Practicality Analysis of eBPF/uBPF-based Network Intrusion Detection and Prevention Systems for Distributed Denial of Service Attacks

Takanori Hara and Shoji Kasahara
Graduate School of Science and Technology, Nara Institute of Science and Technology,
8916-5 Takayama-cho, Ikoma, Nara 630-0192, Japan.
Email: hara@ieee.org, kasahara@ieee.org

***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 practical deployment. In this paper, we propose a machine learning (ML)-based NIDPS that leverages userspace BPF (uBPF) and integrates it with kernel-bypassing frameworks such as AF_XDP and Data Plane Development Kit (DPDK) to achieve portability across diverse runtime environments while accurately and efficiently detecting and preventing Distributed Denial of Service (DDoS) attacks. Evaluation results show that the binary classification performance of the proposed NIDPS is comparable to that of existing userspace NIDPSs and also demonstrate the potential and limitations of uBPF over AF_XDP/DPDK for ML-based packet processing.**

***Index Terms*—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)**

I. Introduction

Distributed Denial of Service (DDoS) attacks pose significant threats to network security, causing service disruptions and financial losses [1]. To effectively mitigate these attacks, *Network Intrusion Detection and Prevention Systems* (NIDPSs) are essential components of modern cybersecurity infrastructures, monitoring and analyzing network traffic to detect and prevent malicious activities and policy violations. In particular, NIDPSs assisted by *Machine Learning* (ML) techniques have shown promise in accurately detecting DDoS attacks by analyzing network traffic patterns and identifying anomalies [2].

Many ML-based NIDPSs operate in user space, which introduces performance bottlenecks due to frequent context switching between kernel and user spaces, especially under high network loads. To address this issue, 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 [3]–[7]. In particular, eBPF-based NIDPSs leveraging neural networks have demonstrated the ability to perform real-time anomaly detection with high accuracy [5]–[7]. Modern userspace NIDPSs suffer from a lack of program safety mechanisms, resulting in unreliability and the occasional occurrence of unexpected crashes. On the other hand, in-kernel NIDPSs offer extremely high reliability thanks to the eBPF verifier.

Modern networks span diverse environments, including cloud infrastructures, edge computing, and Internet of Things (IoT) devices. These heterogeneous environments make it difficult to deploy ML-based NIDPSs across different platforms and architectures due to compatibility issues arising from complex ML functionality. The lack of compatibility increases maintenance costs, complicates large-scale deployments, and hinders the reuse of detection logics across platforms. Therefore, in addition to detection performance and processing capabilities, practical NIDPSs require portability. In-kernel NIDPSs offer compatibility across different kernel versions through *BPF Compile Once-Run Everywhere* (CO-RE) [8]. However, the existing in-kernel NIDPSs remain incompatible with kernel-bypassing frameworks (e.g., *Data Plane Development Kit* (DPDK) [9] and *AF_XDP* [10]), both of which are widely used in high-performance network applications. *Userspace BPF* (uBPF) has the potential to address this compatibility issue by enabling the execution of eBPF programs in user space [11]–[13]. Nevertheless, the practicality of uBPF-based packet processing for ML-based NIDPSs has not yet been explored.

In this paper, we propose a portable and lightweight ML-based NIDPS that leverages both eBPF and uBPF to achieve high detection performance and packet processing capability across diverse runtime environments[1]. Evaluation results reveal both the potential and limitations of the uBPF-based NIDPS assisted by lightweight ML techniques, compared with the existing in-kernel NIDPS approaches [4]–[7].

The main contribution of this manuscript is 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

This work was supported in part by the Japan Society for the Promotion of Science (JSPS) KAKENHI (B) under Grant 24K02931, the JSPS Grant-in-Aid for Young Scientists under Grant 23K16869.

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

limitations in terms of DDoS detection performance and packet processing capability.

2) We demonstrate that the proposed NIDPSs achieve high DDoS detection performance comparable to existing ones while mitigating the calculation errors introduced by fixed-point representations. We further show that the uBPF-based NIDPSs over AF_XDP/DPDK data planes can achieve a higher packet processing rate than the XDP-based NIDPSs.
3) The proposed NIDPSs operate in distinct runtime environments such as XDP, AF_XDP, and DPDK, without modifications or runtime source-code compilation, ensuring high stability and compatibility thanks to the eBPF/uBPF sandbox features.

The rest of this manuscript is organized as follows. Section II gives related work. Section III presents preliminaries. Section IV describes the proposed NIDPS. Section V presents evaluation results. Finally, Section VI concludes this paper.

