REAPER: Real-Time Detection of Malicious Traffic via Deep Time-Series Embedding Analysis

Dan Tang, Boru Liu, Zheng Qin, *Associate Member, IEEE*, Wei Liang, Keqin Li, *Fellow, IEEE*, and Wenqiang Jin, *Member, IEEE*

***Abstract*—Existing malicious traffic detection methods face challenges in achieving real-time detection in high-bandwidth networks, suffer from information loss due to conventional discrete statistical feature extraction, and remain vulnerable to evasion by short-flow-based malicious traffic resembling benign behavior. To this end, we present REAPER, a high-performance system for real-time detection of malicious traffic at the network flow level, built on Intel DPDK. REAPER leverages an IP-trie to dynamically aggregate short flows in real time, exposing hidden sending patterns of malicious traffic. It leverages a deep time-series embedding analysis (DTEA) method, which uses the RNN algorithm to directly convert network flow data into embeddings, followed by outlier analysis on the embeddings using an unsupervised variational autoencoder (VAE) algorithm, capable of detecting zero-day attacks. Experiments show REAPER outperforms the baseline, especially for encrypted malware traffic, achieves an average AUC of 0.9429 and EER of 0.0348 under evolving cyber attack patterns, and supports up to 5 Gbps traffic throughput, with scalability to higher rates.**

***Index Terms*—Deep learning, deep time-series embedding analysis, malicious traffic detection, variational autoencoder.**

I. Introduction

TRADITIONAL malicious traffic detection solutions (e.g., signature identification [1], [2], [3], etc.), leverage configured prior rules to perform feature analysis on the network traffic. However, prior rules-based solutions are not extensible and robust. They rely on manual feature engineering to update prior rules (signature libraries, etc.) and cannot detect zero-day attacks.

Since Machine Learning (ML) methods, especially Deep learning (DL), possess powerful capabilities in modeling and analyzing data. They can effectively identify complex patterns existing in network flows, enabling the accurate identification of malicious patterns [4], [5], [6]. However, the majority of ML-based detection systems in current research, e.g., [7] and [8], utilize existing labeled datasets for supervised learning, limiting their ability to detect only known attacks. In contrast, only a few detection approaches, e.g., [9], [10], and [11], use ML-based outlier analysis in unsupervised manners. While these approaches have the potential to detect zero-day attacks, they still employ traditional statistical feature extraction that exhibits significant information loss, thereby reducing their effectiveness in distinguishing among numerous network flows [12].

Furthermore, traditional detection systems [13], [14] utilizing end-to-end solutions cannot detect ongoing line-speed traffic in real time, especially within high-bandwidth networks, e.g., within backbone networks. Aiming for real-time detection requirements [15], programmable data plane (PDP) as a novel networking paradigm, provides in-network solutions for real-time malicious traffic detection. Concretely, these solutions are mainly developed by Programming Protocol-Independent Packet Processors (P4) [16] or Intel data plane development kit (Intel DPDK).

P4-based in-network solutions [17], [18] are deployed on the network devices integrating the P4 target hardware, e.g., Tofino ASIC and SmartNIC, etc. However, due to the limitations of both P4 primitives and hardware resources [19], [20], P4 cannot highly support developing sophisticated ML-based general detection systems against malicious traffic, even if there are studies about the P4-based in-network ML inference [21], [22]. In practice, P4 is mainly oriented toward developing lightweight systems that address specific tasks, like DDoS attack detection [23], [24], and DNS security defense [25]. Besides, P4 target hardware can be the accelerator for sampling fine-grained traffic statistics [26].

Subsequently, Intel DPDK-based in-network solutions [27] are deployed on x86-based platforms powered by CPUs, e.g., software-defined middleboxes. Intel DPDK utilizes the userspace I/O (UIO) kernel module to bind NIC ports to the poll mode driver (PMD) running in userspace, thereby masking kernel interrupts triggered by arrival packets. Additionally, Intel DPDK employs hugepages as memory pools to cache large volumes of arrival packets, reducing the overhead associated with memory page switching during packet access. Developers can leverage the Intel DPDK API, wrapped as a

Received 25 June 2024; revised 5 February 2025 and 18 August 2025; accepted 14 September 2025; approved by IEEE TRANSACTIONS ON NETWORKING Editor Y. Liu. Date of publication 3 October 2025; date of current version 2 January 2026. This work was supported in part by the National Natural Science Foundation of China under Grant 62472153 and in part by Hunan Provincial Natural Science Foundation of China under Grant 2025JJ50350. *(Corresponding author: Wenqiang Jin.)*

Dan Tang, Zheng Qin, and Wenqiang Jin are with the College of Cyber Science and Technology, Hunan University (HNU), Changsha 410082, China (e-mail: Dtang@hnu.edu.cn; zqin@hnu.edu.cn; wqjin@hnu.edu.cn).

Boru Liu is with the College of Computer Science and Electronic Engineering (CSEE), Hunan University (HNU), Changsha 410082, China (e-mail: liuboru@hnu.edu.cn).

Wei Liang is with the School of Computer Science and Engineering, Hunan University of Science and Technology (HNUST), Xiangtan 411199, China (e-mail: wliang@hnust.edu.cn).

Keqin Li is with the Department of Computer Science, The State University of New York, New Paltz, NY 12561 USA, and also with Hunan University, Changsha 410082, China (e-mail: lik@newpaltz.edu).

Digital Object Identifier 10.1109/TON.2025.3610853

C library, to mirror arrival packets from the memory pool and process them efficiently. Therefore, Intel DPDK exhibits the capacity to provide in-network solutions for achieving real-time malicious traffic detection driven by ML. However, existing Intel DPDK-based in-network solutions [12] only detect individual network flows by extracting statistical features, which can be evade by the disguised malicious traffic.

To this end, we propose REAPER, a real-time, robust malicious traffic detection system prototyped with Intel DPDK. It uses network flows as detection units and adopts DL-based methods to robustly identify malicious traffic.

Real-time Collection for Network Flow Data. We enable the UIO for NIC ports so that REAPER can online extract per-packet features from the packets mirrored from NIC ports. We use a flow table to assemble the per-packet features into network flow data. Specifically, each table entry is indexed by a network flow and stores a record containing both bidirectional and unidirectional data for the network flow. When a network flow is completed, we evict it from the flow table while collecting the corresponding network flow record.

Dynamic Aggregation for Short Flows. We analyzes the composition of backbone network traffic and observes that short flows contain significantly limited information. When performing detection on individual network flows, short flows lack sufficient distinguishability, which can lead to the confusion of malicious short flows with benign ones. This enables malicious traffic, such as encrypted malware traffic, to evade detection by manipulating large botnets to send volumes of short flows. To address this challenge, we dynamically aggregate short flows, amplifying the analytical granularity to robustly identify the sending patterns associated with malicious traffic. Specifically, we have developed an IP prefix tree (IP-Trie) for REAPER to dynamically aggregate short flows into aggregated flows based on their IP prefixes, and we propose efficient algorithms for runtime IP-Trie maintenance.

Deep Time-series Embedding Analysis. We propose the Deep Time-series Embedding Analysis (DTEA) method for REAPER. Unlike traditional discrete statistical feature extraction, the DTEA method directly converts time-series-shaped network flow data into the continuous embeddings using the RNN algorithm, thereby preserving comprehensive information from the original data. Additionally, the DTEA method uses an unsupervised variational autoencoder (VAE) algorithm to perform outlier analysis on the embeddings to identify malicious network flows. The VAE overcomes two key limitations exhibited by traditional autoencoders (AE) [28], [29]: (I) Deterministic modeling of the latent space lacks generalization ability. (II) The deterministic latent representation must be decoded during outlier analysis, which introduces additional computational overhead in real-time detection. In contrast, the VAE algorithm encodes data into an uncertain latent representation, and its uncertainty can directly indicate whether the data is an outlier, without the need for decoding.

Evaluation. We conduct comprehensive experiments to thoroughly evaluate REAPER.

We gather a total of 20 distinct malicious traffic datasets to assess the detection performance of REAPER. To further evaluate the robustness of REAPER's detection capabilities, we go beyond traditional cyber attack traffic datasets and collect 14 sets of encrypted malware traffic, including ransomware and adware traffic. Additionally, we mix various cyber attack traffic datasets to examine REAPER's ability to adapt to evolving cyber attack patterns. The experimental results indicate that REAPER outperforms the baselines, particularly in detecting encrypted malware traffic. Moreover, REAPER maintains an average AUC of 0.9429 and an average EER of 0.0348, even when faced with evolving cyber attack patterns.

Furthermore, we measure REAPER's throughput on our testbed using 3 different traffic loads. The throughput for parsing packets and updating the flow table reaches up to 9.5 Gbps. The throughput for dynamically aggregating short flows reaches 4.5 Gbps, while the throughput for performing network flow detection reaches 5 Gbps. Besides, the system's runtime memory footprint is 4.1 GB.

Overall, our paper has five main contributions:

- We propose REAPER, a real-time, robust malicious traffic detection system that leverages deep learning (DL)-based methods to identify malicious traffic in units of network flows.
- We develop the IP-Trie to perform dynamic aggregation on short flows and propose efficient algorithms for maintaining the IP-Trie in real-time.
- We propose the deep time-series embedding analysis (DTEA) method for robustly analyze network flow data.
- We utilize Intel DPDK to prototype REAPER, capable of handling high-bandwidth traffic.
- We evaluate the detection performance of REAPER on 20 groups of different malicious traffic datasets, and also measure its throughput and runtime overhead.

The remaining content is organized as follows: Section II recommends the threat model and design goals of REAPER. Section III demonstrates the top-level design of REAPER. Section IV expands the design details of REAPER. Section V presents the implementation of REAPER and all conducted experiments. Additionally, Section VI reviews the related work, Section VII presents the limitation and discussion of REAPER, as well as Section VIII summarizes this paper.

II. Threat Modeling and Design Goals

A. Threat Modeling

First, we implement REAPER as a plug-and-play virtualization function module on a software-defined middlebox. REAPER is isolated from basic functions such as routing. Therefore, it does not interfere with ongoing traffic.

Second, we highlight that REAPER has no prior knowledge about any attacks, its knowledge is derived only from the real-world traffic, which is recognized as benign. As a result, it can detect zero-day attacks with unknown patterns.

Third, for resource exhaustion attacks, its traffic consists of extensive short flows, which can easily be confused with benign traffic. These short flows are exploited to request the resources of victim servers until they are exhausted. However, REAPER can dynamically aggregate short flows, effectively countering resource exhaustion attacks.

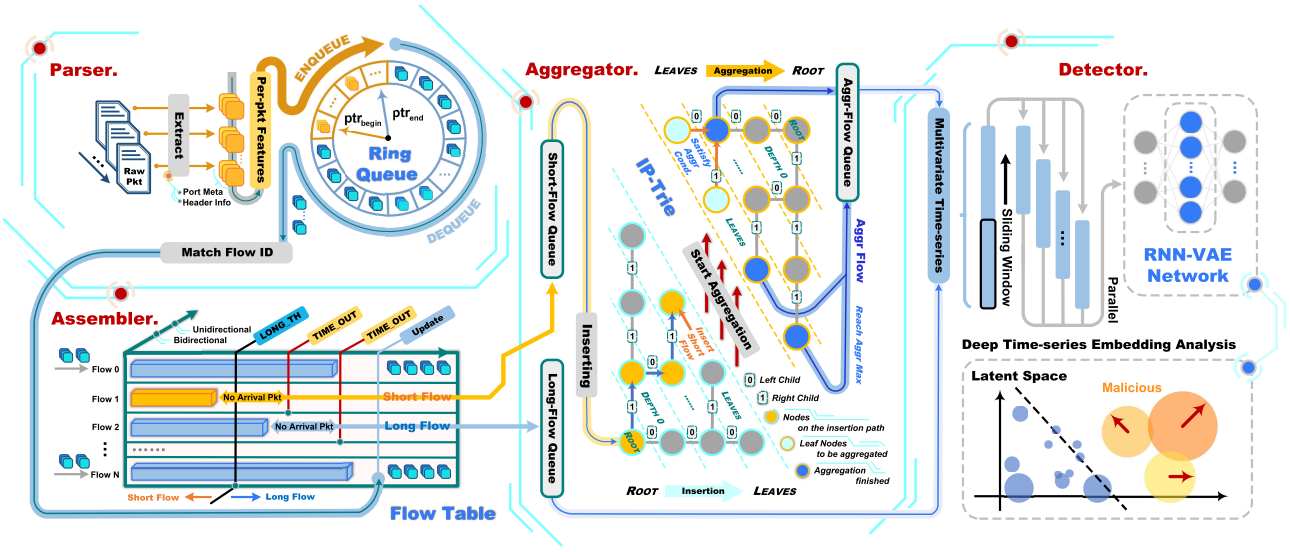


Fig. 1. The Overview of REAPER.

TABLE I
LIST OF ACRONYMS

Acronym	Description	Acronym	Description
DTEA	**D**eep **T**ime-series **E**mbedding **A**nalysis	VAE	**V**ariational **A**uto**E**ncoder
MTS	**M**ultivariate **T**ime **S**eries	FCL	**F**ully **C**onnected **L**ayer
ELBO	**E**vidence **L**ower **BO**und	KL	**K**ullback-**L**eibler divergence
EWMP	**E**xponentially **W**eighted **M**oving **P**ercentile	UIO	**U**serspace **I/O**

Furthermore, REAPER is oriented toward detecting malicious traffic but does not take specific mitigation strategies. To further enhance its capabilities, we can integrate existing solutions for mitigating malicious traffic [30], to further discard identified malicious traffic or limit its rate.

