On Practicality of Kernel Packet Processing Empowered by Lightweight Neural Network and Decision Tree

Takanori Hara
Graduate School of Science and Technology
Nara Institute of Science and Technology
Ikoma, Japan
hara@ieee.org

Masahiro Sasabe
Faculty of Informatics
Kansai University
Takatsuki, Japan
m-sasabe@ieee.org

***Abstract*—Kernel packet processing such as extended Berkeley Packet Filter (eBPF) and eXpress Data Path (XDP) is a promising framework that can speedily/efficiently process packets without passing them to conventional packet processing software running on the user space. Several studies pointed out the possibility of eBPF empowered by simple machine learning techniques (e.g., decision tree (DT)) to realize intelligent packet processing (e.g., intrusion detection) in the kernel space. Note that the quantitative evaluation of both packet processing and detection performance has not been conducted sufficiently. In addition, to ensure the kernel stability and safety, the eBPF program must process packets under strict constraints such as prohibition of floating-point number, which is usually used in neural networks (NNs). In this paper, we examine the possibility of NN-empowered eBPF/XDP based packet processing. More specifically, we first train a floating-point NN and quantize it as a fixed-point NN using 8-bit integers in the user space. Then, we implement the lightweight NN in the eBPF/XDP program to achieve fast packet processing with integer-arithmetic-only inference in the kernel space. Experimental results show that (1) the integer-arithmetic-only NN (resp. DT) classifier can drastically reduce the inference time to 15.3% (resp. 1.6%) while suppressing degradation of classification performance, (2) the lightweight NN classifier can improve the inference performance in case of multi-class classification, and (3) the kernel-based method with NN (resp. DT) classifier can process received packets in a real-time manner under a certain transmission rate, i.e., 300,000 pps (resp. 450,000 pps).**

***Index Terms*—Kernel packet processing, extended Berkeley Packet Filter (eBPF), eXpress Data Path (XDP), intrusion detection system (IDS), machine learning (ML), quantization, fixed-point neural network (NN).**

I. Introduction

The *extended Berkeley Packet Filter* (eBPF) framework can control a Linux kernel through an arbitrary program [1]. This technology not only improves the observability of kernel logs but also enables faster packet processing on the kernel [2]. In eBPF-based networking, when a new packet arrives at a network interface, the eBPF can process it according to predefined processing rules. For example, the eBPF program can record the packets only on TCP port 80 and drop the packets received only from a certain IP address. The *eXpress Data Path* (XDP) is also one of the Linux kernel frameworks for fast programmable packet processing [2], [3]. It runs an eBPF program at the network driver, which is the closest point to the network interface.

Recently, *machine learning* (ML) has been applied to various domains in networking [4]. Application of ML can realize intelligent networking such as automation of network operations and recognition of traffic characteristics. For example, there have been ML-based *intrusion detection systems* (IDSs) [5], [6], most of which can successfully improve the detection accuracy of attacks. Note that these programs are designed to run on the user space. In [7], the authors proposed a flow-based IDS using the eBPF program, which adopts the *decision tree* (DT) algorithm. In addition, they showed that the DT-empowered eBPF program has higher processing efficiency than the conventional program running on the user space.

Although eBPF is a promising framework, it also has several challenging issues. Since eBPF programs are designed to run on the kernel, they must run on a single thread and restricted by various kinds of strict constraints to ensure the kernel stability and safety [2], [8]. These constraints include the limits on the number of instructions and stack space, the prohibitions against unbounded loops, non-static global variables, variadic functions, and floating-point numbers, and the array bound checking. Table I summarizes these constraints, which are not concerned in the user-space programs. The details of these constraints will be described in Section III-B.

These constraints make the eBPF program difficult to execute conventional ML approaches, which are originally designed to run on the user space. In [7], the authors realized the flow-based IDS using the eBPF program empowered by the *64-bit fixed-point (integer)* DT algorithm. In this paper, we newly examine the possibility of eBPF/XDP packet processing empowered by a neural network (NN), which is one of the most common ML methods. Similarly to [7], we adopt a flow-

This work was supported in part by the Japan Society for the Promotion of Science (JSPS) KAKENHI (B) under Grant 22H03586, the JSPS Grant-in-Aid for Young Scientists under Grant 23K16869, and the Support Center for Advanced Telecommunications Technology Research (SCAT).

TABLE I: Constraints in eBPF (kernel-space) programming and there relation to user-space programming.

Constraint	User space	eBPF (Kernel space)	Alternatives in eBPF
Unbounded loop	Supported	Not supported	Bounded loop
Floating-point number	Supported	Not supported	Fixed-point number
Non-static global variables	Supported	Not supported	BPF map
Variadic functions	Supported	Not supported	-
Multi-threaded programming	Supported	Not supported	Single-threaded programming
Array bound checking	Not required	Required	-
Maximum number of instructions	Not limited	1 million since kernel version 5.1	-
Maximum number of branches	Not limited	8192	-
Maximum stack space	8192 KB (default)	512 bytes	Tail call, BPF map, and quantization

based IDS as a case study. More specifically, we adopt a multi-layer perceptron (MLP) as a floating-point NN classifier. We first train the floating-point NN classifier with high accuracy and quantize it as a fixed-point NN one using 8-bit integers in the user space. Then, we implement the quantized fixed-point NN classifier in the eBPF/XDP program, which realizes fast packet processing with integer-arithmetic-only inference in the kernel space.

Another important concern in ML-empowered eBPF/XDP packet processing is the relationship between processing speed and resource efficiency. From the viewpoint of high-speed packet processing, kernel bypassing methods, represented by *Data Plane Development Kit* (DPDK) [9], are one of the promising approaches. They can almost achieve the theoretical performance limit of packet processing with the help of multi-threaded accelerators in the user space, at the cost of CPU-intensive polling of packets even under no packet arrival. On the contrary, an eBPF program runs on a single thread in the kernel space, which limits the packet processing speed but achieves high resource efficiency (i.e., CPU utilization proportional to packet arrival rate). In this paper, we quantitatively reveal the potential of ML-empowered eBPF/XDP packet processing for IDS, in terms of packet processing speed, resource efficiency, and detection performance.

Experimental results will show that (1) the integer-arithmetic-only NN (resp. DT) classifier running on the kernel space can reduce the inference time to 15.3% (resp. 1.6%) while suppressing degradation of classification performance compared with the floating-point NN (resp. DT) running on the user space, and (2) the kernel-based method with NN (resp. DT) classifier can process received packets in a real-time manner under a certain transmission rate, i.e., 300,000 pps (resp. 450,000 pps).

The rest of the manuscript is organized as follows. Section II gives the related work. In Section III, we introduce some preliminaries. In Section IV, we propose the NN-empowered eBPF for packet processing. Section V shows the fundamental characteristics of the proposed method. Finally, Section VI gives the conclusion and future work.

