Real-Time Intrusion Detection and Prevention with Neural Network in Kernel using eBPF

Junyu Zhang*, Pengfei Chen†, Zilong He*, Hongyang Chen*, and Xiaoyun Li*
*†*School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China*
*{zhangjy297, hezlong, chenhy95, lixy223}@mail2.sysu.edu.cn, †chenpf7@mail.sysu.edu.cn

***Abstract*—With the development of public cloud, real-time intrusion detection is becoming necessary. Current methods neither address the overhead of real-time network data capturing, nor effectively balance security level with performance. These issues can be addressed by offloading intrusion detection and prevention to the extended Berkeley Packet Filter (eBPF). However, current eBPF-based methods suffer from shortcomings in model performance or inference overhead. Moreover, they overlook the issues of eBPF in real-time scenarios, such as maximum eBPF instruction limitations. In this paper, we redesign the Neural Network inference mechanism to address the limitations of eBPF. Then, we propose a thread-safe parameter hot-updating mechanism without explicit spin lock. Evaluations indicate that our method achieves model performance comparable to the current best eBPF-based method while reducing memory overhead (5KB) and inference time (3000-5000ns per flow). Our method achieve F1-scores of 0.933 and 0.992 on the offline and online datasets, respectively.**

***Index Terms*—Real-Time Intrusion Detection, eBPF, Deep Learning, Neural Network Quantization**

I. Introduction

Due to the cost-effective advantages of cloud computing, an increasing number of companies are migrating their data and business to the cloud [1]. Tenants access the cloud products via the internet, thus exposing the cloud services to the threat of network intrusions [2]. Consequently, it is necessary to detect and prevent network intrusions in real time before any potential impact on system performance and functionality [3].

On the one hand, current real-time intrusion detection methods [4]–[7] focus on the performance of detection models but overlook the overhead associated with network data capturing (§II-A). On the other hand, there are tools automatically generating rules for active defense tools such as *iptables* [8] from offline detection models [9], but the performance of these tools rapidly degrades as the number of rules increases [10]. Conversely, inadequate rules may allow more intrusions to evade prevention [9]. Therefore, current approaches have not effectively integrated intrusion detection and prevention in the real-time scenarios.

The extended Berkeley Packet Filter (eBPF) enables the dynamic and sandboxed programs execution totally in the Linux kernel, without any changes to the kernel source code [11]. Linux kernel hide diverse hardware architectures, which makes eBPF a suitable programmable network data plane. At present, there is insufficient research on how to use eBPF for real-time intrusion detection and prevention (§V-E). Bachl et al. [12] firstly use eBPF to implement decision tree (DT) for intrusion detection. However, storing all nodes of DT brings exponential memory overhead to the kernel. Linear Support Vector Machine [13] is inadequate for fitting non-linearly separable intrusion detection problems. Neural networks (NN) based on *int8* quantization [14] not only exhibit high inference complexity but also introduce significant errors. [12]–[14] do not consider many important issues in eBPF, such as race conditions and the maximum eBPF instruction limitation [15] (§II-B2). Additionally, these works lack an real-time evaluation of the performance, effectiveness, and reliability.

Insights. In this paper, we implement a real-time intrusion detection and prevention prototype with eBPF (§IV-A). We redesign the NN inference mechanism to address the limitation of the integer-only arithmetic (§IV-B) and maximum instruction number (§IV-C) in eBPF, and reduce the memory overhead while maintaining the performance. We propose a thread-safe parameter hot-updating mechanism without terminating the intrusion detection system and explicit eBPF spin lock [16] (§IV-D).

Recently, deep learning-based intrusion detection systems have shown outstanding performance on existing and unseen intrusions [9], while the mainstream of these deep learning-based systems relies on NN [17]–[19]. Compared with [14], our NN inference method not only reduces the inference complexity but also improves the performance. We decompose the NN inference process into several sequential stages, and then utilize chain eBPF Tail Calls [20] to implement them, which overcomes the maximum instruction limitation in a single eBPF program. We implement an integer-only NN inference algorithm in eBPF, reducing the required memory overhead while ensuring high performance.

Concept drift can lead to critical failures in deployed deep learning-based detection models, and thus it is essential to update the parameters of the models when concept drift occurs [21]. If the NN inference program is terminated for parameter updates, the system is exposed to unprotected risks during the update. Conversely, if the program is not terminated when updating parameters (Parameters Hot-Updating), it can easily lead to read-write race condition issues. We implement a thread-safe hot-updating algorithm that does not require explicit eBPF locks.

In this paper, we make the following contributions.

- **Study**: we identify overlooked issues associated with integrating real-time intrusion detection and prevention

in existing methods.

- **Framework**: we redesign NN inference in kernel using eBPF and propose a thread-safe parameter hot-updating mechanism.
- **Evaluation**: our method reduces memory overhead while maintains the detection performance and time overhead comparable to the existing methods.

Our code is available in the public repository[1].

II. Background & Motivation

A. Overhead of Traditional Packet Capturing

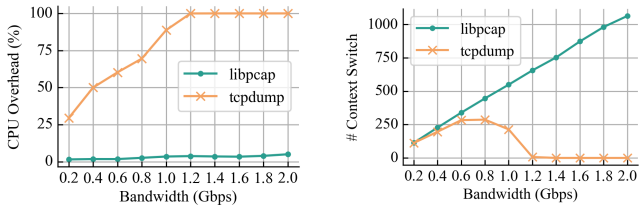
Typically, *tcpdump* [22] is employed for packet capturing, followed by the utilization of tools like *CICFlowMeter* [23] for feature extraction [24]. However, without the programmability, feature extraction will not start until all network packets are captured by *tcpdump*. The time overhead of feature extraction is significantly affected by the total size of the captured packets. Table I shows that, as network traffic increases, feature extraction time changes notably, surpassing the packet capturing time. Consequently, the serial execution mode of *tcpdump* and *CICFlowMeter* proves unsuitable for real-time scenarios.

TABLE I: Impact of size on feature extraction time

Traffic Size (MB)	100	200	300	400	500
Packet Capturing Time (s)	4.3	8.4	12.7	17.4	21.1
Feature Extraction Time (s)	21.3	41.9	78.3	93.6	145.4

libpcap [25] is the foundation of *tcpdump* and provides the programmability during the real-time data capturing [4]. After *libpcap* captures each packet, it will execute a callback function containing the intrusion detection algorithm written in advance, which avoids the problem that packet capture and detection can only be executed serially.

However, *libpcap* introduces significant context switch and CPU overhead. We use the interfaces [26] provided by *libpcap* to implement the callback function. In order to measure the overhead of real-time packet capturing, the callback function returns immediately without performing any operation when it is called. We evaluate *libpcap* and *tcpdump* on a 8 core virtual machine, and use *pidstat* [27] to measure the overhead.



(a) CPU Overhead (b) Context Switch

Fig. 1: Overhead of *libpcap* and *tcpdump*

[1]https://github.com/IntelligentDDS/NN-eBPF

As shown in Figure 1, with the bandwidth increasing, the CPU overhead of *libpcap* grows from 1.53% to 5.00%, while the CPU overhead of *tcpdump* grows from 29.49% to 100%. Since *tcpdump* performs additional packet parsing, the cpu overhead is significantly greater than that of *libpcap* program. Furthermore, when the bandwidth exceeds 1.2 Gbps, *tcpdump* begins to drop packets and the corresponding context switches decreases significantly to 0. Context switches of *libpcap* increases linearly with bandwidth, and can eventually reach thousands of times per second.

The root cause of the overhead is that traditional packet capturing is based on the user-space programs. Packets captured in the kernel require frequent context switches and memory copies before they can reach the user-space.

B. eBPF for real-time intrusion detection and prevention

1) ***Introduction of the eBPF technique****:* eBPF enables programmability in the Linux kernel [11]. User-defined eBPF program is attached to the kernel hook point like system calls, function entry/exit and network events. When events on the corresponding hook point are triggered, the pre-defined eBPF program is run. For eBPF programs completely in the kernel, context switches per second can be almost non-existent.

There are two most common network hooks for packet filtering: XDP and TC [28]. eXpress Data Path (XDP) serves as the initial hook of the kernel network stack [29]. XDP hook enables eBPF program to process packets only in RX direction and decide whether the packets can be received. Traffic Control (TC) [30] is another hook after the execution of XDP. Different from XDP hook, TC hook can filter packets in both RX and TX directions. However, TC hook is slightly worse than XDP hook because TC hook requires additional memory allocation or entering software socket queues before it is triggered [31]. In order to minimize the overhead caused by packet filtering, we choose XDP as the eBPF hook point.

