [38] Koroniotis N, Moustafa N, Sitnikova E, Turnbull B. Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: bot-IoT dataset. *Future Gener Comput Syst* 2019;100:779–96.
- [39] Moustafa N. A new distributed architecture for evaluating AI-based security systems at the edge: network TON\_iiot datasets. *Sustain Cities Soc* 2021;72:102994.
- [40] Moustafa N, Slay J. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In: 2015 Military communications and information systems conference (MilCIS). IEEE; 2015, p. 1–6.
- [41] Ravichandiran S. Hands-on reinforcement learning with Python: master reinforcement and deep reinforcement learning using OpenAI gym and tensorflow. Packt Publishing Ltd; 2018.
- [42] Hill A, Raffin A, Ernestus M, Gleave A, Kanervisto A, Traore R, et al. Stable baselines. 2018. <https://github.com/hill-a/stable-baselines>.