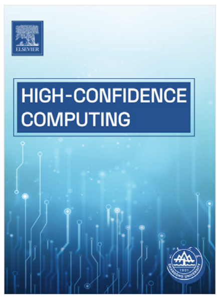
Enhanced Meta-IDS: Adaptive multi-stage IDS with sequential model adjustments

Nadia Niknami, Vahid Mahzoon, Slobadan Vucetic, Jie Wu



PII:	S2667-2952(25)00002-9
DOI:	https://doi.org/10.1016/j.hcc.2025.100298
Reference:	HCC 100298
To appear in:	*High-Confidence Computing*
Received date :	16 October 2024
Revised date :	11 December 2024
Accepted date :	19 December 2024

Enhanced Meta-IDS: Adaptive Multi-Stage IDS with Sequential Model Adjustments

Nadia Niknami[a], Vahid Mahzoon[b], Slobadan Vucetic[b], Jie Wu[a]

[a]*Center for Networked Computing, Temple University, Philadelphia, PA, USA*
[b]*Center for Hybrid Intelligence, Temple University, Philadelphia, PA, USA*

Abstract

Traditional single-machine Network Intrusion Detection Systems (NIDS) are increasingly challenged by rapid network traffic growth and the complexities of advanced neural network methodologies. To address these issues, we propose an *Enhanced Meta-IDS* framework inspired by meta-computing principles, enabling dynamic resource allocation for optimized NIDS performance. Our hierarchical architecture employs a three-stage approach with iterative feedback mechanisms. In real-world scenarios with intermittent data batches, we leverage these intervals to enhance our models. Outputs from the third stage provide labeled samples back to the first and second stages, allowing retraining and fine-tuning based on the most recent results without incurring additional latency. By dynamically adjusting model parameters and decision boundaries, our system optimizes responses to real-time data, effectively balancing computational efficiency and detection accuracy. By ensuring that only the most suspicious data points undergo intensive analysis, our multi-stage framework optimizes computational resource usage. Experiments on benchmark datasets demonstrate that our *Enhanced Meta-IDS* improves detection accuracy and reduces computational load or CPU time, ensuring robust performance in high-traffic environments. This adaptable approach offers an effective solution to modern network security challenges.

Keywords: Adaptive IDS, CPU time, dynamic adaptation, Intrusion Detection System(IDS), Meta-computing.

1. Introduction

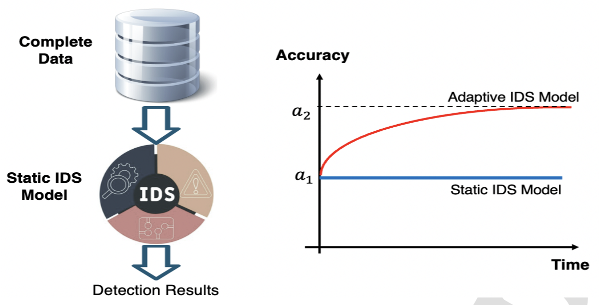
The crucial function of Network Intrusion Detection Systems (NIDS) in protecting network perimeters by diligently monitoring and quickly identifying security breaches remains essential [1, 2, 3, 4, 5]. The dynamics of large-scale networks introduce significant complexities for NIDS performance due to varied traffic patterns with diverse application mixes and temporal fluctuations. These factors require adaptive strategies, as different application protocols require varying depths of analysis. Furthermore, heavy-tailed data transfers cause sudden traffic volume peaks, posing substantial challenges for NIDS operations. In addition, in high-performance environments, the efficacy of NIDS is based not only on the handling of average traffic loads but also on their capacity to manage frequent traffic bursts. Robust mechanisms are essential to ensure NIDS effectiveness under dynamic and demanding conditions. One key measure of effectiveness is how quickly an IDS can identify security incidents. The time to detect a security incident is influenced by the effectiveness of monitoring systems, and the configuration and capabilities of the IDS.

Simpler models may outperform complex models in some data sets because of superior generalization capabilities. However, for large and complex datasets, simple models may fall short, necessitating complex models to achieve acceptable efficiency. Although complex models can deliver higher accuracy, they require significant computational resources for training and inference, which may not be feasible in resource-constrained environments. To address these challenges, there is a necessity for a multi-stage and adaptive IDS that can balance accuracy and computational efficiency. The advent of Deep Learning has significantly enhanced the accuracy and resilience of intrusion detection models. However, despite their accuracy, these models face implementation challenges on resource-constrained devices due to high computational overhead and large model size. The dynamic nature of large-scale networks further complicates effective intrusion detection. Traditional single-machine NIDS struggles with increasing traffic volumes, and deployment on resource-constrained devices remains a challenge. An adaptive, multi-stage IDS can dynamically adjust its processing based on the complexity of the data and available resources.

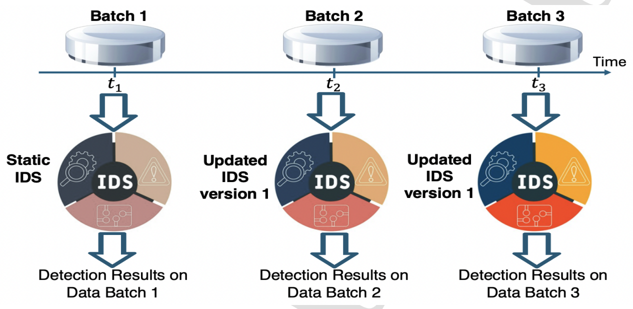
Meta-computing is an effective approach for managing large-scale computational tasks by efficiently acquiring and utilizing resources. Leveraging this concept, we present a multi-stage hierarchical NIDS architecture that dynamically allocates analysis tasks across multiple levels of complexity. In our proposed system, the first stage employs a simple model that quickly processes incoming traffic. When an attack is detected or if there is uncertainty with a low detection probability, the suspicious traffic is forwarded to the second stage for a more in-depth analysis. Similarly, the second stage forwards traffic with low detection probability to a third stage, which utilizes even more sophisticated and computationally intensive models for analysis.

In real-world scenarios, data arrive continuously in batches with intervals between them. Since intrusion detection models need to be updated and adapted to new data, we can leverage these intervals to implement an iterative feedback mechanism from the last stage to the previous ones in the multi-stage IDS. This approach enhances the system's adaptability and responsiveness without incurring additional latency. We introduce an iterative feedback mechanism in which the outputs from the third stage are used to provide labeled samples back to the first and second stages. This feedback loop allows us to retrain and fine-tune the models in the earlier stages based on the most recent results, improving their detection accuracy for subsequent data batches. By leveraging the time gaps between data arrivals, we enhance the system's adaptability and responsiveness without incurring additional latency.

Fig. 1 illustrates the difference between processing the entire dataset at once and handling data in incremental batches with iterative model updates. In the traditional scenario, depicted in Fig. 1(a), the IDS operates on a complete dataset using a static model. This model processes all incoming data simultaneously without the ability to adapt to new threats over time. The IDS model does not adapt or update based on new data. It continues to use the same initial model for all future data batches, without any further learning or updates from new information. In contrast, Fig. 1(b) showcases a multi-stage IDS architecture designed for real-world network environments where data arrives in batches over time. The first data batch passes through the *Static IDS*, which employs a simple anomaly detection model for rapid initial assessment. Static IDS analyzes *Data Batch 1* at time t_1 and produces detection results for this batch. However, instead of remaining static, the IDS model is updated to *Updated IDS version 1* at t_2, reflecting the model's adaptation or learning from the previously seen data. The updated version of the IDS now analyzes *Data Batch 2* and produces detection results. The model is further updated to a newer version (possibly incorporating learnings from both *Data Batches 1* and *Data Batch 2*) when analyzing *Data Batch 3* at t_3, leading to more refined detection results. Static IDS does not change or adapt after deployment, while the Incremental IDS evolves by learning from new data batches over time, potentially improving its detection accuracy and adaptability as new threats and behaviors emerge.



(a) Static IDS Model.


Batch 1
Batch 2
Batch 3
Time
t_1
t_2
t_3
Static IDS
Updated IDS version 1
Updated IDS version 1
IDS
Detection Results on Data Batch 1
Detection Results on Data Batch 2
Detection Results on Data Batch 3

(b) IDS with Incremental Data and Model Updates

Figure 1: Static IDS with one-time data vs. adaptive IDS.

The core concept of our proposed extension to the Meta-IDS [6] framework focuses on improving the adaptability and efficiency of the system by iteratively improving its performance with each incoming batch. This iterative process allows the system to fine-tune its detection capabilities over time, ensuring that it can handle evolving threats with greater accuracy. By incorporating dynamic adjustments to model parameters and decision boundaries, the system can optimize its responses based on real-time data, effectively balancing computational efficiency and detection accuracy. By ensuring that only the most suspicious data points undergo the most computationally intensive analyses, our multi-stage framework optimizes the use of computational resources, balancing accuracy and efficiency. The feedback mechanism between stages not only improves the models' performance over time but also allows the system to quickly adapt to new and evolving threats. The main contributions of this paper are as follows.

- We propose an adaptive multi-stage NIDS that hierarchically detects suspicious traffic, optimizing resource use at each stage.
- We integrate meta-computing with intrusion detection, dynamically adjusting parameters and features to prevent performance declines in IDS implementations.
- We introduce two approaches, *Feedback Loop* and *Adaptive Threshold Tuning*, that dynamically adjust model parameters to enhance detection accuracy and efficiency. This enhances the system's adaptability and efficiency without incurring additional latency.
- Extensive experiments show our adaptive framework outperforms static IDS, offering a scalable solution for large-scale network security.