2) ***Challenges of employing eBPF for real-time intrusion detection****:* The first challenge comes from ensuring feature extraction time. XDP conducts feature extraction for each packet upon receipt. If the time taken for feature extraction is less than the average time interval between two adjacent packets, then feature extraction is nearly imperceptible to the network flow to which the packet belongs. However, if the time equals or exceeds the average time interval, for protocols like TCP with acknowledgment mechanisms, the transmission time of the network flow increases. In the case of protocols like UDP without acknowledgment mechanisms, feature extraction errors may occur.

The second challenge arises from the selection of features. XDP offers programmability to implement feature extraction algorithms. Due to the constraints of the 512 bytes eBPF stack size and 1M instruction number [20], the number of features that can be extracted at the XDP layer is limited. Moreover, XDP is capable of inspecting packets only in the RX direction, meaning it can filter packets received by itself but is unable to observe packets sent by itself (TX direction). Therefore, minimizing the number of features extracted in XDP and

determining the most important and effective features only in the RX direction present another challenge.

The third challenge is race condition. Network interfaces maintain multiple RX queues, with each RX queue assigned to a specific CPU core. Upon receiving a packet, each RX queue executes the XDP program on its allocated CPU core. Hence, shared data structures within the XDP program give rise to race conditions [31]. Although eBPF provides a spin lock mechanism to address race conditions, but using spin locks indiscriminately can introduce unpredictable latency overhead to the kernel and eBPF do not allow any function calling before the lock released. Effectively avoiding or resolving race conditions requires specific design techniques.

III. Threat Model

The system implemented in this paper is designed for real-time intrusion detection scenarios. Therefore, it is necessary to perform benign and intrusion behavior in real-time in a local environment to validate the effectiveness, reliability, availability, and overhead of our system in the real-time scenarios. Thus, we need to define what constitutes benign behavior and what constitutes intrusion behavior.

Benign Behavior [32]:

- Using *ssh* to successfully log in to the system.
- Safe execution of common Linux commands: for example, using *ping* to test reachability, using *ps* to list all processes, and using *docker* to manage containers.
- Normal HTTP requests and TCP traffic.
- Uploading and downloading files using FTP.

Intrusion Behavior [33]:

- Brute-force attacks on SSH and FTP with repeated password attempts.
- Port Scan: for example, multiple execution of *nmap* to scan all network ports.
- Denial of Service (DoS) and Distributed Denial of Service (DDoS) attacks.
- Brute-force and Cross-Site Scripting (XSS) attacks on HTTP applications .

IV. System Design & Implementation

We initially provide an overview of how intrusion detection and prevention are accomplished through NN in the kernel space. Subsequently, we elucidate the quantization of NN parameters and inference, and resolve challenges specific to eBPF implementation. Then, we present the inference based on the chained eBPF Tail Calls. Finally, we discuss strategies to mitigate race conditions during the hot update of parameters from user space to kernel space.

A. Overview

Figure 2 illustrates the overall architecture of the system. The following analysis delves into the functions of each module.

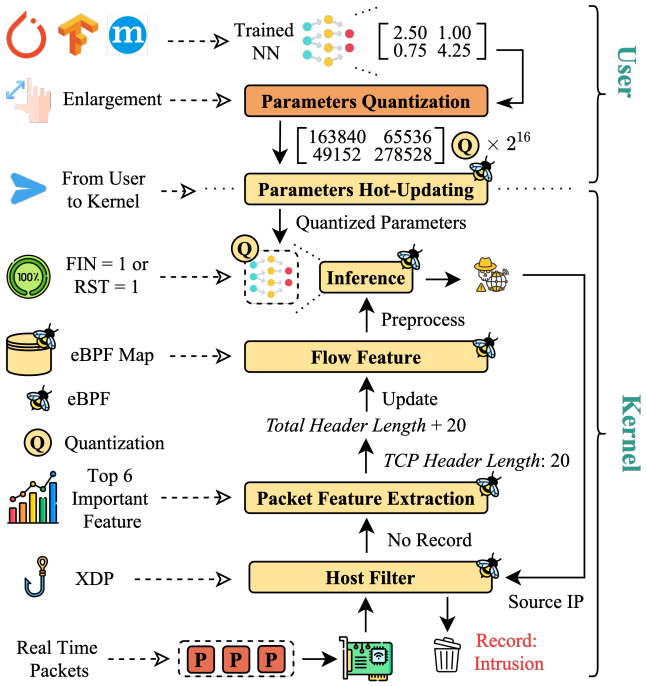


Fig. 2: Overview of our system

*1) **Training and Quantization in User Space**:* To reduce the overhead of training NN in kernel space, we relocate the NN training process to user space, while keeping the inference in the kernel space. Initially, NN is trained using historical data on PyTorch, TensorFlow or MXNet. Subsequently, its parameters are quantized from floating-point numbers to integers for inference in kernel. The basic idea of quantization involves multiplying a coefficient in order to shift the decimal point to the right. Then, quantized parameters are loaded into eBPF program in kernel through module *Parameter Hot-Updating*.

*2) **Parameter Hot-Updating**:* Module *Parameter Hot-Updating* addresses the issue of race conditions when updating parameters from user space to kernel space. The NN quantized for inference solely using integers has been loaded into the kernel and it utilizes the hot-updated parameters to conduct inference.

*3) **Host Filter**:* In the real-time scenario, packets are initially processed by the eBPF program attached to the XDP. The first module of our eBPF program is *Host Filter*. *Host Filter* checks the source IP of packets. If a packet originates from a previously identified host performing intrusions, the it is discarded. Otherwise, it has no detection record yet, and is passed to module *Packet Feature Extraction*. Each record is assigned the same validity period, and it is removed once it expires.

*4) **Packet Feature Extraction and Flow Feature Updating**:* Module *Packet Feature Extraction* extracts features from the packet, such as *TCP Header Length*, and updates the corresponding flow features stored in eBPF map, such as *Total Header Length*. The selection of flow features is not

based on expert knowledge or arbitrary choices. Rather, it is derived from the intrusion detection task. We calculate the importance of each feature from historical data, choose the top 6 most important features (*Fwd packet length Max*, *Fwd IAT Max*, *Fwd packet length Min*, *Destination Port*, *Fwd Header Length*, *Total Fwd packets*, details are shown in Table IV) and subsequently implement corresponding feature extraction algorithms in eBPF.

5) ***Inference in Kernel Space using eBPF****:* We implement an integer-only NN inference algorithm in eBPF. The input to the *Inference* module is the normalized flow features stored in eBPF map *flow feature*, and the output is a binary classification result: whether it is an intrusive or normal flow. To reduce the overhead of kernel-space inference, we perform a binary classification without distinguishing between different intrusion types. Additionally, the inference is only conducted after the completion of a flow. The indication of flow completion is marked by setting the FIN and RST flags in the TCP header to 1.

6) ***Intrusion Host Record and Correction****:* Whenever NN in kernel detects an intrusion, the source IP of the corresponding flow is used in *Host Filter*, resulting in subsequent packets from that source to be discarded. To improve the recall of the model, we have the ability to update *host filter* by adding or removing IP addresses.

B. Parameters and NN Inference Quantization

In this paper, the term NN refers to Multilayer Perceptron (MLP), and ReLU are implemented in all activation layers. The quantization of NN parameters is performed using a simple technique called *enlargement* method. The core idea is to multiply the floating-point number by an integer s (enlargement factor) and subsequently round it to the nearest integer stored in *int32*. To provide a comprehensive description of the method, some notations are defined in Table II.