II. Related Work

There have been studies on eBPF-based networking in terms of performance analysis [3], packet filtering [10], virtual network functions [11], service chaining [12], and IPv6 segment routing [13]. In [3], the authors focused on XDP. They revealed the performance limit of XDP through the comparison with DPDK [9] and network stack on the Linux kernel. DPDK, which is one of the kernel bypassing methods, can almost achieve the theoretical limit of packet processing performance using multi-threaded accelerators in the user space [6], [9], [14]. As a result, it also has a potential drawback of CPU-intensive polling of packets even under no packet arrival. On the contrary, the kernel-based methods such as eBPF and XDP cannot utilize the multi-threading property to ensure the kernel stability and safety. Note that they can achieve highly resource-efficient packet processing proportional to packet arrival rate and coexist with the kernel network stack. In this paper, we focus on the lightweight eBPF approach to realize energy-efficient edge computing. The recent eBPF-related survey can be found in [15].

Applying ML to networking has been expected to realize automation of network operations and recognition of traffic characteristics [4]. For example, there have been studies on ML-based IDSs to improve the detection performance [5], [6]. As for IDSs, there are several datasets reflecting the current trends of attacks [16], [17]. There have been studies on fast packet processing architectures using a programmable hardware accelerator [18]–[21]. These architectures can mitigate the packet processing overhead by offloading the CPU-intensive tasks to the dedicated hardware. There are other studies on implementing NN-based IDSs using the DPDK library, which realize fast packet processing at the expense of a large amount of computational resources [6], [14]. In recent years, several studies focused on the applicability of ML to eBPF [7], [22]. In [22], the authors presented only the concept of an IDS with both eBPF and ML. In [7], the authors proposed a flow-based IDS by implementing the DT algorithm on the eBPF program. However, these existing studies did not sufficiently reveal the possibility of the eBPF/XDP program with the NN, which is one of the most popular ML methods. In this paper, we mainly focus on the possibility of eBPF/XDP packet processing empowered by a lightweight NN for the IDS. To the best of our knowledge, this is the first work on anomaly packet detection using the NN-empowered eBPF/XDP.

Due to the above-mentioned eBPF constraints, ML-empowered eBPF has several challenging issues. To ensure the kernel stability and safety, eBPF programs must run on a single thread and adopt special variables, which require array bound checking. On the contrary, conventional programs running

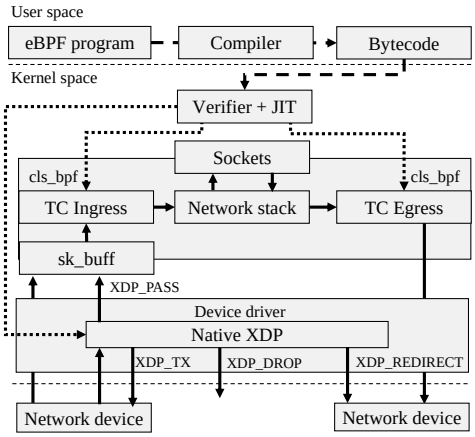


Fig. 1: eBPF workflow.

on the user space do not have such constraints but suffer overhead from packet between the kernel and user spaces. This overhead becomes the bottleneck because the passing process runs on a single thread. In [7], the authors showed that the DT-empowered eBPF program has higher processing efficiency than the conventional program running on the user space. In this paper, we quantitatively reveal the potential of eBPF/XDP packet processing empowered by ML (i.e., NN or DT), compared with the conventional program running on the user space in terms of packet processing speed, resource efficiency, and detection performance.

There are two challenging issues to adopt the NN architecture to eBPF-based packet processing: (1) inference speed and (2) model size. As for the inference speed, there have been many studies on accelerating the NN inference [23]–[25]. The fixed-point NN is one of the approaches for accelerating the NN inference, which represents the data type of the NN as the fixed-point (integer) type [24]. As for the model size, many studies adopted the quantization approaches to reduce the model size and memory usage, which is one of the NN compression approaches [24], [25]. The quantization converts the data type used in the NN architecture (i.e., weight parameters and input values) into the low-bitwidth data type (e.g., converting the 32-bit floating-point number (*float32*) to 8-bit integer number (*int8*)), which can reduce the model size to 1/4 with a risk of performance degradation of inference. Focusing on these characteristics, we expect that the fixed-point NN and the quantization method have a key role to overcome the eBPF constraints. By integrating these approaches, we implement the integer-only lightweight NNs in the eBPF/XDP program.

III. Preliminaries

A. eBPF Workflow

Fig. 1 illustrates the eBPF workflow. An eBPF program must be verified by an eBPF verifier to check the violation of eBPF constraints before its injection into the kernel code, so as to avoid crashes and infinite loops. More specifically, an eBPF program written in a restricted C language is first converted into the corresponding eBPF byte code by a compiler in the user space. After this eBPF byte code is verified by the eBPF verifier, it is further converted into a native instruction set by a Just-In-Time (JIT) compiler. The converted eBPF program runs as efficiently as any other kernel code and kernel module (dotted lines in Fig. 1). Finally, the eBPF program is attached to an arbitrary event point (dashed lines in Fig. 1). When the attached event (e.g., the packet arrival) occurs, the eBPF/XDP program is executed (solid lines in Fig. 1).

Traffic control (TC) is an architecture to schedule traffic on the kernel and processes arrival packets via *sk_buff* [26]. sk_buff is a metadata structure to access the packet data in the buffer. TC consists of the *queuing discipline* (*qdisc*) and the *classifier*. The qdisc is a scheduler to schedule, classify, filter, and shape packets outgoing to an interface. It enqueues packets to be transmitted into the queuing buffer and processes them. Then, it dequeues the packets from the queuing buffer and passes them to the network driver. *cls_bpf* is a programmable classifier, which can attach the eBPF program to TC ingress and egress. In other words, the cls_bpf enables the eBPF program to process both incoming and outgoing packets.

The XDP program is a special case of the eBPF program, which realizes the programmable fast packet processing at the closest point (i.e., network driver) to the network interface card (NIC) [2], [3]. Similarly to the eBPF program, the XDP program is strictly restricted to ensure the kernel stability and safety. Therefore, it is also difficult to execute the ML-empowered program, which requires floating-point arithmetic with large model size, on the kernel space. The (native) XDP attaches the eBPF program to the network driver and conducts high-performance packet processing before passing the packets from sk_buff. Different from TC, XDP only supports the incoming packets and requires the dedicated network driver to execute the XDP program. Recently, the *generic XDP* has been proposed to execute the XDP program without the dedicated network driver by passing the packets through a kernel stack emulating the XDP functionality at the sacrifice of packet processing speed [27]. The generic XDP processes the packets immediately after passing the packets from sk_buff.