II. Related Work

eBPF/XDP has gained significant attention in the networking community due to its ability to execute custom programs within the Linux kernel safely and efficiently [14], [15]. Leveraging the high performance and reliability of eBPF, several eBPF-based NIDPSs have been proposed [3]–[7]. In [4]–[7], the authors have introduced ML techniques into eBPF-based NIDPSs to accurately detect network attacks in real time. These NIDPSs operate directly in kernel space, maintaining the detection performance while reducing context-switching overhead and improving throughput compared to traditional userspace NIDPSs. Furthermore, eBPF programs offer not only high performance and reliability but also compatibility across different kernel versions and distributions through BPF CO-RE [8]. However, these eBPF-based NIDPSs cannot operate within the kernel-bypassing frameworks, hindering their portability across diverse runtime environments.

To address the portability issue, uBPF has been implemented as a lightweight eBPF virtual machine running in the user space [12]. The authors further integrate uBPF with kernel-bypassing frameworks (i.e., AF_XDP and DPDK) to enable the execution of eBPF programs in high-performance data planes [13]. They demonstrate the practicality of uBPF over AF_XDP/DPDK for basic packet processing applications (e.g., firewall and load balancer). However, applying uBPF over AF_XDP/DPDK to ML-based NIDPSs has not been explored yet. In this paper, we explore the potential of uBPF over AF_XDP/DPDK for ML-based NIDPSs to achieve portability across diverse runtime environments.

DDoS attack detection techniques leveraging ML have been extensively studied in the field of network security [2]. In addition, ML-based NIDPSs using kernel-bypassing frameworks have been proposed to achieve high detection performance and throughput [16]–[18]. These kernel-bypassing frameworks enable highly accurate DDoS detection by executing complex ML algorithms in user space without context-switching overhead. However, these NIDPSs lack reliability due to the absence of kernel safety mechanisms, making them prone to unexpected crashes. 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 high performance with verified safety.

III. Preliminaries

A. 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 [15]. 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's networking stack [14]. 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. eBPF constraints enforced by the eBPF verifier include (1) bounded loops, (2) limited stack size, (3) no floating-point arithmetic, and (4) restricted instruction and jump counts [15]. These constraints limit ML implementations within eBPF programs.

To address them, BPF maps, key-value stores shared between kernel and user spaces, are used to store large data structures (i.e., trained parameters). 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 [5]–[7], the tail call technique is used to implement layer-wise operations of neural networks within eBPF programs, enabling them to overcome the eBPF constraints.

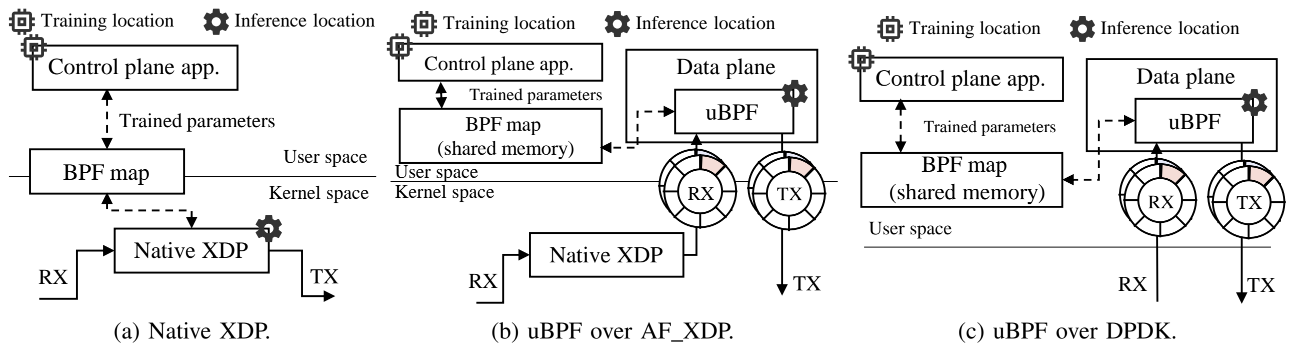
B. uBPF

uBPF is a lightweight userspace implementation of the eBPF virtual machine [11], ensuring program safety. `bpftime` is one of the sophisticated libraries for uBPF [12], enabling the execution of eBPF programs in user space. In [13], 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 a shared memory, enabling efficient data sharing between userspace applications and uBPF programs. However, it only supports a userspace-to-kernel tail call[2], which prevents executing neural-network-based classifiers only in uBPF programs.

C. Packet Processing Mechanisms

Fig. 1 illustrates a comparison of packet processing mechanisms and their relation to ML tasks. In Fig. 1, the locations of training and inference differ depending on the packet processing mechanisms. ML classifiers are trained in user

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



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

Fig. 1: Comparison of packet processing mechanisms.

space because intricate training tasks cannot be executed in the kernel due to the eBPF constraints. In contrast, the inference 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 inference.

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 kernel programmability on 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 [13].