TABLE II: Notations of enlargement method

Notation	Meaning
$\boldsymbol{x}^{(k)} = [x_j^{(k)}] \in \mathbb{R}^{n^{(k-1)}}$	Input of k-th linear layer
$\boldsymbol{y}^{(k)} = [y_i^{(k)}] \in \mathbb{R}^{n^{(k)}}$	Output of k-th linear layer
$\boldsymbol{W}^{(k)} = [w_{ij}^{(k)}] \in \mathbb{R}^{n^{(k)} \times n^{(k-1)}}$	Weight matrix of k-th linear layer
$n^{(k)} (k \geq 1)$	Size of k-th linear layer
$n^{(0)}$	Number of input features
$\boldsymbol{x} = [x_j] \in \mathbb{R}^{n^{(0)}}$	Initial input of NN
$\boldsymbol{\mu} = [\mu_j] \in \mathbb{R}^{n^{(0)}}$	Mean of $\boldsymbol{x}$
$\boldsymbol{\sigma} = [\sigma_j] \in \mathbb{R}^{n^{(0)}}$	Standard deviation of $\boldsymbol{x}$
$relu(x)$	Function ReLU
$round(x) = \lfloor x \rfloor$	Rounding down x
$ars(x, b)$	Arithmetic right-shift x by b bits

1) ***Preprocessing*** *(Standardization):* If normalization is performed during the training process, it implies that normalization is also required before real-time inference in the kernel space.

We employ standard normalization (standardization) for preprocessing. However, the normalization process may involve signed division, which is not supported by eBPF. Moreover, if the data follow a normal distribution, the standardized data follow the standard normal distribution $\mathcal{N}(0, 1)$, with data concentrated around 0 according to the 3σ rule. Since eBPF only supports integer division, direct standardization in eBPF leads to significant precision loss.

To address the issue of precision loss, we incorporate the *enlargement* method into the standardization formula, resulting in the following expression:

$$x_j^{(1)} = \begin{cases} 0 & \sigma_j = 0 \\ -round(\frac{s \cdot (\mu_j - x_j)}{\sigma_j}) & x_j < \mu_j, \sigma_j \neq 0 \\ round(\frac{s \cdot (x_j - \mu_j)}{\sigma_j}) & x_j \geq \mu_j, \sigma_j \neq 0 \end{cases} \tag{1}$$

$\frac{s \cdot (x_j - \mu_j)}{\sigma_j}$ and $s \cdot \frac{x_j - \mu_j}{\sigma_j}$ are two different computation methods. The latter involves division followed by multiplication, and since eBPF performs integer division, significant precision loss occurs in this case. The former, on the other hand, involves multiplication followed by division to preserve precision. Moreover, the above formula first performs unsigned division and then converts the result into the corresponding signed number, thereby circumventing the limitations posed by the lack of support for signed number division in eBPF. Since eBPF performs integer operations exclusively, the above formula does not necessitate the use of the *round* operation.

2) ***Inference****:* Parameters of each linear layer need to be quantized before inference, and the formula is as follows:

$$\boldsymbol{W}_E^{(k)} \triangleq round(s \cdot \boldsymbol{W}^{(k)}) = [round(s \cdot w_{ij}^{(k)})] \tag{2}$$

For the k-th ($k > 1$) linear layer, the input tensor satisfies:

$$\boldsymbol{x}^{(k)} = relu(\boldsymbol{y}^{(k-1)}) \tag{3}$$

The output tensor of the k-th layer satisfies the following formula:

$$\begin{aligned} \boldsymbol{y}^{(k)} &= \frac{1}{s} \cdot \boldsymbol{W}_E^{(k)} \cdot \boldsymbol{x}^{(k)} \\ &= \frac{1}{s} \cdot [\sum_{j=1}^{n^{(k-1)}} w_{E,ij}^{(k)} \cdot x_j^{(k)}] \\ &\overset{s=2^b}{=} ars(\sum_{j=1}^{n^{(k-1)}} w_{E,ij}^{(k)} \cdot x_j^{(k)}, b) \end{aligned} \tag{4}$$

In the above formula, as we have multiplied each element in $\boldsymbol{W}_E^{(k)}$ by s, and the elements in $\boldsymbol{y}^{(k)}$ are obtained by multiplying corresponding elements of $\boldsymbol{W}_E^{(k)}$ and $\boldsymbol{x}^{(k)}$ and then summing them up. Therefore, to eliminate the s from the result, we need to multiply by $\frac{1}{s}$. To reduce the time overhead associated with multiplication and division instructions, we utilize shift instructions to perform these operations. Specifically, let $s = 2^b$, multiplication by s is achieved by left shifting by b bits, while division by s is achieved by right shifting by b bits.

XDP concurrently filters multiple network flows, and performing parallel inference on these flows may introduce issues related to race conditions. The primary concern lies in how to store the hidden layer outputs $\boldsymbol{y}^{(k)}$ in a parallelized manner. If $\boldsymbol{y}^{(k)}$ is stored using global variables, there inevitably is a race condition between read and write operations. To address the race condition problem, we employ local variables for storing $\boldsymbol{y}^{(k)}$ of each flow. However, local variables may pose limitations as the maximum stack space is restricted.

*3) **Classification**:* We consider intrusion detection as a binary classification task, where the label for the *Intrusion* class is 1, and the label for the *Benign* class is 0. For the final linear layer output $\boldsymbol{y}^{(K)}$, the decision criterion is as follows:

$$prediction = \begin{cases} 1 & y_1^{(K)} > y_0^{(K)} \\ 0 & y_1^{(K)} \leq y_0^{(K)} \end{cases} \tag{5}$$

C. Inference based on the chained eBPF Tail Calls

The constraint of 1 million eBPF instructions poses a significant challenge to the implementation of NN in eBPF. In practice, we observe that even for a NN with small dimensions like $[6, 32, 32, 2]$, the compiled number of instructions still exceeds 1 million, leading to rejection by the eBPF verifier during loading.

To address the instruction number limitation, we adopt the eBPF Tail Call mechanism for inference. As illustrated in Figure 3, the inference process is split into alternating eBPF programs for the *Linear Layer* and *ReLU*. Upon completion of one program, it uses *bpf_tail_call* to invoke the next adjacent program.

The advantage of using *tail call* is that each program has an instruction limit of 1 million, treating *Linear Layer* and *ReLU* as independent programs. For an individual *Linear Layer* and *ReLU*, the number of instructions does not exceed 1 million, satisfying the conditions of the eBPF verifier.

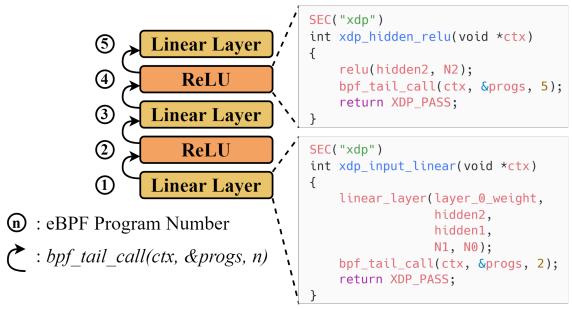
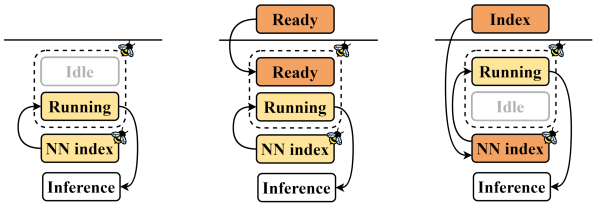


Fig. 3: Inference using three-layer MLP in eBPF

The stack size limit for the entire tail call chain is set to 256 bytes per subprogram, and the maximum call depth is 33 [20]. The outputs of the Linear Layer and ReLU layer are stored as *int32*, with each element occupying 4 bytes. Hence, the size limit for each layer is 64 ($\frac{256}{4} = 64$) elements. Since Linear Layer and ReLU layers appear in pairs except for the last layer, the maximum depth 17 ($\frac{33-1}{2} + 1 = 17$).

D. Parameter Hot-Updating

As shown in Figure 4(a), we utilize an eBPF map named *nn_parameters* to store NN used for inference in XDP. *nn_parameters* consists of two elements: one is named *Running*, representing the active NN parameters for inference, and the other is named *Idle*, reserved for subsequent updating. An eBPF map named *nn_index* is employed to store the index of the *Running* in *nn_parameters*, and thus, *nn_index* comprises only one element.



(a) Inference (b) NN Updating (c) Index Updating

Fig. 4: Parameter hot-updating

The pinned map *nn_parameters* can be found in the corresponding file located at */sys/fs/bpf*. To access *nn_parameters* in kernel space, the user-space eBPF code can employ the helper function *bpf_obj_get* to load the file */sys/fs/bpf/nn_parameters*. In Figure 4(b), the hot-updating process begins by loading the new NN parameters, named *Ready*, from user space into *Idle* in kernel space. The index of *Idle* within *nn_parameters* can be determined using the following formula:

$$idle\_index = (running\_index + 1) \mod 2 \tag{6}$$

In the above formula, the value of $running\_index$ is stored within the *nn_index* pinned map, which can be loaded and accessed in user space.

