uBPF-based Network Intrusion Detection and Prevention Systems Empowered by Machine Learning Algorithms*

Takanori Hara[a,*], Shoji Kasahara[a]

[a]*Graduate School of Science and Technology, Nara Institute of Science and Technology, 8916-5 Takayama-cho, Ikoma, 630-0192, Nara, Japan*

Abstract

Network Intrusion Detection and Prevention Systems (NIDPSs) play a crucial role in identifying and blocking malicious traffic. In-kernel NIDPSs leveraging extended Berkeley Packet Filter (eBPF) have achieved high detection performance and throughput by operating directly in kernel space. However, eBPF-based NIDPSs face portability challenges across diverse runtime environments, limiting their deployment on kernel-bypassing frameworks such as AF_XDP and the Data Plane Development Kit (DPDK). In this paper, we propose a userspace BPF (uBPF)-based NIDPS empowered by machine learning (ML) algorithms and integrate it with kernel-bypassing frameworks to achieve portability across diverse runtime environments while accurately and efficiently detecting and preventing Distributed Denial of Service (DDoS) attacks. We evaluate the proposed NIDPSs from both DDoS detection performance and packet processing performance perspectives. Evaluation results demonstrate that the proposed NIDPS achieves high classification performance for DDoS detection, while enabling the same uBPF-based detection logic to operate across different packet processing frameworks. Moreover, the uBPF-based NIDPSs running on the DPDK data plane achieve packet processing rates 3.38–8.60× higher than the evaluated in-kernel NIDPS implementations.

Keywords:
extended Berkeley Packet Filter (eBPF), userspace BPF (uBPF), network intrusion detection and prevention systems (NIDPSs), eXpress Data Path (XDP), AF_XDP, Data Plane Development Kit (DPDK)

1. Introduction

Distributed Denial of Service (DDoS) attacks are a significant threat to network security, causing service disruptions and financial losses [2]. To mitigate these attacks, *Network Intrusion Detection and Prevention Systems* (NIDPSs) play an essential role in cybersecurity infrastructures by monitoring and analyzing network traffic to detect and prevent malicious activities and policy violations. In particular, *Machine Learning* (ML) techniques have shown promise for accurate DDoS detection by learning network traffic patterns and identifying anomalies [3].

Despite their effectiveness, ML-based NIDPSs still face two major challenges: packet processing performance and deployment portability. First, conventional ML-based NIDPSs typically operate in user space, which incurs performance bottlenecks due to frequent context switching between kernel and user spaces. This overhead becomes a severe performance bottleneck under high network loads, hindering real-time detection and prevention of DDoS attacks. Moreover, ML-based NIDPSs often lack program safety check mechanisms, which can lead to unreliable behavior and unexpected crashes. Second, modern networks span diverse environments, including cloud infrastructures, edge computing platforms, and Internet of Things (IoT) devices. These heterogeneous environments make it difficult to deploy ML-based NIDPSs across different platforms and architectures because complex ML functionality often depends on specific runtime environments. This lack of portability increases maintenance costs, complicates large-scale deployments, and limits the reuse of detection logic across platforms.

To address these challenges, recent studies have explored the use of *extended Berkeley Packet Filter* (eBPF) and *eXpress Data Path* (XDP) for designing NIDPSs that operate directly in kernel space, reducing context-switching overhead and improving throughput [4, 5, 6, 7, 8, 9]. In-kernel NIDPSs also offer high reliability thanks to the eBPF verifier, which ensures that eBPF programs are safely executed in kernel space. Moreover, in-kernel NIDPSs partially address the portability issue through *BPF Compile Once-Run Everywhere* (CO-RE) [10], which allows eBPF programs to run across different kernel versions without recompilation. However, existing in-kernel NIDPSs remain incompatible with kernel-bypassing frameworks (i.e., *Data Plane Development Kit* (DPDK) [11] and *AF_XDP* [12]), both of which are widely used in high-performance network applications.

Userspace BPF (uBPF) has the potential to bridge this gap by enabling eBPF programs to run in user space while pre-

*This work was supported in part by the Japan Society for the Promotion of Science (JSPS) KAKENHI (B) under Grant 26K02903 and 24K02931. This paper is an extended version of the conference paper [1], which was published in the proceedings of the First Workshop on AI Native Distributed Intelligence for 6G Networks (6G AI-RAN 2026) organized in conjunction with IEEE International Conference on Computer Communications (INFOCOM 2026).

*Corresponding author.

serving the safety and portability benefits of eBPF ecosystems [13, 14, 15]. `bpftime` [14], which is a sophisticated uBPF runtime, has demonstrated the potential of uBPF for packet processing performance comparable to AF_XDP/DPDK-based applications [15]. Therefore, uBPF enables a unified ML-based NIDPS implementation that can be reused across both in-kernel and kernel-bypassing packet processing frameworks. Nevertheless, the practicality of uBPF-based packet processing for ML-based NIDPSs has not yet been fully explored.

In this paper, we propose an ML-based NIDPS that leverages uBPF to achieve high detection performance and high packet processing capability across diverse runtime environments[1]. Through extensive evaluation, we reveal both the potential and limitations of uBPF-based NIDPSs assisted by lightweight ML techniques, in comparison with existing in-kernel NIDPS approaches [5, 6, 7, 8, 9].

The main contributions of this manuscript are as follows:

1. To the best of our knowledge, this is the first work to explore the practicality of uBPF over AF_XDP/DPDK for ML-based NIDPSs. We reveal its potential and limitations in terms of DDoS detection performance and packet processing performance.
2. We demonstrate that the proposed NIDPSs achieve DDoS detection performance competitive with that of existing floating-point counterparts while mitigating the calculation errors introduced by fixed-point representations. We also show that the uBPF-based NIDPSs over AF_XDP/DPDK achieve 2.07–2.31×/3.38–8.60× higher packet processing rates than in-kernel NIDPSs.
3. The proposed NIDPSs operate in distinct runtime environments such as XDP, AF_XDP, and DPDK while reusing the same classifier-specific logic and BPF map layout, although runtime-dependent packet I/O glue code is required.

The rest of this manuscript is organized as follows. Section 2 gives related work. Section 3 presents preliminaries. Section 4 describes the proposed NIDPS. Section 5 presents evaluation results. Section 6 discusses limitations of the proposed NIDPS. Finally, Section 7 concludes this paper.

2. Related Work

Table 1 summarizes the related work on NIDPSs based on eBPF/uBPF. To reduce context-switching overhead between kernel and user spaces, several studies have proposed in-kernel IDSs that operate directly in kernel space using eBPF/XDP [4, 5, 16, 7, 8, 6, 9]. In particular, ML algorithms have been integrated into eBPF-based NIDPSs to accurately detect network attacks in real time [5, 6, 7, 8, 9]. These studies have demonstrated that in-kernel NIDPSs can achieve high packet processing performance by reducing context-switching overhead while maintaining detection performance, compared to traditional userspace NIDPSs. Moreover, eBPF-based NIDPSs leveraging neural networks have demonstrated the feasibility of real-time anomaly detection with high accuracy [6, 7, 8]. In addition, eBPF programs provide reliability through verifier-based safety checks and portability across different kernel versions through BPF CO-RE [10]. However, these eBPF-based NIDPSs are designed for in-kernel execution and cannot be directly deployed on kernel-bypassing frameworks, which hinders their portability across diverse runtime environments.

To address the portability issue, uBPF has been developed as a lightweight eBPF virtual machine running in user space [14]. Recent work has further integrated uBPF with kernel-bypassing frameworks (i.e., AF_XDP and DPDK) to execute eBPF programs on AF_XDP/DPDK data planes [15]. This approach demonstrated the practicality of uBPF over AF_XDP/DPDK for basic packet processing applications (e.g., firewall and load balancer). However, its applicability to ML-based NIDPSs has not yet been sufficiently explored. In this paper, we investigate the potential and limitations of uBPF over AF_XDP/DPDK for ML-based NIDPSs, aiming to achieve both packet processing performance and portability across diverse runtime environments.

DDoS attack detection techniques leveraging ML have been extensively studied in the field of network security [3]. In addition, ML-based NIDPSs using kernel-bypassing frameworks have been proposed to achieve high detection performance and throughput [17, 18]. These frameworks can reduce network I/O overhead by executing packet processing in userspace data planes without traversing the conventional kernel network stack. However, these userspace NIDPSs generally lack verifier-based safety check mechanisms, making them more vulnerable to unexpected crashes or unsafe program behavior. Unlike prior ML-based NIDPSs that rely on userspace network stacks without safety mechanisms, our approach executes ML classifiers within uBPF programs integrated with AF_XDP/DPDK, combining kernel-bypassing packet processing with verified safety and portability benefits of the eBPF ecosystem.

The conference version of this work developed a uBPF-based NIDPS for DDoS detection using three ML algorithms (i.e., Decision Tree (DT), Random Forest (RF), and Multi-Layer Perceptron (MLP)) and evaluated its detection performance using the CIC-DDoS2019 [19] dataset as well as the packet processing performance of uBPF over AF_XDP and DPDK [1]. This paper extends the conference version in three major aspects: (1) We evaluate the detection performance of the proposed NIDPS using both the CIC-IDS2017 [20] and CIC-DDoS2019 [19] datasets; (2) We provide a more detailed analysis of the potential and limitations of uBPF over AF_XDP/DPDK for ML-based packet processing by considering detection performance, packet processing performance, CPU utilization, packet processing latency, packet length, and the number of active flows; (3) In addition to DT, RF, and MLP classifiers, we add eXtreme Gradient Boosting (XGB) and Support Vector Machine (SVM) to broaden the classifier evaluation. This extension enables an analysis of how different model structures affect detection performance under fixed-point arithmetic, eBPF instruction con-

[1]The source code is available at `https://github.com/oakeshott/ubpf-ml-ids`.

Table 1: A comparison of eBPF/uBPF-based NIDPSs and related kernel-bypassing packet processing frameworks.