B. eBPF Constraints

An eBPF program is written in a restricted C language. The main eBPF constraints can be summarized in Table I. To ensure the kernel stability and safety, the eBPF program must run on a single thread. Since all instructions must be designed in the integer arithmetic, the eBPF program must adopt the fixed-point number instead of the floating-point number. The maximum number of instructions per eBPF program is restricted to 1 million BPF instructions and the maximum number of branches caused by jump instructions is limited to 8192. The maximum stack space of eBPF program is restricted to 512 bytes, which is much smaller than the default value 8192 KB of the maximum stack space in the user space. The eBPF framework provides a mechanism called *tail call* to alleviate the problem of stack space limitation. An eBPF program can call another one through a tail call, which does not return to the caller. Note that the maximum

number of tail calls is limited to 32. A BPF map is a key-value store sharing data between the user and kernel spaces, which can be in substitution for non-static global variables. Quantization techniques, which make the size of NN smaller, can also be used to alleviate the problem of stack space limitation. These constraints prevent the conventional use-space programs from running as eBPF programs and make the eBPF programs difficult to execute the conventional ML approaches. In addition, they may cause the performance degradation when checking boundary values of arrays.

C. CIC-IDS2017 Dataset

In this paper, we adopt the CIC-IDS2017 dataset [17], which has commonly been used for the IDS performance, as in [7]. The CIC-IDS2017 dataset contains the information about normal packets and six types of attack packets i.e., botnet, brute force, DoS, infiltration, portscan, and web attack. The total amount of packet capture data is more than 50 GB. In this paper, we assume that each packet belongs to a certain *flow*, which is defined as 5-tuple packet information of source and destination IP addresses, source and destination port numbers. The total number of flows becomes 2,315,197 where the number of normal (resp. attack) flows is 1,729,999, (resp. 585,198). In [7], the authors conducted feature selection to improve the performance of attack detection. As a result, they showed that the flow arrival time and packet length are the important features.

D. Quantization

Quantization is a process of converting the data type used in the NN architecture into the low-bitwidth data type. For instance, the 8-bit quantization transforms the 32-bit floating-point NN (*float32*) into the 8-bit integer NN (*int8*), which can reduce the model size and its memory usage to 1/4. From the viewpoint of eBPF/XDP program, the quantization can increase the possibility to satisfy the stack space limit.

Scale quantization [28], which is one of the quantization schemes, maps a real value $x \in [-\alpha, \alpha]$ into a b-bit signed integer value $x_q \in [-2^{b-1}+1, 2^{b-1}-1]$ as follows:

$$x_q = \mathrm{clip}(\mathrm{round}(\sigma x), -2^{b-1}+1, 2^{b-1}-1), \tag{1}$$

where $\sigma = (2^{b-1}-1)/\alpha$ denotes a scale factor, $\mathrm{round}(x)$ is a function to round the value x, and $\mathrm{clip}(x, y, z)$ is a function that outputs x, y, z for $z \leq x \leq y$, $x < y$, and $z > x$, respectively.

On the contrary, *dequantization* tries to restore the quantized the b-bit signed integer value x_q to the original value x', which can be defined as follows:

$$x' = \sigma^{-1} x_q. \tag{2}$$

There can be an error between the dequantized value x' and the original one x.

Post-training quantization (PTQ) aims at quantizing the learned model by determining the quantized parameters of the learned model's weights and activations [29]. To suppress the inference performance degradation of the quantized model as much as possible, we need to adjust the quantized parameter α. This is done by the calibration, which adjusts the quantized parameter α for the inputs and weights in an offline manner such that Kullback-Leibler (KL) divergence [30] between the original distribution and quantized one is minimized.

We assume that a fully-connected layer performs matrix multiplication, i.e., $\mathbf{Y} = \mathbf{X}\mathbf{W}$, where $\mathbf{Y} = (y_{n,m}) \in \mathbb{R}^{N \times M}$ is an output matrix, $\mathbf{X} = (x_{n,k}) \in \mathbb{R}^{N \times K}$ is an input matrix, and $\mathbf{W} = (w_{k,m}) \in \mathbb{R}^{K \times M}$ is a weight matrix. In case of the scale quantization, the output matrix $\mathbf{Y}$ can be approximated to the matrix multiplication of the dequantized matrices $\mathbf{X}_q = (x_{q,n,k}) \in \mathbb{Z}^{N \times K}$ and $\mathbf{W}_q = (w_{q,k,m}) \in \mathbb{Z}^{K \times M}$ by using the scale factors $\boldsymbol{\sigma} = (\sigma_{n,k}) \in \mathbb{R}^{N \times K}$ and $\boldsymbol{\sigma}_{\mathrm{weight}} = (\sigma_{\mathrm{weight},k,m}) \in \mathbb{R}^{K \times M}$:

$$y_{n,m} = \sum_{k \in K} x_{n,k} \cdot w_{k,m} \approx \sum_{k \in K} \frac{x_{q,n,k} \cdot w_{q,k,m}}{\sigma_{n,k} \sigma_{\mathrm{weight},k,m}},$$

where $\sigma_{n,k}$ and $\sigma_{\mathrm{weight},k,m}$ denote the scale factor of $x_{n,k}$ and that of $w_{k,m}$, respectively.

To control the balance between the model accuracy and computational complexity, we can adjust α in a different quantization granularity. The quantization with the finest granularity prepares individual quantization parameters for all elements of the matrix. The quantization with the coarsest granularity shares one quantization parameter among all elements of the matrix. The quantization with the moderate granularity shares one quantization parameter per one or more columns of the matrix.

E. Integer-arithmetic-only Neural Network

In general, an MLP is constructed by a floating-point NN, which cannot be supported by the eBPF/XDP program. Therefore, we need to convert the floating-point NN into the fixed-point one. Here, the fixed-point number can be represented by an integer value in which the lower n bits are treated as the decimal part [23]. More specifically, a floating-point value x_{float} can be converted into the fixed-point value x_{fixed} with the lower n bits of decimal part according to the following equation:

$$x_{\mathrm{fixed}} = \mathrm{round}(x_{\mathrm{float}} \cdot (1 << n)). \tag{3}$$

In Eq. (3), the fixed-point value x_{fixed} is obtained by multiplying x_{float} and 2^n, which is realized by n-bit shift left operation, and then rounding the decimal part by the $\mathrm{round}()$ function.