2. Background

2.1. Network Intrusion Detection (NIDS)

A Network Intrusion Detection System identifies potential security breaches and communicates alerts through various means such as textual alerts, log files, or graphical user interfaces. These alerts are subsequently analyzed by human analysts or automatic postprocessing systems. Accurate identification of intrusions is classified as true positives, while false alarms are false positives. Failure to detect an intrusion results in a false negative, whereas correctly identifying no breach is a true negative. Deep Learning approaches [7, 8] have shown significant promise in enhancing IDS performance, particularly in terms of accuracy and load balancing [2, 4, 5]. However, deploying complex DL models on resource-constrained devices poses challenges due to their high computational demands.

Niknami *et al.* [9] proposed a Reinforcement Learning (RL)-based IDS with an adaptive distributed sampling (ADS) to enhance anomaly detection accuracy while minimizing the increase in controller overhead. They provided a controller leveraging information gathered from each sampled traffic flow to determine whether the flow's state is malicious, suspicious, or benign based on underlying anomaly detection algorithms. Once the flow state is determined, the controller takes the appropriate action with the help of the RL agent. Yazdinejadna *et al.* [10] introduced a zone-based architecture for intrusion detection that enhances scalability and anomaly detection capabilities in NIDS. Zhao *et al.* [11] developed a lightweight IDS model with reduced detection rates. Wang *et al.* [12] proposed a knowledge distillation model to reduce model complexity, though designing effective teacher and student models remains challenging.

Yang *et al.* [13] presented a lightweight intrusion detection method that balances accuracy and efficiency through self-knowledge distillation. Ge *et al.* [14] introduced MetaCluster, a flexible classification framework for cybersecurity that filters and combines classification semantics through feature prototypes and dynamic graph learning layers, providing a comprehensive solution for interpretable classification tasks. Niknami *et al.* in [15] proposed a Prototypical Network-based IDS within a meta-learning framework. Their method adopts a Few-Shot Learning approach, aiming to distinguish and compare network traffic samples to classify them as either normal or malicious. The model not only identifies benign or malicious traffic but also accurately identifies the specific types of attacks. Their analysis showed that PTN-IDS, particularly with 5-shot learning, significantly outperformed the baseline method across different scenarios.

2.2. Multi-Stage Intrusion Detection System

Multi-stage intrusion detection systems are designed to detect complex, evolving attacks by breaking down the detection process into multiple layers or stages. Each stage analyzes network traffic or system behavior with increasing levels of granularity, allowing for a more thorough examination of potential threats. By using multiple detection techniques at different stages, these systems can identify various attack vectors that might be missed by single-stage IDS. Multi-stage IDS are particularly effective at handling advanced persistent threats (APTs) and multi-step attacks, as they track the progression of suspicious activities over time, ensuring early detection and response. Additionally, the layered approach enhances resource efficiency by filtering out benign traffic in earlier stages, focusing computational power on more likely threats in the later stages.

Hocine *et al.* [16] described a collaborative NIDS based on a multi-agent framework, using dynamic load balancing of traffic analysis to enhance detection, particularly against DDoS attacks, while minimizing excessive communication. Niknami *et al.* [17] proposed deploying a chain of IDSs within the data plane, interconnected with switches, to efficiently group data flows and balance the load on the controller. Verkerken *et al.* [18] introduced a multistage, scalable IDS capable of detecting unknown and zero-day attacks.

Niknami *et al.* in [19] proposed a multi-stage network attack detection algorithm that enhances Multi-Stage-Attack (MSA) detection through the analysis of high-dimensional alerts and the integration of various alert aspects. They incorporated semantic similarity, anomaly scores, and feature extraction to identify relevant entities, detect intricate relationships, and uncover hidden patterns, enhancing the detection of multi-stage network attacks.

To tackle the challenge of balancing detection accuracy with model complexity, which often leads to increased CPU usage, this paper proposes a lightweight, multistage NIDS. By leveraging meta-computing, we introduce dynamic resource allocation for each level of the NIDS. In addition, we present a novel feature selection method aimed at reducing the complexity of the model.

2.3. Meta-computing

Meta-computing entails the dynamic and adaptive management of computing resources to optimize performance, efficiency, and resource utilization [20]. A critical aspect of meta-computing is the dynamic allocation of resources based on workload demands and system conditions, which ensures efficient use of available resources and adaptation to changing computational needs. By leveraging meta-computing, computing resources are utilized in a flexible and efficient manner to effectively address computational challenges. When combined with few-shot learning, meta-computing techniques can further enhance resource allocation, improve parallel processing capabilities, and boost overall computational efficiency.

Y. Liu *et al.* in [21] addresses the deployment of machine learning-based intrusion detection systems in environments with limited resources. It proposes a framework for optimal task assignment and capacity allocation, with the aim of improving detection accuracy while efficiently utilizing available resources. This work aligns with meta-computing principles by focusing on the dynamic allocation of tasks and resources to optimize system performance. A. Gupta *et al.* in [22] explores resource allocation strategies for intrusion detection systems within edge computing environments. It introduces a fair resource allocation mechanism that balances detection performance with the computational limitations of edge nodes, ensuring efficient and equitable resource distribution. This study reflects the meta-computing approach of integrating and managing diverse computing resources to achieve optimal performance.

3. Methodology

The proposed framework incorporates an adaptive approach to intrusion detection, iteratively improving its performance with each incoming batch of data. This dynamic adaptation is facilitated through the use of meta-computing principles, which allocate resources based on the current load and complexity of the incoming network traffic. The hierarchical structure of Meta-IDS ensures that each level of the detection process is tailored to handle varying scales of traffic efficiently, thereby optimizing the use of computational resources. This design allows for a robust and scalable intrusion detection system capable of maintaining high accuracy and efficiency in detecting threats within high-performance environments.

In the proposed approach, the detection process is divided into three distinct stages. The first stage employs a simple binary model to quickly filter out benign traffic, thus reducing the load on subsequent stages. If a data point is deemed suspicious, it progresses to the second stage, where a multi-class model performs a more detailed analysis to classify the traffic into known attack categories or benign. The final stage involves a complex neural network model that conducts on in-depth analysis of data points that remain ambiguous after the first two stages. This multi-tiered approach ensures that only the most computationally intensive analyses are reserved for the most suspicious data points, thereby optimizing overall system performance. The multi-stage detection framework is designed to progressively refine the analysis of network traffic. In the initial stage, a simple binary classification model quickly filters out benign traffic, reducing the load on subsequent stages. The second stage involves a more detailed multi-class classification, identifying specific types of attacks. Finally, the third stage utilizes a complex neural network to analyze the remaining ambiguous data points. This hierarchical approach ensures that only the most computationally intensive analyses are performed on the most suspicious data, optimizing overall system performance.

To enhance the system's robustness, we employ techniques such as combining multiple feature selection methods and utilizing deep learning-based feature selection. This combination allows us to leverage the strengths of each method, ensuring comprehensive and accurate feature selection.

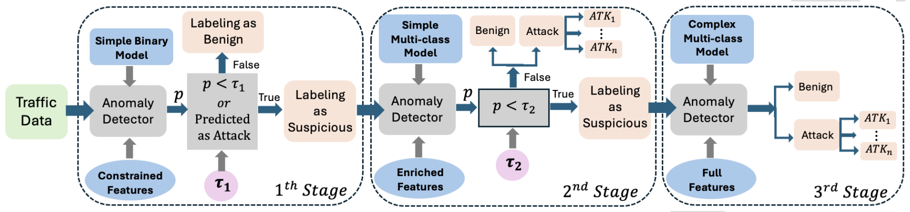


Figure 2: The architecture of the multi-stage hierarchical IDS.

3.1. Feature Selection for Enhanced Detection

The effectiveness of the IDS framework is further enhanced by employing advanced feature selection techniques. Niknami *et al.* in [6] integrated various feature selection methods such as mutual information [23], Minimum Redundancy Maximum Relevance [24], and Sparse Sensor Placement Optimization for Reconstruction [25] to identify the most relevant features for a multi-stage IDS. PCA as a feature extraction algorithm exhibited the highest accuracy compared to different feature selection methods, particularly for both binary and multi-class classification tasks. Their results showed that feature selection has a significant impact on the model's ability to classify both binary and multi-class data, with some methods outperforming others depending on the complexity of the task. Unlike other methods where the chosen features can be readily applied through straightforward slicing operations, PCA necessitates transforming the original data into a new coordinate system via matrix multiplication. This process, inherently more computationally demanding, leads to increased processing time when applied to new data. Consequently, while PCA can effectively reduce complexity and retain important information, its slower computational speed compared to feature selection methods led us to exclude it from our experiment.

In addition, the quality of features is equally crucial in influencing detection accuracy. For instance, approximately 35 features are adequate to achieving satisfactory accuracy in binary classification, while a minimum of 50 features is necessary for acceptable accuracy in multi-class classification. This discrepancy highlights the greater complexity and the challenge associated with multi-class classification compared to binary classification. Therefore, we adopt the feature selection method that yields the highest accuracy for a given number of features.