Research	Detection techniques	Runtime environments	Contribution
[4]	Threshold-based filtering	eBPF/XDP	This work applied eBPF/XDP to network traffic visualization and threshold-based DoS mitigation.
[5]	DT	eBPF	This work implemented a flow-based IDS using DT entirely in eBPF and improved throughput compared to userspace IDSs.
[16]	Statistical framework with dynamic thresholding	eBPF/XDP	This work developed a detection mechanism of LDoS attacks at the kernel-level using lightweight features and dynamic thresholding.
[7, 8]	MLP	eBPF/XDP	These studies implemented real-time in-kernel ML-based packet processing using lightweight neural networks.
[6]	Autoencoders	eBPF	This work proposed an unsupervised in-kernel IDS with autoencoders and quantization-aware training.
[9]	DT, RF, and SVM	eBPF/XDP	This work integrated multiple ML classifiers with eBPF for real-time DoS/DDoS detection.
[17]	DT and MLP	eBPF/XDP and AF_XDP	This work compared eBPF/XDP/AF_XDP-based IDSs leveraging ML algorithms in terms of packet processing performance, resource efficiency, and detection performance.
[18]	Statistical framework	DPDK	This work proposed a DPDK-based DDoS detection framework using a lightweight statistical anomaly metric.
[15]	N/A	uBPF over AF_XDP/DPDK	This work executed unmodified, verified XDP programs in userspace with AF_XDP/DPDK.
[1]	DT, RF, and MLP	uBPF over AF_XDP/DPDK	This work evaluated the feasibility of uBPF-based DDoS detection over AF_XDP/DPDK using three lightweight ML classifiers.
This work	DT, RF, XGB, SVM, MLP	uBPF over AF_XDP/DPDK	This work is an extended version of [1], providing a comprehensive evaluation of uBPF-based NIDPSs leveraging multiple ML algorithms in terms of detection performance, packet processing performance, and impacts of packet length and active flows.

straints, and packet processing rate. In particular, the evaluation reveals the tradeoffs between the simple but fixed-point-sensitive SVM, the instruction-intensive XGB, and the substantially more computationally demanding MLP classifier.

3. Preliminaries

3.1. eBPF/XDP

eBPF is an abstract virtual machine embedded in the Linux kernel, enabling the safe and efficient execution of user-defined programs in kernel space [21]. XDP is a high-performance packet processing framework built on top of eBPF, allowing eBPF programs to run at the earliest point in the kernel network stack [22].

To ensure kernel safety and stability, eBPF programs must adhere to several constraints imposed by the eBPF verifier. The eBPF verifier performs static analysis on eBPF programs before they are loaded into the kernel, checking for potential issues that could compromise system integrity [21]. Table 2 summarizes the constraints enforced by the eBPF verifier. The eBPF constraints include (1) bounded loops, (2) limited stack size, (3) no floating-point arithmetic, and (4) restricted instruction and jump counts. These constraints make ML implementations within eBPF programs difficult.

To address them, fixed-point representations are used to perform arithmetic operations without floating-point support. BPF maps, key-value stores shared between kernel and user spaces, enable large data structures (e.g., trained parameters) to be stored. Tail calls are a mechanism that allows an eBPF program to jump to another eBPF program without returning to the caller. When a tail call is executed, the kernel resets the program context, including the instruction and jump counts as well as the consumed stack space. In [6, 7, 8], the tail call technique is used to implement layer-wise operations of neural networks within eBPF programs, enabling them to overcome the eBPF constraints.

3.2. uBPF

uBPF is a lightweight userspace implementation of the eBPF virtual machine [13], designed to execute eBPF programs safely in user space. `bpftime` is one of the sophisticated libraries for uBPF [14], enabling the execution of eBPF programs in user space. In [15], the authors integrate eBPF/XDP programs with kernel-bypassing frameworks (i.e., AF_XDP and DPDK) by leveraging bpftime. bpftime provides BPF maps implemented as shared memory, enabling efficient data sharing between userspace applications and uBPF programs. uBPF programs are designed to satisfy the eBPF constraints considered in this work. Moreover, bpftime currently supports only userspace-to-kernel tail calls[2], which prevents neural-network-based classifiers from being executed entirely within uBPF programs.

3.3. BPF maps

BPF maps are key-value stores that enable data sharing between kernel and user spaces in eBPF programs [21]. They are used to store various types of data, such as configuration parameters, counters, and trained ML model parameters. BPF maps can be accessed and modified by both eBPF programs and user-space applications, facilitating communication and data exchange between the two spaces. In uBPF, BPF maps are implemented as shared memory, allowing for efficient data sharing

[2] `https://github.com/eunomia-bpf/bpftime/pull/131`

Table 2: eBPF verifier constraints considered in this work and their implementation alternatives.

Constraints	Traditional programs	eBPF/uBPF programs	Alternatives
Unbounded loops	Supported	Not supported	Bounded loops
Floating-point representations	Supported	Not supported	Fixed-point representations
Non-static global variables	Supported	Not supported	BPF maps
Array bounds checking	Not required	Required	N/A
Maximum instruction count	Not limited	1 million	Tail calls
Maximum jump instruction count	Not limited	8,192	Tail calls
Maximum stack space	8,192 KB (default)	512 B	BPF maps and tail calls

between userspace applications and uBPF programs [14, 15]. The overhead of accessing BPF maps has been analyzed in [23].

3.4. Packet Processing Mechanisms

In this section, we compare three packet processing mechanisms: native XDP, uBPF over AF_XDP, and uBPF over DPDK. Fig. 1 illustrates a comparison of packet processing mechanisms and their relation to ML tasks. In Fig. 1, the locations of training and classification differ depending on the packet processing mechanisms. For all packet processing mechanisms, ML classifiers are trained in user space because intricate training tasks cannot be executed in the kernel due to the eBPF constraints. In contrast, the classification tasks are executed within eBPF/uBPF programs to minimize context-switching overhead between kernel and user spaces and to avoid unexpected ML-based NIDPS crashes. BPF maps are used to store the trained classifier parameters, which are accessed by the eBPF/uBPF programs during classification. In the case of uBPF, BPF maps are implemented as a shared memory, enabling efficient data sharing between userspace applications and uBPF.

In Fig. 1a, native XDP processes incoming packets at the driver level in the kernel space. In Fig. 1b, uBPF is integrated with AF_XDP, a partially kernel-bypassing framework that allows XDP programs to redirect packets to userspace applications, bypassing the traditional network stack. In Fig. 1c, uBPF is integrated with DPDK, a fully kernel-bypassing framework that offers high-performance packet processing capabilities in user space by directly accessing Network Interface Cards (NICs). These bypassing approaches reduce network I/O latency by leveraging direct userspace packet processing, albeit at the cost of increased CPU utilization and limited kernel functionality. In contrast, integrating uBPF with AF_XDP/DPDK restores eBPF-like programmability to their data planes. This integration allows eBPF programs to be hooked into the AF_XDP/DPDK data planes as uBPF, enabling user-defined packet processing [15].

Table 3 presents the main characteristics of packet processing mechanisms. Generic XDP is a flexible packet processing mechanism that allows eBPF programs to be attached to hook points in the kernel, while native XDP processes packets at the driver level. Native XDP offers higher performance than generic XDP by processing packets at the earliest point in the kernel's networking stack, but it is limited to specific NICs and drivers. Both generic and native XDP execute eBPF programs in an interrupt mode using *New Application Programming Interface* (NAPI), which is a mechanism that allows the kernel to handle incoming packets in batches, reducing the overhead of frequent interrupts.

uBPF over AF_XDP can operate in three modes: standard, syscall, and poll, depending on the combinations of driver execution mode and userspace checking mode. In the standard mode, AF_XDP operates in interrupt mode using NAPI while the userspace application uses a busy loop to check for incoming packets in the RX ring of AF_XDP socket. In the syscall mode, hardware interrupts are disabled and the userspace program checks the presence of a newly incoming packet in the RX ring by periodically executing a system call in a busy loop. AF_XDP syscall can reduce context-switching and cache-coherency overhead while the busy-loop based AF_XDP (i.e., AF_XDP syscall and standard) would consume more CPU resources as in DPDK. In the poll mode, AF_XDP operates in interrupt mode using NAPI, while the userspace application uses the `poll()` system call to wait for incoming packets. This mode allows the userspace application to sleep until a packet arrives, reducing CPU utilization compared to the busy-loop approaches.

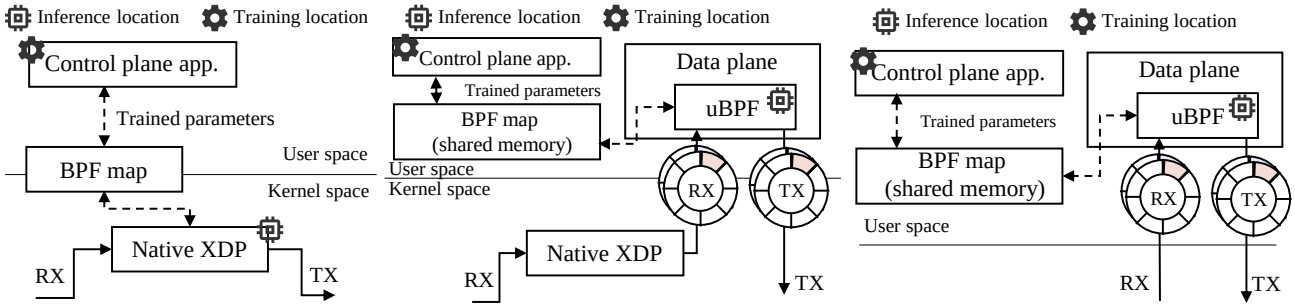
3.5. Fixed-Point Arithmetic

Floating-point arithmetic is not supported in eBPF programs due to the eBPF constraints. Alternatively, fixed-point arithmetic [24] is used to perform calculations involving real numbers within eBPF/uBPF programs [5, 6, 7, 8]. Fixed-point numbers are represented as integer values in which lower n bits are treated as the fractional part. A floating-point number x_{float} is transformed into a fixed-point number x_{fixed} with the lower n bits as fractional part using the equation:

$$x_{\text{fixed}} = \text{round}(x_{\text{float}} \times 2^n),$$

where $\text{round}(x)$ is a function to round a value x.