IV. NN-Empowered eBPF-based Packet Processing

A. Overview

Given a large amount of training data, each of which is a pair of a D-dimensional feature vector $\boldsymbol{x} = (x_1, \ldots, x_D) \in \mathcal{X}$ and the target score $y \in \mathcal{Y}$, an NN aims at representing a function f such that $y = f(\boldsymbol{x})$. More specifically, the function f is expressed by an NN with the fully-connected layers where the weight parameters are adjusted by the training data. In general, the weight parameters are expressed by floating-point numbers. Due to the eBPF constraints aforementioned

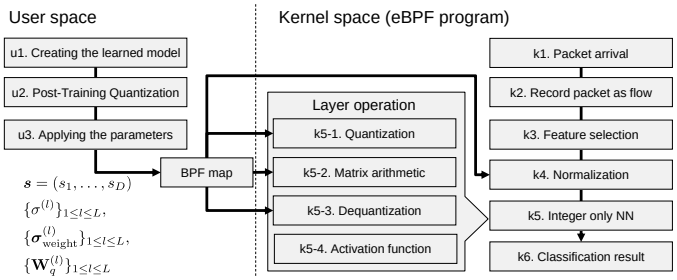


Fig. 2: Workflow of the proposed scheme.

before, we cannot directly implement the floating-point NN in the eBPF/XDP program. In addition, since the eBPF/XDP program should be lightweight and fast, we need to transform the floating-point NN to the 8-bit fixed-point (integer) NN to reduce the learned model size.

In this paper, we attempt to solve this problem by combining highly accurate and flexible learning in the user space and lightweight and fast inference with the eBPF/XDP program in the kernel space. In particular, we realize the lightweight and fast inference by ML on the eBPF/XDP program by (1) applying the PTQ-based 8-bit quantization to the learned model in the user space (See the details in Section IV-B.) and (2) performing the integer-arithmetic-only NN in the kernel space (See the details in Section IV-C). In what follows, we consider two types of classification: binary classification and multi-class classification. Note that the integer-arithmetic-only NN designed for the eBPF program is compatible with that for the XDP program. Fig. 2 illustrates the workflow of the proposed scheme.

B. Learning and Quantization of Neural Network in User Space

In this paper, we adopt supervised learning and realize a classification model as an NN classifier, which estimates the type of an arrival packet, i.e., normal/attack in the binary classification and the seven labels in the multi-class classification. We realize the NN classifier as an NN with $L = 3$ layers plus one input layer, where each of the outputs at lth ($l = 1, \ldots, L-1$) hidden layer employs an activation function of a rectified linear unit (ReLU) function.

The dimension M_l of the lth layer is set to be $M_0 = D$, $M_1 = 16$, $M_2 = 16$, and $M_L = M_{\text{class}}$, respectively, where $l = 0$ is used to represent the input layer and M_{class} means the number of outputs, i.e., $M_{\text{class}} = 2$ for the binary classification and $M_{\text{class}} = 7$ for the multi-class classification. In what follows, we mainly explain the binary classification.

We first explain how to generate the learned model in the user space. As for the training data, we adopt the $D = 12$ *float64* features, where the source and destination port numbers, protocol number, packet length, and packet direction are denoted by x_1–x_5, which are directly obtained from CIC-IDS2017 dataset, and the remaining are statistically calculated per flow. (See the details in Section IV-C.) The packet direction x_5 takes one if the packet is an incoming packet and zero otherwise. Since each feature $x_i \in \boldsymbol{x}$ has the different representation range, we apply the min-max normalization to each feature $x_i \in \boldsymbol{x}$ by using the corresponding normalization parameter s_i.

Here, we define $\boldsymbol{s} = (s_1, \ldots, s_D)$. Since the D-dimensional normalization parameter $\boldsymbol{s}$ will be used for the inference in the kernel space, it is stored in the BPF map. In the user space, we use a set $\mathcal{X}$ of features and a set $\mathcal{Y}$ of target scores in the training data as inputs and train the NN classifier by updating weight parameters $\{\mathbf{W}^{(l)}\}_{1 \le l \le L}$with the help of the backpropagation method [31]. The lth layer operation can be interpreted as the matrix multiplication $\mathbf{Y}^{(l)} = \mathbf{X}^{(l)} \cdot \mathbf{W}^{(l)}$, where $\mathbf{X}^{(l)} = (x_{i,j}) \in \mathbb{R}^{N \times M_{l-1}}$, $\mathbf{W}^{(l)} = (w_{i,j}) \in \mathbb{R}^{M_{l-1} \times M_l}$, $\mathbf{Y}^{(l)} = (y_{i,j}) \in \mathbb{R}^{N \times M_l}$ are the input, weight, and output matrices of the lth layer, respectively. Note that N means the batch size. After updating $\mathbf{W}^{(l)}$ of every lth layer, we can obtain the learned model (step u1 in Fig. 2).

Next, we introduce how to derive the information required for the NN classifier running as the eBPF/XDP program in the kernel space (step u2 in Fig. 2). The details of the NN-empowered eBPF/XDP program will be described in Section IV-C. We first quantize the parameters required for the learned NN classifier. For each layer l ($l = 1, \ldots, L$), we obtain the quantization parameter of $\mathbf{X}^{(l)}$ by adopting the quantization with the coarsest granularity, which shares one quantization parameter among all elements of $\mathbf{X}^{(l)}$. More specifically, we first adjust the quantization parameter $\alpha^{(l)}$ of $\mathbf{X}^{(l)}$ such that the KL-divergence between the original distribution and quantized one, with the help of the calibration. Next, the scale factor $\sigma^{(l)}$ of $\mathbf{X}^{(l)}$ is calculated, which is used to convert *float64* into *int8* by Eq. (1).

Similarly, for each layer l ($l = 1, \ldots, L$), we quantize the weight matrix $\mathbf{W}^{(l)}$ of the learned model. In case of the weight matrix $\mathbf{W}^{(l)}$, we adopt the quantization with the moderate granularity, which shares one quantization parameter per column of $\mathbf{W}^{(l)}$. More specifically, we first adjust the quantization parameters $\boldsymbol{\alpha}^{(l)}_{\text{weight}} = (\alpha^{(l)}_{\text{weight},j}) \in \mathbb{R}^{M_l}$ of $\mathbf{W}^{(l)}$ by minimizing the KL divergence between the original distribution and quantized one, where j means a column index. Next, we calculate the scale factors $\boldsymbol{\sigma}^{(l)}_{\text{weight}} = (\sigma^{(l)}_{\text{weight},j}) \in \mathbb{R}^{M_l}$ of $\mathbf{W}^{(l)}$, which are used to convert *float32* into *int8*. Finally, we quantize the weight matrix $\mathbf{W}^{(l)}$ by $\boldsymbol{\sigma}^{(l)}_{\text{weight}}$ and obtain the quantized weight matrix $\mathbf{W}^{(l)}_q = (w_{q,i,j}) \in \mathbb{R}^{M_{l-1} \times M_l}$.

These parameters (i.e., the D-dimensional normalization parameter $\boldsymbol{s}$, the scale factors $\{\sigma^{(l)}\}_{1 \le l \le L}$ and $\{\boldsymbol{\sigma}^{(l)}_{\text{weight}}\}_{1 \le l \le L}$, and the quantized weight matrix $\{\mathbf{W}^{(l)}_q\}_{1 \le l \le L}$) are stored in the BPF map such that they can be used by the NN classifier running on the kernel space (step u3 in Fig. 2). Note that these learned parameters can also be applied to the eBPF/XDP program through the BPF map in a real-time manner.