B. Design Goals

Robust Detection. The malicious traffic detection system we developed should have high accuracy to identify malicious traffic and strong generalization ability to adapt evolving cyber attack patterns, and the system should possess capacity to counter covert malicious traffic.

Real-time & High Throughput. The system needs to be deployed in high-bandwidth network scenarios in real-time and requires high processing throughput.

III. Overview of REAPER

Table I lists the main acronyms in this paper. In this section, we present the design overview of REAPER, a system designed to detect malicious traffic at the granularity of network flows. Based on insights into network traffic composition, REAPER adopts distinct strategies for handling long and short flows. In particular, REAPER dynamically aggregates short flows based on IP prefixes, thereby amplifying the analytical granularity to aggregated flows at the network segment level. This approach improves its capability to detect malicious network flows that use camouflage techniques to evade detection, e.g., low-rate port flooding [31] and botnet manipulation. Additionally, to analyze the network flow data, REAPER departs from the traditional approach of extracting discrete statistical features. Instead, it employs the DTEA method to directly transform the network flow data into continuous embeddings, preserving the original information of the network flows. Besides, the DTEA method leverages the VAE algorithm to perform outlier analysis on the embeddings, enabling malicious traffic detection in an unsupervised manner. Fig. 1 illustrates the top-level design of REAPER, which comprises four key modules: 1) Parser, 2) Assembler, 3) Aggregator, and 4) Detector.

Parser. This module monitors the RX queues of NIC ports, mirroring the arrival packets and extracting their per-packet features, including the header information (e.g., IP, port, protocol, length, flags, etc.), and the NIC port metadata (e.g., arrival timestamp, etc.). The extracted per-packet features are orderly stored in a ring queue. The ring queue design will be discussed in Section IV-A.

Assembler. This module retrieves the per-packet features from the ring queue. It maintains a flow table that assembles the per-packet features into network flow records based on the flow ID, i.e., the five-tuple hash key: $<\{$[SrcIP, SrcPort], [DstIP, DstPort]$\}_{\text{symmetric}}$, protocol$>$, where [SrcIP, SrcPort] and [DstIP, DstPort] are symmetric hash key elements. Each table entry maps to a network flow record that caches both bidirectional and unidirectional flow data for corresponding network flow. Once the data of a network flow is fully collected, its record will be enqueued into the short/long-flow queue. The details of the flow table design and network flow classification will be presented in Section IV-B.

Aggregator. This module dequeues short flow records from the short-flow queue and inserts the unidirectional flow data from the records into the leaf nodes of an IP prefix tree (IP-Trie), based on the SrcIP of unidirectional flows. The inserted flow data is incorporated into the aggregated flow record at the leaf nodes. When the aggregation process is enabled, the

TABLE II
LIST OF SYMBOLS USED IN DESIGN DETAIL

Symbol	Description
$\mathrm{ptr_{end}}$	The pointer variable points to the slot where the next dequeuing operation will begin in the ring queue.
$\mathrm{ptr_{begin}}$	The pointer variable points to the slot where the next enqueuing operation will begin in the ring queue.
$\mathrm{slot_i}$	The $\mathrm{i^{th}}$ slot in ring queue.
$\mathrm{Set^{hist}}$	The set of flow IDs of incomplete network flows accumulated historically in the current flow table snapshot.
$\mathrm{Set^{hist}_{next}}$	The $\mathrm{Set^{hist}}$ of the next flow table snapshot.
$\mathrm{Set_{last}}$	The set of flow IDs of newly arrived network flows in the current flow table snapshot.
$\mathrm{Set_{next}}$	The $\mathrm{Set_{last}}$ of the next flow table snapshot.
$\mathrm{Set^{left}_{leaf}}$	The set of left leaf nodes in IP-Trie.
$\mathrm{Set^{right}_{leaf}}$	The set of right leaf nodes without siblings in IP-Trie.
$\mathrm{Set^{next,all}_{leaf}}$	The set of leaf nodes that need to be accessed in next dynamic aggregation iteration.
$\mathrm{Set^{next,left}_{leaf}}$	The set of left leaf nodes that need to be accessed in next dynamic aggregation iteration.
$\mathrm{Set^{next,right}_{leaf}}$	The set of right leaf nodes that need to be accessed in next dynamic aggregation iteration.
$\mathrm{n^i_{leaf}}$	The $\mathrm{i^{th}}$ leaf node accessed in current dynamic aggregation iteration.
$\mathrm{n^{i,sibl}_{leaf}}$	The sibling of $\mathrm{i^{th}}$ leaf node accessed in current dynamic aggregation iteration.
$\mathrm{n^i_{leaf''}}$	The $\mathrm{i^{th}}$ leaf node accessed in next dynamic aggregation iteration.
$\mathrm{m^i}$	The $\mathrm{i^{th}}$ per-packet feature in MTS
$\mathrm{h_i}$	The hidden state in GRU cell in the $\mathrm{i^{th}}$ time step.
$\mathrm{u_i}$	The update gate variable in GRU cell in the $\mathrm{i^{th}}$ time step.
$\mathrm{r_i}$	The reset gate variable in GRU cell in the $\mathrm{i^{th}}$ time step.
$\hat{\mathrm{h}}_\mathrm{i}$	The candidate hidden state in GRU cell in the $\mathrm{i^{th}}$ time step.
$\hat{\mathrm{m}}_\mathrm{i}$	The $\mathrm{i^{th}}$ per-packet feature in $\hat{\mathrm{M}}\mathrm{TS}$ reconstructed within RNN-VAE network.

module iteratively merges sibling leaf nodes whose aggregated flow record length is below a specified threshold. The cached aggregated flow record of these nodes is then combined and stored in their parent node. The aggregation process terminates once the height of the current IP-Trie reaches the minimum value. Finally, the aggregated flow records of the leaf nodes are saved into the aggregated-flow queue. Details of the aggregation process will be discussed in Section IV-C.

Detector. This module fetch long flow records from the long-flow queue and aggregated flow records from the aggregated-flow queue. It preprocesses the network flow data from these records through the following steps: 1) converting arrival timestamps into arrival intervals, 2) normalizing the data, and 3) applying a sliding window technique. Subsequently, this module employs the DTEA method, which first transforms the preprocessed network flow data into embeddings and then inputs these embeddings into a VAE network for inference, determining whether the related network flows are benign or malicious. The details of the DTEA method will be elaborated in Section IV-D.

IV. DESIGN DETAIL

In this section, we expand on the details of the four key modules included in REAPER. The symbols used in this section are summarized in Table II.

A. Parser Design

In this module, we sequentially extract per-packet features from packets mirrored from NIC ports. Each per-packet feature abstracts a packet into the following key information: 1) Src/DstIP, Src/DstPort, and protocol, which associate a packet with a network flow; 2) flags from the protocol header, such as the SYN and ACK flags in the TCP header, whose sequence can represent the communication pattern of a network flow; 3) packet length, whose sequence can reflect traffic volume of a network flow; and 4) packet arrival timestamp, which is used to compute arrival intervals. The sequence of arrival interval can be used to analyze the periodicity within a network flow.

Ring Queue. We utilize a ring queue to store extracted per-packet features. Concretely, we allocate an array memory of size RQ_SIZE for the ring queue, referred to as ArrMem. We use $\mathrm{slot_i}$ to indicate $\mathrm{i^{th}}$ slot of ArrMem. Each slot holds a single per-packet feature. Besides, we adopt pointer variables, namely $\mathrm{ptr_{begin}}$ and $\mathrm{ptr_{end}}$, to index the slots for enqueuing and dequeuing operations, respectively. Since ArrMem is structured as a ring, both $\mathrm{ptr_{begin}}$ and $\mathrm{ptr_{end}}$ are taken modulo RQ_SIZE after each enqueuing or dequeuing operation.

Concretely, we perform enqueuing operation immediately after each per-packet feature is extracted. Therefore, this operation only need to enqueue the newest per-packet feature in the $\mathrm{slot_{ptr_{end}}}$ and update $\mathrm{ptr_{end}}$ with:

$$\mathrm{ptr_{end}} = (\mathrm{ptr_{end}} + 1) \bmod \mathrm{RQ\_SIZE} \tag{1}$$

Besides, we dequeues FETCH_NUM number of per-packet features each time in the following Assembler. We consider the situation when the dequeuing operation is across the boundary of the first and last slots, that is:

$$\mathrm{ptr_{end}} < \mathrm{ptr_{begin}},\ \ \mathrm{ptr_{begin}} + \mathrm{FETCH\_NUM} > \mathrm{RQ\_SIZE} \tag{2}$$

Under this situation, we take dequeuing operation on a set of slots, i.e., $\{\mathrm{slot_i} \mid \mathrm{i} \in \mathrm{I_0}\}$, where $\mathrm{I_0}$ is:

$$\begin{aligned} I_0 = \{i \mid [ptr_{begin},\ \mathrm{RQ\_SIZE}) \\ \cup [0,\ \mathrm{FETCH\_NUM} - ptr_{begin})\} \end{aligned} \tag{3}$$

Without this situation, we directly dequeue another set of slots, i.e., $\{\mathrm{slot_i} \mid \mathrm{i} \in \mathrm{I_1}\}$, where $\mathrm{I_1}$ is:

$$\mathrm{I_1} = \{\mathrm{i} \mid [\mathrm{ptr_{begin}},\ \mathrm{ptr_{begin}} + \mathrm{FETCH\_NUM})\} \tag{4}$$

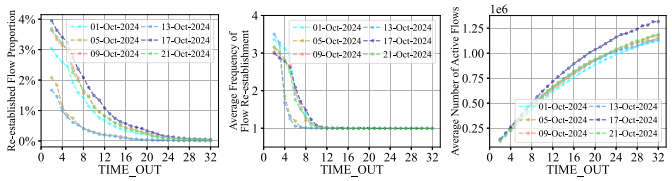
Additionally, we update the $\mathrm{ptr_{begin}}$ with:

$$\mathrm{ptr_{begin}} = (\mathrm{ptr_{begin}} + \mathrm{FETCH\_NUM}) \bmod \mathrm{RQ\_SIZE} \tag{5}$$

B. Assembler Design

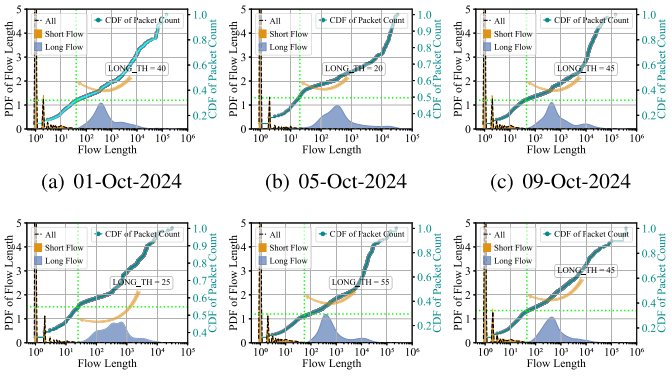
In this module, we assemble the per-packet features dequeued from the ring queue into network flow data using a flow table. Each entry in the table is a key-value pair corresponding to a specific network flow, where the key is the flow ID defined as the five-tuple hash key: <[SrcIP, SrcPort], $[\mathrm{DstIP, DstPort}]_{\mathrm{symmetric}}$, protocol>, and the value is the associated network flow record.

Each network flow record maintains both bidirectional and unidirectional flow data. Here, unidirectional flow data involves the forward flow data (packets from [SrcIP, SrcPort] to [DstIP, DstPort]) and backward flow data (packets in the



(a) (b) (c)

Fig. 2. TIME_OUT configuration exploration.



(a) 01-Oct-2024 (b) 05-Oct-2024 (c) 09-Oct-2024

(d) 13-Oct-2024 (e) 17-Oct-2024 (f) 21-Oct-2024

Fig. 3. Flow length distributions of WIDE backbone network traffic datasets.

opposite direction). Each per-packet feature first hits a table entry and then triggers an update of both the bidirectional and unidirectional flow data in the corresponding flow record.

Notably, before being cached in an entry, each per-packet feature is preserved as a 3D vector containing only the packet length, packet arrival timestamp, and a composite flag field. This flag field includes the protocol, flags from the protocol header (e.g., ACK), and a 1-bit indicator denoting whether the packet belongs to the forward or backward flow. To facilitate high-concurrency access, we adopt a hash table where each entry is protected by an independent lock, enabling efficient flow table access.

Flow Completion. The flow table utilizes a timeout mechanism to determine when a network flow has completed. If no packets arrive for a network flow within the TIME_OUT period, it is considered complete. To select an appropriate TIME_OUT value, we analyze the real-world benign traffic datasets. Specifically, we use six WIDE backbone network traffic datasets collected on October 1, 5, 9, 13, 17, and 21, 2024. They are all daily traces captured at the transit link and the BBIX link of WIDE backbone network. We use the first 10^9 packets in each dataset for analysis.As shown in fig. 2, a larger TIME_OUT value results in a smaller re-established flow proportion and a decreased average frequency of flow re-establishment. However, it also leads to an increase in the average number of active flows in the flow table, requiring more runtime memory allocation. As a trade-off, we observe that re-established flow proportion across the six datasets drops below 1% for the first time when TIME_OUT is set to 16s. Therefore, we select 16s as the final value for TIME_OUT.