Fixed-point arithmetic introduces errors because of the limited precision of fixed-point representations. The error can be reduced by increasing the fractional bit-width n, but it also increases the risk of overflow and the computational complexity [25]. To balance the trade-off between precision and overflow risk, we set $n = 16$.



(a) Native XDP. (b) uBPF over AF_XDP. (c) uBPF over DPDK.

Figure 1: Comparison of packet processing mechanisms.

Table 3: Main characteristics of ML-based packet processing mechanisms.

Mechanism	Training location	Inference location	Runtime	Driver execution mode	Userspace checking mode
Generic XDP	User space	Kernel space	eBPF programs	Interrupt (NAPI)	N/A
Native XDP	User space	Kernel space	eBPF programs	Interrupt (NAPI)	N/A
uBPF over AF_XDP standard	User space	User space	uBPF programs	Interrupt (NAPI)	Busy loop
uBPF over AF_XDP syscall	User space	User space	uBPF programs	Busy polling	Busy loop
uBPF over AF_XDP poll	User space	User space	uBPF programs	Interrupt (NAPI)	Blocking mode
uBPF over DPDK	User space	User space	uBPF programs	Busy polling	Busy loop
AF_XDP standard	User space	User space	AF_XDP data plane (Native C)	Interrupt (NAPI)	Busy loop
AF_XDP syscall	User space	User space	AF_XDP data plane (Native C)	Busy polling	Busy loop
AF_XDP poll	User space	User space	AF_XDP data plane (Native C)	Interrupt (NAPI)	Blocking mode
DPDK	User space	User space	DPDK data plane (Native C)	Busy polling	Busy loop

Table 4: All extracted features.

Category	Features
5-tuple flow	Source port number, destination port number, protocol number, and total number of packets.
Packet length	Mean packet length, maximum packet length, minimum packet length, total packet length, variance of packet lengths, and header length.
Packet inter-arrival time (IAT)	Mean IAT, maximum IAT, minimum IAT, total IAT, and variance of IATs.
TCP	Initial TCP window size, SYN flag count, ACK flag count, FIN flag count, RST flag count, and total flag count.

4. eBPF/uBPF-based Network Intrusion Detection and Prevention

4.1. Feature Extraction

For training and testing the NIDPSs, we use CIC-IDS2017 [20] and CIC-DDoS2019 [19] datasets, which are widely used for evaluating DDoS detection performance. These datasets provide flow-level labeled network traffic statistics captured in a realistic network environment, including various types of DDoS attacks as well as benign traffic. Details of the datasets can be found in [19, 20].

When deploying an eBPF/uBPF-based NIDPS, we cannot use these datasets directly due to the following limitations: (1) XDP handles only incoming packets; (2) online and lightweight feature extraction is required; and (3) eBPF programs cannot process a large number of features due to stack-size and jump-count constraints. These limitations are not inherent to these datasets.

To address these limitations, we apply the following considerations for feature extraction and selection: (1) we extract 5-tuple flow-based features using only statistics derived from incoming packets; (2) although the datasets include features such as the offline computation of standard deviation, the calculation of standard deviation itself requires a square root operation, which is difficult to implement efficiently in eBPF programs. To mitigate this difficulty, we use variance instead of standard deviation to reduce computational complexity and apply Welford's online algorithm [26] to compute the variance in an online manner; and (3) we use the feature importance [27], one of the feature selection techniques, to reduce the number of features so that the selected features can be efficiently computed within the constraints of eBPF programs while maintaining the detection performance.

The extracted features are categorized into four groups: 5-tuple flow, packet length, packet inter-arrival time (IAT), and TCP features. Table 4 lists all extracted features.

4.2. Feature Selection

To select a subset of features that can be efficiently computed within eBPF programs while maintaining high detection perfor-

Table 5: Selected features.

Symbol	Feature	Symbol	Feature
x_1	Mean packet length	x_9	Total flag count
x_2	Source port number	x_{10}	Total IAT
x_3	Destination port number	x_{11}	Total number of packets
x_4	Maximum packet length	x_{12}	SYN flag count
x_5	Total packet length	x_{13}	Protocol number
x_6	Minimum packet length	x_{14}	Maximum IAT
x_7	Initial TCP window size	x_{15}	Header length
x_8	FIN flag count	x_{16}	Variance of packet lengths

mance, we apply a feature importance ranking technique [27] to the extracted features. For each dataset, we first compute feature rankings using four importance measures: Pearson correlation, mutual information, random forest variable importance, and permutation importance based on a random forest classifier. We then average the ranks obtained from the two datasets and select the top sixteen features $\boldsymbol{x} = (x_1, \ldots, x_{16})$ with the best average ranks. The selected sixteen features are listed in Table 5.

4.3. Model Training in User Space

To ensure compatibility with eBPF and uBPF, we design lightweight ML classifiers that can be efficiently executed under the eBPF constraints. In this paper, we adopt five classifiers: Decision Tree (DT), Random Forest (RF), eXpress Gradient Boosting (XGB), Support Vector Machine (SVM), and Multi-Layer Perceptron (MLP), which have been used in the existing in-kernel NIDPSs [5, 7, 8, 9].

All training tasks are performed in user space in advance. For the DT, RF, and XGB classifiers, we train the classifiers using the training data represented in the fixed-point format with the lower $n = 16$ bits as fractional part. For the SVM and MLP classifiers, we transform the input features into a standard normal distribution with a mean of zero and a standard deviation of one using a standard scaler.

We also train the SVM and MLP classifiers using the floating-point training data. After all training tasks, the resulting floating-point trained parameters are converted into fixed-point representation with the lower $n = 16$ bits as fractional part. These trained parameters and standardization parameters, represented in fixed-point numbers, are stored in BPF maps to be accessed by eBPF/uBPF programs during classification.

4.4. Anomaly Detection and Prevention in eBPF/uBPF Programs

During online packet processing, the eBPF/uBPF program is invoked for each incoming packet and updates the corresponding flow statistics. Anomaly detection and prevention are then performed within the eBPF/uBPF programs using the current flow statistics. Fig. 2 illustrates the classification workflow. Algorithm 1 presents the pseudocode for online packet processing in the eBPF/uBPF-based NIDPS.

Upon the arrival of a new packet p at a hook point, the corresponding eBPF/uBPF program is invoked. The program first

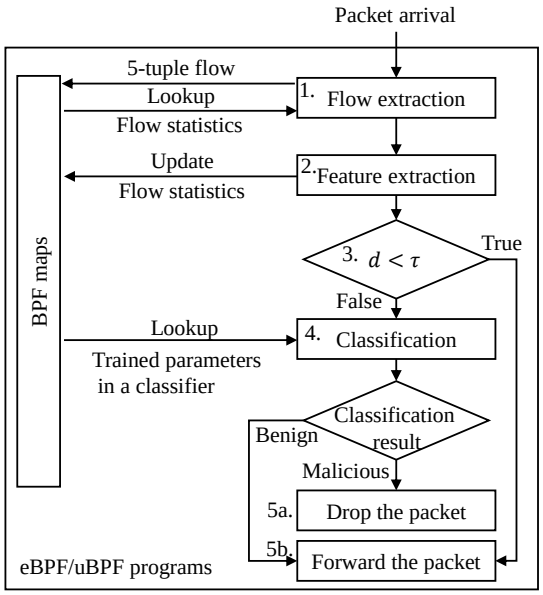


Figure 2: Classification workflow in eBPF/uBPF programs.

obtains current timestamp t using the BPF_KTIME_GET_NS() function, which outputs a timestamp with nanosecond precision (line 1 in Algorithm 1). It then inspects the packet header to extract a 5-tuple flow identifier $k = (sip_p, dip_p, sport_p, dport_p, proto_p)$, consisting of the source and destination IP addresses sip_p and dip_p, source and destination ports $sport_p$ and $dport_p$, and protocol type $proto_p$ (step 1 in Fig. 2 and line 2 in Algorithm 1). The program also retrieves the packet length ℓ_p and TCP flags $\boldsymbol{f}_p$ using the PacketLength(p) and TCPFlags(p) functions, respectively, which return the length and TCP flags of the packet p (lines 3 and 4 in Algorithm 1). If $proto_p$ is not TCP, the TCP flags $\boldsymbol{f}_p$ are set to zeros.

Using this flow identifier k as a key, the program retrieves the corresponding flow statistics s from the BPF map M_F, which maintains the state of each active flow (line 5 in Algorithm 1). The state (i.e., flow statistics) s contains the number of observed packets, the timestamps of the first and most recent packets, and the online statistics required for feature extraction. For packet lengths, it maintains the sum, minimum, maximum, and variance-related state. For IATs, it maintains the sum, minimum, maximum, and variance-related state. It also maintains counters for each TCP flag and their total count. The mean packet length and mean IAT are derived from the corresponding sums and observation counts, while the current flow duration d is derived from the first and most recent timestamps. The packet-header values used as features, such as the protocol number, header length, and initial TCP window size, are obtained from the packet header and retained as required for feature extraction.

If this retrieval fails, indicating that p is the first packet of the flow, the program initializes the flow statistics s and stores them in the BPF map M_F using the InitializeFlowState(p, t) function (lines 6–9 in Algorithm 1). In the InitializeFlowState(p, t) function, all statis-

tics are initialized with zeros and $s.first_timestamp$ and $s.last_timestamp$ are initialized to t (lines 7 and 8 in Algorithm 1).