C. Fast Inference with eBPF and Neural Network in Kernel Space

We can realize the lightweight and fast NN classifier as the eBPF/XDP program running on the kernel space by integrating

both the quantized learned model and the integer-arithmetic-only inference. The NN classifier running on the kernel space refers to the parameters $\boldsymbol{s}$, $\{\sigma^{(l)}\}_{1\le l\le L}$, $\{\boldsymbol{\sigma}^{(l)}_{\rm weight}\}_{1\le l\le L}$, and $\{\mathbf{W}^{(l)}_q\}_{1\le l\le L}$ defined in the user space through the BPF map. In what follows, each variable is represented by a 64 bit integer fixed-point number (*int64*) with the lower 16-bit decimal part.

The eBPF/XDP program performs the integer-arithmetic-only inference whenever a new packet arrives at the network interface (step k1 in Fig. 2). It manages the flow by aggregating arrival packets with help of the BPF map. To manage the flows, the BPF map stores a collection $\mathcal{V} = \{\boldsymbol{v_k}\}_{\forall \boldsymbol{k}\in\mathcal{K}}$ of per-flow values, each of which is referenced by a key (flow ID) $\boldsymbol{k} \in \mathcal{K}$ and a mapping function $h : \mathcal{K} \to \mathcal{V}$. Note that $\mathcal{K}$ and $\mathcal{V}$ denote a set of keys and that of values, respectively. Here, the key $\boldsymbol{k}$ is the 5-tuple packet information of source and destination IP addresses ip_s and ip_d, source and destination port numbers $port_s$ and $port_d$, and protocol number $proto$, i.e., $\boldsymbol{k} = \langle ip_s, ip_d, port_s, port_d, proto\rangle$. The corresponding value $\boldsymbol{v_k}$ is the 5-tuple flow-related information of last packet arrival time $t_{\rm last}$, total sum of packet length len, total sum of packet direction $direction$, total sum of packet arrival interval $interval$, and number of packets num, i.e., $\boldsymbol{v_k} = \langle t_{\rm last}, len, direction, interval, num\rangle$.

The eBPF/XDP program first extracts source and destination IP addresses, source and destination port numbers, and protocol number from the header of received packet, and then generates the corresponding key $\boldsymbol{k}$. It then obtains the value $\boldsymbol{v_k}$ from the BPF map using the key $\boldsymbol{k}$. If it successfully obtains the corresponding value $\boldsymbol{v_k}$ from the BPF map, it updates $\boldsymbol{v_k}$ according to the arrival time t and information $(x_1, \ldots, x_5)$ of received packet. Since $(x_1, \ldots, x_5)$ in the IP packet header and t are represented by the integer values (*int*), they are transformed into the fixed-point numbers using 64-bit integer values (*int64*) by Eq. (3). After updating the BPF map, it further derives $(x_6, \ldots, x_{12})$ according to the following rules: $x_6 = t - t_{\rm last}$, $len \mathrel{+}= x_4$, $direction \mathrel{+}= x_5$, $interval \mathrel{+}= t - t_{\rm last}$, $t_{\rm last} = t$, $x_7 = len/num$, $x_8 = direction/num$, $x_9 = interval/num$, $x_{10} = \mathrm{abs}(x_4 - x_7)$, $x_{11} = \mathrm{abs}(x_5 - x_8)$, and $x_{12} = \mathrm{abs}(x_6 - x_9)$, where the $\mathrm{abs}(x)$ function gives the absolute value of x, i.e., $|x|$. If the eBPF/XDP program cannot find the value $\boldsymbol{v_k}$ in the BPF map, it records $\boldsymbol{k}$ and $\boldsymbol{v_k}$ into the BPF map and then derives $(x_6, \ldots, x_{12})$ in the same way (steps k2 and k3 in Fig 2). (Note that the flow-related information recorded in the BPF map can also be used as the training data.) The eBPF/XDP program normalizes the matrix $\mathbf{X}^{(0)} = ((x_1, \ldots, x_{12})) \in \mathbb{R}^{1\times M_0}$ by using the D-dimensional normalization parameter $\boldsymbol{s}$ (step k4 in Fig. 2).

Given the normalized $\mathbf{X}^{(0)}$, the NN classifier performs the integer-arithmetic-only inference in the kernel space (step k5 in Fig. 2). For each layer l ($l = 1, \ldots, L$), it first quantizes the lth input matrix $\mathbf{X}^{(l)}$ and obtains the 8-bit quantized matrix $\mathbf{X}^{(l)}_q$ by the scale factor $\sigma^{(l)}$ according to Eq. (1) (step k5-1 in Fig. 2). It then performs the matrix multiplication of the 8-bit quantized input matrix $\mathbf{X}^{(l)}_q$ and the 8-bit quantized weight matrix $\mathbf{W}^{(l)}_q$ to obtain the 8-bit output matrix $\mathbf{Y}^{(l)}_q = \mathbf{X}^{(l)}_q \cdot \mathbf{W}^{(l)}_q$ (step k5-2 in Fig. 2). Next, it dequantizes $\mathbf{Y}^{(l)}_q$ by the scale factor $\sigma^{(l)}\boldsymbol{\sigma}^{(l)}_{\rm weight}$ according to Eq. (2) and obtains $\mathbf{X}^{(l+1)} = \mathbf{Y}^{(l)}$ by applying the activation function in the lth layer ($l = 1, \ldots, L-1$) (steps k5-3 and k5-4 in Fig. 2). After repeating these procedures for all layers l ($l = 1, \ldots, L$), it obtains $\mathbf{Y}^{(L)} = (y^{(L)}_1, \ldots, y^{(L)}_{M_L})$. Finally, it obtains the classification result $m^* = \underset{m\in\{0,\ldots,M_L-1\}}{\arg\max}\ y^{(L)}_{m+1}$ (step k6 in Fig. 2). In case of the binary classification ($M_L = 2$), the NN classifier classifies the received packet as the normal (resp. abnormal) packet if $m^* = 0$ (resp. $m^* = 1$). In case of the multi-class classification ($M_L = 7$), it outputs one of the seven label indices.

V. Evaluation

In this section, we evaluate the potential of eBPF/XDP packet processing empowered by the NN and DT, respectively, through the comparison with the conventional program running on the user space, in terms of classification performance, packet processing speed, and resource efficiency.

A. Classifier Performance

1) Evaluation Scenario: We first evaluate the performance of NN classifier in terms of inference performance and speed. As for this evaluation, we use the computation server with Intel Xeon Gold 5317 CPU (24 core), 64 GB memory, Intel Ethernet Converged Network Adapter 710X, and Ubuntu 20.04.1 LTS (kernel version 5.15). We divide the CIC-IDS2017 dataset into the testing, training, and validation dataset, where the testing dataset is assigned to 33% of the dataset and the remaining part is further divided into the training one (80%) and validation one (20%).