Network Flow Classification. Based on the definition of long flows [32], which are network flows that occupy more than 1% of the total network bandwidth, we further investigate the distribution of long and short flows across the six datasets discussed earlier, as shown in Fig. 3.We find that the PDF curve of short flows closely aligns with the PDF curve of all flows, indicating that short flows constitute the vast majority of network traffic. Meanwhile, the CDF curve of packet count reveals that long flows, despite their negligible number, account for at least half of the total packets. These findings highlight the extreme imbalance in the information carried by short and long flows, which leads us to adopt distinct analysis strategies for each.

For long flows, which contain sufficient information, we only preserve the bidirectional flow data in their records, which subsequently output to the Detector for analysis. In contrast, short flows are numerous, and each of them carry limited information, making it easier to confuse malicious short flows with benign ones. For example, attackers can control a host to launch low-rate port flooding or even conduct botnet manipulation to send lots of malicious short flows but seemly benign ones at a five-tuple network flow level. To address this, we use traffic sending patterns as the basis for detection, aggregating the data from short flows to amplify the analytical granularity. This increases the amount of information contained in each analysis unit, thereby improving the effectiveness of malicious network traffic identification, as detailed in Section IV-C.

Additionally, we observe that there is a clear boundary between the distributions of long and short flows, which allows us to use the LONG_TH threshold for efficient flow classification. As shown in Fig. 3, different datasets have different LONG_TH values.

Flow Table Snapshot. Each flow table snapshot contains all the flow IDs of network flows, which survived in the flow table at a certain moment. The flow table takes a snapshot each INSPECT_INTERVAL period, while the Assembler module scans the snapshot to assist in evicting completed network flows from the flow table and enqueueing their data in the short/long-flow queue. Traditional snapshot methods involve locking the entire flow table at the end of each INSPECT_INTERVAL period and copying its entire content, which interrupts the runtime Assembler module and incurs significant copying overhead each time. To achieve more efficient snapshot operation, we adapt the concept of move semantics from C++11 to eliminate copying and dynamically generate snapshots during each INSPECT_INTERVAL period.

As shown in Fig. 4, we maintain four sets: $\mathrm{Set}^{\mathrm{hist}}$, $\mathrm{Set}^{\mathrm{hist}}_{\mathrm{next}}$, $\mathrm{Set}_{\mathrm{last}}$, and $\mathrm{Set}_{\mathrm{next}}$, all sets are empty initially. During an INSPECT_INTERVAL period, the flow table adds the flow IDs of newly arrival network flows to $\mathrm{Set}_{\mathrm{next}}$. When the INSPECT_INTERVAL period ends, we move the contents of $\mathrm{Set}_{\mathrm{next}}$ to $\mathrm{Set}_{\mathrm{last}}$, and $\mathrm{Set}_{\mathrm{next}}$ naturally becomes an empty set. At this moment, the snapshot of the flow table is described by $\mathrm{Set}_{\mathrm{last}}$. The Assembler scans this snapshot, evicting complete network flows and adding the flow IDs of incomplete network flows to $\mathrm{Set}^{\mathrm{hist}}_{\mathrm{next}}$. Once the scanning is complete, we move the contents of $\mathrm{Set}^{\mathrm{hist}}_{\mathrm{next}}$ to $\mathrm{Set}^{\mathrm{hist}}$, and $\mathrm{Set}^{\mathrm{hist}}_{\mathrm{next}}$ naturally becomes an empty set. Finally, the flow table snapshot can be represented as a combination of $\mathrm{Set}^{\mathrm{hist}}$ and $\mathrm{Set}_{\mathrm{last}}$.

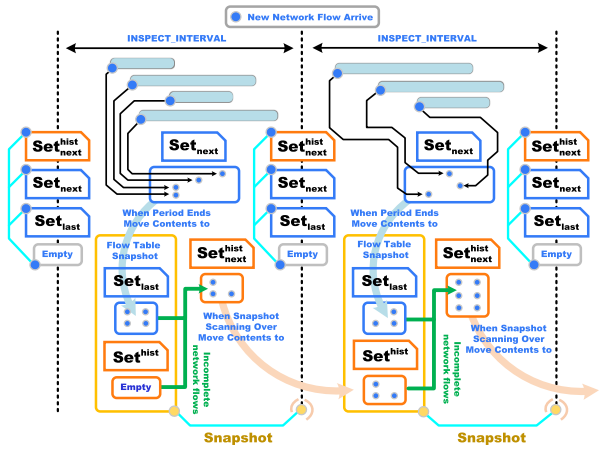


Fig. 4. Flow table snapshot.

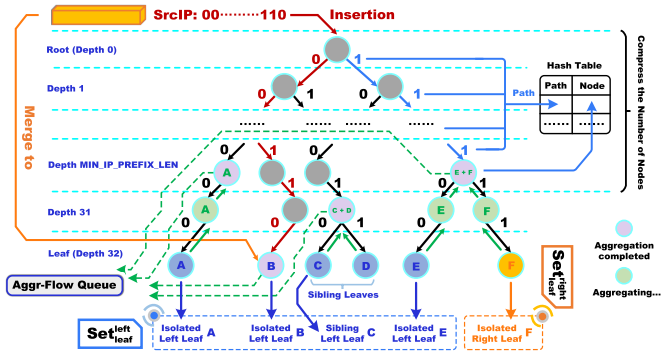


Fig. 5. The IP-Trie insertion and dynamic aggregation process with MIN_IP_PREFIX_LEN Set to 30 as an example.

C. Aggregator Design

In this module, to detect malicious traffic that exploits short flows for concealment, we approach the detection from the perspective of traffic sending patterns. Specifically, we dynamically aggregate the unidirectional flows, including both forward and backward flows, within all short flows using the SrcIP prefix to form aggregated flows. Based on this, we perform detection on the aggregated flows, which contains more comprehensive information, at a coarser analytical granularity.

We create a IP prefix tree (IP-Trie) to perform the dynamic aggregation on the unidirectional flows within all short flows. For each short flow record, we extract its unidirectional flow data. The binary string of the SrcIP from the unidirectional flow is used as the matching string for the IP-Trie insertion. Subsequently, the unidirectional flow data is merged into the aggregated flow record at the matched leaf node. When the number of inserted unidirectional flows reaches AGGR_CYCLE, we enable dynamic aggregation on the leaf nodes to obtain aggregated flow records with different IP prefix lengths. The following content details the processes of the IP-Trie insertion and dynamic aggregation. An example of these processes is presented in Fig. 5.

IP-Trie Insertion. When inserting a unidirectional flow into the IP-Trie, we perform a path match based on the binary string of its SrcIP. Starting from the root node, if the current matched bit is 0, the path extends to the left child node; if the bit is 1, the path extends to the right child node. After matching all bits of the binary string, the unidirectional flow data is merged into the aggregated flow record of a leaf node.

During the IP-Trie insertion, we dynamically maintain two sets of leaf nodes, i.e., $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{left}}$ and $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{right}}$. We store isolated left and right leaf nodes in $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{left}}$ and $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{right}}$, respectively. For sibling leaf nodes that appear in pairs, only the left leaf node is retained in $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{left}}$, as the right leaf node can be accessed through their common parent node. This design allows us to access all leaf nodes in both sets simultaneously, enabling parallel execution of the following dynamic aggregation. Specifically, when a unidirectional flow is inserted into a left leaf node, we add this leaf node to $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{left}}$ while removing its sibling right leaf node from $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{right}}$. If the flow is inserted into a right leaf node that has no sibling left leaf node, we insert this leaf node into $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{right}}$.

Additionally, we define MIN_IP_PREFIX_LEN as the minimum IP prefix length for dynamic aggregation. That is, when the height of current IP-Trie is MIN_IP_PREFIX_LEN, dynamic aggregation will stop. After this, we employ a hash table to map all paths from the root node to those at a depth of MIN_IP_PREFIX_LEN, which avoids the unnecessary creation and maintenance of the nodes on these paths.

Dynamic Aggregation. We initially create 3 auxiliary sets, namely $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{next,all}}$, $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{next,left}}$, and $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{next,right}}$. Once the dynamic aggregation begins, we iteratively execute the following process until both $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{left}}$ and $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{right}}$ are empty:

- ***Step 1***: We concurrently access all leaf nodes in $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{left}}$ and $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{right}}$, where the i^{th} leaf node is denoted as $\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i}}$ and the length of aggregated flow record cached in $\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i}}$ is denoted as $\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i}} \rightarrow \mathrm{len}$. If $\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i}}$ belongs to $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{left}}$, it proceeds to *Step 1.1*; if it belongs to $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{right}}$, it proceeds to *Step 1.2*.
- ***Step 1.1***: Determine whether $\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i}}$ has a sibling right leaf node, denoted as $\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i,sibl}}$. If it does not, proceed to *Step 1.2*. Otherwise, check the condition:

$$\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i}} \rightarrow \mathrm{len} < \mathrm{AGGR\_MAX} \;||\; \mathrm{n}_{\mathrm{leaf}}^{\mathrm{i,sibl}} \rightarrow \mathrm{len} < \mathrm{AGGR\_MAX} \tag{6}$$

  If the condition holds, the aggregated flow records of $\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i}}$ and $\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i,sibl}}$ will be merged into their common parent node, which will then be added as a new leaf node to $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{next,all}}$. Otherwise, both $\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i}}$ and $\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i,sibl}}$ are fully aggregated, their aggregated flow records will be added to the aggregated-flow queue. Proceed to *Step 2*.
- ***Step 1.2***: If the depth of $\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i}}$ is MIN_IP_PREFIX_LEN, the aggregation for $\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i}}$ terminates, and its aggregated flow record is added to the aggregated-flow queue. Otherwise, proceed to *Step 1.3*.
- ***Step 1.3***: Check the condition:

$$\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i}} \rightarrow \mathrm{len} < \mathrm{AGGR\_MAX} \tag{7}$$

  If the condition is met, the aggregated flow record of $\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i}}$ is moved to its parent node, and the parent node is added as a new leaf node to $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{next,all}}$. Otherwise, the aggregation for $\mathrm{n}_{\mathrm{leaf}}^{\mathrm{i}}$ is finished, and its aggregated flow record is added to the aggregated-flow queue. Proceed to *Step 2*.

- ***Step 2***: After synchronizing the aggregation execution of all leaf nodes up to this step, concurrently access all nodes in $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{next,all}}$, where the i^{th} leaf node is denoted as $\mathrm{n}_{\mathrm{leaf}''}^{\mathrm{i}}$. If $\mathrm{n}_{\mathrm{leaf}''}^{\mathrm{i}}$ is a left leaf node, it is added to $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{next,left}}$. If $\mathrm{n}_{\mathrm{leaf}''}^{\mathrm{i}}$ is a right leaf node that has no sibling node, it is added to $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{next,right}}$. Proceed to *Step 3*.
- ***Step 3***: Move the contents of $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{next,left}}$ and $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{next,right}}$ to $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{left}}$ and $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{right}}$, respectively. Then clear the auxiliary sets.

D. Detector Design

In this module, we fetch network flow records from the aggr/long-flow queue and apply the DTEA method for malicious traffic detection. We use $\mathrm{MTS} \in \mathrm{R}^{\mathrm{L}\times 3}$ to represent the network flow data from a record with the length of L, and we employ $\mathrm{m}^{\mathrm{i}} \in \mathrm{R}^{1\times 3}$ to represent the i^{th} per-packet feature that contains 3 columns of packet information, including the packet arrival timestamp ($\mathrm{m}_{\mathrm{ts}}^{\mathrm{i}}$), packet legnth ($\mathrm{m}_{\mathrm{len}}^{\mathrm{i}}$), and flags ($\mathrm{m}_{\mathrm{flags}}^{\mathrm{i}}$). Consequently, MTS is denoted as a multivariate time series and m^{i} is denoted as the i^{th} vector along the time dimension of MTS:

$$\mathrm{MTS} = \left[\mathrm{m}^{0}, \ldots, \mathrm{m}^{\mathrm{i}}, \ldots, \mathrm{m}^{\mathrm{L}-1}\right],\ \mathrm{m}^{\mathrm{i}} = \left(\mathrm{m}_{\mathrm{ts}}^{\mathrm{i}}, \mathrm{m}_{\mathrm{len}}^{\mathrm{i}}, \mathrm{m}_{\mathrm{flags}}^{\mathrm{i}}\right) \tag{8}$$

PreProcessing. We indicate the number of collected MTS is M and use $\mathrm{MTS}_{\mathrm{i}}$ to represent the i^{th}MTS. For $\mathrm{MTS}_{\mathrm{i}}$, we perform 3 steps of preprocessing on it:

- ***Step 1***: We sort all per-packet features in $\mathrm{MTS}_{\mathrm{i}}$ in ascending order based on their packet arrival timestamp using the merge sort algorithm. Then, we convert their packet arrival timestamps into arrival intervals:

$$\mathrm{m}_{\mathrm{ts}}^{1:\mathrm{L}} = \mathrm{m}_{\mathrm{ts}}^{1:\mathrm{L}-1} - \mathrm{m}_{\mathrm{ts}}^{0:\mathrm{L}-2},\ \ \mathrm{m}_{\mathrm{ts}}^{0} = 0 \tag{9}$$

- ***Step 2***: We perform the min-max normalization on $\mathrm{MTS}_{\mathrm{i}}$.
- ***Step 3***: Using the sliding window technique, we segment $\mathrm{MTS}_{\mathrm{i}}$ into multiple overlapping blocks, the j^{th} block is denoted as $\mathrm{MTS}_{\mathrm{ij}}$. These blocks collectively form an input batch for the RNN-VAE network, denoted as $\mathrm{Batch}_{\mathrm{i}}$:

$$\mathrm{Batch}_{\mathrm{i}} = \left\{ \mathrm{MTS}_{\mathrm{ij}} = \mathrm{MTS}_{\mathrm{i}}^{\mathrm{j}\cdot\mathrm{SL}:(\mathrm{j}+1)\cdot\mathrm{SL}} \mid \mathrm{j} \in \left[0, \lceil \frac{\mathrm{L_i}}{\mathrm{SL}} \rceil\right] \right\} \tag{10}$$

Here, $\mathrm{L_i}$ is the length of $\mathrm{MTS}_{\mathrm{i}}$, the sliding window size is SL, and the step size by which the window moves forward along the time dimension of $\mathrm{MTS}_{\mathrm{i}}$ is Stride.