Otherwise, IAT iat_p is computed as $iat_p = t - s.last_timestamp$, the IAT-related statistics are updated using the UpdateIATStatistics(s, iat_p) function, after which $s.last_timestamp$ is set to t, where the UpdateIATStatistics(s, iat_p) function updates the maximum, minimum, variance, sum of IATs (lines 11–13 in Algorithm 1). Similarly, the packet-length statistics and TCP-flag counters are updated using the UpdatePacketLengthStatistics(s, ℓ_p) and UpdateTCPFlagCounters$(s, \boldsymbol{f}_p)$ functions, respectively, where the UpdatePacketLengthStatistics(s, ℓ_p) function updates the maximum, minimum, variance, sum of packet lengths while the UpdateTCPFlagCounters$(s, \boldsymbol{f}_p)$ function updates the TCP flag counts $\boldsymbol{f}$ of the statistics s as $\boldsymbol{f} = \boldsymbol{f} + \boldsymbol{f}_p$ and computes the total TCP flag count $f_{\text{total}} = \sum_{i=1}^{|f|} f_i$ (lines 14 and 15 in Algorithm 1). Both the UpdateIATStatistics(s, iat_p) and UpdatePacketLengthStatistics(s, ℓ_p) functions employ Welford's online algorithm [26] to incrementally update the corresponding variance statistics. Finally, the updated flow statistics s are written back to the BPF map M_F using the Update(M_F, k, s) function (step 2 in Fig. 2 and line 16 in Algorithm 1). Note that the packet length ℓ_p, IAT iat_p, and TCP flags $\boldsymbol{f}_p$ and their statistics are represented in integer values.

If flow duration d is shorter than τ, it continues collecting packets to update the flow statistics, otherwise, it proceeds with the inference process (line 19 in Algorithm 1). After updating the flow statistics s, the program checks whether the current flow duration exceeds the predefined flow observation period τ (step 3 in Fig. 2 and line 18 in Algorithm 1). During inference, the program extracts the feature vector $\boldsymbol{x}$ from the flow statistics s using getFeatures(s) function and converts it into a fixed-point representation x_{fixed} (lines 20 and 21 in Algorithm 1). The program then retrieves the classifier parameters Θ_C and standardization parameters S_C from the BPF map M_C that stores the parameters of classifier C (line 22 in Algorithm 1). If the classifier C is either an SVM or an MLP, the program standardizes the fixed-point features x_{fixed} using S_C (lines 23 and 24 in Algorithm 1). The program then performs fixed-point inference using the FixedPointInference$(C, x_{\text{fixed}}, \Theta_C)$ function, producing the classification result $\hat{y}$ (step 4 in Fig. 2 and line 25 in Algorithm 1). If the classifier identifies the flow as malicious (i.e., $\hat{y} = 1$), the program drops the packet p to mitigate the detected DDoS attacks (step 5a in Fig. 2 and line 27 in Algorithm 1). Otherwise, the program forwards the packet p to the next network function (step 5b in Fig. 2 and line 29 in Algorithm 1).

Recall that the MLP classifier requires tail calls to implement layer-wise operations within eBPF programs, as discussed in Section 3.1. However, `bpftime` does not support userspace-to-userspace tail calls. To address this limitation, we implement a uBPF caller module that sequentially invokes multiple uBPF programs from the userspace application in required order.

Algorithm 1 Online packet processing in the eBPF/uBPF-based NIDPS

Require: Packet p, flow-state map M_F, classifier map M_C, classifier C, classifier parameters Θ_C, scaler parameters S_C, flow-observation period τ
Ensure: DROP or FORWARD

1: $t \leftarrow$ BPF_KTIME_GET_NS()
2: $k \leftarrow (sip_p, dip_p, sport_p, dport_p, proto_p)$
3: $\ell_p \leftarrow$ PacketLength(p)
4: $\boldsymbol{f}_p \leftarrow$ TCPFlags(p)
5: $s \leftarrow$ Lookup(M_F, k)
6: **if** $s = \varnothing$ **then**
7: $\quad s \leftarrow$ InitializeFlowState(p, t)
8: $\quad s.first_timestamp \leftarrow t$
9: $\quad s.last_timestamp \leftarrow t$
10: **else**
11: $\quad iat \leftarrow t - s.last_timestamp$
12: $\quad$ UpdateIATStatistics(s, iat)
13: $\quad s.last_timestamp \leftarrow t$
14: UpdatePacketLengthStatistics(s, ℓ)
15: UpdateTCPFlagCounters$(s, \boldsymbol{f})$
16: Update(M_F, k, s)
17: $d \leftarrow s.last_timestamp - s.first_timestamp$
18: **if** $d < \tau$ **then**
19: $\quad$ **return** FORWARD
20: $\boldsymbol{x}_{\text{int}} \leftarrow$ GetFeatures(s)
21: $\boldsymbol{x}_{\text{fixed}} \leftarrow$ EncodeFixedPoint$(\boldsymbol{x}_{\text{int}})$
22: $\Theta_C, S_c \leftarrow$ GetClassifier(M_C)
23: **if** $C \in \{\text{SVM}, \text{MLP}\}$ **then**
24: $\quad \boldsymbol{x}_{\text{fixed}} \leftarrow$ FixedPointStandardize$(\boldsymbol{x}_{\text{fixed}}, S_C)$
25: $\hat{y} \leftarrow$ FixedPointInference$(C, \boldsymbol{x}_{\text{fixed}}, \Theta_C)$
26: **if** $\hat{y} = 1$ **then**
27: $\quad$ **return** DROP
28: **else**
29: $\quad$ **return** FORWARD

Table 6: BPF maps used in the proposed NIDPS.

Map name	Map type	Key	Key size	Value	Value size
Flow statistics map	Hash map	5-tuple flow identifier	16 B	Flow statistics	208 B
DT parameters map	Array map	Integer value	4 B	Trained parameters of the DT classifier	6.52 KB
RF parameters map	Array map	Integer value	4 B	Trained parameters of the RF classifier	27.4 KB
XGB parameters map	Array map	Integer value	4 B	Trained parameters of the XGB classifier	117.2 KB
SVM parameters map	Array map	Integer value	4 B	Trained parameters of the SVM classifier	136 B
MLP parameters map	Array map	Integer value	4 B	Trained parameters of the MLP classifier	6.92 KB

4.5. BPF Maps for Storing Flow Statistics and Trained Parameters

BPF maps are used to store both flow statistics and trained parameters of the classifiers. Table 6 summarizes the BPF maps

Table 7: Information on the CIC-IDS2017 and CIC-DDoS2019 datasets used in the evaluation.

Dataset	CIC-IDS2017	CIC-DDoS2019
# of benign/malicious flows for training	184,294/152,070	207,492/207,492
# of benign/malicious flows for validation	46,074/38,017	51,873/51,873
# of benign/malicious flows for testing	47,522/47,522	23,427/23,427

used in the proposed NIDPS. The flow statistics are stored in a hash map, where the key is a 5-tuple flow identifier with 16 B and the value is a structure containing the flow statistics with 208 B. The trained parameters of all classifiers are stored in an array map, where the key is an integer value and the value is a structure containing the trained parameters. The sizes of the trained parameters are based on the evaluation settings described in Section 5.

5. Evaluation

5.1. Detection Performance

5.1.1. Evaluation Setup

We evaluate the binary classification performance of the proposed NIDPS using the CIC-IDS2017 [20] and CIC-DDoS2019 [19] datasets. The sixteen features listed in Table 5 are extracted from both datasets and used for training and testing the classifiers. Since it is difficult to evaluate the detection performance directly within eBPF/uBPF programs, we simulate feature extraction by replaying the pcap files, as mentioned in Section 4.1. Table 7 summarizes the information on these datasets used in the evaluation.

For CIC-IDS2017, the datasets captured on July 3 and July 7, which contain DoS-related traffic, are used for training/validation and testing, respectively. From the July 3 dataset, we randomly select 80% of the flows and split them into the training and validation sets at an 8:2 ratio. For the test set, we use the remaining malicious flows and randomly sample benign flows from the remaining benign flows such that the numbers of benign and malicious flows are equal.

For CIC-DDoS2019, the dataset captured on Dec. 1 is used for training and validation, whereas the dataset captured on Nov. 3 is used for testing. Since CIC-DDoS2019 is highly imbalanced, containing substantially more malicious flows than benign flows, we augment its benign training samples with the benign training samples from CIC-IDS2017. We then randomly sample malicious flows from the Dec. 1 dataset such that the number of malicious flows matches the number of benign flows. We split the resulting balanced dataset into the training and validation sets at an 8:2 ratio. For the test set, we randomly sample flows from the Nov. 3 dataset such that the numbers of benign and malicious flows are equal.

In the evaluation metrics, accuracy, precision, recall, and false positive rate (FPR) are used, each of which is defined as follows:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}},$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}},$$

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}},$$

where TP, TN, FP, and FN denote the numbers of true positives (the number of correctly classified malicious flows), true negatives (i.e., the number of correctly classified benign flows), false positives (i.e., the number of benign flows incorrectly classified as malicious flows), and false negatives (i.e., the number of malicious flows incorrectly classified as benign flows), respectively.

A DT with a maximum depth of 10, an RF with a maximum depth of 10 and five estimators, an XGB classifier with a maximum depth of 10 and ten estimators, and a linear SVM are trained using scikit-learn [28] while an MLP with a 32×32 hidden layer is trained by using PyTorch [29]. We use an SGD optimizer with a learning rate of 10^{-2} and a batch size of 1024 to train the MLP classifier for 200 epochs. We also implement the fixed-point DT, RF, XGB, SVM, and MLP classifiers in the C language, for simulating the eBPF/uBPF programs. All calculations in the classifiers are performed using integer arithmetic according to the fixed-point arithmetic rules [24]. For comparison with the detection performance of the proposed NIDPS, we also evaluate the detection performance of the floating-point classifiers implemented in the Python language.