Fig. 2 illustrates the multi-stage hierarchical architecture IDS, *Meta-IDS*[6]. The architecture is structured into three stages, each contributing

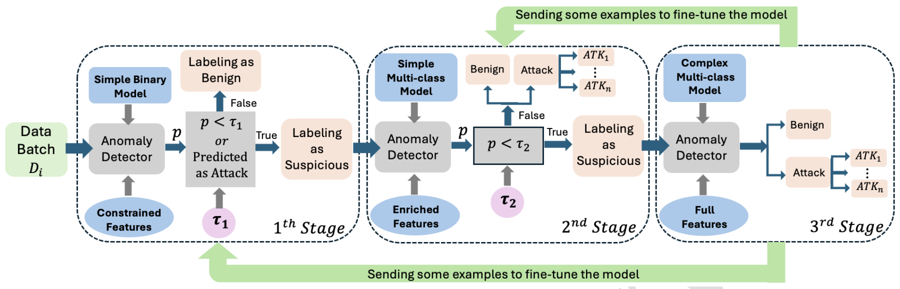


Figure 3: The architecture of the multi-stage hierarchical IDS with feedback loop.

to the detection and classification of network intrusions with increasing complexity and accuracy. In the first stage, traffic data is fed into an anomaly detector, which operates on constrained features. The system evaluates whether the probability p of an event being an attack is less than a predefined threshold τ_1. If $p < \tau_1$ or if the event is directly predicted as an attack, the event is labeled as suspicious. Otherwise, it is labeled as benign and does not proceed further in the detection pipeline. Suspicious events are passed to the second stage for more refined analysis. The second stage re-evaluates suspicious events using a more complex anomaly detector operating on enriched features. Here, the probability p is compared against a second threshold τ_2. If $p < \tau_2$, the event continues to be labeled as suspicious. A higher p triggers classification into either a benign category or one of several attack categories (denoted as ATK_1 to ATK_n). In the third Stage, the system performs the most detailed analysis using a complex multi-class model, leveraging full feature sets and an advanced anomaly detector. This stage thoroughly classifies the event as either benign or as one of several potential attack types (again denoted as ATK_1 to ATK_n). The output represents the most accurate and detailed assessment within the system.

To further improve the performance of Meta-IDS, we propose two dynamic approaches: *Feedback Loop* and *Adaptive Threshold Tuning*. These methods adjust the model parameters in real-time to optimize detection accuracy and efficiency. Below, we provide detailed explanations of each approach.

3.2. Enhanced Meta-IDS: Feedback Loop

In the proposed *Feedback Loop* approach, *Meta-IDS* is equipped with a feedback loop, which allows information from the last (third) stage to be fed back into earlier stages. This feedback improves detection accuracy by refining thresholds τ_1 and τ_2, or fine-tuning the earlier stage models. Fig. 3 illustrates the *Enhanced Meta-IDS*. Feedback loop mechanism is designed to prioritize low-confidence samples, which naturally includes the most severe misclassification cases. The feedback loop addresses edge cases by: 1) Retraining on low-confidence samples to improve generalization and 2) Dynamically adjusting thresholds to handle uncertain scenarios. By selecting these cases for retraining and fine-tuning earlier-stage models, the system systematically improves its ability to handle challenging scenarios. As demonstrated in our results, the accuracy progressively improves with each task, indicating that not only are previously misclassified samples addressed, but new samples with similar distributions to these challenging cases are now correctly classified.

The hierarchical approach divides the detection process into four Phases:

- **Phase 1 (Simple IDS in Stage 1)**: Simple binary classification to quickly filter out benign traffic.
- **Phase 2 (Moderate IDS in Stage 2)**: Detailed multi-class classification to identify specific types of attacks.
- **Phase 3 (Complex IDS in Stage 3)**: Complex neural network analysis for ambiguous data points, ensuring comprehensive intrusion detection.
- **Phase 4 (Feedback Loop)**: Following the evaluation of our multi-stage model on each data batch, we will implement a feedback loop with fine-tuning the earlier stage models to enhance the model's efficiency and accuracy.

Overall, this hierarchical approach ensures a scalable and accurate intrusion detection system. By progressing from simple to complex analyses across multiple stages, and incorporating a feedback loop that refines the performance of each stage, the system effectively balances computational efficiency with detection accuracy. This feedback mechanism enhances the decision-making process at each stage, reducing false positives and improving the system's ability to identify even the most subtle attack patterns, thereby optimizing overall performance.

Fig. 4 illustrates the process of data by the traditional Meta-IDS model and an enhanced version of it. In the first approach, the Meta-IDS model

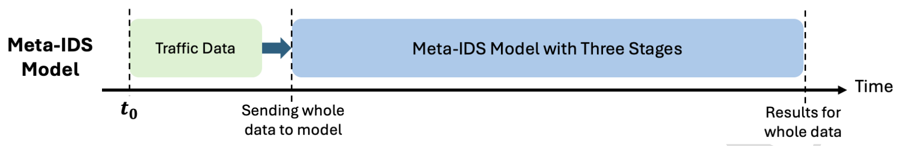


Figure 4: The timeline of Meta-IDS approach.

processes the entire dataset in one go. At an initial time, t_0, the complete traffic data is sent to the model. This data passes through a pipeline consisting of three stages, which are not detailed in the figure, and the system generates results only after the entire dataset has been processed. While this method can handle large-scale data, it processes everything at once, potentially leading to resource inefficiency and a long wait time before results are available.

Fig. 5 represents the *Enhanced Meta-IDS* model, where data arrives sequentially, requiring the method to dynamically adapt and process the data in smaller, incremental batches. At time t_0, the model processes the first data batch D_i, producing results for that specific batch. After this, an updating process occurs, allowing the model to adapt based on the newly processed data. Once updated, the system moves to the next batch D_{i+1}, repeating this process for subsequent tasks. This approach enables the model to continually evolve with the incoming data, making it more adaptive and flexible compared to the traditional method. By processing data in small batches and updating the model frequently, the *Enhanced Meta-IDS* ensures that the model remains robust and capable of detecting new threats in real-time. Each task is treated as a separate learning process, contributing to the overall system performance by ensuring that previous knowledge is retained while adapting to new information.

The continuous learning strategy depicted in Fig. 5 allows the *Enhanced Meta-IDS* to excel in dynamic environments where threats and data patterns are constantly changing. Unlike static models that require retraining with large datasets, this incremental approach ensures efficiency in resource-constrained environments while maintaining high detection accuracy. Through this cyclical process of receiving new data batches, updating the model, and applying the enhanced model to future tasks, the system stays resilient and agile in a fast-evolving cybersecurity landscape. The key difference between these two methods is that while the Meta-IDS model processes the entire dataset in one go, the *Enhanced Meta-IDS* model processes data incremen-

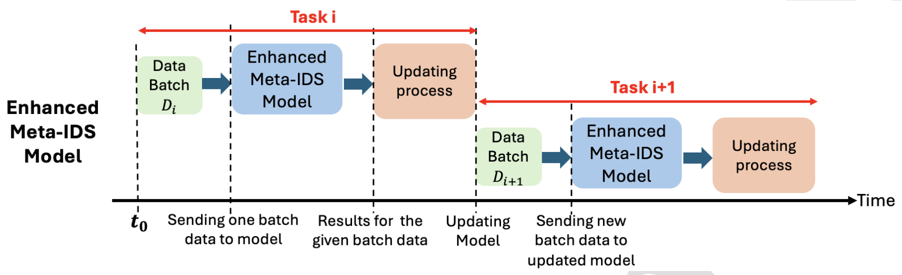


Figure 5: The timeline of *Enhanced Meta-IDS* approach.

tally. This allows the system to be more responsive and scalable, particularly in environments where data is continuously arriving and real-time adaptation is necessary.

The *Enhanced Meta-IDS* algorithm, outlined in Algorithm 1. Given a sequence of tasks $\{T_1, T_2, \ldots, T_k\}$, where each task $T_j = \{(x_1, y_1), ..., (x_N, y_N)\}$ consists of input features x and the corresponding label y, the Meta-IDS algorithm employs a multi-stage process using feature sets $\mathcal{F}_1$ for *Stage 1* and $\mathcal{F}_2$ for *Stage 2*, with thresholds τ_1 and τ_2. In *Stage 1*, each data point x_i is processed by a binary classifier (`BinaryModel`) using $\mathcal{F}_1$. If the probability of malicious behavior is below τ_1 or explicitly classified as malicious, the algorithm advances to *Stage 2*. In *Stage 2*, a simple multi-class model (`SimpleMultiClassModel`) with $\mathcal{F}_2$ is applied. If the classification probability p_2 is below τ_2, the algorithm moves to *Stage 3*, where a complex multi-class model (`ComplexMultiClassModel`) is used for the final prediction. Otherwise, the prediction from the simple multi-class model is used. This sequential approach ensures efficient and accurate intrusion detection.

After processing through all the stages (binary, simple multi-class, and complex multi-class models) for a task T_j, a feedback loop is introduced by calling the `Feedback Loop for Fine-tuning Models` function. This function is responsible for updating the models based on the results from *Stage 3*. `Feedback Loop for Fine-tuning Models` function outlined in Algorithm 2. The procedure takes as input the predicted labels from three stages of the IDS: $\hat{Y}_1$, $\hat{Y}_2$, and $\hat{Y}_3$, along with their corresponding prediction probabilities P_1 and P_2. The output of the process is fine-tuned models at *Stage 1* and *Stage 2*. In Step 1, the algorithm begins by ranking the predicted labels from *Stage 1* and *Stage 2* based on their probabilities. From $\hat{Y}_1$, it selects the lowest N samples according to the ranked probabilities in

Algorithm 1 Enhanced Meta-IDS Algorithm

```
1: Input: Tasks $T = \{T_1, T_2, ..., T_k\}$, $T_j = \{(x_1, y_1), ..., (x_N, y_N)\}$, Feature sets $F_1, F_2, F_3$, Thresholds $\tau_1, \tau_2$
2: Output: Predicted Labels $\hat{Y} = \{\hat{y}_1, ..., \hat{y}_N\}$
3: Step 1: Evaluate model on each task
4: for each task $T_j$ in $T$ do
5:     for each data point $x_i$ in task $T_j$ do
6:         $p_1, c_1 \leftarrow$ BinaryModel($x_i, F_1$)
7:         if $p_1 < \tau_1$ or $c_1$ is malicious then
8:             $p_2, c_2 \leftarrow$ SimpleMultiClassModel($x_i, F_2$)
9:             if $p_2 < \tau_2$ then
10:                $p_3, c_3 \leftarrow$ ComplexMultiClassModel($x_i, F_3$)
11:                $\hat{y}_i \leftarrow c_3$
12:            else
13:                $\hat{y}_i \leftarrow c_2$
14:            end if
15:        else
16:            $\hat{y}_i \leftarrow c_1$
17:        end if
18:    end for
19:    Step 2: Fine-tune models using the feedback loop
20:    Call Algorithm 2 to fine-tune models using obtained results from $T_j$
21: end for
```

P_1, forming the set S_1. Similarly, from $\hat{Y}_2$, it selects the lowest N samples based on the ranked probabilities in P_2, forming the set S_2. These samples represent predictions where the models showed low confidence, indicating a higher likelihood of misclassification.