DTEA Method. This method integrates RNN and VAE algorithms to construct an RNN-VAE network for inferring whether a network flow is benign or malicious. Specifically, the RNN is built using the GRU cell, which is for the transformation between MTS (i.e., the network flow data) and the embedding. The RNN-VAE network consists of two sub-networks: the variational network and the generative network, as shown in Fig. 6.

Variational Network. The variational network consists of the GRU cell and the fully connected layers (FCL). For each MTS, this sub-network transforms the MTS as the uncertain latent representation, i.e., the probability distribution representation. Specially, its uncertainty can be directly used for robust outlier analysis under inference mode. When this sub-network performs forward propagation, MTS is first serially fed into the GRU cell including NUM_LAYER stacked K-dimensional hidden layers. m^{i} and the historical hidden state $\mathrm{h}_{\mathrm{i}-1}$ are utilized to calculate the GRU cell variables at the i^{th} time step, including the update gate $\mathrm{u_i}$, reset gate $\mathrm{r_i}$ and candidate hidden state $\hat{\mathrm{h}}_{\mathrm{i}}$:

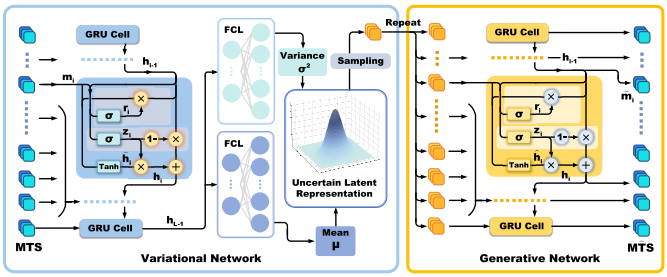


Fig. 6. The structure of GRU RNN-based VAE network.

$$\mathrm{u_i} = \sigma\left(\mathrm{W_u} \cdot \left[\mathrm{h_{i-1}}, \mathrm{m^i}\right]\right),\ \mathrm{r_i} = \sigma\left(\mathrm{W_r} \cdot \left[\mathrm{h_{i-1}}, \mathrm{m^i}\right]\right) \tag{11}$$

$$\hat{\mathrm{h}}_{\mathrm{i}} = \tanh\left(\mathrm{W_h} \cdot \left[\mathrm{r_i} \odot \mathrm{h_{i-1}}, \mathrm{m^i}\right]\right) \tag{12}$$

Then, $\mathrm{h_i}$ (the hidden state of the i^{th} time step) is updated by:

$$\mathrm{h_i} = (1 - \mathrm{u_i}) \odot \mathrm{h_{i-1}} + \mathrm{u_i} \odot \hat{\mathrm{h}}_{\mathrm{i}} \tag{13}$$

The final hidden state $\mathrm{h_{L-1}}$ is used as an embedding, containing the overall information of MTS.

To ensure the smoothness and continuity of the RNN-VAE latent space, we model the uncertain latent representation as a diagonal log-normal distribution with a dimension of LATENT_DIM. We indicate this distribution as $\mathrm{q}(\mathrm{s} \mid \mathrm{MTS})$, and it has two parameters consisting of the mean μ and the variance σ^2, where μ_{i} and ${\sigma^2}_{\mathrm{i}}$ are the values of μ and σ^2 in the i^{th} dimension respectively:

$$\mu = \mathrm{g_0}\left(\mathrm{h_{L-1}}\right) = \left(\mu_0, \ldots, \mu_{\mathrm{i}}, \ldots, \mu_{\mathrm{LATENT\_DIM}-1}\right) \tag{14}$$

$$\sigma^2 = \mathrm{g_1}\left(\mathrm{h_{L-1}}\right) = \left({\sigma^2}_0, \ldots, {\sigma^2}_{\mathrm{i}}, \ldots, {\sigma^2}_{\mathrm{LATENT\_DIM}-1}\right) \tag{15}$$

To determine the parameters of $\mathrm{q}(\mathrm{s} \mid \mathrm{MTS})$, the two FCLs ($\mathrm{g_0}$ and $\mathrm{g_1}$) separately transform $\mathrm{h_{L-1}}$ into μ and σ^2. To this end, the variational network samples a vector s with a dimension of LATENT_DIM, i.e., $\mathrm{s} = (\mathrm{s_0}, \ldots, \mathrm{s_i}, \ldots, \mathrm{s_{LATENT\_DIM}-1})$, from $\mathrm{q}(\mathrm{s} \mid \mathrm{MTS})$ and delivers it to the generative network.

Generative Network. The generative network only contains a GRU cell for seq2seq task. This sub-network aims to generate the $\hat{\mathrm{MTS}}$ from s and ensure that $\hat{\mathrm{MTS}}$ closely resembles MTS. In the forward propagation, s is repeated for L times and serially fed into the GRU cell which is stacked with NUM_LAYER K-dimensional hidden layers. For the i^{th} time step, the current hidden state $\mathrm{h_i}$ is output and linearly transformed to $\hat{\mathrm{m}}^{\mathrm{i}} \in \mathrm{R}^{1\times \mathrm{N}}$ by the dense matrix $\mathrm{Q} \in \mathrm{R}^{\mathrm{K_2}\times \mathrm{N}}$, where $\hat{\mathrm{m}}^{\mathrm{i}}$ is the i^{th} element in $\hat{\mathrm{MTS}}$ and $\hat{\mathrm{MTS}} = \left[\hat{\mathrm{m}}^{0}, \ldots, \hat{\mathrm{m}}^{\mathrm{i}}, \ldots, \hat{\mathrm{m}}^{\mathrm{L}-1}\right]$. We use $\mathrm{q}(\mathrm{MTS} \mid \mathrm{s})$ to represent the probability of reproducing the original MTS by s, i.e., the similarity between MTS and $\hat{\mathrm{MTS}}$.

Training Mode. We only use the MTS derived from the network flows considered benign for training. The RNN-VAE network is trained by optimizing the evidence lower bound (ELBO), where $\mathrm{p}(\mathrm{s})$ is the prior distribution of s:

$$\begin{aligned}\mathrm{ELBO} = \mathbb{E}_{\mathrm{q}(\mathrm{s}\,|\,\mathrm{MTS})}[\log[\mathrm{p}(\mathrm{MTS}\,|\,\mathrm{s})] \\ + \log[\mathrm{p}(\mathrm{s})] - \log[\mathrm{q}(\mathrm{s}\,|\,\mathrm{MTS})]]\end{aligned} \tag{16}$$

Note that $\mathrm{p}(\mathrm{s})$ is set to the standard log-normal distribution. The first term of ELBO is measured by the mean square error between MTS and $\hat{\mathrm{MTS}}$. And next two terms of ELBO are calculated by the Monte Carlo integration leveraging samples from $\mathrm{q}(\mathrm{s}\,|\,\mathrm{MTS})$. Beside, since the back propagation of the network cannot flow through the uncertainty nodes, we apply the reparameterization trick for sampling s from $\mathrm{q}(\mathrm{s}\,|\,\mathrm{MTS})$, where ϵ is the random noise sampled from standard normal distribution and used to maintain the randomness of s:

$$\mathrm{s} = \mu + \epsilon \odot \exp\left(\sigma^2/2\right) = (\mathrm{s}_0, \ldots, \mathrm{s}_\mathrm{i}, \ldots, \mathrm{s}_{\mathrm{LATENT\_DIM}-1}) \tag{17}$$

Inference Mode & Outlier Analysis. The RNN-VAE network performs the outlier analysis in the inference mode. Since the optimization goal of ELBO is to make $\mathrm{q}(\mathrm{s}\,|\,\mathrm{MTS})$ approach the prior distribution $\mathrm{p}(\mathrm{s})$, the similarity between $\mathrm{q}(\mathrm{s}\,|\,\mathrm{MTS})$ and $\mathrm{p}(\mathrm{s})$ can reflect whether a network flow related to the input MTS is an outlier distinguished from the training benign network flows. We compute $\mathrm{KL\,loss}$ to measure the similarity between $\mathrm{q}(\mathrm{s}\,|\,\mathrm{MTS})$ and $\mathrm{p}(\mathrm{s})$, where μ and σ^2 are the parameters of $\mathrm{q}(\mathrm{s}\,|\,\mathrm{MTS})$:

$$\mathrm{KL\,loss}(\mathrm{MTS}) = \sum_{\mathrm{i}=0}^{\mathrm{LATENT\_DIM}-1} \left(1 + \sigma^2{}_\mathrm{i} - \mu_\mathrm{i}^2 - \exp\left(\sigma^2{}_\mathrm{i}\right)\right) \tag{18}$$

In the inference mode, a network flow is classified as malicious if its $\mathrm{KL\,loss}$ exceeds a threshold, i.e., KL_LOSS_TH. We recommend defining this threshold using the x^{th} percentile of the computed $\mathrm{KL\,loss}$ values, denoted as $\mathrm{P_x}$. The initial value of KL_LOSS_TH is set to the $\mathrm{P_x}$ of all $\mathrm{KL\,loss}$ values observed during training.

To enable adaptive threshold tuning throughout REAPER's runtime, we adopt the exponentially weighted moving percentile (EWMP) to dynamically adjust KL_LOSS_TH. Specifically, we maintain a sliding window that stores the most recent 100 computed $\mathrm{KL\,loss}$ values. Each time a new $\mathrm{KL\,loss}$ value is obtained, the window is updated accordingly, and a local $\mathrm{P_x}$, denoted as $\mathrm{P_x^t}$, is computed to update the KL_LOSS_TH:

$$\mathrm{KL\_LOSS\_TH} = \alpha \cdot \mathrm{P_x^t} + \mathrm{KL\_LOSS\_TH} \cdot (1 - \alpha) \tag{19}$$

V. Experimental Evaluation

In this section, we evaluate the detection performance of REAPER by 20 groups malicious traffic and we measure its throughput and runtime overhead. The experimental results will answer the following questions:

1) Does REAPER have stronger detection performance compared to the baselines? (Section V-D)
2) Does REAPER exhibit the robustness to detect encrypted malware traffic that is easily confused with benign traffic? (Section V-E)
3) Can REAPER run in high-bandwidth network scenarios? (Section V-F)

TABLE III
Hyper-Parameter Settings for REAPER

Hyper-parameter	Description	Value
RQ_SIZE	The size of ring queue.	10^7
FETCH_NUM	The number of dequeued ring queue slots.	2^{17}
TIME_OUT	The maximum allowed packet interval.	16s
LONG_TH	The minimum length of the long flow.	Determined by training traffic datasets.
INSPECT_INTERVAL	The flow table snapshot interval.	2s
AGGR_CYCLE	The number of short flows inserted when dynamic aggregation begins.	10000
MIN_IP_PREFIX_LEN	The shortest IP prefix allowed during dynamic aggregation.	24
AGGR_MAX	The maximum length of the aggregated flow.	500
SL	The sliding window size.	LONG_TH
Stride	The step size by which the sliding window moves each time.	0.25 · SL
NUM_LAYER	The number of stacked hidden layers in the GRU cell.	Determined in Section V-C.
LATENT_DIM	The dimension of the latent space in the VAE.	Determined in Section V-C.
K	The dimension of the hidden layer in the GRU cell.	50

A. Implementation

We utilize C/C++ with GCC 9.4.0 to prototype REAPER with Intel DPDK [33]. We adopt the PcapPlusPlus library 22.11 to call the APIs of Intel DPDK 19.11.14 LTS as C++ wrappers. Overall, we consume more than 4000 lines of codes.

We assign each module a group of logical cores on the CPU to execute their respective worker threads in parallel. Each worker thread is an instance of the pcpp::DpdkWorkerThread class and exclusively occupies a logical core, meaning that the number of worker threads of each module is equal to the number of logical cores allocated to it.

We have summarized all the hyper-parameters in REAPER and provided recommended values in Table III.

Parser Impl. We mount the NIC ports to the UIO driver, thereby bypassing kernel interrupts and directly managing NIC ports with userspace mode. Based on this, each parser worker thread handles respective RX queues of NIC ports and calls the APIs of Intel DPDK 19.11.14 LTS to continuously extract the packets arriving at the RX queue.