5.1.2. Classification Results

Figs. 3 and 4 illustrate the impact of the flow observation period τ on detection performance for the CIC-IDS2017 and CIC-DDoS2019 datasets, respectively, where t denotes the elapsed time since a new flow was generated. All classifiers exhibit similar trends: accuracy, precision, and recall improve as the flow observation period τ increases, while the FPR decreases. This is because only a limited number of packets are available immediately after a new flow is generated, resulting in insufficient and unstable flow statistics. As τ increases, the flow statistics become more informative, allowing the classifiers to distinguish malicious flows from benign ones more accurately.

For the CIC-IDS2017 dataset, the fixed-point DT, RF, XGB, and MLP classifiers achieve stable classification performance when $\tau \geq 0.1$ s. Specifically, these four classifiers achieve an accuracy of 0.877–0.915, a precision of 0.878–0.965, a recall of 0.861–0.877, and an FPR of 0.03–0.12 when $\tau = 0.1$ s. The fixed-point SVM classifier shows lower performance than the other classifiers, except in terms of recall; when $\tau = 0.1$ s, its accuracy, precision, recall, and FPR are 0.734, 0.676, 0.906, and 0.434, respectively. These results suggest that the tree-based classifiers and the MLP classifier are more suitable than the linear SVM classifier for the proposed NIDPS on the CIC-IDS2017 dataset.

For the CIC-DDoS2019 dataset, the classifiers achieve better classification performance than for the CIC-IDS2017 dataset. The majority of flows in the CIC-DDoS2019 dataset have a

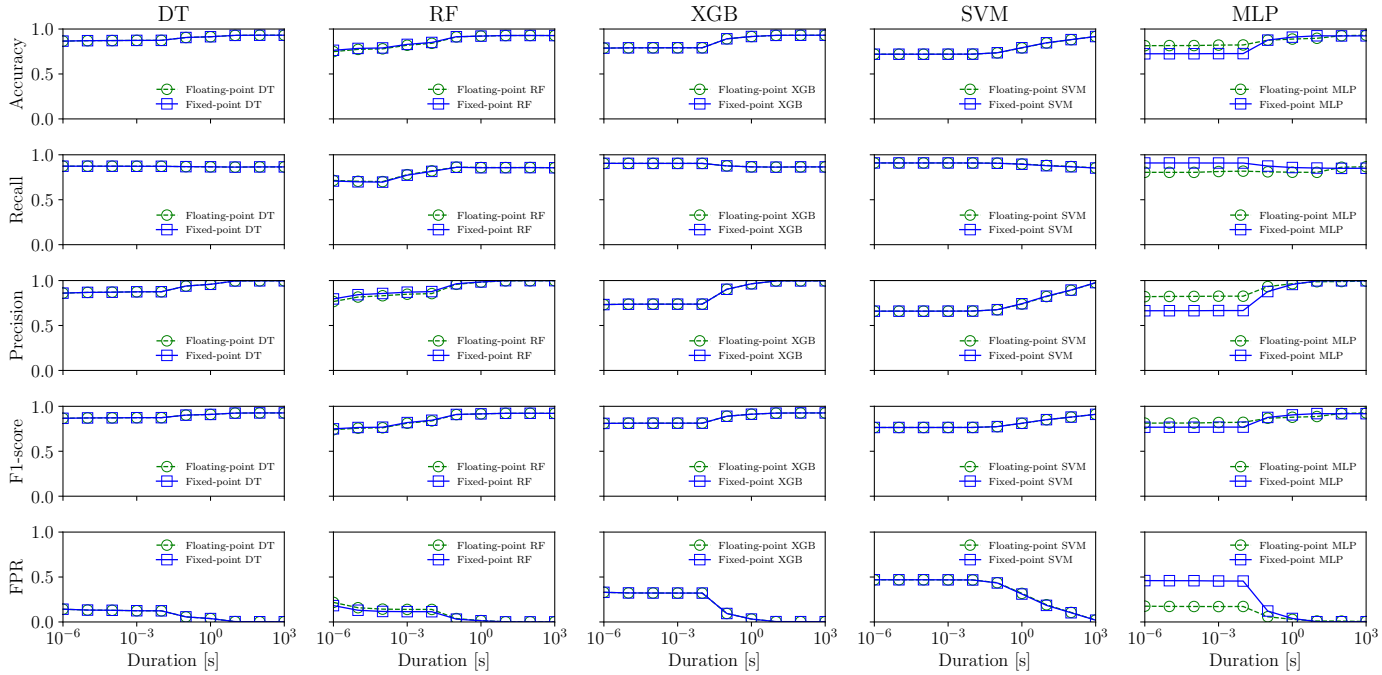


Figure 3: Detection performance over flow observation period t (CIC-IDS2017 dataset).

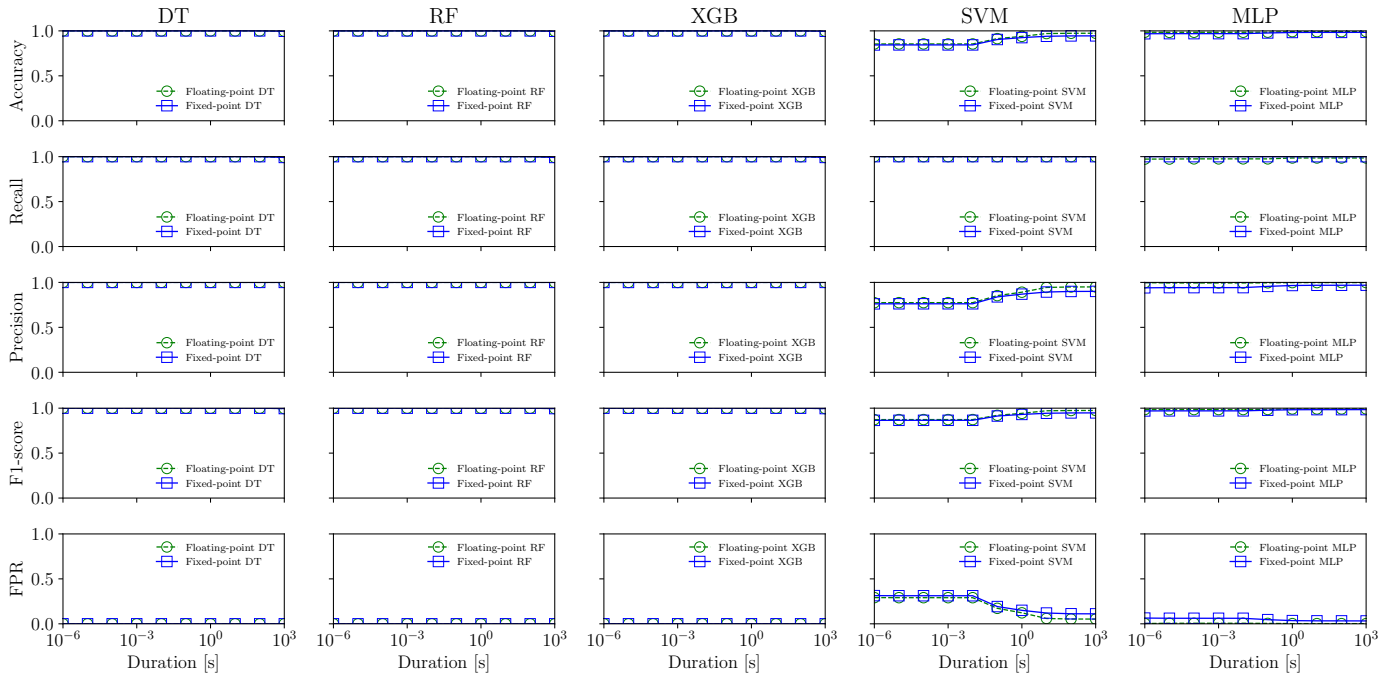


Figure 4: Detection performance over flow observation period t (CIC-DDoS2019 dataset).

duration of one microsecond or less, suggesting that sufficient flow statistics can already be collected at $\tau = 10^{-6}$ s. Consequently, the classifiers maintain high detection performance even with this short observation period. At $\tau = 0.1$ s, all fixed-point classifiers achieve an accuracy of at least 0.924, a precision of at least 0.870, and a recall of 1.00. These results indicate that the classifiers successfully detect all malicious flows while maintaining a relatively low false-positive rate. Specifically, the FPR is at most 0.150 at $\tau = 0.1$ s.

The fixed-point classifiers show detection performance comparable to that of their floating-point counterparts in many cases, although some performance degradation is observed for the MLP and SVM classifiers due to fixed-point arithmetic errors. These results indicate that fixed-point arithmetic can maintain practical DDoS detection performance under the constraints of eBPF/uBPF programs, where floating-point arithmetic is not supported. However, the impact of fixed-point arithmetic errors depends on the classifier type. The tree-based

classifiers show almost identical results between the floating-point and fixed-point implementations, whereas the MLP and SVM classifiers show larger performance degradation. This is because the MLP classifier accumulates fixed-point arithmetic errors across layer operations, while the SVM classifier is sensitive to such errors due to its linear decision boundary. Therefore, classifiers should be selected by considering not only classification accuracy but also robustness to fixed-point implementation errors under the constraints of eBPF/uBPF programs.

5.2. Packet Processing Performance

5.2.1. Evaluation Setup

We use two computing servers: a sender equipped with a 13th Gen Intel(R) Core(TM) i7-13700K CPU and 64 GB memory, and a receiver equipped with Intel Xeon Gold 5317 CPU, 64 GB memory, Intel Ethernet Converged Network Adapter 710X, and Ubuntu 22.04 LTS with kernel version 6.8. These servers are directly connected via a 10 Gbps link. The receiver is responsible for executing a network function (NF), which consists of the proposed NIDPS and L2 forwarding. To evaluate single-core packet processing performance, we configure the receiver to use a single RX queue. We send 64-byte UDP packets from the sender to the receiver over a 100-second interval using the `pktgen` tool [30] with 8 cores. In this evaluation, to confirm the performance bottleneck, we assume worst-case scenarios where all packets are classified by the NIDPS, indicating the flow observation period $\tau = 0$.