In the final step, as depicted in Figure 4(c), updating the value in *nn_index* with $idle\_index$ completes the parameter hot-updating process.

Although the helper function *bpf_map_update_elem* can be used to atomically update the NN parameters stored in the eBPF map, accessing the NN during inference is not atomic, which can result in potential race condition issues between reads and writes. Hence, we choose the approach illustrated in Figure 4 to address the race condition without relying on spin locks.

V. Evaluation

In this section, our goal is to evaluate the performance of NN in eBPF and address the following research questions (RQs).

- **RQ I:** Compared to existing methods, what is the effectiveness of the proposed NN in accomplishing intrusion detection tasks (§V-B)?
- **RQ II:** When errors occur in real-time detection, to what extent can the proposed NN maintain the reliability and availability of the system (§V-C)?

- **RQ III:** How much overhead is caused to the system by extracting features from the flow in real time and performing NN inference (§V-D)?
- **RQ IV:** What are the overlooked issues in the existing methods (§V-E)?

A. Dataset & Environment Setup

CIC-IDS-2017 [33] is a commonly used benchmark for evaluating intrusion detection models, and we remove all the invalid data from the CIC-Flow-2017 dataset. To evaluate the real-time performance using eBPF, we reproduce intrusion datasets (eBPF-Reproduction Dataset) for both benign and intrusion using the feature extraction algorithm in eBPF XDP. Below are the generation method of each intrusion:

- **Benign**: use *ssh* to log in shell and execute various common commands, including *ping*, *ps*, *docker*, and *curl*, among others. Additionally, utilize *httperf* [34] for simulating typical HTTP traffic and *iperf* [35] for simulating regular TCP traffic. Furthermore, take into account activities such as logging in, uploading, and downloading within an FTP application.
- **PortScan**: utilize the Linux *nmap* tool to conduct port scanning.
- **Dos GoldenEye and Slowhttptest**: implement two distinct Denial of Service (DoS) attack methods using GoldenEye [36] and Slowhttptest [37].
- **FTP and SSH Patator**: use patator [38] to perform dictionary-based brute force attack on SSH and FTP passwords.
  zh
- **Web Brute Force and XSS**: automate Brute Force and Cross-Site Scripting (XSS) attacks on Damn Vulnerable Web App (DVWA) [39] using selenium [40].

TABLE III: CIC-IDS-2017 and reproduction dataset

Intrusion Type		CIC-IDS-2017	eBPF-Reproduction Train	eBPF-Reproduction Test
Benign	FTP Download	2271320	1287	990
	FTP Upload		1568	3136
	Http Traffic		1024	2048
	TCP Traffic		923	1846
	SSH		2160	2160
Intrusion	FTP-Patator	7935	1972	1913
	SSH-Patator	5897	972	1920
	Dos GoldenEye	10293	2807	6772
	PortScan	158804	896	896
	Dos Slowhttptest	5499	1100	2222
	Web Brute Force	967	800	1600
	Web XSS	1507	800	1600
Total		**2462222**	**16257**	**27103**

Table III illustrates the datasets used in this study, namely CIC-IDS-2017 and eBPF-Reproduction dataset. To minimize the influence of the local environment, the training and testing datasets of eBPF-Reproduction are collected on different dates. Some intrusion types from the CIC-IDS-2017 dataset are not reproduced due to three reasons: outdated intrusion types (Heartbleed), insufficient quantity of instances (Infiltration, Web SQL Injection), and similarities to the already reproduced intrusions (DoS Hulk, DDoS, DoS Slowloris).

We conduct the evaluation using two Linux virtual machines. Host A deploys the real-time intrusion detection system proposed in this paper, while host B is responsible for sending benign and intrusion network traffic to host A according to the generation methods. Each host is configured with 8 cores, 16 GB memory, 2,000 MHz CPU frequency, and 6.1.43 kernel version.

B. Effectiveness

We use classical metrics for intrusion detection to evaluate the model performance, namely $Accuracy = \frac{TP+TN}{TP+FP+TN+FN}$, $Precision = \frac{TP}{TP+FP}$, $Recall = \frac{TP}{TP+FN}$, and $F1\text{-}score = \frac{2 \cdot Precision \cdot Recall}{Precision+Recall}$. True Positive (TP) represents the instances correctly identified as intrusion, False Positive (FP) represents the instances incorrectly identified as intrusion, True Negative (TN) represents the instances correctly identified as benign, and False Negative (FN) represents the instances incorrectly identified as benign. We regard intrusion detection as a binary classification task, that is, the type of intrusion is not distinguished.

We compare the effectiveness among Decision Tree (DT) [12], Support Vector Machine (SVM) [13], Neural Network using *int8* quantization (NN-int8) [14], and our method (NN-int32). Table V demonstrates the evaluation results on the CIC-IDS-2017 and eBPF-Reproduction dataset.

*1) **Effectiveness on CIC-IDS-2017**:* Because CIC-IDS-2017 is a widely utilized intrusion dataset with a diverse range of categories, we firstly evaluate the effectiveness of each method on it. we set the *max_depth* of decision tree to 10, the same configuration as [12], and the neural network is a three-layer perceptron, with the sizes of each layer being 32, 32, and 2 respectively. The batch size is set to 512, the learning rate is set to 0.001, and Nvidia Tesla V100 is used to iteratively train the NN for 32 times. The enlargement factor s is set to 2^{16}.

Due to the *linear* kernel method used in the implementation of SVM [13], its non-linear fitting capability is limited. Consequently, SVM tends to classify all flows as intrusions, leading to high recall but low precision results. We now focus on comparing DT, NN-int8, and NN-int32.

We initially train the model (WQ) using all features (ALL), and subsequently quantize it (Q). Both NN-int8 and NN-int32 employ the same model (WQ) but with different quantization methods. DT can achieve good performance even without preprocessing the input data using standardization (WQ+ALL). Moreover, many features are actually integers, for instance, *Total Fwd Packets* (Table IV). Additionally, apart from the threshold values at each node, the model parameters of DT are represented using integers [12]. Therefore, the performance remains consistent before and after quantization (Q+ALL).

The performance of NN before quantization is comparable to that of DT (WQ+ALL). However, the differences in quantization methods lead to variations in performance (Q+ALL).

NN-int8 leads to a decrease in recall, while the decline in other metrics is smaller. In other words, NN-int8 tends to classify flows as benign. This is because, compared to the unquantized model, NN-int8 loses too much information during the quantization process, as 8-bit integers are insufficient to represent parameters and variables of each layer. Furthermore, NN-int8 requires both *quantize* and *dequantize* operations at each layer [14], both of which involve approximation, leading to further loss of precision. NN-int32 (Our Method) and NN-int8 use the same unquantized model, but the performance of NN-int32 remains consistent with that before quantization. This is because parameters and inputs of each layers multiplied by the enlargement factor s do not overflow the representation range of 32-bit integers, while maintaining precision within the maximum range.

To further assess if employing solely RX features is sufficient for intrusion detection, features associated with TX and the overall flow are removed from the CIC-IDS-2017 dataset, resulting in 24 RX-specific features detailed in Table IV.

TABLE IV: RX-specific features. **fwd** and **forward** indicate "in the forward direction", which refers to the RX.

Number	Feature	Description
0	Destination Port	Destination Port
2	Total Fwd Packets	Total packets
4	Total Length of Fwd Packets	Total size of packet
6	Fwd Packet Length Max	Maximum size of packet
7	Fwd Packet Length Min	Minimum size of packet
8	Fwd Packet Length Std	Standard deviation size of packet
9	Fwd Packet Length Mean	Mean size of packet
20	Fwd IAT Total	Total time between two packets
21	Fwd IAT Mean	Mean time between two packets
22	Fwd IAT Std	Standard deviation time between two packets
23	Fwd IAT Max	Maximum time between two packets
24	fwd IAT Min	Minimum time between two packets
30	Fwd PSH Flags	Number of PSH flag
32	Fwd URG Flags	Number of URG flag
34	Fwd Header Length	Total bytes used for headers
36	Fwd Packets/s	Number of packets per second
53	Avg Fwd Segment Size	Average size observed
56	Fwd Avg Bytes/Bulk	Average number of bytes bulk rate
57	Fwd Avg Packets/Bulk	Average number of packets bulk rate
58	Fwd Avg Bulk Rate	Average number of bulk rate
62	Subflow Fwd Packets	The average number of packets in a sub flow
63	Subflow Fwd Bytes	The average number of bytes in a sub flow
68	act_data_pkt_fwd	Count of packets with at least 1 byte
69	min_seg_size_forward	Minimum segment size observed

Before quantization, compared to the model using all features (WQ+ALL), both DT and NN experience only a slight decline in performance when utilizing only RX-specific features (WQ+RX). This is attributed to the reduction in features related to TX and the overal flow, impacting the detection capability of model. However, due to the strong correlation between RX and TX-specific features, the decrease in performance is minimal. After quantization, both DT and NN-int32 (Q+RX) maintain consistent performance with the unquantized model (WQ+RX). However, the quantization error of NN-int8 amplifies the performance decline caused by the reduced features, resulting in a significant decrease in precision.