Additionally, we implement the ring queue in the Parser module as an array to store per-packet features. We use the std::memcpy() API in the C standard library to implement the dequeuing operation in batch.

Assembler Impl. We implement the flow table in the Assembler module using the tbb::concurrent_hash_map container provided by oneTBB 2022.0.0. Each table entry has its own independent lock, allowing the assembler worker threads to update the flow table with high concurrency after fetching the per-packet features from the ring queue in the Parser module. Additionally, We also utilize the tbb::parallel_for_each() and tbb::parallel_invoke() APIs to orchestrate the assembler worker threads for parallel scanning of the flow table snapshot.

The flow IDs of the incomplete network flows are then placed into $\mathrm{Set}_{\mathrm{next}}^{\mathrm{hist}}$, which is implemented by the concurrency-supported tbb::concurrent_unordered_set container.

Aggregator Impl. We achieve the mapping between the path and the node at depth MIN_IP_PREFIX_LEN in the IP-Trie through the tbb::concurrent_hash_map container. These nodes are evenly distributed to multiple aggregator worker threads for management. As a result, the worker threads are independently responsible for maintaining distinct sets of non-overlapping subtrees, where the root of each subtree is a node at depth MIN_IP_PREFIX_LEN in the IP-Trie. This design enables the parallel execution of IP-Trie insertion. Beside, we employ the tbb::concurrent_hash_map container to implement the $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{left}}$ and $\mathrm{Set}_{\mathrm{leaf}}^{\mathrm{right}}$. During the leaf node dynamic aggregation, we call the tbb::parallel_for_each( ) and tbb::parallel_invoke( ) APIs to arrange the aggregator worker threads for parallel access to the leaf nodes in each aggregation iteration.

Detector Impl. In training mode, once the training data has been fully collected, the Detector module invokes a script written in PyTorch 2.0.1 and Python 3.8.0 to train the RNN-VAE network model using a GPU. In inference mode, the Detector module loads the trained RNN-VAE network model via LibTorch C++ 2.0.1, performing real-time parallel inference for network flows by multiple detector worker threads.

Others. We use the std:::move( ) API, introduced in C++11, to adopt move semantics for transferring set contents, thereby eliminating deep copies. Network flow queues, i.e., short-flow, long-flow, and aggregated-flow queues, are implemented using tbb::concurrent_queue from oneTBB 2022.0.0, ensuring efficient and thread-safe inter-module communication.

B. Experiment Setup

Baselines. We establish three baselines, which all adopt unsupervised methods, for evaluating the improvement brought by REAPER:

- **Whisper.** Whisper [12] is a lightweight cluster-based detection method prototyped by Intel DPDK. It divides the traffic into network flows based on SrcIP and uses discrete Fourier transform to extract network flow features. Finally, outlier analysis is performed using the K-means clustering method. We build on its original open source code [34] and only tune its hyper-parameters to obtain acceptable performance.
- **Kitsune.** Kitsune [29] is an autoencoder-based detection method. It accumulates stateful statistical features at four different scales: <SrcMac, SrcIP>, < SrcIP >, <SrcIP, DstIP>, and <SrcPort, DstPort>. When a packet arrives, all statistical features related to the packet will be used for per-packet inference using integrated autoencoders. Since it cannot handle high-bandwidth traffic, we use its open source offline implementation [35] with built-in hyper-parameteters.
- **FlowLens.** FlowLens [26] is a compact traffic feature extraction method deployed by P4 switches. It collects the distributions of both packet lengths and arrival intervals towards each network flow identified by <SrcIP, DstIP, SrcPort, DstPort, protocol>. Then, the counts of the distribution buckets are treated as extracted features to describe a network flow compactly. We build FlowLens by its open source code [36] and incorporate with the K-means algorithm to analyze its extracted features, resulting in unsupervised malicious traffic detection.

Testbed. The REAPER and Whisper prototypes are implemented using Intel DPDK, and deployed on a testbed server compromising Ubuntu 20.04 operating system (Linux 5.4.0), one Intel Xeon E5-2680 v4 CPU (32GB RAM, 14 physical cores, and 28 logical cores), as well as one Intel I210 NIC (1000 Mbps, one port with 4 RX queues, and Intel DPDK supported). We allocate 1024 huge pages, each 2MB in size, and use the IGB_UIO kernel module to enable userspace I/O for the Intel DPDK runtime environment. In particular, receive side scaling (RSS) is enabled to achieve load balancing across the 4 RX queues. Additionally, we connect another the same server to the testbed server via a 1000 Mbps network cable and utilize it to generate traffic for testbed server. Concretely, we leverage Tcpreplay tool with the '-K' flag to replay traffic datasets at their original rates.

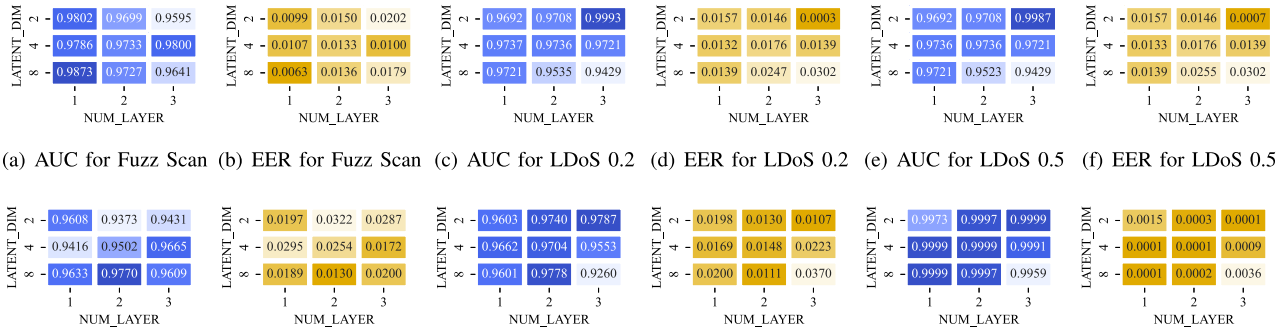
The FlowLens prototype is deployed on a software P4 switch, i.e., Behavioral Model v2, running on the testbed server. Traffic datasets are also replayed using Tcpreplay and injected into the switch port. As for Kitsune, which is unable to handle high-bandwidth traffic in real time, we prototype it by a Python script and evaluate it on the testbed server offline.

The configuration of the REAPER prototype is as follows. The main thread occupies a single logical core. The Parser module is allocated 4 logical cores, while the Assembler module is assigned 8 logical cores, with 4 dedicated to updating the flow table and the remaining 4 responsible for scanning flow table snapshots and performing flow table eviction. The Aggregator module is allocated 8 logical cores, with 4 dedicated to the IP-Trie insertion and the other 4 handling the dynamic aggregation of IP-Trie leaf nodes. Finally, the Detector module is assigned 6 logical cores to execute MTS preprocessing and inference. In total, REAPER occupies 27 logical cores out of 28 available logical cores. In particular, REAPER exhibits the scalability, as it can flexibly increase the number of assigned logical cores in each module to achieve higher performance, with support from more powerful CPUs.

Traffic Dataset. It is important to first declare that the payload of all traffic datasets has been truncated to ensure that, when using a 1000 Mbps network cable, the traffic can be replayed at the original rate of the dataset using Tcpreplay.

We used six different WIDE backbone network traffic datasets as background benign traffic, which are collected on 2024.10.01, 2024.10.05, 2024.10.09, 2024.10.13, 2024.10.17, and 2024.10.21, respectively.

In the detection performance evaluation, we use the first 30% of packets from the benign background traffic dataset to train the detection models, including the RNN-VAE network in REAPER, the K-means model in Whisper, and the autoencoders in Kitsune. Additionally, these first 30% of packets were also utilized to determine the LONG_TH hyper-parameter for REAPER. Subsequently, the remaining 70%



(a) AUC for Fuzz Scan (b) EER for Fuzz Scan (c) AUC for LDoS 0.2 (d) EER for LDoS 0.2 (e) AUC for LDoS 0.5 (f) EER for LDoS 0.5

(g) AUC for SSDP DoS (h) EER for SSDP DoS (i) AUC for UDP DoS (j) EER for UDP DoS (k) AUC for SSL DoS (l) EER for SSL DoS

Fig. 7. AUC and EER measurement of REAPER on Cyber attacks with different configurations for RNN-VAE network Hyper-parameters.

of packets, along with the malicious traffic dataset will be replayed together using Tcpreplay to evaluate the detection performance of both REAPER and baselines.

Overall, the traffic dataset configurations for evaluating detection performance across different types of malicious traffic are listed in Table IV. We select 20 groups of malicious traffic datasets, covering cyber attacks and malware traffic, where cyber attacks include flooding attacks and stealthy attacks:

- **Flooding Attacks.** These attacks utilize flooding techniques to send high-volume traffic or large magnitude requests to overwhelm the victim network. We collect four types of flooding attack traffic from the Kitsune dataset [29] and the CIC-DDoS2019 dataset [37]. These attacks include SrcPort flooding (Fuzz Scan), link flooding (UDP DoS attacks), request flooding (SSDP DoS attacks and SSL Renegotiation DoS attacks).
- **Steathy Attacks.** These attacks intermittently send malicious traffic to stealthily exploit vulnerabilities in network protocols, causing the target network to experience denial of service. We implement a specific type of stealthy attack, namely Low-rate DoS (LDoS) attacks [38], [39], which send periodic burst video traffic that triggers momentary network congestion. This causes the actual bandwidth usage of the victim network to be forced down due to the response of congestion control algorithms (e.g., CUBIC [40]). We launch LDoS attacks with pulse periods of 0.2s and 0.5s in a real-world WAN topology simulated using Mininet [41], and collect the resulting attack traffic as two datasets, referred to as LDoS 0.2 and LDoS 0.5, respectively. The detailed properties of the topology (number of switching nodes, link delays, etc.) are from the Topology Zoo dataset [42].
- **Malware Traffic.** Malware traffic is crafted by attackers and carries malicious software that compromises the host. Owing to its low rate, it can be confused with benign traffic to evade detection. To demonstrate the robustness of REAPER, we further use Malware traffic for additional evaluation. We collect 14 types of Malware traffic from the CIC-AndMal2017 dataset [43], covering Ransomware traffic and Adware traffic.

Metrics. We adopt four metrics: 1) the area under ROC curve (AUC); 2) the true positive rate (TPR); 3) the false positive rate (FPR); 4) the equal error rate (EER). All metrics are computed based on the per-flow scores (i.e., KL loss values) produced by the RNN-VAE network inference.

C. Hyper-Parameter Selection for RNN-VAE Nertwork

We investigate the impact of different RNN-VAE network hyper-parameters, specifically LATENT_DIM and LAYER_NUM, on the detection performance of REAPER for cyber attacks. The candidate values for LATENT_DIM are 2, 4, and 8, while those for LAYER_NUM are 1, 2, and 3. The number of training iterations is limited to 50. As illustrated in Fig. 7, we present REAPER's detection performance for each cyber attack across 9 hyper-parameter combinations.

When NUM_LAYER is set to 1, increasing LATENT_DIM enhances REAPER's detection performance for Fuzz Scan and SSL DoS, whereas the performance for other cyber attacks fluctuates only slightly, with minimal change.

When LATENT_DIM is set to 2, increasing the value of NUM_LAYER improves REAPER's detection performance for most cyber attacks, except for Fuzz Scan and SSDP DoS. The detection performance for these two attacks even obliviously declines, which contrasts with the effects observed when increasing LATENT_DIM while keeping NUM_LAYER fixed. This is because increasing NUM_LAYER results in a greater increase in the model's parameter count compared to increasing LATENT_DIM. With a limited number of training iterations, the RNN-VAE network is more difficult to converge.

In short, both increasing NUM_LAYER and LATENT_DIM improve REAPER's detection performance. However, when the required number of training iterations is insufficient, the RNN-VAE network model struggles to converge, and further increasing NUM_LAYER or LATENT_DIM can have the opposite effect. The selection of RNN-VAE network hyper-parameters is therefore a trade-off between REAPER's detection performance and the model's training time.