In Step 2, the algorithm proceeds to improve the models by leveraging the final predictions from *Stage 3*. For each sample s_i in S_1, the algorithm finds the associated label y_i in $\hat{Y}_3$, the output of *Stage 3*. The sample s_i along with its corresponding label y_i is then sent back to the *BinaryModel* at *Stage 1* for retraining. Similarly, for each sample s_i in S_2, the algorithm finds the corresponding label in $\hat{Y}_3$ and sends the sample-label pair back to the *SimpleMultiClassModel* at *Stage 2* for retraining. In Step 3, the algorithm performs the fine-tuning of the models. Both the *BinaryModel* and the *SimpleMultiClassModel* are retrained using the selected N samples and

Algorithm 2 Feedback Loop for Fine-tuning Models

```
1: Input: Predicted labels $\hat{Y}_1$, $\hat{Y}_2$, $\hat{Y}_3$ and Probabilities $P_1$, $P_2$
2: Output: Fine-tuned models at Stages 1 and 2
3: Step 1: Rank and select low-confidence samples
4: $S_1 \leftarrow N$ samples from $\hat{Y}_1$ with lowest $P_1$
5: $S_2 \leftarrow N$ samples from $\hat{Y}_2$ with lowest $P_2$
6: Step 2: Find associated Labels in Stage 3
7: for each sample $s_i \in S_1$ do
8:     $y_i \leftarrow$ Associated label for $s_i$ in $\hat{Y}_3$
9:     Send $(s_i, y_i)$ to BinaryModel for training Stage 1
10: end for
11: for each sample $s_i \in S_2$ do
12:     $y_i \leftarrow$ Associated label for $s_i$ in $\hat{Y}_3$
13:     Send $(s_i, y_i)$ to SimpleMultiClassModel in Stage 2
14: end for
15: Step 3: Updating Models with $N$ Trained Samples
16: Fine-tune the Binary and Simple Multi-Class models with corresponding
    selected $N$ samples
```

their associated labels from *Stage 3*. This targeted retraining is intended to improve the performance of the models by focusing on the low-confidence samples where the models had previously struggled, thereby enhancing the models' ability to correctly classify similar samples in future predictions.

3.3. Enhanced Meta-IDS: Adaptive Threshold Tuning

In this section, we explore an alternative approach for implementing the feedback loop. As demonstrated in our previous work [6], the thresholds in the first and second stages significantly influence the efficiency and accuracy of Meta-IDS. Therefore, after evaluating the model on each task, we aim to update the thresholds to enhance the model's utility. Since there are no known analytical relationships between the utility and the thresholds, we propose using a greedy algorithm as the most practical approach for this optimization. The approach involves two thresholds: one is kept fixed, while the other is incrementally adjusted to optimize the utility. We define two thresholds, τ_1 and τ_2, and the two scenarios are as follows:

1. The value of τ_1 is fixed, and τ_2 is incrementally adjusted by a step size $\Delta_{\tau_2} = 0.2$.

- **Initialization:** Set the initial values of τ_1 and τ_2.
- **Increase Step:** Increment τ_2 by the step size $\Delta_{\tau_2} = 0.2$, and evaluate the *Utility*.
- **Utility Evaluation:**
  - If *Utility* improves, increase τ_2 by Δ_{τ_2} for the next task and repeat the evaluation.
  - If *Utility* decreases, decrease τ_2 by either a half of the step size $\Delta_{\tau_2} = 0.1$ or a quarter of the step size, $\Delta_{\tau_2} = 0.05$, and re-evaluate the utility.
  - **Termination:** The process is terminated if any of the following criteria are met:
    * **Upper Bound Condition**: If the τ_2 reaches an upper bound and still improving, stop further increases and retain the current threshold, as no further improvement is expected beyond this point.
    * **Convergence Condition**: If utility improves after the decrease, stop adjusting the tau_2 and retain the current value.
    * **Cycle Condition**: If the τ_2 adjustments lead back to a previously evaluated value without yielding further improvement, terminate the adjustment process and retain the τ_2 value at which the utility was highest.

2. The value of τ_2 is fixed, and τ_1 is incrementally adjusted by a step size $\Delta_{\tau_1} = 0.2$.
   - **Initialization:** Set the initial values of τ_1 and τ_2.
   - **Increase Step:** Increment τ_1 by the step size $\Delta_{\tau_1} = 0.2$, and evaluate the *Utility*.
   - **Utility Evaluation:**
     - If *Utility* improves, increase τ_1 by Δ_{τ_1} for the next task and repeat the evaluation.
     - If *Utility* decreases, decrease τ_1 by either a half of full step size $\Delta_{\tau_1} = 0.1$ or a quarter of the step size, $\Delta_{\tau_1} = 0.05$, and re-evaluate the utility.
     - **Termination:** The process is terminated if any of the following criteria are met:

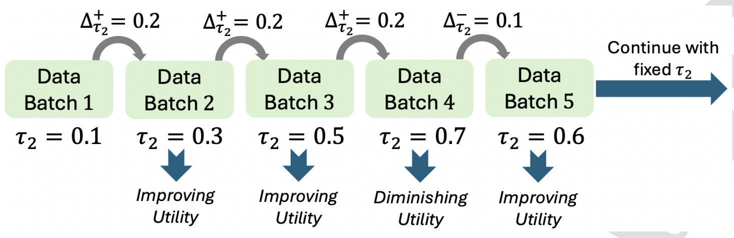


Figure 6: Adjusting τ_2 when the value of τ_1 is fixed to 0.5.

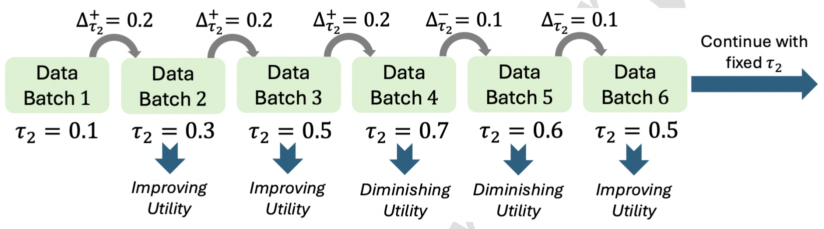


Figure 7: Adjusting τ_2 when the value of τ_1 is fixed to 0.7.

* **Upper Bound Condition**: If the τ_1 reaches an upper bound and still improving, stop further increases and retain the current threshold, as no further improvement is expected beyond this point.
* **Convergence Condition**: If utility improves after the decrease, stop adjusting the tau_1 and retain the current value.
* **Cycle Condition**: If the τ_1 adjustments lead back to a previously evaluated value without yielding further improvement, terminate the adjustment process and retain the τ_2 value at which the utility was highest.

This adaptive threshold tuning process aims to iteratively adjust the threshold τ_1 or τ_2 to maximize the utility function. The approach provides flexibility in fine-tuning the threshold while minimizing the risk of overshooting the optimal value. Note that τ_1 is associated with a binary classification problem and is therefore bounded below by 0.5, while τ_2 corresponds to a multi-class classification problem and is bounded below by 0.1. Additionally, since values greater than 0.9 are inefficient—leading to nearly all samples being flagged as suspicious and requiring further processing—we impose an upper bound of 0.9 on both τ_1 and τ_2.

Figs. 6 and 7 illustrate the iterative process of adjusting τ_2 across data batches to optimize utility. In both figures, the adjustment begins with an initial value of $\tau_2 = 0.1$, followed by increments using a positive step size $\Delta^+_{\tau_2} = 0.2$ over the first three data batches. This adjustment leads to continuous improvements in utility, as indicated by the "Improving Utility" annotations. However, in Data Batch 4, further increasing τ_2 to 0.7 results in diminishing utility. To address this, a smaller step size $\Delta^-_{\tau_2} = 0.1$ is used to reduce τ_2 to 0.6 in Data Batch 5, which leads to an improvement in utility once again. In Fig. 6, after reaching this point, the value of τ_2 is fixed at 0.6 for subsequent tasks, as the optimal range for utility has been achieved. In Fig. 7, however, in Data Batch 5, setting τ_2 to 0.6 still results in diminishing utility. To address this, τ_2 is further reduced to 0.5 in Data Batch 6, which leads to an improvement in utility once again. These figures demonstrate the adaptive tuning of τ_2, where increasing and decreasing adjustments are made based on the observed utility trend to reach an optimal utility value.

3.4. Complexity Analysis

The implementation of the Enhanced Meta-IDS Algorithm and its associated feedback loop for fine-tuning models is inherently complex, as it heavily depends on data-specific factors such as the feature space, the distribution of low-confidence samples, and the underlying model architecture. These dependencies make a generalized complexity analysis challenging. However, a structured complexity analysis for the algorithm's feedback loop is feasible due to its hierarchical design. The complexity at each stage is described as follows:

- **Stage 1 (Binary Model):** The computational complexity for this stage is $O(n \cdot f_1)$, where n is the number of input samples, and f_1 is the number of features utilized at this stage.
- **Stage 2 (Simple Multi-Class Model):** The complexity is $O(m \cdot f_2)$, where m is the subset of samples forwarded from Stage 1, and f_2 represents the feature set at this stage, where $f_2 > f_1$.
- **Stage 3 (Complex Multi-Class Model):** The complexity is $O(p \cdot f_3)$, where p denotes the subset of samples passed from Stage 2, and f_3 is the feature set at this stage, where $f_3 > f_2$.