In what follows, we compare the NN classifier with DT one. To confirm the inference performance limit, we create the learned model by using the Python language. More specifically, we train the ML classifiers (i.e., NN and DT classifiers) by using the training data defined by the 64-bit floating-point number (*float64*), respectively. The learned NN (resp. DT) classifier is implemented by Pytorch [32] (resp. scikit-learn [33]). In the training phase, we use the Adam optimizer [34] with the initial learning rate of 10^{-3}. To evaluate the performance of ML classifiers running on the kernel space, we implement them using the restricted C language and train them according to the training data defined by the 64-bit integer fixed-point number (*int64*). We implement the DT classifier according to [7] and set the tree depth to be 5. Note that the DT classifier does not require the quantization process. As for the inference performance, we evaluate the accuracy, precision, recall, and f1-score of ML classifiers. In addition, we evaluate the inference speed by the inference time per packet, which is measured in the user space.

2) Binary Classification Performance: We first focus on the binary classification performance for attack packets with label index 1, as shown in Table II. We confirm that the Python-based implementation (i.e., NN (Pytorch) and DT (sklearn)

TABLE II: Binary classification performance.

Classifier	Accuracy	Precision	Recall	F1 score	Inference time [μs]
DT (sklearn)	0.976	0.957	0.946	0.952	52.88
NN (Pytorch)	0.981	0.962	0.962	0.962	46.09
DT (C)	0.977	0.944	0.964	0.954	0.86
NN (C)	0.975	0.928	0.979	0.952	6.38

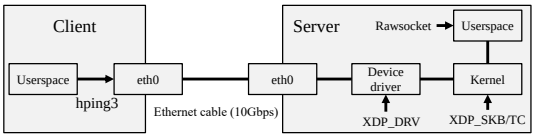


Fig. 3: Network environment.

classifiers) perform high inference performance in terms of accuracy, precision, recall, and f1-score. As for the C-based implementation (i.e., NN (C) and DT (C)), DT (C) keeps the inference performance even under the fixed-point number arithmetic. On the other hand, NN (C) shows the slight decrease of inference performance, due to both fixed-point number arithmetic and quantization. Focusing on the inference time, we confirm that NN (C) and DT (C) reduce the inference time to 13.8% and 1.7% compared with NN (Pytorch) and DT (sklearn), respectively. These results indicate that the DT classifier is suitable for a simple task such as detecting the abnormal packet, compared with the NN one. In Section V-B, we will further evaluate the impact of inference overhead on packet processing performance.

3) Multi-Class Classification Performance: Table III presents the multi-class classification performance. Focusing on the overall performance, i.e., accuracy and inference time, we observe that the multi-class classification results show a similar tendency to the binary classification ones. In particular, the C-based implementation (i.e., NN (C) and DT (C)) can reduce the inference time to 15.3% and 1.6% while suppressing the accuracy degradation by 3.7% and 2.5%, compared with the Python-based implementation (i.e., NN (Pytorch) and DT (sklearn)), respectively. This is because the same reason mentioned in Section V-A2.

Next, focusing on the inference performance of the individual labels, we confirm that the rich representation capabilities of NNs contribute to improving the inference performance for most of the attacks. More specifically, we observe the following characteristics: (1) Both NN and DT work well to classify DoS and Portscan as well as Normal, (2) NN can drastically improve the inference performance for Botnet and Brute force, compared with DT, and (3) DT is slightly effective to detect Infiltration compared with NN. However, we also observe that both NN and DT completely fail to detect Web attack. This may be caused by the insufficient feature selection and/or imbalanced data, which will be further examined in future work.

B. Packet Processing Performance

1) Evaluation Scenario: In this section, we evaluate the packet processing performance under the ML classification. Fig. 3 illustrates the network environments built on the two nodes, i.e., *client* and *server*, to evaluate the packet processing performance. The *client* is with Intel Core i9-12900 CPU (24 core), 64 GB memory, Marvell AQtion AQC113, and Ubuntu 20.04.1 LTS. On the other hand, the *server* is the same computation server used in the evaluation of Section V-A1. We restrict the number of CPUs of the *server* to one by the *chcpu* command. Recall that the eBPF and XDP programs are single-threaded programs. We send TCP packets from the *client* to the *server* for 100 s by using *hping3* [35]. We select hping3 for evaluations because it tends to be used for executing DoS and Portscan attacks. Since hping3 can control the packet sending interval I_s in the order of microseconds, we change I_s in the range of [0, 10] μs. Note that the setting of $I_s = 0$ results in the upper limit of packet sending rate, i.e., about 800,000 pps. We measure the packet sending (resp. receiving) rate R_{TX} (resp. R_{RX}), which is the number of outgoing (resp. incoming) packets from the network interface *eth0* of the *client* (resp. to the program of the *server*) per second. The packets arriving at the network interface *eth0* of the *server* are processed.

To evaluate the performance limit of each classifier, we compare three kinds of native XDP (XDP_DRV) methods, each of which applies the XDP program to the device driver on the network interface *eth0* of the *server*. XDP_DRV (Flow Extraction) appiles the XDP program that only aggregates the received packets into the corresponding flow. XDP_DRV (NN) and XDP_DRV (DT) apply the XDP program that has the NN and DT classifiers for binary classification, respectively, in addition to the flow extraction function. To confirm the performance limit of packet processing, we further prepare No Filter, which does not apply the XDP or eBPF program.

To evaluate the impact of packet processing mechanisms, we additionally prepare TC (NN), XDP_SKB (NN), and Rawsocket (NN) by applying the NN classifier to the ingress of TC, generic XDP, and rawsocket, respectively. TC (NN) (resp. XDP_SKB (NN)) attaches the eBPF (resp. XDP) program with the NN classifier to the ingress of TC (resp. generic XDP) on *eth0* of the *server*. On the other hand, Rawsocket (NN) applies the NN classifiers to the raw packet obtained from the socket in the user space. We implement each eBPF/XDP program by using the BPF Compiler Collection [36], which is one of the helper tools for eBPF. As for Rawsocket (NN), we also implement the rawsocket-based packet processing in the C language. We prepare two types of Rawsocket (NN): 1-core Rawsocket (NN) and 24-core Rawsocket (NN). 1-core (resp. 24-core) Rawsocket (NN) runs with 1-core CPU (resp. 24-core CPU.) Note that we consider 1-core/24-core Rawsocket (NN) under the single-threaded packet passing between kernel and user spaces. The performance comparison with the multi-threaded kernel bypassing methods will be future work.

2) Throughput Analysis: Fig. 4 illustrates the impact of R_{TX} on R_{RX} among three classifiers and packet processing mechanisms. Fig 5 illustrates the impact of R_{TX} on CPU utilization. In Fig 5, *usr* and *sys* mean the CPU utilization caused by the application overhead in the user space and that in the kernel space. *soft* means the CPU utilization caused by

TABLE III: Multi-class classification performance.