Not all features in Table IV contribute to the intrusion detection model. In other words, it is possible to achieve similar performance as using all features from Table IV by choosing the most important subset (Top K) of features. To provide interpretability to the feature importance, we estimate the importance of each feature based on the Gini gain of DT. Then, we compute the cumulative feature performance to investigate how many of the most important features are required to achieve optimal performance. We incorporate features based on their importance in descending order, train NN, and evaluate the performance metrics. Cumulative results displayed in Figure 5 indicate that training the model with only features $\{6, 23, 7, 0, 34, 2\}$ already achieves performance comparable to using all features. Consequently, the number of feature can be reduced from 24 to 6.

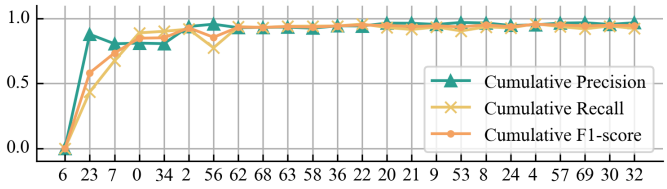


Fig. 5: Cumulative feature performance

We select the Top 6 RX-specific features. Before quantization, compared to models using all RX-specific features (WQ+RX), both NN and DT experience a decrease in performance (WQ+Top K), primarily reflected in precision and recall. However, the maximum decrease is only 0.0047 (NN-int32 recall). After quantization, the performance of NN-int32 and DT remains nearly unchanged (Q+Top K), while the performance of NN-int8 decreases by almost 50%, indicating that the quantization error of NN-int8 is significantly influenced by the number of features.

Since CIC-IDS-2017 is imbalanced, we also use Matthew Correlation Coefficient (MCC) [41] to evaluate the performance of models with quantization and top 6 features (Q+Top K). MCC for DT and our method are 0.958 and 0.917, respectively, indicating optimal agreement between predicted and actual values. MCC for NN-int8 is 0.344, suggesting poor classification performance. MCC for SVM is 0.015, much close to the random guess classifier. Results of PR-AUC are almost consistent with MCC. DT and our method perform the best, achieving 0.996 and 0.975, respectively, while SVM and NN-int8 are close, with values of 0.408 and 0.366, respectively.

*2) **Effectiveness on eBPF-Reproduction**:* In §V-B1, we evaluate the effectiveness utilizing Top K RX-specific features (WQ/Q+Top K). Firstly, we implement the extraction algorithm for the Top 6 features in eBPF XDP. Subsequently, following the methodology outlined in §V-A, we generate training and testing data for benign and intrusion, resulting in the dataset presented in Table III. Afterwards, we implement NN inference algorithm with eBPF. NN parameters obtained from the training dataset are loaded after quantization. Results on the testing dataset are illustrated in Table V.

The performance of SVM (WQ/Q+Top K) is the worst among the models due to the non-linearity of the data. It is consistently biased towards classifying flows as benign, which leads to a high recall but a low precision.

TABLE V: Effectiveness evaluation. **WQ** denotes models implemented in the PyTorch and scikit-learn frameworks without any quantization, while **Q** denotes the models quantized from models of **WQ** and then implemented in eBPF. **ALL**, **RX**, and **Top K** respectively denote the evaluation results using all features, only the RX-specific features, and only the top K most important features from the RX-specific set.

Model		Offline Evaluation				Real-Time Evaluation			
		Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score
DT [12]	WQ + ALL	0.997	0.995	0.989	0.992				
	WQ + RX	0.993	0.989	0.975	0.982				
	WQ + Top K	0.987	0.962	0.972	0.967	0.999	0.999	0.999	0.999
	Q + ALL	0.988	0.995	0.946	0.970				
	Q + RX	0.993	0.988	0.975	0.981				
	Q + Top K	0.987	0.962	0.972	0.967	0.999	0.999	1.000	0.999
SVM [13]	WQ + ALL	0.227	0.201	0.984	0.334				
	WQ + RX	0.197	0.197	0.999	0.329				
	WQ + Top K	0.198	0.197	1.000	0.329	0.824	0.788	0.982	0.874
	Q + ALL	0.339	0.144	0.477	0.221				
	Q + RX	0.282	0.158	0.611	0.251				
	Q + Top K	0.368	0.191	0.686	0.299	0.624	0.624	1.000	0.769
NN-int8 [14]	WQ + ALL	0.994	0.978	0.993	0.985				
	WQ + RX	0.988	0.952	0.988	0.970				
	WQ + Top K	0.974	0.927	0.941	0.934	0.991	0.988	0.998	0.993
	Q + ALL	0.902	0.956	0.527	0.679				
	Q + RX	0.864	0.720	0.502	0.592				
	Q + Top K	0.838	0.622	0.444	0.518	0.671	0.663	0.961	0.785
NN-int32 (Our Method)	WQ + ALL	0.994	0.978	0.993	0.985				
	WQ + RX	0.988	0.952	0.988	0.970				
	WQ + Top K	0.974	0.927	0.941	0.934	0.991	0.988	0.998	0.993
	Q + ALL	0.994	0.977	0.992	0.985				
	Q + RX	0.988	0.952	0.988	0.970				
	Q + Top K	0.974	0.926	0.941	0.933	0.994	0.986	0.999	0.992

NN and DT achieve excellent results before quantization (WQ+Top K), with performance of NN slightly trailing behind DT, but the difference is no more than 0.011. After quantization, both NN-int32 and DT maintain comparable performance (Q+Top K), indicating that the quantization process largely preserves accuracy. However, NN-int8 exhibits a significant reduction in all metrics except recall after quantization, indicating a substantial decrease in the ability to detect intrusions. Moreover, when compared to results on the CIC-IDS-2017 dataset, the performance (CIC-IDS-2017, Q+Top K) shows a higher decline in recall, suggesting that NN-int8 is sensitive to the choice of the dataset.

We also calculate the MCC on the eBPF reproduction dataset. We find that both DT and our method still perform the best, with MCC reaching 0.998 and 0.981 respectively. However, MMC for NN-int8 degrade to -0.099 and 0, respectively. SVM predicts all test samples as intrusions, resulting in an MCC of 0. For PR-AUC, DT and our method achieve 0.999 and 0.966 respectively, while SVM and NN-int8 only reach 0.735 and 0.707 respectively.

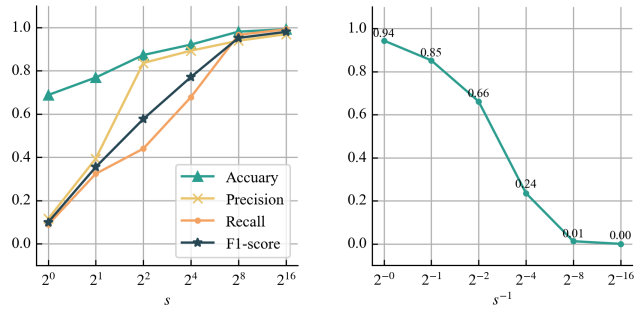
*3) **Hyper-parameter Settings**:* To investigate the impact of the NN-int32 (Q+TOP K) structure on performance, experiments are conducted with different depths and sizes of hidden layers on CIC-IDS-2017. Table VI suggests that modifications to the structure have minimal impact on performance. Therefore, to strike an optimal balance between model performance and complexity, a three-layer neural network with a hidden layer size of 32 is employed.