The feedback loop for fine-tuning models introduces an additional computational cost of $O(k \cdot (f_1 + f_2))$, where k represents the number of samples selected for fine-tuning. However, for adaptive threshold strategy, there is no extra computational cost. This mechanism refines the model iteratively, leveraging the feedback from previous predictions. The hierarchical structure of the Enhanced Meta-IDS Algorithm ensures that computational efficiency is maintained by early-stage filtering, which reduces the number of samples passed to the computationally intensive stages. Experimental results demonstrate that this design achieves a favorable trade-off between computational cost and accuracy, validating the practical feasibility of the approach.

4. Evaluation

Through extensive experiments and empirical evaluations, we have demonstrated that our *Enhanced Meta-IDS* framework achieves significant improvements in both accuracy and efficiency compared to traditional single-stage IDS models. By leveraging dynamic resource allocation, hierarchical processing, and advanced feature selection techniques, our system can effectively handle large-scale network traffic while maintaining high levels of detection accuracy and minimizing false positives. This makes our proposed Meta-IDS framework a robust and scalable solution for modern network security challenges. We conducted our experiments on a system equipped with an Intel(R) Xeon(R) W-2225 CPU @ 4.10GHz, featuring an $x86\_64$ architecture with 8 CPUs, each with 2 threads per core and 4 cores per socket, reaching a maximum frequency of 4.6 GHz. We conduct each experiment multiple times and then calculate the average to present the results. To evaluate our method, we conducted experiments using the CICIDS2017 dataset generated from real network traffic recordings. This dataset includes benign samples alongside the most up-to-date common attacks. The dataset was generated over a period of five days using 14 machines. The benign traffic was simulated based on the behavior of 25 individuals, utilizing statistical techniques and machine learning. In contrast, malicious traffic was generated by executing known attack tools within specific time windows. The preprocessing steps are exactly the same as what is done in [6].

After preprocessing, we simplified the dataset by grouping related attacks attack types from the CICIDS2017 dataset, and a label was assigned to each group. For instance, various Denial of Service (DoS) attacks, such as DoS Hulk and DoS Slowloris, were grouped under the label DoS. Similarly, Web attacks like Brute Force - XSS were grouped under Web Attack. This approach simplifies the dataset by consolidating different types of attacks into broader categories, making it more manageable for analysis and machine learning applications. For the binary model, we trained on 10% of the data, while the multi-class models were trained on just 2%. This reduced dataset size significantly minimized training times and CPU usage. For testing, we selected 1 million records not used during training, consisting of 500,000 benign instances and 500,000 attacks. To simulate larger traffic volumes, we processed a total of 8 million samples by repeatedly using this 1 million test dataset. To evaluate the proposed method, we considered two scenarios with different numbers of features in the first stage and second stage of our proposed method. The scenarios are as follows:

- *Scenario 1*: the number of features in *Stage 1* is $\mathcal{F}_1 = 5$ and the number of features in *Stage 2* is $\mathcal{F}_2 = 10$.
- *Scenario 2*: the number of features in *Stage 1* is $\mathcal{F}_1 = 10$ and the number of features in *Stage 2* is $\mathcal{F}_2 = 30$.

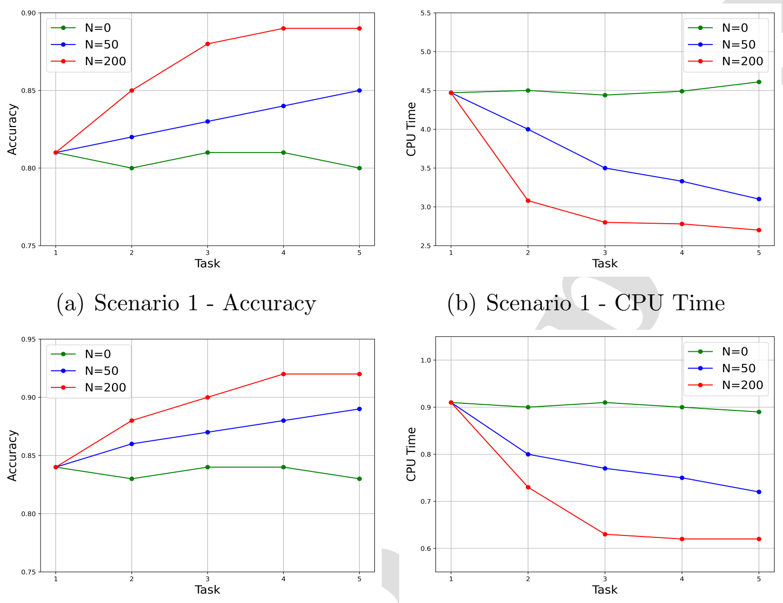
For the sake of simplicity, in this section, we will refer to a "Task" instead of a "Data Batch". We measured both *Accuracy* and *Inference time*. Inference time, in the context of deep learning, refers to the duration a trained model takes to generate predictions on new, unseen data. During this phase, the model processes the input data and produces an output, such as a classification or regression result. Inference time is a critical factor for deploying deep learning models in real-world applications, as it directly affects the system's speed and responsiveness. In addition, we define another measurement to display the effectiveness of the method based on both accuracy and CPU time. It is *Utility* function which is defined as follows:

$$Utility(\tau_1, \tau_2) = \begin{cases} Score(\tau_1, \tau_2), & \text{if } Accuracy(\tau_1, \tau_2) > 0.8 \\ 0 & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} Score(\tau_1, \tau_2) =& (W_{ACC} \times Accuracy(\tau_1, \tau_2)) \\ & -(W_T \times Time(\tau_1, \tau_2)). \end{aligned} \tag{1}$$

Here, W_{ACC} represents the weight of accuracy, and W_T denotes the weight of CPU time. These two parameters assign importance to the accuracy and CPU time in the utility calculation, respectively.



(a) Scenario 1 - Accuracy (b) Scenario 1 - CPU Time

(c) Scenario 2 - Accuracy (d) Scenario 2 - CPU Time

Figure 8: Accuracy and CPU Time across tasks for Scenario 1 and Scenario 2.

The number of training data points sent to *Stage 1* and *Stage 2* for updating the models are the effective parameters. We define N corresponding to the number of training data points used to update models, represented as:

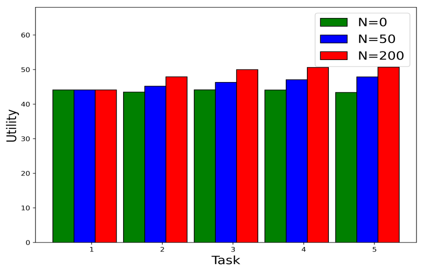
- $N = 0$: No feedback loop for updating the models.
- $N = 50$: Feedback loop with 50 instances.
- $N = 200$: Feedback loop with 200 instances

4.1. Impact of Feedback Loop Size on Accuracy and CPU Time

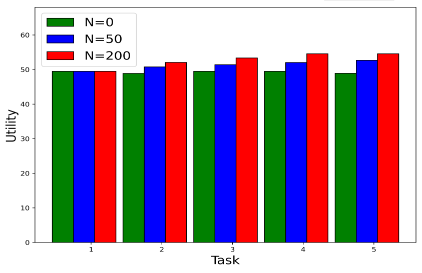
Fig. 8 displays how accuracy and CPU time change across five tasks in Scenario 1 and Scenario 2 under three different configurations: $N = 0$ (no feedback loop), $N = 50$ (moderate feedback loop), and $N = 200$ (larger feedback loop). For both scenarios, we considered thresholds as $\tau_1, \tau_2 = 0.5$. Fig. 8(a) shows as the progress of the task, accuracy improves for the configurations with feedback loop. The model with the $N = 200$ feedback loop achieves the highest accuracy, reaching about 0.9. The $N = 50$ configuration also shows steady improvement, ending with an accuracy of around 0.86. However, the $N = 0$ model, which has no feedback loop, performs the worst, maintaining accuracy between 0.80 and 0.82 without much improvement as tasks go on. Fig. 8(b) shows the amount of time needed by the model decreases as tasks progress. The model with the $N = 200$ feedback loop becomes more efficient, with CPU time dropping from 4.5 to 2.5 by Task 5. The $N = 50$ model also sees a reduction in CPU time, stabilizing at 3.5. On the other hand, the $N = 0$ configuration, with no feedback loop, doesn't improve much and stays around 4.5 in CPU time.

Fig. 8(c) shows that the $N = 200$ feedback loop leads to the highest accuracy, reaching nearly 0.95 by Task 5. The $N = 50$ model improves moderately, hitting around 0.88, while the $N = 0$ configuration stays at around 0.80, again showing minimal improvement. The two scenarios, Scenario 1 and Scenario 2, differ significantly in the number of features used at each stage of the model. In Scenario 1, there are fewer features, with 5 features in *Stage 1* ($\mathcal{F}_1 = 5$) and 10 features in *Stage 2* ($\mathcal{F}_2 = 10$). In contrast, Scenario 2 has a more complex feature space, with 10 features in *Stage 1* ($\mathcal{F}_1 = 10$) and 30 features in *Stage 2* ($\mathcal{F}_2 = 30$). The more features the model can leverage, the better it can generalize across tasks, leading to higher accuracy. The fact that Scenario 2 reaches an accuracy of nearly 0.95 with $N = 200$ suggests that the additional features help the model learn more effectively compared to Scenario 1, where accuracy caps at around 0.9.