Scheme	DT (sklearn)			NN (Pytorch)			DT (C)			NN (C)		
Label	Precision	Recall	F1 score	Precision	Recall	F1 score	Precision	Recall	F1 score	Precision	Recall	F1 score
Botnet	1.0	0.070	0.130	0.930	0.879	0.904	1.0	0.066	0.124	0.929	0.868	0.898
Brute force	0.0	0.0	0.0	0.874	0.873	0.873	0.0	0.0	0.0	0.924	0.699	0.796
DoS	0.998	0.965	0.980	0.971	0.996	0.983	0.998	0.752	0.858	0.990	0.806	0.888
Infiltration	0.746	0.675	0.709	0.902	0.619	0.734	0.896	0.665	0.764	0.599	0.639	0.618
Normal	0.980	0.987	0.984	0.987	0.988	0.987	0.944	0.993	0.968	0.952	0.973	0.963
Portscan	0.882	0.942	0.911	0.871	0.951	0.910	0.882	0.985	0.930	0.857	0.973	0.912
Web attack	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Accuracy			0.969			0.974			0.944			0.937
Inference time [μs]			52.73			45.53			0.876			6.96

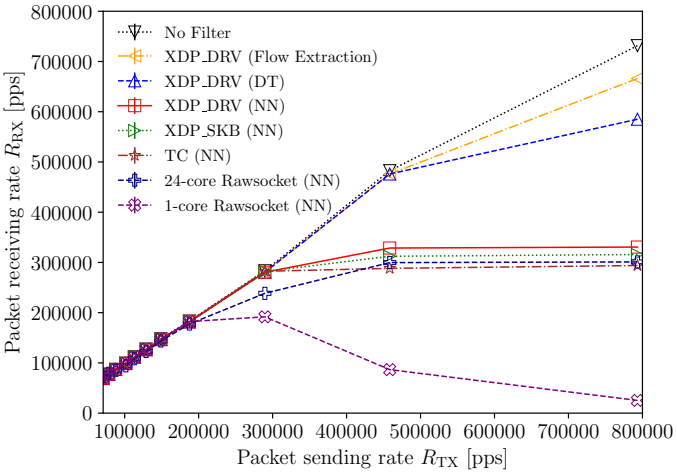
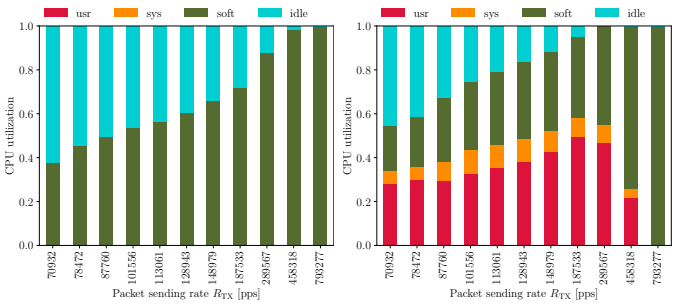


Fig. 4: Impact of R_{TX} on R_{RX}.



(a) XDP_DRV (NN). (b) 1-core Rawsocket (NN).

Fig. 5: Impact of R_{TX} on the CPU utilization.

the software interruption and *idle* is the unused CPU ratio. Note that *sys* and *soft* are overheads induced by the kernel. Recall that No Filter gives the performance limit.

We observe from Fig. 4 that all methods show similar tendency: 1) R_{RX} is competitive with the performance limit in the range of $R_{TX} = [0, \hat{R}_{TX}]$ and 2) R_{RX} eventually saturates or drops with further increase of R_{TX}. The value of $\hat{R}_{TX}$ is, however, different among the classifiers and packet processing mechanisms. We first focus on the impact of R_{TX} on R_{RX} among three schemes. XDP_DRV (Flow Extraction) has $\hat{R}_{TX} \approx$ 450,000 and shows 11.7% performance degradation from the performance limit when $\hat{R}_{TX} \approx$ 800,000. XDP_DRV (NN) (resp. XDP_DRV (DT)) has $\hat{R}_{TX} \approx$ 300,000 (resp. $\hat{R}_{TX} \approx$ 450,000) and shows 56.1% (resp. 22.6%) performance degradation from the performance limit when $\hat{R}_{TX} \approx$ 800,000. The performance degradation mainly comes from lack of CPU resource, as shown in Fig. 5a. In other words, the XDP packet processing empowered by NN (resp. DT) can work well under moderate packet sending rate.

Next, we focus on the impact of R_{TX} on R_{RX} among packet processing mechanisms for NN classifier. Focusing on the kernel-based methods, we confirm that XDP_SKB (NN) and TC (NN) show the same tendency as XDP_DRV (NN). Although XDP_DRV (NN), XDP_SKB (NN), and TC (NN) have the higher packet processing performance in this order, their differences are limited, indicating that the special XDP hardware is not necessarily required for the ML-empowered packet processing. On the other hand, 1-core Rawsocket (NN) running on the user space has $\hat{R}_{TX} \approx$ 200,000, which is smaller than those of kernel-based methods. It also shows performance degradation when $R_{TX} > \hat{R}_{TX}$, due to lack of CPU resource, as shown in Fig. 5b. Comparing Figs. 5a with 5b, we confirm that 1-core Rawsocket (NN) requires more CPU resource than the kernel-based methods, which is caused by the overhead of packet passing between kernel space and user space.

Since above-mentioned methods are single-threaded programs, we finally examine the contribution of multi-threaded NN computation in the user space. We observe from Fig. 4 that 24-core Rawsocket (NN) can achieve competitive packet processing performance with the kernel-based methods, at the expense of more CPU cores. Please note that the performance of 24-core Rawsocket (NN) is still limited, due to the bottleneck of single-threaded packet passing between kernel space and user space. The kernel bypassing methods will overcome this bottleneck but they will show full CPU utilization, regardless of R_{TX}.

VI. Conclusion

In this paper, we have examined the practicality of combination of kernel packet processing and machine learning for flow-based intrusion detection system (IDS). As for the kernel packet processing, we have focused on extended Berkeley Packet Filter (eBPF) and eXpress Data Path (XDP). Inspired by the existing decision tree (DT) empowered eBPF program,

we have newly proposed a lightweight neural network (NN) applicable to eBPF/XDP programs. More specifically, we have trained the NN with high accuracy and quantized the learned model to obtain the lightweight NN in the user space. We have further implemented the quantized fixed point NN, as so to realize the fast packet processing with integer-arithmetic-only inference in the kernel space.