Additionally, as shown in Fig. 8, we measure the inference latency of the RNN-VAE network model across the 9 hyper-parameter combinations. As expected, increasing NUM_LAYER significantly increases the model's parameter

TABLE IV

THE CONFIGURATIONS OF TRAFFIC DATASETS FOR DETECTION PERFORMANCE EVALUATION

Malicious Traffic Scope	Type	Malicious Traffic Description	Malicious Traffic Dataset Source	Malicious Traffic Bandwidth	Benign Background Traffic	Benign Traffic Bandwidth	Ratio of Malicious Packets
Cyber Attacks (Flooding Attacks)	SSDP DoS	Flooding by spoofed SSDP requests, and reflecting extensive replies.	Online Open Source Dataset	29.922 Mbps	WIDE 2024.10.13	1.1286 Gbps	20.817%
	UDP DoS	Saturating network links with high volumetric UDP traffic.	Online Open Source Dataset	620.10 Mbps	WIDE 2024.10.21	3.1550 Gbps	32.296%
	SSL DoS	Flooding by SSL renegotiation message without reply for occupying resources.	Online Open Source Dataset	1.4610 Mbps	WIDE 2024.10.17	4.5147 Gbps	0.495%
	Fuzz Scan	Scanning for potential protocol vulnerabilities.	Online Open Source Dataset	0.8299 Mbps	WIDE 2024.10.05	1.5121 Gbps	0.0212%
Cyber Attacks (Stealthy Attacks)	LDoS 0.2	Triggering congestion event by UDP pulse traffic with a period of 0.2/0.5s. (TCP-targeted LDoS Attacks)	Attack Simulation	11.391 Mbps	WIDE 2024.10.09	2.7052 Gbps	10.218%
	LDoS 0.5			8.1305 Mbps			7.158%
Malware Traffic	Ransomware	Including the malicious software of Charger, Jisut, Koler, Lockerpin, Pletor, Porndroid, and Wannalocker	Online Open Source Dataset	Average 0.0692 Mbps	WIDE 2024.10.01	3.0281 Gbps	Average 0.331%
	Adware	Including the malicious software of Dowgin, Ewind, Kemoge, Koodous, Mobidash, Shuanet, and Youmi	Online Open Source Dataset	Average 0.1358 Mbps			Average 0.317%

TABLE V

DETECTION PERFORMANCE MEASUREMENT OF REAPER AND BASELINES ON CYBER ATTACKS

Methods	Kitsune				Whisper				FlowLens				REAPER														
Metrics	AUC	TPR	FPR	EER	AUC	TPR	FPR	EER	AUC	TPR	FPR	EER	AUC	vs.K↑	vs.W↑	vs.F↑	TPR	vs.K↑	vs.W↑	vs.F↑	FPR	vs.K↓	vs.W↓	vs.F↓	EER	vs.K↓	vs.W↓	vs.F↓
SSDP DoS	0.8168	0.7880	0.2345	0.2297	0.9468	0.9999	0.0532	0.0266	0.9644	0.9859	0.0750	0.0578	0.9770	↑0.1602	↑0.0303	↑0.0126	0.9999	↑0.2119	-	↑0.0140	0.0260	↓0.2084	↓0.0272	↓0.0490	0.0130	↓0.2167	↓0.0136	↓0.0448
UDP DoS	0.8693	0.9560	0.2825	0.2256	0.9720	0.9911	0.0372	0.0369	0.8910	0.9959	0.1246	0.1462	0.9787	↑0.1094	↑0.0067	↑0.0877	0.9999	↑0.0439	↑0.0086	↑0.0004	0.0213	↓0.2612	↓0.0159	↓0.1033	0.0107	↓0.2150	↓0.0262	↓0.1355
SSL DoS	0.8351	0.8920	0.2377	0.2089	0.9399	0.9999	0.0895	0.0448	0.6444	0.7081	0.4289	0.4151	0.9999	↑0.1648	↑0.0600	↑0.3555	0.9999	↑0.1079	-	↑0.2918	0.0001	↓0.2376	↓0.0894	↓0.4288	0.0001	↓0.2088	↓0.0447	↓0.4150
Fuzz Scan	0.9957	0.9980	0.0026	0.0023	0.9855	0.9999	0.0387	0.0564	0.6440	0.7172	0.2506	0.2720	0.9873	-	↑0.0018	↑0.3433	0.9999	↑0.0019	-	↑0.2827	0.0127	-	↓0.0260	↓0.2379	0.0063	-	↓0.0501	↓0.2657
LDoS 0.2	0.6006	0.4180	0.2522	0.4875	0.9510	0.9519	0.1112	0.1026	0.9999	0.9999	0.0001	0.0001	0.9993	↑0.3988	↑0.0483	-	0.9999	↑0.5819	↑0.0480	-	0.0007	↓0.2515	↓0.1105	-	0.0003	↓0.4871	↓0.1023	-
LDoS 0.5	0.7789	0.6380	0.2818	0.3314	0.9340	0.9347	0.1218	0.1099	0.9999	0.9999	0.0001	0.0001	0.9987	↑0.2198	↑0.0647	-	0.9999	↑0.3619	↑0.0652	-	0.0013	↓0.2805	↓0.1205	-	0.0007	↓0.3308	↓0.1092	-

[1] We use "K", "W", and "F" to represent the abbreviations of Kitsune, Whisper, and FlowLens, respectively.
[2] We use blue to mark the **metric improvement** compared with baselines. Meanwhile, we use yellow to mark the **best** results and red to mark the **worst** results.

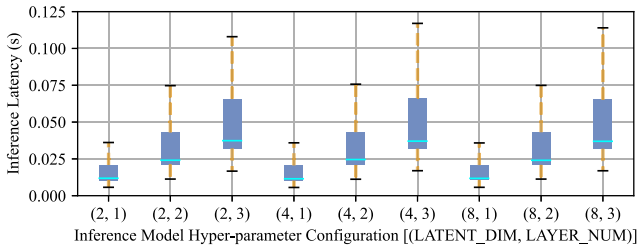


Fig. 8. Measurement of RNN-VAE network inference latency.

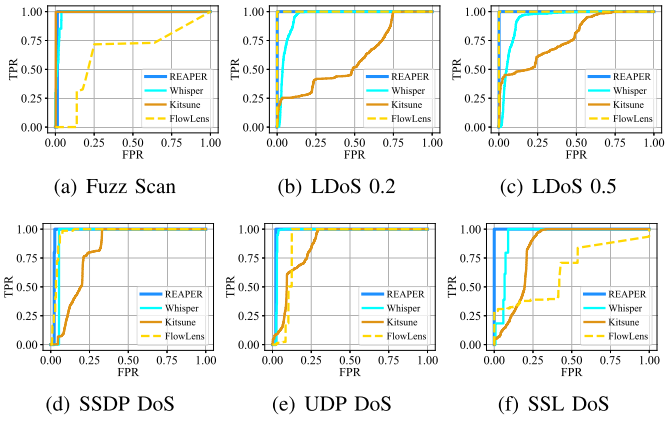


Fig. 9. ROC curves when detecting cyber attacks.

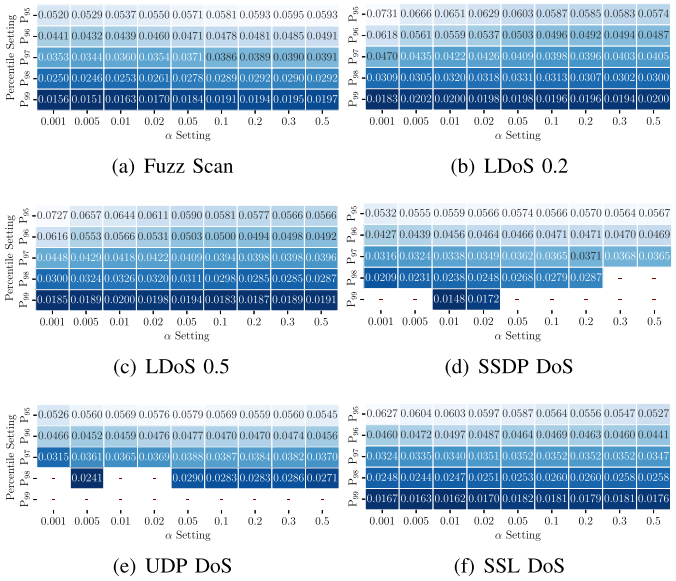
count, which in turn raises this latency, while increasing LATENT_DIM has a negligible effect on this latency.

D. Detection Performance

As shown in Table V, we present the best detection metrics of REAPER and the baselines for cyber attacks. REAPER outperforms the baselines in most metrics for detecting 6 types of cyber attacks. For the SSDP DoS, although Whisper achieves a comparable TPR to REAPER, all other metrics of Whisper are lower than those of REAPER. For the Fuzz Scan, Kitsune achieves the best AUC (0.9957), slightly higher than REAPER, but REAPER achieves the best TPR (0.9999). In the case of LDoS 0.2/0.5 that consists mainly of long flows, FlowLens shows the best AUC (0.9999). However, for attacks primarily composed of short flows (e.g., SSDP DoS, UDP DoS, SSL DoS, and Fuzz Scan), REAPER significantly outperforms FlowLens across all metrics.

As shown in Fig. 9, we present the ROC curves for REAPER and the baselines in detecting cyber attacks. Both REAPER and Whisper achieve a balance of low FPR and high TPR across all 6 cyber attack detection tasks. However, at the same TPR, REAPER maintains a lower FPR. In contrast, Kitsune and FlowLens are only able to achieve a balance of low FPR and high TPR when detecting the Fuzz Scan traffic and LDoS traffic, respectively.

These posterior detection metrics indicate that the scores (i.e., KL, loss values) produced during REAPER's runtime effectively distinguish between benign and malicious traffic, demonstrating REAPER's strong overall detection performance. In real-world deployments, configuring the threshold (i.e., KL_LOSS_TH) during REAPER's runtime is essential. We further enable it with EWMP-based adaptive tuning to assess REAPER's detection performance in terms of TPR and FPR. We independently configure P_x as P_{95}, P_{96}, P_{97}, P_{98}, and P_{99}, each combined with a set of α values: 0.001, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.3, and 0.5. As shown in Fig. 10, we report the FPR of REAPER under various combinations of P_x and α, considering only those combinations where REAPER achieves a TPR exceeding 0.9999. Smaller α can reduce false positives (FPs) in scenarios with frequent traffic fluctuations. For example, the LDoS 0.5/0.2, which exhibits periodic burst behavior, achieved the lowest FPR at $\alpha = 0.001$, while UDP



(a) Fuzz Scan (b) LDoS 0.2

(c) LDoS 0.5 (d) SSDP DoS

(e) UDP DoS (f) SSL DoS

Fig. 10. Impact of different percentile and α settings on REAPER's FPR when enabling EWMP-based adaptive threshold tuning.

DoS, which causes traffic surges, attained the lowest FPR at $\alpha = 0.005$. In contrast, moderately larger α can better identify malicious traffic and mitigate FPs in stable traffic scenarios. For instance, both SSL DoS and SSDP DoS rely on request flooding, which causes slight traffic fluctuations and achieves the lowest FPR at $\alpha = 0.01 \sim 0.02$.

E. Robustness of Detection

To further evaluate the robustness of REAPER in detecting malicious traffic, we conduct evaluation from: 1) malware traffic detection, and 2) hybrid cyber attack traffic detection.

Malware Traffic Detection. We first use encrypted malware traffic to evaluate the robustness of REAPER's malicious traffic detection. As shown in Fig. 11, the results demonstrate that REAPER has acceptable detection performance on encrypted malware traffic, with an average AUC of 0.9538, an average EER of 0.1156, an average TPR of 0.8989, and an average FPR of 0.0908. The malware traffic is generated by infected bots sending extensive short flows, which can easily be confused with benign traffic. However, REAPER can effectively detect encrypted malware traffic by dynamically aggregating short flows and identifying the traffic sending patterns of infected bots. On the other hand, the average AUC of both Whisper and Kitsune is close to 0.5, indicating that they struggle to identify encrypted malware traffic.

Hybrid Cyber Attack Traffic Detection. To explore the impact of evolving cyber attack patterns on REAPER's detection performance, we mix the 6 cyber attack traffic datasets and replay them alongside the benign background traffic dataset to measure REAPER's detection performance. Similarly, we use the first 30% of the packets from each benign background traffic dataset to train the RNN-VAE network model, while the remaining 70% are replayed alongside the hybrid cyber attack traffic dataset. After mixing the cyber attack traffic datasets, as shown in Fig. 12, the results indicates that REAPER's AUC and EER in detecting 6 types of cyber attacks exhibit slight reductions compared to the optimal values listed in Table V. Nonetheless, REAPER maintains an average AUC of 0.9429 and an average EER of 0.0348, demonstrating its robustness in handling evolving cyber attack patterns.

F. Throughput and Memory Footprint

Throughput. We utilize 3 traffic loads to measure the throughput of REAPER, namely the WIDE backbone network traffic datasets from 2024.10.01, 2024.10.13, and 2024.10.17. Tcpreplay is utilized to replay the traffic loads, with the '-t' flag employed to maximize the replay rate, enabling us to measure the peak throughput of REAPER. For each module of REAPER, we perform 30 throughput measurements.

As shown in Fig. 13, for the Parser module, the packet parse throughput under the 3 traffic loads is significantly concentrated around 8 Gbps, 7 Gbps, and 9.5 Gbps, respectively. While the flow table update throughput of the Assembler module under the 3 traffic loads is distributed around 8 Gbps, 5.5 Gbps, and 9.5 Gbps, respectively.

Compared to packet parse throughput, the distribution of flow table update throughput exhibits a greater variance. Despite setting independent locks for each flow table entry, concurrent access to the same entry still causes the flow table update throughput to decrease at certain measurement points, leading to wider range fluctuations in the measurement results. Nevertheless, the Parser and Assembler modules still exhibit similar peak throughput, both reaching up to approximately 10 Gbps, although the throughput of the Assembler module shows fluctuations with wider range.

For the Aggregator and Detector modules, which are computation-intensive, the number of logical cores allocated to them directly determines their maximum achievable throughput. To explore the impact of increasing the number of allocated logical cores on the throughput of both modules, we strategically reallocate the total of 14 logical cores originally assigned to them (8 for the Aggregator and 6 for the Detector). Specifically, we investigate the throughput of the Detector module when allocated 12, 10, 8, 6, 4, and 2 logical cores, meanwhile, we explore the throughput of Aggregator module when allocated 2, 4, 6, 8, 10, and 12 logical cores. We measure the throughput of each sub-step in both the Aggregator and Detector modules. The sub-steps in the Aggregator module include IP-Trie insertion and IP-Trie leaf node aggregation, while those of the Detector module include MTS preprocessing and RNN-VAE network inference.