To explore distinct packet processing mechanisms, we implement ten types of NIDPSs: generic XDP, native XDP, uBPFs over AF_XDP standard, syscall, and poll, uBPF over DPDK, AF_XDP standard, syscall, and poll, and DPDK, as shown in Table 3. The generic and native XDP-based NIDPSs are implemented as eBPF programs, which follow the existing in-kernel NIDPS implementations [5, 7, 8, 9]. We use the uBPF implementation provided in [15] and implement eBPF programs equipped with either the five classifiers or the flow extraction only. The uBPF runtime is configured to use an LLVM-based backend [14]. To ensure a fair comparison, we implement the same five classifiers and flow extraction only in the AF_XDP and DPDK dataplanes without uBPF, which are implemented in native C language without uBPF. For the configurations of the kernel-bypassing frameworks, AF_XDP enables a zero-copy mode and uses 64-packet batches. AF_XDP syscall also enables `SO_PREFER_BUSY_POLL` and `XDP_USE_NEED_WAKEUP`. DPDK is configured with a memory buffer pool of 512 MB.

As the evaluation metric, we use the packet processing rate R, the packet processing efficiency ρ, the packet processing latency, and the CPU utilization. The packet processing rate R is defined as the number of packets processed by the receiver per second. The packet processing efficiency ρ is the ratio of the packet processing rate R to the maximum packet processing rate $R_{\max}$ (i.e., $\rho = R/R_{\max}$). The maximum packet processing rate $R_{\max}$ is defined as the idealized upper-bound packet processing rate, which is defined as follows:

$$R_{\max} = \frac{C}{8(L+14)}.$$

Table 8: Instruction counts of eBPF/uBPF programs.

Scheme	Instruction count
Flow extraction only	547
DT	560
RF	948
XGB	1,298
SVM	560
MLP	3,635

Here, C is the link capacity in bits per second and L is the packet length in bytes and the term $8(L + 14)$ represents the bit length of a packet, including the Ethernet frame header (14 bytes). The packet processing latency D is defined as the time taken to conduct flow extraction, feature extraction, classification and packet forwarding. Note that the packet processing latency D does not include the driver execution and userspace checking costs. The 99th-percentile (P99) packet processing latency is estimated using a fixed-width latency histogram with 100-ns bins. The processing time of each packet is measured and assigned to the corresponding latency bin. After all samples are collected, the P99 rank is determined as 99% of the total number of packets. The histogram is then scanned in ascending order of latency while accumulating the number of samples. The upper bound of the first bin whose cumulative count reaches the P99 rank is reported as the P99 latency.

5.2.2. Instruction Count

Table 8 presents the instruction counts of eBPF/uBPF programs for six NFs (i.e., flow extraction only and the five classifiers), where instruction counts are measured using the `bpftool` command. The instruction count of the flow extraction only is 547, which is the minimum instruction count among the six NFs. The DT and SVM classifiers have smaller instruction counts than the other classifiers, with 560 instructions each. This is because the DT classifier only requires a series of comparisons and branching operations, while the SVM classifier only requires a linear combination of the input features and a threshold comparison for classification. RF and XGB classifiers have larger instruction counts of 948 and 1,298, respectively, due to the additional instructions required for traversing multiple decision trees. Furthermore, the MLP classifier has the largest instruction count of 3,635, which is significantly higher than that of the other classifiers due to the tail calls and the multiple layer-wise operations, including matrix multiplications and activation functions. The instruction counts of the classifiers affect the packet processing rate, as discussed in Section 5.2.3.

5.2.3. Packet Processing Rate

Fig. 5 compares the packet processing rate R across ten packet processing mechanisms and six NFs. Native XDP achieves rates of 0.72 Mpps, 0.72 Mpps, 0.68 Mpps, 0.64 Mpps, 0.72 Mpps, and 0.27 Mpps for flow extraction only, DT, RF, XGB, SVM, and MLP, respectively. Generic XDP achieves slightly lower rates than native XDP, consistent with the additional overhead of allocating and initializing a socket buffer for each incoming packet.

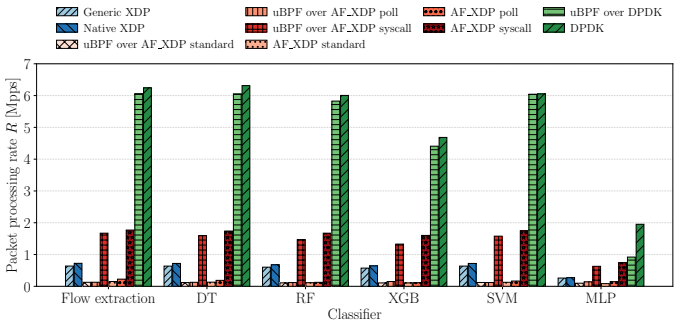


Figure 5: Packet processing rate.

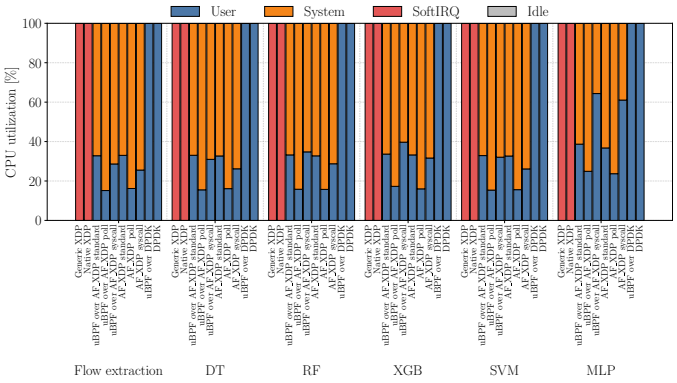


Figure 6: CPU utilization.

The packet processing rates generally reflect the computational complexity of the classifiers. DT and SVM achieve the same rate as flow extraction only because both require 560 instructions and introduce relatively little additional computational overhead. However, their detection performance is not equivalent. As discussed in Section 5.1.2, on the CIC-IDS2017 dataset, SVM achieves an accuracy of 0.734, while DT, RF, XGB, and MLP achieve higher and more stable detection performance. RF and XGB achieve lower rates of 0.68 Mpps and 0.64 Mpps, respectively, because they require 948 and 1,298 instructions and involve traversing multiple decision trees. Although these classifiers can provide high detection performance, their additional computational cost reduces the packet processing rate. MLP achieves the lowest rate, 0.27 Mpps, because it requires 3,635 instructions, including layer-wise matrix operations and activation functions. Thus, MLP can provide high detection performance at the cost of a substantial reduction in packet processing throughput. In other words, there is a trade-off between detection performance and packet processing performance. SVM provides low computational cost but relatively poor detection performance on CIC-IDS2017, whereas RF, XGB, and MLP can provide higher detection performance with increasing computational overhead.

The uBPF-based implementations over AF_XDP syscall and DPDK achieve higher packet processing rates than native XDP. Specifically, uBPF over AF_XDP syscall achieves rates of 1.67 Mpps, 1.59 Mpps, 1.46 Mpps, 1.33 Mpps, 1.58 Mpps, and 0.63 Mpps for flow extraction only, DT, RF, XGB, SVM, and MLP, respectively. These rates are 2.31, 2.21, 2.16, 2.07, 2.19, and 2.31 times those of native XDP. uBPF over DPDK achieves corresponding rates of 6.06 Mpps, 6.05 Mpps, 5.82 Mpps, 4.41 Mpps, 6.02 Mpps, and 0.92 Mpps, or 8.37, 8.40, 8.60, 6.86, 8.39, and 3.38 times those of native XDP. The improvement is attributed to DPDK's ability to fully bypass the kernel network stack and perform packet I/O entirely in user space. These comparisons reflect differences in the complete packet processing paths, including network I/O, polling, and program execution.

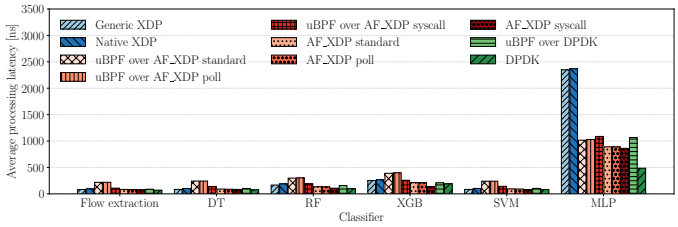
In contrast, uBPF over AF_XDP standard and AF_XDP poll achieve substantially lower packet processing rates than native XDP. Native XDP executes the NF at the driver hook, whereas AF_XDP additionally delivers packets to user space through RX rings. Under the single-core configuration used in this evaluation, NAPI processing, userspace packet handling, and classifier execution share the same CPU core. The additional delivery and coordination costs can therefore reduce the achievable packet processing rate. A similar trend is observed in the corresponding native implementations of AF_XDP standard and AF_XDP poll, suggesting that this limitation is associated with their packet processing paths rather than solely with uBPF execution.

Across the evaluated mechanisms, the MLP classifier achieves substantially lower packet processing rates than the other classifiers. This result is consistent with its larger instruction count and the computational cost of layer-wise matrix operations and activation functions. For MLP, uBPF over DPDK achieves 3.38 times the rate of native XDP, whereas the corresponding ratios range from 6.86 to 8.60 for the other NFs. Thus, the relative benefit of the DPDK packet-I/O path is smaller when classifier execution accounts for a larger share of the per-packet processing cost. For uBPF-based MLP execution, the sequential invocation of multiple programs by the userspace caller may also contribute to this cost.