Through the experiments for binary classification and multi-class one, we have demonstrated that the integer-arithmetic-only lightweight NN and DT classifiers drastically reduce the inference time to 13.8% and 1.7% while suppressing the inference performance degradation compared with the floating-point ones in the user space, with the help of the fixed-point arithmetic and the quantization. In addition, the NN classifier can improve the inference performance for most of the attacks thanks to its rich representation capabilities in case of the multi-class classification. Furthermore, we have unveiled the eBPF/XDP program with the NN (resp. DT) classifier can process received packets in a real-time manner under a certain transmission rate, i.e., 300,000 pps (resp. 450,000 pps).

In future work, we plan to reconsider the way to improve the inference performance (e.g., feature selection) under the eBPF constraints and investigate the generalization capabilities using other datasets. Quantitative comparison between kernel-based methods and kernel-bypassing methods (e.g., Data Plane Development Kit (DPDK)) is also one of the future directions.

REFERENCES

[1] B. Gregg, *BPF Performance Tools: Linux System and Application Observability*, 1st ed. Hoboken: Pearson Education, Inc, 2019.

[2] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications," *ACM Computing Surveys*, vol. 53, no. 1, pp. 1–36, Jan. 2021.

[3] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel," in *Proc. of the ACM CoNEXT*. ACM, Dec. 2018, pp. 54–66.

[4] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A Comprehensive Survey on Machine Learning for Networking: Evolution, Applications and Research Opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, p. 16, Dec. 2018.

[5] P. Mishra, V. Varadharajan, U. Tupakula, and E. S. Pilli, "A Detailed Investigation and Analysis of Using Machine Learning Techniques for Intrusion Detection," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 686–728, 2019.

[6] L. Chen, X. Kuang, A. Xu, S. Suo, and Y. Yang, "A Novel Network Intrusion Detection System Based on CNN," in *Proc. of CBD*, Feb. 2020, pp. 243–247.

[7] M. Bachl, J. Fabini, and T. Zseby, "A Flow-Based IDS Using Machine Learning in eBPF," Mar. 2022.

[8] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating Complex Network Services with eBPF: Experience and Lessons Learned," in *Proc. of IEEE HPSR*. Bucharest, Romania: IEEE, Jun. 2018, pp. 1–8.

[9] L. Foundation, " Data Plane Development Kit," https://www.dpdk.org/, 2018, Accessed 18 Dec. 2022.

[10] Y. Choe, J.-S. Shin, S. Lee, and J. Kim, "eBPF/XDP Based Network Traffic Visualization and DoS Mitigation for Intelligent Service Protection," in *Proc. of Advances in Internet, Data and Web Technologies*. Springer International Publishing, 2020, pp. 458–468.

[11] N. Van Tu, J.-H. Yoo, and J. Won-Ki Hong, "Accelerating Virtual Network Functions With Fast-Slow Path Architecture Using eXpress Data Path," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1474–1486, Sep. 2020.

[12] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu, "A Framework for eBPF-Based Network Functions in an Era of Microservices," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 133–151, Mar. 2021.

[13] M. Xhonneux, F. Duchene, and O. Bonaventure, "Leveraging eBPF for Programmable Network Functions with IPv6 Segment Routing," in *Proc. of the ACM CoNEXT*. ACM, Dec. 2018, pp. 67–72.

[14] A. Mishra, R. Shrivastava, and P. Yadav, "A Modified Cascaded Feed Froward Neural Network Distributed Denial of Service Attack Detection using Improved Regression based Machine Leaning Approach," in *Proc. of ICOEI*, Apr. 2022, pp. 1292–1299.

[15] H. Sharaf, I. Ahmad, and T. Dimitriou, "Extended Berkeley Packet Filter: An Application Perspective," *IEEE Access*, vol. 10, pp. 126 370–126 393, 2022.

[16] N. Moustafa and J. Slay, "UNSW-NB15: A Comprehensive Data Set for Network Intrusion Detection Systems (UNSW-NB15 Network Data Set)," in *Proc. of MilCIS*, Jan. 2015, pp. 1–6.

[17] I. Sharafaldin, A. Habibi Lashkari, and A. A. Ghorbani, "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization:," in *Proc. of the ICISSP*, 2018, pp. 108–116.

[18] P. Salva-Garcia, R. Ricart-Sanchez, E. Chirivella-Perez, Q. Wang, and J. M. Alcaraz-Calero, "XDP-Based SmartNIC Hardware Performance Acceleration for Next-Generation Networks," *Journal of Network and Systems Management*, vol. 30, no. 4, p. 75, Sep. 2022.

[19] M. Attig and G. Brebner, "400 Gb/s Programmable Packet Parsing on a Single FPGA," in *Proc. of ACM/IEEE ANCS*, Oct. 2011, pp. 12–23.

[20] D.-M. Ngo, C. Pham-Quoc, and T. N. Thinh, "Heterogeneous Hardware-based Network Intrusion Detection System with Multiple Approaches for SDN," *Mobile Networks and Applications*, vol. 25, no. 3, pp. 1178–1192, Jun. 2020.

[21] L. Le Jeune, T. Goedemé, and N. Mentens, "Towards Real-Time Deep Learning-Based Network Intrusion Detection on FPGA," in *Proc. of ACNS*. Springer International Publishing, 2021, pp. 133–150.

[22] I. Ben-Yair, P. Rogovoy, and N. Zaidenberg, "AI & eBPF Based Performance Anomaly Detection System," in *Proc. of the ACM SYSTOR*. ACM, May 2019, pp. 180–180.

[23] H. Benmaghnia, M. Martel, and Y. Seladji, "Code Generation for Neural Networks Based on Fixed-Point Arithmetic," *ACM Transactions on Embedded Computing Systems*, pp. 1–28, Sep. 2022.

[24] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the Speed of Neural Networks on CPUs," in *Proc. of NIPS*, 2011.

[25] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and Quantization for Deep Neural Network Acceleration: A Survey," *Neurocomputing*, vol. 461, pp. 370–403, Oct. 2021.

[26] Linux, "tc(8) – Linux manual pages," https://man7.org/linux/man-pages/man8/tc.8.html, 2023, Accessed 2 May 2023.

[27] D. Miller, "Generic XDP," https://lwn.net/Articles/720072/, 2017, Accessed 18 Dec. 2022.

[28] R. Krishnamoorthi, "Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper," Jun. 2018.

[29] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation," Apr. 2020.

[30] S. Kullback and R. A. Leibler, "On Information and Sufficiency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-Propagating Errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986.

[32] Pytorch, "Pytorch," https://pytorch.org/, 2023, Accessed 2 May 2023.

[33] Scikit-learn, "Scikit-learn," https://scikit-learn.org/, 2023, Accessed 2 May 2023.

[34] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," Jan. 2017.

[35] antirez, "antirez/hping," https://github.com/antirez/hping, 2023, Accessed 2 May 2023.

[36] iovisor, "BPF Compiler Collection (BCC)," https://github.com/iovisor/bcc, 2023, Accessed 2 May 2023.