Fig. 8(d) shows that CPU time decreases significantly for the $N = 200$ model, dropping from 0.9 to 0.5 by Task 5. The $N = 50$ model also sees a reduction in CPU time, ending at around 0.8. However, the $N = 0$ model remains constant, using about 0.9 CPU time throughout the tasks. In terms of CPU time, the larger number of features in Scenario 2 ($\mathcal{F}_1 = 10$ and $\mathcal{F}_2 = 30$) means that more computational resources are initially required. This is evident in the first tasks, where CPU time is generally higher for Scenario 2 than Scenario 1. However, as tasks progress, the feedback loop mechanism allows the models to reduce CPU time more effectively in Scenario 2, particularly for the $N = 200$ configuration, where the CPU time drops from 0.9 to 0.5. This suggests that while Scenario 2 begins with higher computational demands due to the greater number of features, the feedback loop enables more significant optimization as the model fine-tunes itself across tasks. In Scenario 1, CPU time reduction is less pronounced, likely because the smaller feature set limits the model's ability to optimize as dramatically.



(a) Scenario 1



(b) Scenario 2

Figure 9: Impact of feedback loop size on model utility.

In summary, models with larger feedback loops ($N = 200$) perform much better in terms of accuracy and become more efficient in CPU time as tasks progress. The models without feedback ($N = 0$) perform the worst in both accuracy and CPU time. Feedback loops help the models improve both their performance and efficiency as they tackle more tasks. In addition, we see trade-offs between feature complexity and computational efficiency. Scenario 1, with fewer features, has lower initial CPU times, especially for the $N = 0$ and $N = 50$ configurations, which remain relatively constant throughout the tasks. While Scenario 1's simplicity results in lower CPU demands, it also limits the model's accuracy potential, as seen in the lower accuracy scores compared to Scenario 2. On the other hand, Scenario 2's higher feature count initially increases computational demand but leads to better accuracy and more efficient CPU usage over time, especially when the feedback loop is employed effectively.

4.2. Impact of Feedback Loop Size on Utility

Throughout the paper, we set the utility weights as $W_{ACC} = 60$ and $W_T = 1$. As demonstrated in Figure 8, our methodology enhances both accuracy and computational time. Therefore, regardless of the specific values chosen for W_{ACC} and W_{T}, our algorithm remains superior. The weights $W_{\mathrm{ACC}} = 60$ and $W_{\mathrm{T}} = 1$ prioritize detection accuracy over computational efficiency, reflecting the importance of minimizing false negatives. Step sizes $(\Delta_{\tau_1}, \Delta_{\tau_2})$ were chosen for practical convergence.

Fig. 9 displays the impact of Feedback Loop Size on Model Utility Across Tasks in Scenario 1 and Scenario 2 under different number of training data points. In Scenario 1, the utility values vary depending on the feedback loop size across the tasks. For Task 1, the utility for all configurations ($N = 0$, $N = 50$, and $N = 200$) remains similar, with values just below 50. The reason is that the feedback loop has not yet been executed. From Task 2 onward, there is a clear upward trend as the feedback loop size increases. $N = 200$ consistently achieves the highest utility across all tasks, particularly in Task 4 and Task 5, where it exceeds 50. The $N = 0$ case, where no feedback loop is used, consistently performs the worst, suggesting the crucial role of feedback in improving model utility. $N = 50$ shows better performance than $N = 0$ but does not reach the level of $N = 200$, indicating that a moderate feedback loop improves performance, though not as much as a larger one. In Scenario 2, the pattern is similar to Scenario 1, but with some variations in utility. For Task 1, the utility values for all configurations remain close, slightly below 50. In Tasks 2 and 3, the utility across different configurations stays consistent, with $N = 200$ performing marginally better. By Task 4 and Task 5, the differences between $N = 0$, $N = 50$, and $N = 200$ become more pronounced, with $N = 200$ achieving the highest utility, around 55, while $N = 50$ shows a moderate improvement over $N = 0$.

Overall, larger feedback loops, particularly $N = 200$, lead to consistently higher utility in both scenarios, especially in later tasks, indicating that feedback significantly enhances model performance as the number of instances increases. Models without feedback loops, represented by $N = 0$, result in lower utility, emphasizing the importance of incorporating feedback for model improvement. Although $N = 50$ improves performance compared to $N = 0$, it does not reach the performance levels of $N = 200$, highlighting the benefit of larger feedback loops in maximizing utility across tasks.

4.3. Fine-tuning Time vs. Average Accuracy

Fig. 10 compares two variables across four scenarios: fine-tuning time and average accuracy. The overall fine-tuning time remains consistent for each stage since the number of samples fine-tuned is the same across stages. By dividing the total fine-tuning time by the number of tasks, one can estimate the time required per stage. Furthermore, the fine-tuning process is designed to occur during intervals between tasks, ensuring it does not interfere with real-time system operation. Therefore, this overhead was not included in the evaluation of our methodology. For scenarios where such intervals are not available, the second algorithm (adaptive threshold tuning) can be employed as an alternative to maintain efficiency without introducing additional fine-tuning delays.

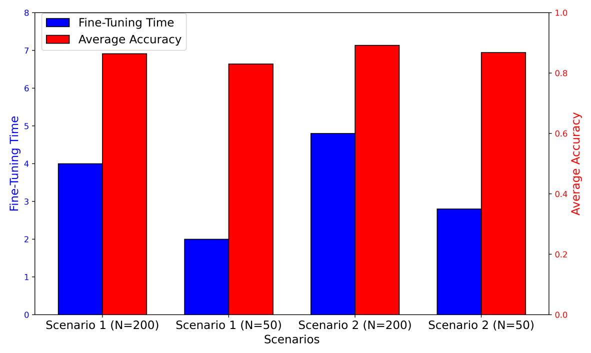


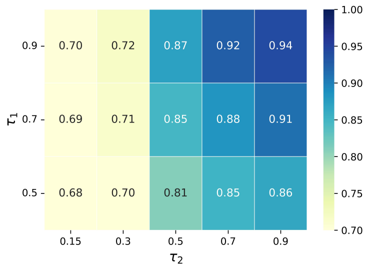
Figure 10: Comparison of Fine-Tuning Time and Average Accuracy across different scenarios and different N.

In Scenario 1 with 200 training data points ($N = 200$), the fine-tuning time is approximately 4.2, and the average accuracy reaches around 0.87. When the number of training data points is reduced to 50 ($N = 50$), the fine-tuning time drops to just under 2, while the average accuracy decreases slightly to about 0.82. In Scenario 2 with $N = 200$, the fine-tuning time increases to just over 5, and the accuracy improves, reaching close to 0.90. Similarly, in Scenario 2 with $N = 50$, the fine-tuning time drops to just under 1, but the average accuracy remains strong at around 0.85.

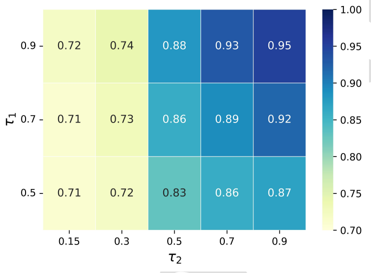
A notable observation is that Scenario 2 consistently requires more fine-tuning time compared to Scenario 1, suggesting that it demands more computational effort. However, it also results in higher average accuracy. Additionally, larger training data points ($N = 200$) tend to increase both the fine-tuning time and accuracy, demonstrating the positive impact of more training data on performance. Overall, Scenario 2 with $N = 200$ achieves the highest accuracy but at the cost of increased fine-tuning time, while smaller datasets require less time but generally offer slightly lower accuracy.

4.4. Impact of N on Accuracy with Varying τ_1 and τ_2

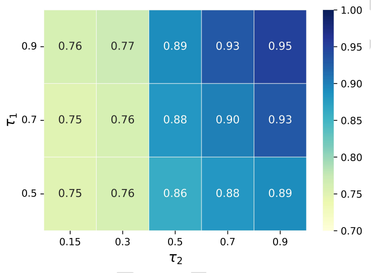
Figs. 11 show the average accuracy under varying values of N ($N = 0$, $N = 50$, and $N = 200$) and different threshold combinations of τ_1 and τ_2. In the case where $N = 0$, shown in Fig. 11(a), the accuracy ranges from 0.68 to 0.94, with the highest accuracy achieved when $\tau_1 = 0.9$ and $\tau_2 = 0.9$. The results indicate that the models perform better when higher thresholds are used, but lower values of τ_2 result in reduced accuracy, especially for smaller



(a) $N = 0$



(b) $N = 50$

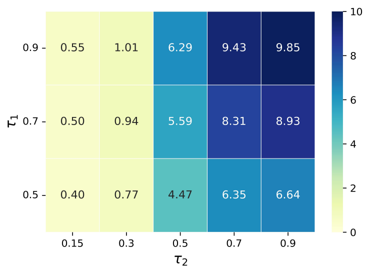


(c) $N = 200$

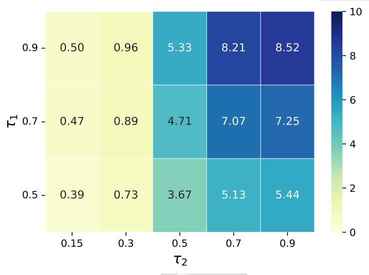
Figure 11: Average accuracy over 5 tasks for different N values in scenario 1.

values of τ_1. This suggests that without the feedback mechanism, the model struggles to generalize well across the tasks, particularly when the threshold values are set lower. Fig. 11(b), where $N = 50$, the accuracy ranges from 0.71 to 0.95. The most substantial improvements are observed in the low threshold ranges, such as $\tau_1 = 0.5$ and $\tau_2 = 0.15$, where accuracy increases to 0.71 compared to 0.68 when $N = 0$. This demonstrates that selecting 200 low-confidence samples for retraining in the feedback loop enhances the model's ability to generalize, especially for tasks that were more difficult to classify without the benefit of retraining.