As shown in Fig. 15, Fig. 14, Fig. 16, and Fig. 17, under the 3 traffic loads, the throughput of each sub-step in the Aggregator and Detector modules is positively correlated with the number of allocated logical cores. Specifically, as for the Aggregator module, with 12 assigned logical cores, the IP-Trie insertion throughput reaches a maximum of 10 Gbps, as shown in Fig. 14(c), while the IP-Trie leaf node aggregation throughput peaks at 4.5 Gbps, as shown in Fig. 15(c). As for the Detector module, with 12 assigned logical cores, the MTS preprocessing throughput reaches a maximum of 225 Gbps, as shown in Fig. 16(b), and the RNN-VAE network inference throughput peaks at 5 Gbps, as shown in Fig. 17(c).

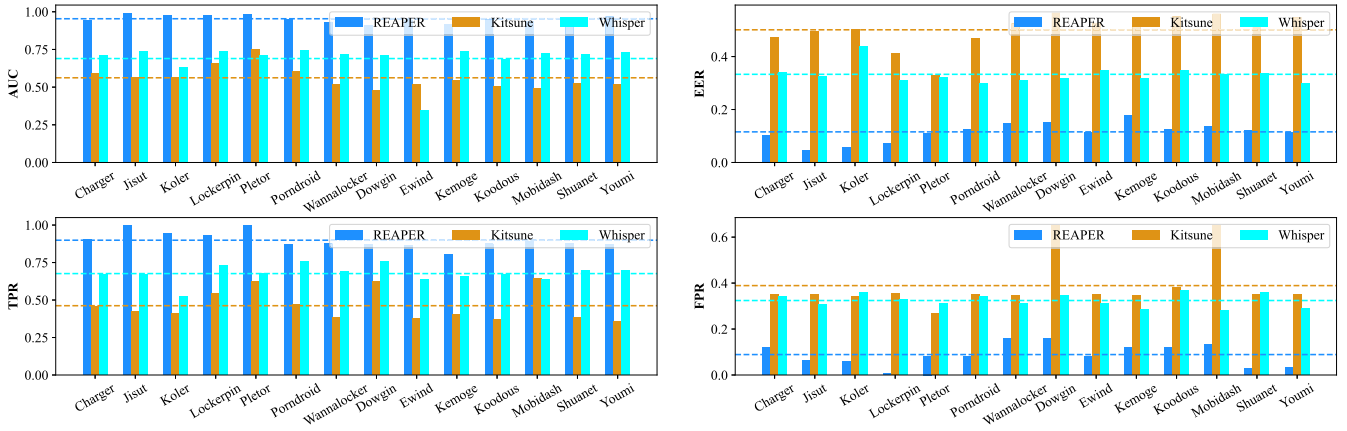


Fig. 11. Detection performance measurement of REAPER and baselines on Malware traffic (The Dotted Lines represent the metric mean values).

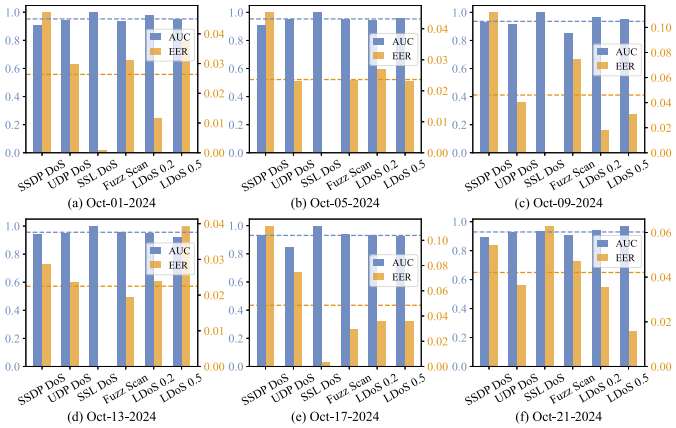


Fig. 12. Detection performance measurement of REAPER on hybrid cyber attacks under different background benign traffic (The Dotted Lines Represent the Metric Mean Values).

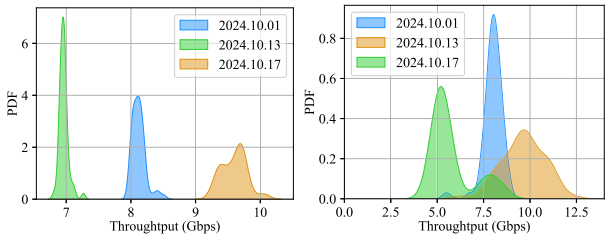


Fig. 13. Throughput of REAPER for packet parse and flow table update.

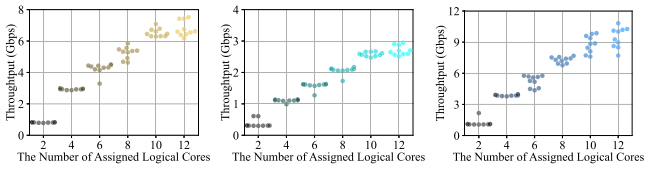


Fig. 14. IP-Trie insertion throughput in the Aggregator module with varying numbers of assigned logical cores.

Therefore, REAPER possesses the potential to handle high-bandwidth throughput traffic, with the scalability to allocate more logical cores to each module on more powerful CPUs.

Runtime Memory Footprint. We use Tcpreplay to replay the WIDE 2024.10.01, WIDE 2024.10.13, and WIDE 2024.10.17 traffic loads at their original rates to measure

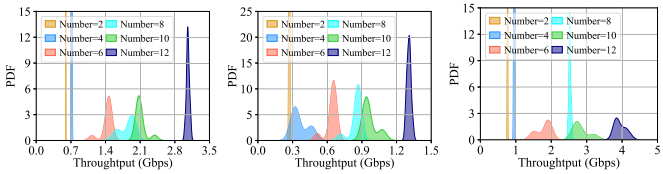


Fig. 15. IP-Trie leaf aggregation throughput in the aggregator module with varying numbers of assigned logical cores.

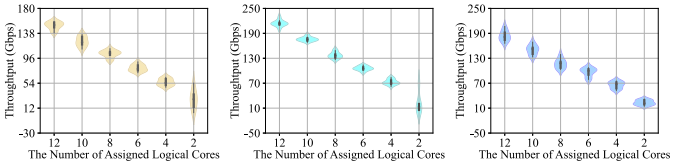


Fig. 16. MTS preprocessing throughput in the detector module with varying numbers of assigned logical cores.

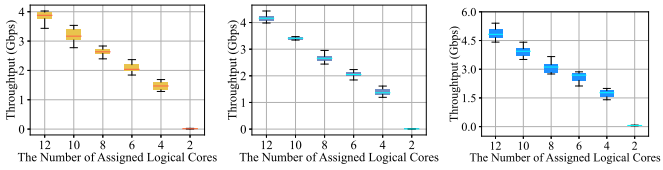


Fig. 17. RNN-VAE network inference throughput in the detector module with varying numbers of assigned logical cores.

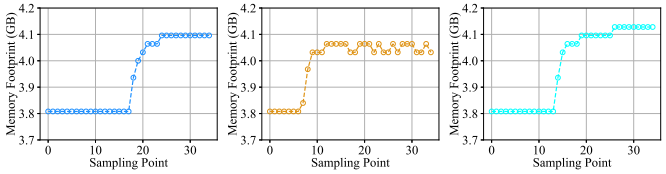


Fig. 18. Memory footprint measurement of REAPER.

REAPER's runtime memory footprint. As shown in Fig. 18, under all 3 traffic loads, the runtime memory footprint ultimately converges to around 4.1 GB.

VI. Related Work

We categorize existing network anomaly detection work into three major types based on detection units: 1) log-based,

TABLE VI
RELATED WORK REVIEW

	Method	Analysis Strategy	Deployment	Real-time Detection over High-bandwidth Networks	Unknown Attack Detection
Log-based	CoToRu [44]	State Transition	Import to Zeek	✗	✓
	Paradise [45]	Provenance	NFV		
	RETSINA [11]	Meta-learning	Offline		
	The work in [10]	Federated PCA			
	Mlog [46]	LSTM and CNN			✗
Network Flow-based	SketchLib [47]	Sketch	P4 Switch	✓	✗
	PLUTO [48]	XGBoost			
	Horuseye [17]	iForest			✓
	The work in [9]	AE and iForest	SDN Switch	✗	✓
	Whisper [12]	DFT and Kmeans	Intel DPDK	✓	
	Kitsune [29] The work in [49]	KitNet	Offline	✗	
	H2ID [50] The work in [51]	OC-SVM and AE			
	The work in [52]	CNN			✗
Interaction Graph-based	Nadege [53]	Graph Kernel	Offline	✗	✗
	The work in [54]	DLGNN			
	Graph2vec+RF [55]	Graph2vec and RF			✓
	HyperVision [27]	DBSCAN and Kmeans	Intel DPDK	✓	

2) network flow-based, and 3) interaction graph-based. The comparison of existing work is shown in Table VI.

Log-based. CoToRu [44] converts normal log-derived behaviors into a state transition table for real-time anomaly detection via Zeek, anomalous behavior is identified when the state transition fails to reach a termination state. Paradise [45] detects APT attacks through online causal provenance analysis using NFV. RETSINA [11] applies meta-learning to identify zero-day vulnerabilities from limited HTTP logs. The work in [10] proposes the federated PCA to unsupervisedly analyze anomalous network behaviors in logs across multiple parties. Mlog [46] represents logs as semantic vectors via transformers, and detects anomalies using LSTM and CNN. However, log-based methods require protecting the logs from tampering, and the log cache needs to occupy significant storage space.

Network Flow-based. SketchLib [47] designs efficient sketch structures in P4 switches for detecting DDoS attacks. PLUTO [48] and Horuseye [17] deploy XGBoost and iForest-based models on P4 switches for per-flow anomaly detection. The work in [9] applies autoencoder-based models for anomaly detection on SDN switches. Whisper [12], built on Intel DPDK, uses DFT for flow feature extraction and K-means for clustering. Kitsune [29] introduces KitNet, an ensemble of autoencoders, for flow-based anomaly detection, later enhanced in [49]. H2ID [50] and the works in [51], [52] further explore ML-based anomaly detection at the flow level. However, these network flow-based methods analyze individual network flows, making it difficult to detect covert malicious traffic.

Interaction Graph-based. Nadege [53] models network flows as hierarchical graphs and introduces a novel graph kernel for feature extraction. HyperVision [27] and the work in [54] represent flows as bidirectional graphs with sockets as vertices. The former uses subgraph structures and edge features for clustering, while the latter employs the Dynamic Line Graph Neural Network (DLGNN). Graph2vec+RF [55] also treats flows as bidirectional graphs, encoding packet lengths as vertices. However, constructing interaction graphs for network flows generates significant overhead, and the interaction graphs of numerous network flows are highly dense. Subgraph decomposition is an NP-complete problem, and its solution exhibits high time complexity.

VII. LIMITATION AND DISCUSSION

A. Inherent Challenges of Unsupervised Methods

Unsupervised methods are inherently challenged by evolving cyber attack patterns and potential FPs. However, REAPER, which employs unsupervised RNN-VAE model trained on benign real-world traffic datasets, still achieves a strong average AUC of 0.9538 against evolving cyber attack patterns (Section V-E). To address potential FPs, inspired by pVoxel [56], which finds that benign network flows misclassified as FPs often lie sparsely in feature space, we propose analyzing sparsity in the VAE's latent space representations to mitigate majority of FPs.

B. Threshold-Based Classification of Long and Short Flows

REAPER classifies network flows based on a threshold LONG_TH. If attackers learn this value, they can evade detection by manipulating short flows to slightly exceed LONG_TH, avoiding aggregation. To address this, we propose: 1) Periodically updating LONG_TH with new training traffic datasets. 2) Using ML algorithms to learn a nonlinear classifier for distinguishing long and short flows.

C. Investigation of Adaptive Smoothing Parameter in EWMP

The smoothing parameter α in EWMP controls the evolution of KL_LOSS_TH, thereby influencing REAPER's runtime detection performance. We recommend two strategies for adaptively tuning α in response to varying traffic conditions. 1) Variance-based. EWMP uses a sliding window to track recent KL loss values. A higher variance within the window indicates greater volatility, prompting the use of a smaller α to ensure smoother updates. Conversely, a lower variance allows for a larger α to improve responsiveness. 2) Reinforcement Learning-based. The adjustment of α could be formulated as a reinforcement learning optimization problem, where the agent observes the distribution of KL loss within the sliding window and the runtime trend of FPR, and selects the optimal α to maximize detection performance.

VIII. CONCLUSION

In this paper, we propose REAPER, a high-performance system for real-time detection of malicious traffic at the network flow level. Noting the limited information in short flows and their low distinguishability, which causes benign and malicious short flows to be easily confused, we develop the

IP-Trie to dynamically aggregate short flows based on their IP prefixes into more informative aggregated flows. Additionally, instead of traditional statistical feature extraction, we propose the DTEA method for detecting network flow data, which uses RNN to directly convert network flow data into embeddings, followed by outlier analysis on the embeddings using VAE.