TABLE VI: Impact of depth and size of hidden layers

	Structure	Precision	Recall	F1-score
Size	[6,16,16,16,2]	0.990	0.926	0.957
	[6,32,32,32,2]	0.974	0.926	0.941
	[6,64,64,64,2]	0.995	0.937	0.965
	[6,128,128,128,2]	0.983	0.969	0.976
Depth	[6,32,32,2]	0.970	0.896	0.932
	[6,32,32,32,2]	0.974	0.926	0.941
	[6,32,32,32,32,2]	0.979	0.928	0.953
	[6,32,32,32,32,32,2]	0.993	0.898	0.943

The enlargement factor s affects the precision of model quantization. A larger factor leads to higher quantization precision, but it also comes with increased storage overhead. For instance, with $s = 2^{16}$, it may be necessary to use *int32* for storage, while $s = 2^8$ might allow the use of *int16*, halving the storage cost. Figure 6(a) indicates that the model achieves optimal performance only when $s = 2^{16}$.

The reason for using a larger s is that the model parameters K are very small. Therefore, a larger s is needed to preserve higher precision during rounding. This heuristic provides a basis for searching a suitable value of s, where the product of K and s should be greater than or equal to 1 to avoid becoming 0 during rounding. However, s cannot be too large to avoid the overflow. Therefore, the most suitable s^* should minimize s under the condition that the probability of the product of K and s being less than 1 is less than a threshold α. This can be expressed in the formula below:



(a) Performance under different s (b) ECDF of NN parameters

Fig. 6: Impact of different s

$$s^* = \arg\min_s Pr(|K| \leq \frac{1}{s}) \leq \alpha. \tag{7}$$

By fitting the empirical cumulative distribution function (ECDF) of the absolute values of model parameters $|K|$ through data, the above formula can be expressed as:

$$s^* = \arg\min_s ECDF(s^{-1}) \leq \alpha. \tag{8}$$

The ECDF is shown in Figure 6(b). If the condition is set to $\alpha = 0.005$, then s needs to be at least greater than 2^{16}. The error of NN-int32 also depends on the quantization of the input data. Given $\alpha = 0.005$ and $s = 2^{16}$ introduced above, we expect that for each feature x_i and its ECDF denoted as $ECDF_i$, the following formula is satisfied:

$$ECDF_i(s^{-1}) \leq \alpha. \tag{9}$$

When computing the ECDF for each feature, we observe that, the ECDF of each features all adhere to the formula mentioned above. Therefore, we can achieve results with NN-int32 that closely unquantized NN.

C. Availability & Reliability

The accuracy of inference is not guaranteed to be 100%. In real-time scenarios, dropping packets to terminate intrusion flow based on the incorrect inference results may have unexpected consequences for the system. Therefore, this section mainly analyzes the reliability and availability of our method. The dataset used for the analysis is the testing dataset as shown in Table III.

First, we analyze the availability. In this context, availability refers to accuracy. Table VII demonstrates that out of the 5 benign categories, 2 have a perfect accuracy of 1.000, while among the 7 intrusion categories, 5 also achieve a perfect accuracy of 1.000. The remaining categories all have an accuracy of at least 0.950, with the intrusion categories consistently outperforming the benign categories. These findings indicate that our method exhibits high availability and is particularly adept at identifying intrusion instances as compared to benign ones.

Next, we analyze the reliability. We select the categories with accuracy less than 1.000 from Table VII, namely *FTP Download*, *Http Traffic*, *SSH*, *SSH-Patator*, and *Web XSS*. We then visualize the detection results for each flow in each category in chronological order and calculate the *Mean Time To Failure* (MTTF) for each category, which represents the number of flows between two consecutive incorrect detection results.

FTP Download, *SSH-Patator*, and *Web XSS* demonstrate robust detection accuracy with minimal errors, as shown in Figure 7. These errors are sparsely scattered. Conversely, *Http Traffic* and *SSH* exhibit relatively lower detection accuracy, resulting in a denser distribution of errors. However, when considering MTTF, *Http Traffic* and *SSH* achieve impressive MTTF values of 1050 and 1021 respectively, implying a high level of reliability despite the lower accuracy. While *FTP Download*, *SSH-Patator*, and *Web XSS* have lower MTTF values, their infrequent occurrence of errors contributes to their overall strong reliability.

D. Overhead

This section analyzes the overhead of our method in terms of time, space and CPU.

*1) **Time**:* to assess the real-time feature extraction and NN inference overhead, we utilize *iperf* [35] to generate varying numbers of concurrent flows, ranging from 1 to 128. Each concurrency level is repeated for 8 iterations. Each flow transmit 10MB TCP traffic at the maximum sending rate, and all the flows are sent concurrently in one iteration.

As shown in Table VIII, with the increase in the number of concurrent flows, the average transmission time (**WO**) also increases. This is because the system has only 8 cores, and the network transmission rate is fixed. Therefore, with a constant amount of data to be transmitted, as the number of concurrent flows increases, the transmission time also increases. However, even with the addition of feature extraction and NN inference (**W**), the average transmission time for **W** and **WO** does not show a significant difference, with instances where one is smaller or larger than the other. This suggests that the implemented feature extraction and NN inference do not introduce a significant delay to the network.

Due to the latency caused by the network stack and transmission link between two packets, as long as the feature extraction time is less than the delay time, it has little impact on the network connection when performing feature extraction upon receiving a packet. In an ideal scenario where there is no significant delay in the transmission link, the latency of the network stack can be approximated using *ping 127.0.0.1*. The measured average latency of the network stack on the experimental machine is 46000 ns, significantly higher than the values of **FEPP** (average feature extraction time per packet) and **IPF** (average NN inference time per flow). Therefore, this explains the close results observed between **W** and **WO**.

IPF exhibits fluctuations in the range of 3000-5000 ns, while **FEPP** shows two stable phases, particularly around 8 concurrent flows. This behavior is attributed to the fact

TABLE VII: Effectiveness of real time detection for different benign and intrusion behaviours

	Benign					Intrusion						
	FTP Download	**FTP Upload**	**Http Traffic**	**TCP Traffic**	**SSH**	**FTP-Patator**	**SSH-Patator**	**Dos Goldeneye**	**Port Scan**	**Dos Slowhttptest**	**Web Brute Force**	**Web XSS**
Accuracy	**0.973**	1.000	**0.950**	1.000	**0.964**	1.000	**0.997**	1.000	1.000	1.000	1.000	**0.983**

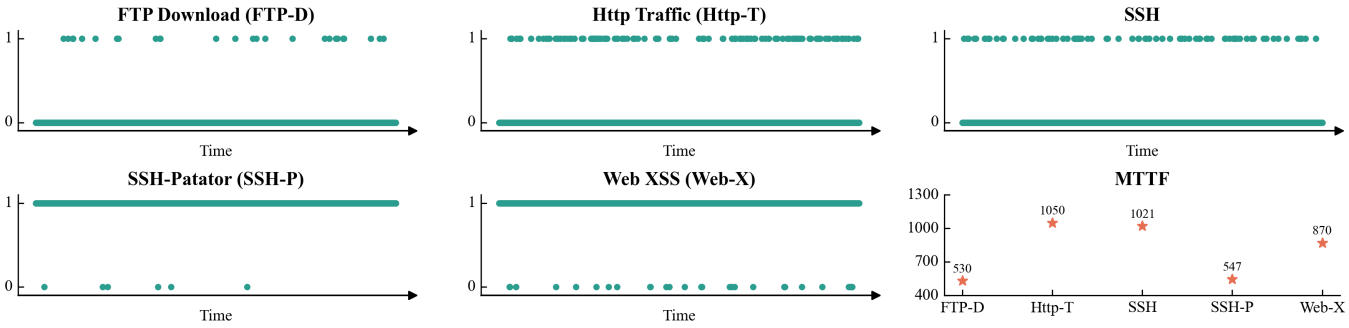


Fig. 7: Reliability in real time detection

that inference is performed only at the completion of each flow, causing **IPF** to be influenced by other concurrently processed flows when using eBPF tail call. On the other hand, feature extraction occurs immediately upon receiving a packet, resulting in a more stable pattern. Given the limitation of 8 cores in the system, when the number of concurrent flows exceeds 8, all cores become fully occupied, leading to an increase in the execution time of **FEPP**.

TABLE VIII: Time overhead of feature extracting and NN inference. **WO** denotes average transmission time without feature extracting and NN inference. **W** denotes average transmission time with feature extracting and NN inference. **FEPP** denotes average feature extraction time per packet. **IPF** denotes average NN inference time per flow. The units of **WO** and **W** are seconds, and the units of **FEPP** and **IPF** are nanoseconds.