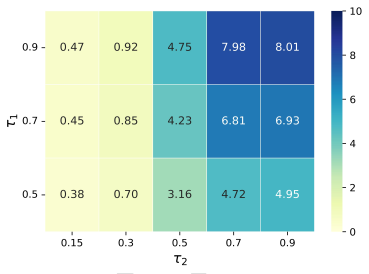
Fig. 11(c), where $N = 200$, the accuracy in this scenario ranges from 0.75 to 0.95, demonstrating further improvements in overall performance. For lower threshold values, such as $\tau_1 = 0.7$ and $\tau_2 = 0.7$, the accuracy reaches 0.90, showing that the feedback loop has significantly helped the model perform better across the board. However, the marginal gains between $N = 50$ and $N = 200$ are smaller compared to the improvements seen from $N = 0$ to $N = 50$. This suggests that while the feedback loop continues to improve



(a) $N = 0$



(b) $N = 50$



(c) $N = 200$

Figure 12: Average CPU Time over 5 tasks for different N values in scenario 1.

performance as more samples are added, the largest performance gains occur with the initial introduction of feedback. It can be observed that the highest accuracy is consistently achieved at $\tau_1 = 0.9$ and $\tau_2 = 0.9$, regardless of the value of N. This indicates that the models perform optimally when the thresholds for classification are set to higher values. The general trend across the experiments is that as the value of N increases, the accuracy improves. This clearly demonstrates the benefit of using the feedback loop for retraining the models, particularly when the models exhibit low confidence in their predictions. As more low-confidence samples are used for retraining, the models are better able to generalize, improving overall accuracy.

4.5. Impact of N on CPU Time with Varying τ_1 and τ_2

Figs. 12 show the average CPU time under varying values of N ($N = 0$, $N = 50$, and $N = 200$) and different threshold combinations of τ_1 and τ_2. In Fig. 12(a), where $N = 0$, the CPU time ranges from 0.40 to 9.85 seconds, with the highest CPU time occurring at $\tau_1 = 0.9$ and $\tau_2 = 0.9$. The CPU time is higher for larger values of τ_1 and τ_2, which likely corresponds to more complex model evaluations as the threshold values increase.

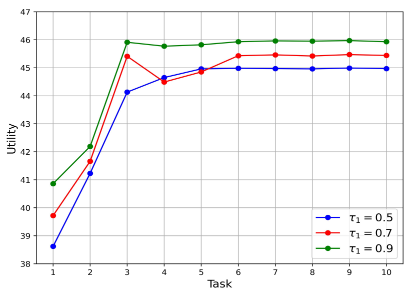
Fig. 12(b) shows the results when $N = 50$. The CPU time now ranges from 0.39 to 8.52 seconds. There is a noticeable reduction in CPU time, particularly for higher threshold values like $\tau_1 = 0.9$ and $\tau_2 = 0.9$, compared to the case when $N = 0$. This could indicate that the feedback loop has optimized certain areas of the model, reducing the need for extensive computations during task execution. However, the overall trend still indicates that higher threshold values correspond to higher CPU times. In Fig. 12(c), where $N = 200$, the CPU time ranges from 0.38 to 8.01 seconds. The highest CPU time, again, is observed at $\tau_1 = 0.9$ and $\tau_2 = 0.9$, although it has decreased compared to both $N = 0$ and $N = 50$. The reduction in CPU time when $N = 200$ suggests that the feedback loop, by retraining with a larger number of low-confidence samples, enhances the model's efficiency in processing tasks. As a result, the model requires less computational time to handle complex tasks, which leads to reduction in CPU time.

It is evident that increasing N leads to a reduction in CPU time, especially at higher threshold values. The feedback loop improves the model's ability to perform the tasks more efficiently, thereby reducing the CPU time required. The general trend suggests that the feedback loop helps optimize the model, with the most significant reductions in CPU time occurring at higher values of N.

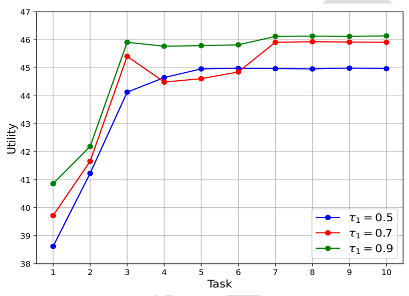
4.6. Impact of Adaptive Threshold Tuning

In Fig. 13(a), the utility function is analyzed as τ_2 is adjusted while τ_1 is kept constant at values: 0.5, 0.7, and 0.9. When τ_1 is fixed at 0.5 and the initial value for τ_2 is 0.1, the utility starts at approximately 38.6 for the first task. As τ_2 is adjusted (with a step size of $\Delta^+_{\tau_2} = 0.2$), the utility improves steadily over the first five tasks. By task 5, the utility reaches approximately 45.0 with $\tau_2 = 0.9$ (terminated by upper bound condition). There are only minor fluctuations in subsequent tasks.

When τ_1 is fixed at 0.7, the utility starts at approximately 39.8 for the first Task. As τ_2 is adjusted, the utility improves steadily over the first three Tasks. By Task 3, the utility reaches approximately 45.5 with $\tau_2 = 0.5$. However, further increases in τ_2 for Task 4 lead to a slight decrease in utility. This indicates that the adjustment of τ_2 to 0.7 was too large and requires a smaller adjustment step to optimize utility. To rectify this, τ_2 is decreased by the step size $\Delta^-_{\tau_2} = 0.1$, leading to $\tau_2 = 0.6$ for Task 5. However, this



(a) Adjusting τ_2 by $\Delta^+_{\tau_2} = 0.2$ and $\Delta^-_{\tau_2} = 0.1$ when τ_1 is fixed.



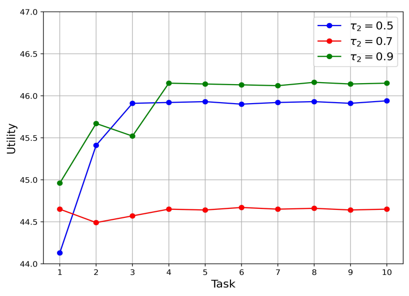
(b) Adjusting τ_2 by $\Delta^+_{\tau_2} = 0.2$ and $\Delta^-_{\tau_2} = 0.05$ when τ_1 is fixed.

Figure 13: Comparison of utility functions based on τ_2 adjustments with different Δ.

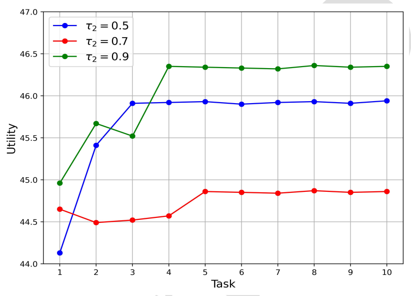
adjustment results in a decrease in utility. For Task 6, τ_2 is again decreased by the step size $\Delta^-_{\tau_2} = 0.1$, resulting in an improvement in utility (terminated by cycle condition). From Task 6 through Task 10, the utility stabilizes around 45.5, indicating that τ_2 has reached an optimal range for improving utility when $\tau_1 = 0.7$.

When τ_1 is fixed at 0.9, the utility starts at approximately 40.9 for the first Task. As τ_2 is adjusted, the utility improves steadily over the first three Tasks. By Task 3, the utility reaches approximately 46.0 with $\tau_2 = 0.5$. However, further increases in τ_2 for Task 4 lead to a slight decrease in utility. This indicates that the adjustment of τ_2 to 0.7 was too large and requires a smaller adjustment step to optimize utility. From this point onward, a similar behavior is observed for $\tau_1 = 0.9$ as seen with $\tau_1 = 0.7$, but the utility reaches a higher value in this case (terminated by cycle condition). Fig. 13(b) illustrates these scenarios with $\Delta^-_{\tau_2} = 0.05$. We observe a slightly better stabilization in utility for both $\tau_1 = 0.7$ and $\tau_1 = 0.9$ compared to the previous scenario, where $\Delta^-_{\tau_2} = 0.1$ was used. In both cases $\tau_1 = 0.7$ and $\tau_1 = 0.9$, the algorithm was able to find $\tau_2 = 0.55$ as the optimal threshold due to the smaller step size.

In Fig. 14(a), the utility function is analyzed as τ_1 is adjusted while τ_2 is kept constant at values: 0.5, 0.7, and 0.9. Initially, for Task 1, we set $\tau_1 = 0.5$. When τ_2 is fixed at 0.5, the utility starts at a relatively low value of 44.2 for the first Task. Moving to Task 2, τ_1 is increased to 0.7 with a step size of $\Delta^+ = 0.1$. For Task 3, τ_1 is further incremented by $\Delta^+_{\tau_1} = 0.2$, resulting in $\tau_1 = 0.9$. However, for Task 4, further increasing is not possible



(a) Adjusting τ_1 by $\Delta^+_{\tau_1} = 0.2$ and $\Delta^-_{\tau_1} = 0.1$ when τ_2 is fixed.



(b) Adjusting τ_1 by $\Delta^+_{\tau_1} = 0.2$ and $\Delta^-_{\tau_1} = 0.05$ when τ_2 is fixed.