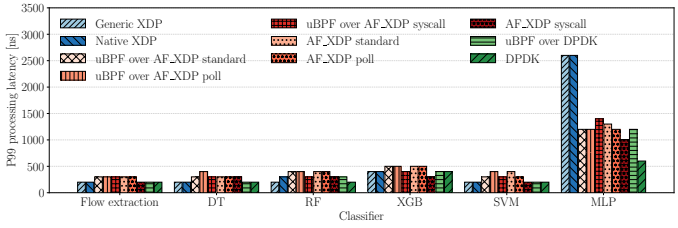
To assess the performance cost of uBPF within the same packet-I/O framework, we compare the uBPF-based implementations with their native C counterparts. The native C implementation over AF_XDP syscall achieves rates of 1.77 Mpps, 1.74 Mpps, 1.67 Mpps, 1.60 Mpps, 1.74 Mpps, and 0.74 Mpps for flow extraction only, DT, RF, XGB, SVM, and MLP, respectively. The native C implementation over DPDK achieves corresponding rates of 6.24 Mpps, 6.31 Mpps, 6.01 Mpps, 4.68 Mpps, 6.05 Mpps, and 1.95 Mpps. For DPDK, the performance gap is relatively small for flow extraction only and the non-MLP classifiers, but is substantial for MLP: the uBPF-based implementation achieves 0.92 Mpps compared with 1.95 Mpps for native C. Potential factors of these differences include program transformations required to satisfy eBPF constraints, BPF-map access costs, and sequential invocation overhead of multiple uBPF programs.

5.2.4. CPU Utilization

Fig. 6 illustrates the breakdown of CPU time into *User*, *System*, *SoftIRQ*, and *Idle* for each combination of packet processing mechanism and NF. Here, *User* and *System* represent the



(a) Average.



(b) 99th percentile.

Figure 7: Packet processing latency.

fractions of CPU time spent executing userspace code and kernel code, respectively. *SoftIRQ* represents time spent handling software interrupts, and *Idle* represents time during which the CPU is idle. Across all evaluated combinations, CPU utilization reaches approximately 100%, with negligible idle time on the measured core. Thus, the differences in packet processing rate reported in Section 5.2.3 occur under similarly high CPU utilization. However, full CPU utilization does not necessarily indicate that all CPU time is spent processing packets, because busy polling also consumes CPU cycles. These results should therefore be interpreted together with the packet processing rates.

For native XDP and generic XDP, most CPU time is accounted for as *SoftIRQ*. Under the evaluated configuration, packet reception and XDP program execution, including feature extraction and classification, predominantly take place in the NAPI receive path within softirq context. For the AF_XDP-based mechanisms, *User* and *System* account for most CPU time. Feature extraction and classification execute in user space, while packet delivery involves kernel processing. The precise breakdown depends on the polling mode and the execution context of receive processing. As classifier complexity increases, the fraction of CPU time spent in *User* generally increases. This trend is particularly evident for NIDPSs using AF_XDP syscall. For the DPDK-based mechanisms, CPU time is predominantly accounted for as *User*, because packet I/O, feature extraction, and classification execute in user space. This category also includes the CPU time spent polling for incoming packets.

5.2.5. Packet Processing Latency

Figs. 7a and 7b illustrate the average and 99th percentile packet processing latency for each combination of packet processing mechanisms and six NFs. We confirm that the average and 99th percentile processing latency achieve nanosecond-level latency for all combinations of packet processing mechanisms and NFs, except for the MLP classifier. However, the MLP classifier exhibits the average and 99th percentile processing latency in the order of microseconds, which is attributed to the high instruction count of the MLP classifier.

Comparing the packet processing mechanisms, we observe that XDP-related mechanisms have higher average and 99th percentile processing latency than AF_XDP and DPDK-related mechanisms. This is because the XDP-related mechanisms require BPF-map accesses for flow extraction and classification and tail-calls for MLP classifier execution, which increases the packet processing latency.

5.2.6. Impact of Packet Length

Fig. 8 shows the impact of packet length on the ratio ρ for three packet processing mechanisms (i.e., native XDP, uBPF over AF_XDP syscall, and uBPF over DPDK) and six NFs (i.e., flow extraction only, DT, RF, XGB, SVM, and MLP). Regardless of the packet processing mechanism and NF, the ratio ρ increases as the packet length L increases. This is because the number of packets sent per second by the sender decreases as the packet length increases, improving packet processing efficiency.

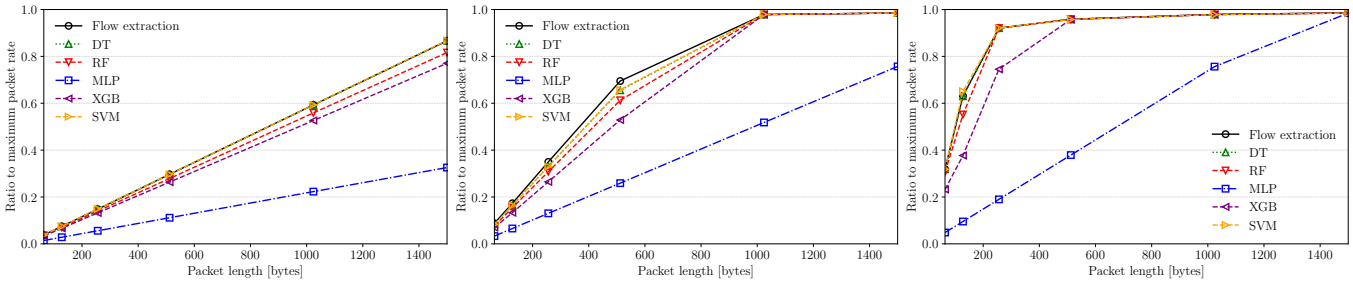
In native XDP (Fig. 8a), the ratio ρ increases almost linearly with the packet length L. In uBPF over AF_XDP syscall (Fig. 8b) and uBPF over DPDK (Fig. 8c), the ratio ρ also increases with L and saturates at approximately $L = 1{,}000$ bytes and $L = 512$ bytes, respectively, except for the MLP classifier.

These results indicate that uBPF over DPDK achieves packet processing rates close to the idealized upper-bound packet rate when $L \geq 512$ bytes, while uBPF over AF_XDP syscall approaches the upper bound when $L \geq 1{,}000$ bytes. Native XDP does not reach the upper bound even when $L = 1{,}500$ bytes for all NFs. Note that ρ may still be smaller than 1.0 because $R_{\max}$ represents the upper bound and does not include software packet I/O overhead, memory access overhead, or classifier execution overhead.

The MLP classifier exhibits a lower ratio ρ than the other classifiers because it requires more CPU resources for classification, leaving insufficient CPU resources for the network I/O process. Therefore, native XDP and AF_XDP syscall with the MLP classifier do not achieve line-rate performance even when $L = 1{,}500$ bytes. However, in the uBPF over DPDK case, the ratio ρ of the MLP classifier becomes competitive with those of the other classifiers when $L = 1{,}500$ bytes.

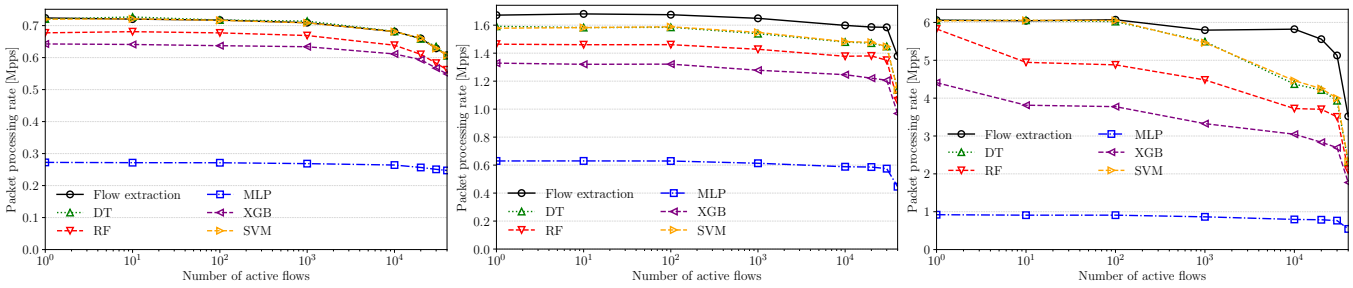
5.2.7. Impact of Active Flow Counts

Fig. 9 illustrates the impact of the number of active flows on the packet processing rate among the three packet processing mechanisms (i.e., native XDP, uBPF over AF_XDP syscall, and uBPF over DPDK) and the six NFs (i.e., flow extraction only, DT, RF, SVM, XGB, and MLP). The packet processing rate remains almost constant until the number of active flows reaches approximately 10,000 and then decreases as the number of active flows further increases. This is because the flow statistics are stored in a hash map, and as the number of active flows increases, the number of map entries increases, leading to increased cache misses.



(a) Native XDP. (b) uBPF over AF_XDP syscall. (c) uBPF over DPDK.

Figure 8: Relationship between the packet length L and the ratio ρ of packet processing rate to the maximum packet processing rate.



(a) Native XDP. (b) uBPF over AF_XDP syscall. (c) uBPF over DPDK.

Figure 9: Impact of the number of active flows on packet processing rate.

In Fig. 9, the packet processing rate with lightweight NFs shows a significant decrease when the number of active flows exceeds 10,000, while the packet processing rate with the MLP classifier shows a slight decrease. Furthermore, for lightweight classifiers, uBPF over DPDK, which achieves the highest packet processing rate, exhibits more significant degradation compared to native XDP and uBPF over AF_XDP syscall. In particular, the combinations of packet processing mechanisms and NFs with higher packet processing rates are more sensitive to the number of active flows because they are more affected by the increased cache misses.

6. Limitations

6.1. Impact of Class Imbalance on Evaluation Metrics

To enable a fair comparison among the classifiers, we used the test set containing equal numbers of benign and malicious flow samples, as mentioned in Section 5.2.1. Although this balanced setting prevents the majority class from dominating aggregate metrics such as accuracy, it does not reflect the class distribution typically observed in operational networks, where benign flows substantially outnumber malicious flows. In particular, precision and the absolute number of false alerts depend on the prevalence of malicious traffic. Therefore, the precision and accuracy reported in Section 5.1.2 may differ from those obtained in deployment, even if recall and the FPR remain unchanged. For example, a low FPR can still produce a large number of false alerts when the classifier is applied to a high volume of benign traffic. Consequently, the results should primarily be interpreted as a comparison of the intrinsic classification capabilities of the evaluated classifiers. In future work, further evaluation using imbalanced test sets that reflect realistic traffic distributions is required to quantify operational precision and false-alert volume as well as to determine appropriate decision thresholds for practical deployment.