# Flows	WO (s)	W (s)	FEPP (ns)	IPF (ns)
1	0.38	0.39	96.50	5189.88
2	0.80	0.78	90.69	4570.06
4	1.65	1.60	90.91	3691.25
8	3.33	3.31	134.81	3697.88
16	6.49	6.52	127.90	2945.18
32	13.27	13.19	142.96	3333.93
64	26.07	26.19	145.11	3872.96
128	50.30	50.59	148.28	4306.88

To compare the inference time overhead, we also measure the **IPF** of the existing methods, as shown in Figure 8. Linear SVM during inference is equivalent to a single-layer NN, hence the minimum **IPF**. Our method and NN-int8 have similar inference times, but we simplify the NN inference algorithm implementation. The decision paths of DT are uncertain and vary with the number of flows. Although DT is structurally simpler, their implementation is constrained by eBPF memory access, necessitating the use of eBPF Maps to store parameters. Accessing parameters through eBPF helper functions at each layer introduces additional memory access overhead, resulting in an increased **IPF**. Overall, our method reduces inference time overhead while maintaining performance.

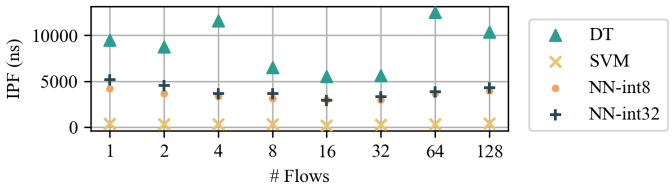


Fig. 8: IPF of the existing methods.

*2) **Space**:* Storage overhead comprises of two components. The initial component comes from storing the features of presently active flows. Each flow has six features stored as *int64*, resulting in a total overhead requirement of $48n$ bytes, where n represents the current number of active flows. The second component involves storing the NN parameters. With the sizes of the three layers in the NN as $6\times32, 32\times32, 32\times2$, and using *int32* for storage, along with the hidden layer output uniformly represented by an *int32* array of 32 elements, a total of 5248 bytes is needed.

*3) **CPU**:* eBPF program is triggered with the XDP hook, so its impact on the CPU is mixed in the kernel process and does not exist as a separate process. In order to isolate the CPU overhead of the eBPF program, we use *perf* [42] to instrument CPU performance. We use *iperf* to send at the maximum rate for 60 seconds, with the number of concurrent processes ranging from 1-128. During the first 0-30 seconds of the network transmission, we randomly start *perf* to samples at 99Hz for 30 seconds, and then record the CPU overhead caused by the eBPF program. Since NN inference starts after

each flow ends, the recorded results is the overhead of real-time feature extraction for each packet.

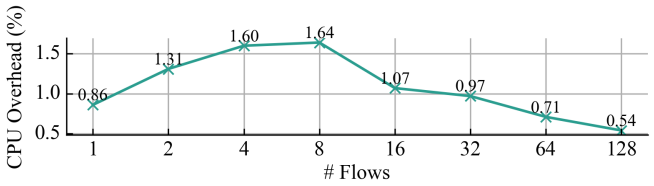


Fig. 9: CPU Overhead of Feature Extraction

As shown in Figure 9, the CPU overhead remains between 0.54% and 1.64%. If the number of concurrent flows does not exceed 8 (total CPU cores), the CPU overhead increases as the number of flows grows because the total bandwidth becomes larger. After exceeding the limit, each flow not only has to compete for bandwidth, but also for CPU cores, so the total bandwidth is declining and the CPU overhead decreases even the number of flows grows.

E. Overlooked Issues in Existing Methods

The worst-case spatial complexity of DT used in [12] is $O(2^n)$, where n represents the depth [43]. Although tree pruning can reduce nodes in the training process, the number of nodes in the trained DT varies even for the same dataset. We train 1024 DT and the training data comprised 80% randomly sampled data from each category of CIC-IDS-2017. As shown in Figure 10, the distribution of the number of nodes is mainly spread between 300 and 600. In order to hot-update

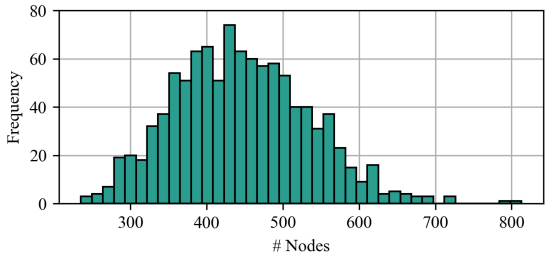


Fig. 10: Distribution of the number of DT nodes.

parameters, eBPF program has to consider the worst case, which introduces exponential memory overheads to the kernel. In addition, the depth of DT also determine its performance. As shown in Figure IX, when using the Top 6 features, if the depth is less than 9, the performance of DT is inferior to our method (NN-int32 Q+Top K). DT implemented in [12] requires at least 4 *int64* arrays. If the depth is set to 10, then the worst-case memory would reach $4 \times 8 \times 2^{10} = 32$ KB. However, our method achieves performance close to that of DT while reducing memory overhead to 5 KB (§ V-D).

Although DT achieves great anomaly detection performance with a simple structure, it still suffer from the problem of concept drift. To illustrate this, we remove one type of intrusion from the training and testing datasets from CIC-IDS-2017 at a time, then train a DT with a depth of 10 and use

TABLE IX: Impact of DT Depth [12]

Depth	Accuracy	Precision	Recall	F1-score
3	0.862	0.599	0.910	0.722
4	0.886	0.990	0.425	0.594
5	0.914	0.829	0.706	0.763
6	0.944	0.799	0.958	0.871
7	0.967	0.898	0.941	0.919
8	0.971	0.966	0.886	0.924
9	**0.983**	**0.964**	**0.951**	**0.957**
10	0.987	0.962	0.972	0.967

the trained DT to detect the removed intrusions. We find that although the average F1-score of the DT on the testing dataset reaches 0.969, the accuracy on each removed intrusion types is low, as shown in Figure 11. We also find that NN generally performs better than DT on the removed intrusion types but still suffer from the problem of concept drift.

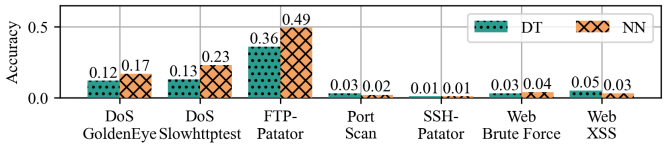


Fig. 11: Accuracy on the removed intrusions

When using non-linear SVM, we find that even with the same training dataset and Top 6 features, the training time of SVM does not stop even after several hours and is much longer compared to DT and NN. DT and NN takes 5s and 71s respectively to achieve the effectiveness shown in Table V. Even we set the maximum number of iterations to 1000 during training, SVM training is still slow. As shown in Table X, results on the testing dataset indicate that non-linear kernels do not improve the performance of the model. Computations involving the exponential function e^x in the *RBF* and *Sigmoid* kernels also pose significant challenges for its integer implementations in eBPF.

TABLE X: Performance of SVM with non-linear kernels

SVM Kernel	Accuracy	Precision	Recall	F1-score	Training Time (s)
Linear	0.262	0.210	0.999	0.348	1289
Polynomial	0.198	0.197	0.999	0.329	1414
RBF	0.222	0.202	0.999	0.336	2147
Sigmoid	0.421	0.196	0.625	0.298	2706

Compared to NN-int8 [14], our method (NN-int32) is not simply increase the length of quantized integers from 8 bits to 32 bits. Hara et al. [14] use *int8* to store the model parameters, while each layer still use *int32* to store the input and output. Thus, for each layer, the input is first quantized into *int8*, then undergoes linear and relu layer, and finally, dequantized to obtain the output in *int32*, which serves as the input for the next layer. The quantization and dequantization steps in each layer add considerable computation overhead and error to the real-time inference. Furthermore, the two steps introduce

extra memory overhead beyond the model parameters [44]. However, our method essentially stores floating-point numbers in the form of fixed-point numbers within int32, eliminating the need for additional quantization and dequantization steps during inference. We design our NN inference algorithm and implementation mechanism to accommodate the limitations of eBPF overlooked in [14]. We significantly improve performance while reducing the computational complexity.