Figure 14: Comparison of utility functions based on τ_1 adjustments with different Δ.

because classification with $\tau_1 > 0.9$ is not efficient for our model (terminated by upper bound condition). Therefore, from this point onward, we continue with $\tau_1 = 0.9$. After reaching the peak of 45.9, the utility remains relatively stable, with minor fluctuations across subsequent Tasks. When $\tau_2 = 0.7$ and with initial value $\tau_1 = 0.5$, the utility function starts at a higher value of 44.7. Moving to Task 2, τ_1 is increased to 0.7 with a step size of $\Delta^+_{\tau_1} = 0.2$. However, increasing τ_1 results in a decrease in utility, indicating that the adjustment was too large. To rectify this, τ_1 is decreased by the step size, $\Delta^-_{\tau_1} = 0.1$, leading to $\tau_1 = 0.6$ for Task 3. Again, it results in a decrease in utility. So for Task 4, we decrease τ_1 by $\Delta^-_{\tau_1} = 0.1$ to 0.5 (terminated by cycle condition). When $\tau_2 = 0.9$, the utility function starts at a higher value of 45. Moving to Task 2, τ_1 is increased to 0.7 with a step size of $\Delta^+ = 0.1$. However, further increases in τ_1 for Task 3 lead to a slight decrease in utility. This indicates that the adjustment of τ_1 to 0.9 was too large and requires adjusting τ_1 with $\Delta^-_{\tau_1}$ to 0.8 to optimize utility (terminated by convergence condition). From Task 4 through Task 10, the utility stabilizes around 46.2. Fig. 14(b) illustrates these scenarios with $\Delta^-_{\tau_1} = 0.05$. We observe a slightly better stabilization in utility for all values of $\tau_1 = 0.7$ and $\tau_1 = 0.9$ compared to the previous scenario, where $\Delta^-_{\tau_1} = 0.1$ was used. In scenarios $\tau_1 = 0.7$ and $\tau_1 = 0.9$, the algorithm was able to find $\tau_2 = 0.55$ and $\tau_2 = 0.85$ as the optimal threshold, respectively, due to smaller step size. The fluctuations in the utility function arise from the following factors:

- **Step size ($\Delta_{\tau_1}, \Delta_{\tau_2}$):** The algorithm incrementally adjusts the threshold values by adding or reducing step sizes to optimize utility. Early in the process, these adjustments can overshoot or undershoot the optimal values, leading to observed fluctuations. Larger step sizes may overshoot optimal thresholds, causing temporary drops in utility.

- **Non-linear relationships:** The thresholds interact non-linearly with the utility function, influenced by batch variability and the underlying data distribution.
- **Feedback loop noise:** Retraining the model on low-confidence samples may introduce variance, contributing to utility fluctuations.

As the algorithm progresses, these step-size adjustments gradually converge toward stability, resulting in steady utility values in later tasks. This adaptive threshold tuning process ensures optimized performance over time. To clarify this in the paper, we will expand the explanation of the adaptive threshold tuning mechanism and its impact on utility function dynamics.

The experiments demonstrate the impact of adjusting τ_1 and τ_2 on the utility function, highlighting the importance of appropriate step sizes in optimizing performance. When adjusting τ_2 while keeping τ_1 fixed at various values (0.5, 0.7, and 0.9), smaller adjustment steps ($\Delta^-_{\tau_2}$) lead to better stabilization in utility, particularly for larger values of τ_1. Similarly, when adjusting τ_1 with τ_2 fixed, the utility tends to stabilize more effectively with smaller step adjustments, avoiding large fluctuations. The results indicate that the choice of step size is crucial for achieving stable and optimal utility values, with smaller steps providing finer control in reaching a stable maximum.

5. Conclusion

The experimental results confirm that increasing the number of trained samples in the feedback loop (N) has a positive impact on the accuracy of the models. The feedback loop is especially effective in improving performance in cases where the model had low confidence, as evidenced by the gains in accuracy for lower threshold values. While increasing N continues to yield improvements, the gains diminish beyond a certain point, suggesting that there may be an optimal range for N that balances performance and computational cost. In addition, the feedback loop significantly affects the computational efficiency of the model. As the number of trained samples N increases, the average CPU time across the tasks decreases, particularly for higher threshold values. While the feedback loop introduces additional computational complexity due to retraining with low-confidence samples, this process ultimately reduces the time required to perform future tasks by improving the model's overall efficiency. The feedback loop not only enhances model accuracy but also plays a crucial role in optimizing the model's performance in terms of computational time. Thus, incorporating the feedback loop with a sufficient number of samples can lead to a more computationally efficient model, especially when processing complex tasks under higher thresholds.

References

[1] V. Chandola, A. Banerjee, V. Kumar, Anomaly detection: A survey, ACM computing surveys (CSUR) 41 (2009) 1–58.

[2] Z. Ahmad, A. Shahid Khan, C. Wai Shiang, J. Abdullah, F. Ahmad, Network intrusion detection system: A systematic study of machine learning and deep learning approaches, Transactions on Emerging Telecommunications Technologies 32 (2021) e4150.

[3] A. A. Ahmad, S. Boukari, A. M. Bello, M. A. Muhammad, A survey of intrusion detection techniques on software-defined networking, Intl. Journal of Innovative Science and Research Technology (2021).

[4] A. Chen, Y. Fu, X. Zheng, G. Lu, An efficient network behavior anomaly detection using a hybrid dbn-lstm network, Computers & Security 114 (2022) 102600.

[5] W. Wei, H. Gu, W. Deng, Z. Xiao, X. Ren, Abl-tc: A lightweight design for network traffic classification empowered by deep learning, Neurocomputing 489 (2022) 333–344.

[6] N. Niknami, V. Mahzoon, J. Wu, Meta-ids: A multi-stage deep intrusion detection system with optimal cpu usage, in: 1st IEEE International Conference on Meta Computing (ICMC), 2024.

[7] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016. `http://www.deeplearningbook.org`.

[8] M. Nayebi Kerdabadi, A. Hadizadeh Moghaddam, B. Liu, M. Liu, Z. Yao, Contrastive learning of temporal distinctiveness for survival analysis in electronic health records, in: Proceedings of the 32nd ACM International Conference on Information and Knowledge Management, 2023, pp. 1897–1906.

[9] N. Niknami, A. Srinivasan, J. Wu, Cyber-ande: Cybersecurity framework with adaptive distributed sampling for anomaly detection on sdns, IEEE Transactions on Information Forensics and Security (2024).

[10] A. Yazdinejadna, R. M. Parizi, A. Dehghantanha, M. S. Khan, A kangaroo-based intrusion detection system on software-defined networks, Computer Networks 184 (2021) 107688.

[11] R. Zhao, G. Gui, Z. Xue, J. Yin, T. Ohtsuki, B. Adebisi, H. Gacanin, A novel intrusion detection method based on lightweight neural network for internet of things, IEEE Internet of Things Journal 9 (2021) 9960–9972.

[12] Z. Wang, Z. Li, D. He, S. Chan, A lightweight approach for network intrusion detection in industrial cyber-physical systems based on knowledge distillation and deep metric learning, Expert Systems with Applications 206 (2022) 117671.

[13] S. Yang, X. Zheng, Z. Xu, X. Wang, A lightweight approach for network intrusion detection based on self-knowledge distillation, in: Proc. of the IEEE Intl. Conf. on Communications (ICC), 2023, pp. 3000–3005.

[14] W. Ge, Z. Cui, J. Wang, B. Tang, X. Li, Metacluster: a universal interpretable classification framework for cybersecurity, IEEE Transactions on Information Forensics and Security (2024) 1–1.

[15] N. Niknami, V. Mahzoon, J. Wu, Ptn-ids: Prototypical network solution for the few-shot detection in intrusion detection systems, in: IEEE 49th Conference on Local Computer Networks (LCN), IEEE, 2024, pp. 1–9.

[16] N. Hocine, C. Zitouni, A multi-agent system based on dynamic load balancing for collaborative intrusion detection, in: Proc. of the IEEE Intl. Conf. on Networking and Advanced Systems (ICNAS), 2023, pp. 1–6.

[17] N. Niknami, J. Wu, Enhancing load balancing by intrusion detection system chain on sdn data plane, in: Proc. of the IEEE Conf. on Communications and Network Security (CNS), 2022, pp. 264–272.

[18] M. Verkerken, L. D'hooge, D. Sudyana, Y.-D. Lin, T. Wauters, B. Volckaert, F. De Turck, A novel multi-stage approach for hierarchical intrusion detection, IEEE Transactions on Network and Service Management (2023).

[19] N. Niknami, V. Mahzoon, J. Wu, Crossalert: Enhancing multi-stage attack detection through semantic embedding of alerts across targeted domain, in: Proc. of the IEEE Conf. on Communications and Network Security (CNS), 2024.

[20] X. Cheng, M. Xu, R. Pan, D. Yu, C. Wang, X. Xiao, W. Lyu, Meta computing, IEEE Network (2023).

[21] Y. Liu, M. Zhang, X. Wang, Task assignment and capacity allocation for ml-based intrusion detection in resource-constrained edge networks, IEEE Xplore (2022).

[22] A. Gupta, S. Singh, R. Kumar, Fair resource allocation in an intrusion-detection system for edge computing, IEEE Xplore (2018).

[23] J. R. Vergara, P. A. Estévez, A review of feature selection methods based on mutual information, Neural Computing and Applications 24 (2014) 175–186.

[24] H. Peng, F. Long, C. Ding, Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy, IEEE Transactions on Pattern Analysis and Machine Intelligence 27 (2005) 1226–1238.

[25] K. Manohar, B. W. Brunton, J. N. Kutz, S. L. Brunton, Data-driven sparse sensor placement for reconstruction: Demonstrating the benefits of exploiting known patterns, IEEE Control Systems Magazine 38 (2018) 63–86.

Declaration of interests

☒ 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.

☒ The author is an Editorial Board Member/Editor-in-Chief/Associate Editor/Guest Editor for *[Journal name]* and was not involved in the editorial review or the decision to publish this article.

☒ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

The authors declare no competing financial interests or personal relationships.

Nadia Niknami	Vahid mahzoon	Jie Wu
Nadia Niknami	[illegible]	[illegible]