We prototype REAPER using Intel DPDK, evaluation Results show REAPER outperforms baselines and can handle high-bandwidth traffic.

In future work, we will focus on optimizing concurrent access to critical sections in REAPER, such as the flow table in the Assembler module, to enhance system stability. We also plan to investigate the impact of deploying REAPER on more powerful CPUs, with the goal of evaluating how an increased number of logical cores affects its overall throughput. In addition, we intend to integrate REAPER with advanced lifelong learning techniques [57] to strengthen its long-term robustness against evolving malicious traffic.

REFERENCES

[1] Z. Liu et al., "Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric DDoS attacks with programmable switches," in *Proc. USENIX Secur. Symp.*, 2021, pp. 3829–3846.

[2] F. Erlacher and F. Dressler, "On high-speed flow-based intrusion detection using snort-compatible signatures," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 1, pp. 495–506, Jan. 2022.

[3] D. Tang, R. Dai, C. Zuo, J. Chen, K. Li, and Z. Qin, "A low-rate DoS attack mitigation scheme based on port and traffic state in SDN," *IEEE Trans. Comput.*, vol. 74, no. 5, pp. 1758–1770, May 2025.

[4] S. Cui, C. Dong, M. Shen, Y. Liu, B. Jiang, and Z. Lu, "CBSeq: A channel-level behavior sequence for encrypted malware traffic detection," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 5011–5025, 2023.

[5] H. Ding, Y. Sun, N. Huang, Z. Shen, and X. Cui, "TMG-GAN: Generative adversarial networks-based imbalanced learning for network intrusion detection," *IEEE Trans. Inf. Forensics Security*, vol. 19, pp. 1156–1167, 2023.

[6] Y. Liang, Q. Wang, K. Xiong, X. Zheng, Z. Yu, and D. Zeng, "Robust detection of malicious URLs with self-paced wide & deep learning," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 2, pp. 717–730, Mar. 2022.

[7] H. Kim, C. Hahn, H. J. Kim, Y. Shin, and J. Hur, "Deep learning-based detection for multiple cache side-channel attacks," *IEEE Trans. Inf. Forensics Security*, vol. 19, pp. 1672–1686, 2023.

[8] M. Wichtlhuber et al., "IXP scrubber: Learning from blackholing traffic for ML-driven DDoS detection at scale," in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 707–722.

[9] P. Zhang et al., "Real-time malicious traffic detection with online isolation forest over SD-WAN," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 2076–2090, 2023.

[10] T.-A. Nguyen, J. He, L. T. Le, W. Bao, and N. H. Tran, "Federated PCA on Grassmann manifold for anomaly detection in IoT networks," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2023, pp. 1–10.

[11] P. Li et al., "Learning from limited heterogeneous training data: Meta-learning for unsupervised zero-day web attack detection across web domains," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2023, pp. 1020–1034.

[12] C. Fu, Q. Li, M. Shen, and K. Xu, "Realtime robust malicious traffic detection via frequency domain analysis," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2021, pp. 3431–3446.

[13] Z. Zhao et al., "ERNN: Error-resilient RNN for encrypted traffic detection towards network-induced phenomena," *IEEE Trans. Dependable Secure Comput.*, pp. 1–18, 2023.

[14] D. Tang, Y. Yan, C. Gao, W. Liang, and W. Jin, "LtRFT: Mitigate the low-rate data plane DDoS attack with learning-to-rank enabled flow tables," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 3143–3157, 2023.

[15] K. Sood, M. R. Nosouhi, D. D. N. Nguyen, F. Jiang, M. Chowdhury, and R. Doss, "Intrusion detection scheme with dimensionality reduction in next generation networks," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 965–979, 2023.

[16] P. Bosshart et al., "p4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[17] Y. Dong et al., "HorusEye: A realtime IoT malicious traffic detection framework using programmable switches," in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur.)*, 2023, pp. 571–588.

[18] V. Shrivastav, "Programmable multi-dimensional table filters for line rate network functions," in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 649–662.

[19] G. Zhou, Z. Liu, C. Fu, Q. Li, and K. Xu, "An efficient design of intelligent network data plane," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 6203–6220.

[20] C. Zheng and N. Zilberman, "Planter: Seeding trees within switches," in *Proc. SIGCOMM Poster Demo Sessions*, Aug. 2021, pp. 12–14.

[21] A. T. J. Akem, M. Gucciardo, and M. Fiore, "Flowrest: Practical flow-level inference in programmable switches with random forests," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2023, pp. 1–10.

[22] T. Swamy, A. Rucker, M. Shahbaz, I. Gaur, and K. Olukotun, "Taurus: A data plane architecture for per-packet ML," in *Proc. 27th ACM Int. Conf. Architect. Support Program. Lang. Oper. Syst.*, 2022, pp. 1099–1114.

[23] H. Zhou, S. Hong, Y. Liu, X. Luo, W. Li, and G. Gu, "Mew: Enabling large-scale and dynamic link-flooding defenses on programmable switches," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2023, pp. 1625–1639.

[24] A. G. Alcoz, M. Strohmeier, V. Lenders, and L. Vanbever, "Aggregate-based congestion control for pulse-wave DDoS defense," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, 2022, pp. 693–706.

[25] S. Gupta, D. Gosain, M. Kwon, and H. B. Acharya, "DeeP4R: Deep packet inspection in P4 using packet recirculation," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2023, pp. 1–10.

[26] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. V. Ramos, and A. Madeira, "FlowLens: Enabling efficient flow classification for ML-based network security applications," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–18.

[27] C. Fu, Q. Li, and K. Xu, "Detecting unknown encrypted malicious traffic in real time via flow interaction graph analysis," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2023, pp. 1–18.

[28] K. Xu et al., "Self-supervised learning malware traffic classification based on masked autoencoder," *IEEE Internet Things J.*, vol. 11, no. 10, pp. 17330–17340, May 2024.

[29] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: An ensemble of autoencoders for online network intrusion detection," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[30] Z. Zhao, Z. Liu, H. Chen, F. Zhang, Z. Song, and Z. Li, "Effective DDoS mitigation via ML-driven in-network traffic shaping," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 4, pp. 4271–4289, Jul. 2024.

[31] D. Tang, R. Dai, Y. Yan, K. Li, W. Liang, and Z. Qin, "When SDN meets low-rate threats: A survey of attacks and countermeasures in programmable networks," *ACM Comput. Surv.*, vol. 57, no. 4, pp. 1–32, 2024.

[32] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 270–313, Aug. 2003.

[33] B. Liu. (2025). *Reaper Code Repository*. [Online]. Available: https://github.com/lbr711/REAPER

[34] C. Fu. (2021). *Whipser Code Repository*. [Online]. Available: https://github.com/fuchuanpu/Whisper

[35] Y. Mirsky. (2018). *Kisune Code Repository*. [Online]. Available: https://github.com/ymirsky/KitNET-py

[36] D. Barradas. (2021). *Flowlens Code Repository*. [Online]. Available: https://github.com/dmbb/FlowLens

[37] CIC-DDoS2019. (2019). *CIC-DDoS2019*. [Online]. Available: https://www.unb.ca/cic/datasets/index.html

[38] D. Tang, S. Wang, B. Liu, W. Jin, and J. Zhang, "GASF-IPP: Detection and mitigation of LDoS attack in SDN," *IEEE Trans. Services Comput.*, vol. 16, no. 5, pp. 3373–3384, Sep./Oct. 2023.

[39] D. Tang, Y. Yan, S. Zhang, J. Chen, and Z. Qin, "Performance and features: Mitigating the low-rate TCP-targeted DoS attack via SDN," *IEEE J. Sel. Areas Commun.*, vol. 40, no. 1, pp. 428–444, Jan. 2022.

[40] S. Ha, I. Rhee, and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, no. 5, pp. 64–74, Jul. 2008.

[41] Mininet.(2010). *Mininet*. [Online]. Available: http://mininet.org/

[42] (2012). *The Internet Topology Zoo*. [Online]. Available: http://www.topology-zoo.org/index.html
[43] CIC-AndMal2017. (2017). *CIC-AndMal2017*. [Online]. Available: https://www.unb.ca/cic/datasets/index.html
[44] H. C. Tan, C. Cheh, and B. Chen, "CoToRu: Automatic generation of network intrusion detection rules from code," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2022, pp. 720–729.
[45] Y. Wu et al., "Paradise: Real-time, generalized, and distributed provenance-based intrusion detection," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 2, pp. 1624–1640, Mar./Apr. 2023.
[46] Y. Fu, K. Liang, and J. Xu, "MLog: Mogrifier LSTM-based log anomaly detection approach using semantic representation," *IEEE Trans. Services Comput.*, vol. 16, no. 5, pp. 3537–3549, Sep./Oct. 2023.
[47] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste, "SketchLib: Enabling efficient sketch-based monitoring on programmable switches," in *Proc. 19th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, Mar. 2022, pp. 743–759.
[48] D. Tang, B. Liu, K. Li, S. Xiao, W. Liang, and J. Zhang, "PLUTO: A robust LDoS attack defense system executing at line speed," *IEEE Trans. Dependable Secure Comput.*, vol. 22, no. 3, pp. 2855–2872, May 2025.
[49] G. Bovenzi, G. Aceto, D. Ciuonzo, A. Montieri, V. Persico, and A. Pescapé, "Network anomaly detection methods in IoT environments via deep learning: A fair comparison of performance and robustness," *Comput. Secur.*, vol. 128, May 2023, Art. no. 103167.
[50] G. Bovenzi, G. Aceto, D. Ciuonzo, V. Persico, and A. Pescapé, "A hierarchical hybrid intrusion detection approach in IoT scenarios," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Taiwan, Dec. 2020, pp. 1–7.
[51] M. Verkerken et al., "A novel multi-stage approach for hierarchical intrusion detection," *IEEE Trans. Netw. Service Manage.*, vol. 20, no. 3, pp. 3915–3929, Sep. 2023.
[52] R. Zhao et al., "A novel intrusion detection method based on lightweight neural network for Internet of Things," *IEEE Internet Things J.*, vol. 9, no. 12, pp. 9960–9972, Jun. 2022.
[53] H. Lesfari and F. Giroire, "Nadege: When graph kernels meet network anomaly detection," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2022, pp. 2008–2017.
[54] G. Duan, H. Lv, H. Wang, and G. Feng, "Application of a dynamic line graph neural network for intrusion detection with semisupervised learning," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 699–714, 2023.
[55] X. Hu, W. Gao, G. Cheng, R. Li, Y. Zhou, and H. Wu, "Toward early and accurate network intrusion detection using graph embedding," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 5817–5831, 2023.
[56] C. Fu, Q. Li, K. Xu, and J. Wu, "Point cloud analysis for ML-based malicious traffic detection: Reducing majorities of false positive alarms," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2023, pp. 1005–1019.
[57] F. Cerasuolo, G. Bovenzi, D. Ciuonzo, and A. Pescapè, "Attack-adaptive network intrusion detection systems for IoT networks through class incremental learning," *Comput. Netw.*, vol. 263, May 2025, Art. no. 111228.

Dan Tang received the B.S., M.S., and Ph.D. degrees from the Huazhong University of Science and Technology in 2014. He is currently an Associate Professor with the College of Cyber Science and Technology, Hunan University (HNU). His research interests include network security, information security, and programmable networks.

Boru Liu received the B.S. degree from the College of Computer Science and Electronic Engineering (CSEE), Hunan University (HNU), Changsha, China, where he is currently pursuing the M.S. degree. He is a Senior with CSEE, HNU. He is majoring in computer science and technology, and his research focuses on network and information security.

Zheng Qin (Associate Member, IEEE) received the Ph.D. degree in computer software and theory from Chongqing University in 2001. He is currently a Professor with the College of Cyber Science and Technology, Hunan University, China. He has accumulated rich experience in product development and application services, such as in the areas of financial, medical, military, and education sectors. His main research interests include computer networks and information security, cloud computing, big data processing, and software engineering. He is a member of CCF and ACM.

Wei Liang received the Ph.D. degree in computer science and technology from Hunan University in 2013. He was a Post-Doctoral Scholar with Lehigh University, Bethlehem, PA, USA, from 2014 to 2016. He is currently a Professor with the School of Computer Science and Engineering, Hunan University of Science and Technology. His research interests include blockchain security technology, network security protection, embedded systems and hardware IP protection, fog computing, and security management in wireless sensor networks.

Keqin Li (Fellow, IEEE) received the B.S. degree in computer science from Tsinghua University in 1985 and the Ph.D. degree in computer science from the University of Houston in 1990. He is currently a SUNY Distinguished Professor at The State University of New York and a National Distinguished Professor at Hunan University, China. He is an AAAS Fellow, an AAIA Fellow, an ACIS Fellow, and an AIIA Fellow. He is a member of European Academy of Sciences and Arts. He is a member of Academia Europaea. Since 2020, he has been among the world's top few most influential scientists in parallel and distributed computing regarding single-year impact (ranked #2) and career-long impact (ranked #4) based on a composite indicator of the Scopus citation database.

Wenqiang Jin (Member, IEEE) received the Ph.D. degree from The University of Texas at Arlington, TX, USA, in 2021. He is currently a Professor with the College of Cyber Science and Technology, Hunan University, China. His research interests include mobile security, the IoT security, and ubiquitous computing.