VI. Related Work

Real-time intrusion detection and prevention can protect system from potential impact on its functionality [3]. However, current methods [4]–[7] focus on algorithmic improvements while neglecting the CPU, memory, and context switch overhead during real-time packet capturing. Unlike these methods, we utilize eBPF to offload the intrusion detection model into XDP hook, enabling real-time packet capturing and subsequent analysis within the kernel. We significantly reduces the kernel-user context switch overhead and make it compatible with a wider range of NIC architectures in a cost-effective way.

There are also similar methods that directly use eBPF for intrusion detection. [12] first implements DT in eBPF, but the structure of DT becomes unfixed after each training, presenting challenges in storing and updating it within eBPF. Linear Support Vector Machine employed by [13] may not be suitable for intrusion detection, which is not a straightforward linearly separable problem. NN has fixed structure and strong fitting capabilities. Therefore, NN is widely used in intrusion detection task now [9]. [14] implements NN in eBPF based on *int8* quantization. However, the quantization method introduces significant errors and unnecessarily complicates the implementation process. Moreover, [12]–[14] neglect the critical analyses such as feature importance, overlooked problems like race conditions and limitations in implementing complex algorithms, and the reproduction real-time intrusions for evaluation. We redesign a new NN inference mechanism based on int32 and implement it in eBPF through chained eBPF tail calls. We then propose a thread-safe parameters hot-updating mechanism. Through comprehensive evaluations, we demonstrate that our method achieves performance and inference overhead comparable to the existing methods while reducing memory overhead.

VII. Limitations & Discussions

Since the detection occurs only at the end of a flow, our current implementation is not suitable for intrusions during persistent connections. *Persistent connection* means that it remains open for a long duration without being disconnected. The detection of intrusions in persistent connections requires detecting every packet or flow features within a specific time window. This limitation can be addressed by replacing the training dataset and conducting detection immediately after each packet. Since current mainstream intrusion detection datasets are primarily based on features of the entire flow [33], and conducting detection for each received packet introduces excessive computational overhead, we choose to perform detection only after the completion of a flow.

VIII. Conclusion

We address the overlooked issues in eBPF such as the maximum instructions number, integer-only arithmetic operations, and race conditions. Subsequently, we implement an efficient real-time intrusion detection and defense prototype within the kernel. First, we redesign an NN inference algorithm based on *int32* quantization and integer-only arithmetic. To overcome the maximum eBPF instruction limitation, we decompose the algorithm and employ chained eBPF Tail Calls for real-time inference in XDP. Furthermore, we implement a thread-safe mechanism for hot-updating model parameters. The evaluation results show that our methods reduce memory and inference time overhead while maintains performance comparable to the existing state-of-the-art method using eBPF.

Acknowledgment

We greatly appreciate the insightful feedback from the anonymous reviewers. The research is sponsored by the National Key Research and Development Program of China (2019YFB1804002), the National Natural Science Foundation of China (No.62272495) and the Guangdong Basic and Applied Basic Research Foundation (No.2023B1515020054), and sponsored by Huawei. The corresponding author is Pengfei Chen.

References

[1] Shih-Wei Li, John S Koh, and Jason Nieh. Protecting cloud virtual machines from hypervisor and host operating system exploits. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1357–1374, 2019.

[2] Amazon. Cloud security software. https://aws.amazon.com/marketplace/solutions/security.

[3] Ravi Sekar and Prem UppuluriR Sekar. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *8th USENIX Security Symposium (USENIX Security 99)*, 1999.

[4] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: an ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[5] Steve TK Jan, Qingying Hao, Tianrui Hu, Jiameng Pu, Sonal Oswal, Gang Wang, and Bimal Viswanath. Throwing darts in the dark? detecting bots with limited data using neural data augmentation. In *2020 IEEE symposium on security and privacy (SP)*, pages 1190–1206. IEEE, 2020.

[6] HyungBin Seo and MyungKeun Yoon. Generative intrusion detection and prevention on data stream. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4319–4335, 2023.

[7] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Recent Advances in Intrusion Detection: 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings 11*, pages 116–134. Springer, 2008.

[8] iptables. https://linux.die.net/man/8/iptables.

[9] Feng Wei, Hongda Li, Ziming Zhao, and Hongxin Hu. Xnids: Explaining deep learning-based network intrusion detection systems for active intrusion responses. In *32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA*, 2023.

[10] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating linux security with ebpf iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 108–110, 2018.

[11] eBPF. Available at: https://ebpf.io/.

[12] Maximilian Bachl, Joachim Fabini, and Tanja Zseby. A flow-based ids using machine learning in ebpf. *arXiv preprint arXiv:2102.09980*, 2021.

[13] NEMALIKANTI ANAND, MA SAIFULLA, and Pavan Kumar Aakula. High-performance intrusion detection systemusing ebpf with machine learning algorithms. 2023.

[14] Takanori Hara and Masahiro Sasabe. On practicality of kernel packet processing empowered by lightweight neural network and decision tree. In *2023 14th International Conference on Network of the Future (NoF)*, pages 89–97. IEEE, 2023.

[15] Linux. Bpf design q&a. Available at: https://www.kernel.org/doc/html/v5.2/bpf/bpf_design_QA.html.

[16] Jonathan Corbet. Concurrency management in bpf. https://lwn.net/Articles/779120/.

[17] Ahmad Javaid, Quamar Niyaz, Weiqing Sun, and Mansoor Alam. A deep learning approach for network intrusion detection system. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*, pages 21–26, 2016.

[18] Jihyun Kim, Jaehyun Kim, Huong Le Thi Thu, and Howon Kim. Long short term memory recurrent neural network classifier for intrusion detection. In *2016 international conference on platform technology and service (PlatCon)*, pages 1–5. IEEE, 2016.

[19] Guansong Pang, Chunhua Shen, Longbing Cao, and Anton Van Den Hengel. Deep learning for anomaly detection: A review. *ACM computing surveys (CSUR)*, 54(2):1–38, 2021.

[20] Cillium. Bpf and xdp reference guide. Available at: https://docs.cilium.io/en/latest/bpf/.

[21] Limin Yang, Wenbo Guo, Qingying Hao, Arridhana Ciptadi, Ali Ahmadzadeh, Xinyu Xing, and Gang Wang. Cade: Detecting and explaining concept drift samples for security applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2327–2344, 2021.

[22] Tcpdump. https://www.tcpdump.org/manpages/tcpdump.1.html.

[23] Mohammad Saiful Islam Mamun Arash Habibi Lashkari, Gerard Draper-Gil and Ali A. Ghorbani. cicflowmeter. Available at: https://www.unb.ca/cic/research/applications.html.

[24] Dongzi Jin, Yiqin Lu, Jiancheng Qin, Zhe Cheng, and Zhongshu Mao. Swiftids: Real-time intrusion detection system based on lightgbm and parallel intrusion detection mechanism. *Computers & Security*, 97:101984, 2020.

[25] Libpcap. https://www.tcpdump.org/manpages/pcap-filter.7.html.

[26] Programming with pcap. Available at: https://www.tcpdump.org/pcap.html.

[27] pidstat. Available at: https://man7.org/linux/man-pages/man1/pidstat.1.html.

[28] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with ebpf. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, 2023.

[29] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pages 54–66, 2018.

[30] Tc-bpf. Available at: https://man7.org/linux/man-pages/man8/tc-bpf.8.html.

[31] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, 2018.

[32] John H Ring IV, Colin M Van Oort, Samson Durst, Vanessa White, Joseph P Near, and Christian Skalka. Methods for host-based intrusion detection with deep learning. *Digital Threats: Research and Practice (DTRAP)*, 2(4):1–29, 2021.

[33] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. *ICISSp*, 1:108–116, 2018.

[34] httperf. Available at: https://github.com/httperf/httperf.

[35] iperf. Available at: https://iperf.fr.

[36] GoldenEye. Available at: https://github.com/jseidl/GoldenEye.

[37] Slowhttptest. Available at: https://github.com/shekyan/slowhttptest.

[38] patator. Available at: https://github.com/lanjelot/patator.

[39] DVWA. Damn vulnerable web app. Available at: https://github.com/digininja/DVWA.

[40] selenium. Available at: https://selenium-python.readthedocs.io/.

[41] Brian W Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405(2):442–451, 1975.

[42] perf. Available at: https://perf.wiki.kernel.org/index.php/Tutorial.

[43] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[44] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation, 2020.