6.2. Multi-Class Classification

This work focuses on binary classification between benign and malicious flows. This evaluation setting is suitable for evaluating the feasibility of eBPF/uBPF-based DDoS detection and prevention because the proposed NIDPS primarily decides whether an incoming packet should be forwarded or dropped. However, practical NIDPSs may require multi-class classification to identify specific attack types and apply attack-specific mitigation policies.

Extending the proposed NIDPS to multi-class classification may increase model complexity, BPF map size, instruction count, and per-packet processing cost. Therefore, multi-class classification introduces a trade-off among detection granularity, classification performance, and packet processing performance. In future work, we plan to evaluate this trade-off under the constraints of eBPF/uBPF programs.

6.3. Concurrent Multi-core and Multi-queue Packet Processing

This work evaluates the single-core packet processing performance by using a single RX queue. This setting clarifies the per-core packet processing capability of the proposed NIDPSs. However, practical NIDPS deployments require concurrent packet processing across multiple cores and multiple RX queues for high-speed networks.

Extending the proposed NIDPS to concurrent packet processing is not straightforward because the current uBPF implementation has several runtime and programming limitations, including limited support for concurrent accesses to shared BPF maps, userspace-to-userspace tail calls, and concurrent invocation of multiple uBPF programs from a userspace caller module. If these limitations are addressed, *Receive-Side Scaling* (RSS) [31] and per-core BPF maps can be used to enable concurrent packet processing across multiple cores and multiple RX queues. Designing and evaluating such concurrent multi-core and multi-queue uBPF-based NIDPSs remain future work.

7. Conclusion

In this paper, we proposed a userspace Berkeley Packet Filter (uBPF)-based NIDPS empowered by machine learning (ML) algorithms to achieve high portability across diverse runtime environments, including native eXpress Data Path (XDP), AF_XDP, and the Data Plane Development Kit (DPDK). The evaluation results demonstrated that (1) the five fixed-point classifiers achieved high DDoS detection performance comparable to that of their floating-point counterparts, despite fixed-point arithmetic errors, and (2) packet processing performance is determined by both network I/O overhead and classifier complexity. In future work, we plan to conduct multi-class classification and multi-core performance evaluations.

References

[1] T. Hara, S. Kasahara, Practicality Analysis of eBPF/uBPF-based Network Intrusion Detection and Prevention Systems for Distributed Denial of Service Attacks, in: Proc. of the First Workshop on AI Native Distributed Intelligence for 6G Networks (6G AI-RAN 2026) organized in conjunction with IEEE International Conference on Computer Communications (INFOCOM 2026), 2026, pp. 1–6. doi:10.1109/INFOCOM59046.2026.11571593.

[2] Y. Ma, L. Liu, Z. Liu, F. Li, Q. Xie, K. Chen, C. Lv, Y. He, F. Li, A Survey of DDoS Attack and Defense Technologies in Multiaccess Edge Computing, IEEE Internet of Things Journal 12 (2) (2025) 1428–1452. doi:10.1109/JIOT.2024.3490897.

[3] N. S. Musa, N. M. Mirza, S. H. Rafique, A. M. Abdallah, T. Murugan, Machine Learning and Deep Learning Techniques for Distributed Denial of Service Anomaly Detection in Software Defined Networks—Current Research Solutions, IEEE Access 12 (2024) 17982–18011. doi:10.1109/ACCESS.2024.3360868.

[4] Y. Choe, J.-S. Shin, S. Lee, 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, Lecture Notes on Data Engineering and Communications Technologies, Springer International Publishing, Cham, 2020, pp. 458–468. doi:10.1007/978-3-030-39746-3_47.

[5] M. Bachl, J. Fabini, T. Zseby, A Flow-Based IDS Using Machine Learning in eBPF, arXiv preprint (arXiv:2102.09980) (Mar. 2022). arXiv:2102.09980, doi:10.48550/arXiv.2102.09980.

[6] H. Taguchi, T. Hara, S. Kasahara, Unsupervised Real-Time In-Kernel Intrusion Detection System Using Autoencoders and eBPF, IEICE Transactions on Communications E109-B (2) (2026) 326–336. doi:10.23919/transcom.2025EBP3028.

[7] T. Hara, M. Sasabe, On Practicality of Kernel Packet Processing Empowered by Lightweight Neural Network and Decision Tree, in: Proc. of International Conference on Network of the Future (NoF), 2023, pp. 89–97. doi:10.1109/NoF58724.2023.10302811.

[8] S. Kostopoulos, D. Papatsaroucha, I. Kefaloukos, E. K. Markakis, eIDPS: A Real-Time eBPF-based and Machine Learning-Powered Network Intrusion Detection and Prevention Solution, in: Proc. of 2025 6th International Conference in Electronic Engineering & Information Technology (EEITE), 2025, pp. 1–8. doi:10.1109/EEITE65381.2025.11165804.

[9] N. Anand, S. M A, P. K. Aakula, R. B. Ponnuru, R. Patan, C. R. P. Reddy, Enhancing Intrusion Detection against Denial of Service and Distributed Denial of Service Attacks: Leveraging extended Berkeley Packet Filter and Machine Learning Algorithms , IET Communications 19 (1) (2025) e12879. doi:10.1049/cmu2.12879.

[10] A. Nakryiko, BPF CO-RE reference guide, `https://nakryiko.com/posts/bpf-core-reference-guide/`, Accessed 18 Jun. 2026 (2018).

[11] Linux Foundation, Data Plane Development Kit, `https://www.dpdk.org/`, Accessed 18 Jun. 2026 (2018).

[12] The Kernel Development Community, AF_XDP — The Linux Kernel documentation, `https://docs.kernel.org/networking/af_xdp.html`, Accessed 18 Jun. 2026 (2023).

[13] iovisor, iovisor/ubpf, `https://github.com/iovisor/ubpf`, Accessed 18 Jun. 2026 (2025).

[14] Y. Zheng, T. Yu, Y. Yang, Y. Hu, X. Lai, D. Williams, A. Quinn, Extending Applications Safely and Efficiently, in: Proc. of 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25), 2025, pp. 557–574.

[15] Y. Zheng, P. Gavriil, M. Kogias, uXDP: Frictionless XDP Deployments in Userspace, in: Proc. of the 3rd Workshop on eBPF and Kernel Extensions , 2025, pp. 8–14. doi:10.1145/3748355.3748360.

[16] M. Elzoghbi, H. He, Kernel-level LDoS attack detection in SDN networks: An eBPF /XDP framework with dynamic thresholding, Computer Networks 275 (2026) 111939. doi:10.1016/j.comnet.2025.111939.

[17] T. Hara, M. Sasabe, Practicality of In-Kernel/User-Space Packet Processing Empowered by Lightweight Neural Network and Decision Tree, Computer Networks 240 (2024) 110188:1–110188:18. doi:10.1016/j.comnet.2024.110188.

[18] J. E. Varghese, B. Muniyal, An Efficient IDS Framework for DDoS Attacks in SDN Environment, IEEE Access 9 (2021) 69680–69699. doi:10.1109/ACCESS.2021.3078065.

[19] I. Sharafaldin, A. H. Lashkari, S. Hakak, A. A. Ghorbani, Developing Realistic Distributed Denial of Service (DDoS) Attack Dataset and Taxonomy, in: Proc. of 2019 International Carnahan Conference on Security Technology (ICCST), 2019, pp. 1–8. doi:10.1109/CCST.2019.8888419.

[20] I. Sharafaldin, A. Habibi Lashkari, A. A. Ghorbani, Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization, in: Proc. of the International Conference on Information Systems Security and Privacy, 2018, pp. 108–116. doi:10.5220/0006639801080116.

[21] B. Gregg, BPF Performance Tools: Linux System and Application Observability, 1st Edition, Hoboken, 2019.

[22] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, D. 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, ACM, Heraklion Greece, 2018, pp. 54–66. doi:10.1145/3281411.3281443.

[23] C. Liu, B. Tak, L. Wang, Understanding Performance of eBPF Maps, in: Proc. of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions, eBPF '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 9–15. doi:10.1145/3672197.3673430.

[24] H. Benmaghnia, M. Martel, Y. Seladji, Code Generation For Neural Networks Based On Fixed-Point Arithmetic , ACM Transactions on Embedded Computing Systems (Sep. 2022). doi:10.1145/3563945.

[25] A. Osaki, M. Poisson, S. Makino, R. Shiiba, K. Fukuda, T. Okoshi, J. Nakazawa, Dynamic Fixed-point Values in eBPF: A Case for Fully In-kernel Anomaly Detection, in: Proc. of the Asian Internet Engineering Conference 2024, AINTEC '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 46–54. doi:10.1145/3674213.3674219.

[26] B. P. Welford, Note on a Method for Calculating Corrected Sums of Squares and Products, Technometrics 4 (3) (1962) 419–420. doi:10.1080/00401706.1962.10490022.

[27] A. Zien, N. Krämer, S. Sonnenburg, G. Rätsch, The Feature Importance Ranking Measure, in: Proc. of Machine Learning and Knowledge Discovery in Databases, Springer, Berlin, Heidelberg, 2009, pp. 694–709. doi:10.1007/978-3-642-04174-7_45.

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, Scikit-learn: Machine Learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[29] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, PyTorch: An Imperative Style, High-Performance Deep Learning Library (Dec. 2019). arXiv:1912.01703, doi:10.48550/arXiv.1912.01703.

[30] The kernel development community, HOWTO for the linux packet generator — The Linux Kernel documentation, `https://docs.kernel.org/networking/pktgen.html`, Accessed 18 Jun. 2026 (2025).

[31] Linux, Scaling in the Linux Networking Stack, `https://docs.kernel.org/networking/scaling.html`, Accessed 18 Jun. 2026 (2023).