D. Fixed-Point Arithmetic

Floating-point arithmetic is not supported in eBPF programs due to the eBPF constraints. To perform calculations involving real numbers, fixed-point arithmetic is used as an alternative. 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 of fractional part using the equation $x_{\mathrm{fixed}} = \mathrm{round}(x_{\mathrm{float}} \times 2^n)$, where $\mathrm{round}(x)$ is a function to round a value x. The rules for fixed-point arithmetic are described in [19]. The fixed-point arithmetic is used for executing ML classifiers within eBPF/uBPF programs [4]–[7].

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

A. Feature Extraction and Selection

For both training and testing the NIDPSs, we use the CIC-DDoS2019 dataset [20], which is widely used for evaluating DDoS detection performance. This dataset provides flow-level labeled network traffic statistics captured in a realistic network environment, including various types of DDoS attacks (e.g., SYN flood, UDP flood, and PortMap) as well as benign traffic. Details of the dataset can be found in [20].

TABLE I: Features used for both training and inference.

Notation	Feature
x_1	Maximum incoming packet length
x_2	Mean of incoming packet lengths
x_3	Minimum incoming packet length
x_4	TCP initial window size
x_5	ACK flag count
x_6	Sum of incoming packet lengths
x_7	FIN flag count
x_8	Mean of incoming packet inter-arrival times (IATs)
x_9	Sum of incoming packet IATs
x_{10}	Variance of incoming packet lengths
x_{11}	SYN flag count
x_{12}	Maximum incoming packet IAT

When deploying an eBPF/uBPF-based NIDPS, we cannot use the CIC-DDoS2019 dataset 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 the CIC-DDoS2019 dataset itself.

To address these limitations, we apply the following considerations for feature extraction and selection. First, we extract 5-tuple flow-based features using only statistics derived from incoming packets. Second, although the CIC-DDoS2019 dataset includes features such as the standard deviation, which is computed offline. However, calculating standard deviation requires a square root operation, making it difficult to implement efficiently in eBPF programs. To mitigate this difficulty, we use variance instead of standard deviation to reduce computational complexity, applying Welford's online algorithm [21] to compute the variance in an online manner. Third, we use feature importance [22], 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 selected twelve features $\boldsymbol{x} = (x_1, \ldots, x_{12})$ are listed in Table I.

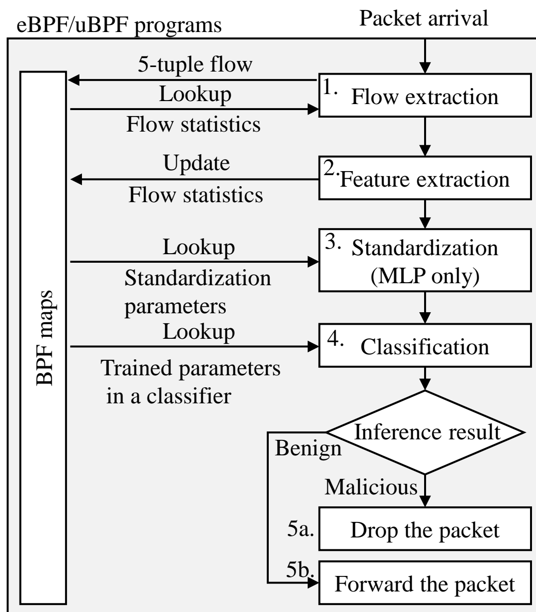


Fig. 2: Classification workflow in eBPF/uBPF programs.

B. 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 three classifiers: Decision Tree (DT), Random Forest (RF), and Multi-Layer Perceptron (MLP), which have been used in the existing in-kernel NIDPSs [4], [6], [7].

All training tasks are performed in the user space in advance. For the DT and RF classifiers, we train the classifiers using the training data represented in the fixed-point format with the lower $n = 16$ fractional bits. For the MLP classifier, we adopt a standard scaler, which transforms the input features into a standard normal distribution with mean of zero and standard deviation of one. We also train the MLP classifier 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 $n = 16$ fractional bits. These trained parameters and standardization parameters, represented in fixed-point numbers, are stored in BPF maps to be accessed by eBPF/uBPF programs during inference.

C. Anomaly Detection and Prevention in eBPF/uBPF Programs

All anomaly detection and prevention operations are executed within eBPF/uBPF programs. Fig. 2 illustrates the classification workflow. Upon the arrival of a new packet at a hook point, the corresponding eBPF/uBPF program is invoked. The program inspects the packet header to extract a 5-tuple flow identifier, which includes the source and destination IP addresses, source and destination ports, and protocol type (step 1 in Fig. 2). Using this flow identifier as a key, the program retrieves the corresponding flow statistics from a BPF map. If this retrieval fails, indicating that this is the first packet of the flow, the program initializes the flow statistics and stores them to the BPF map. Next, the program updates the flow statistics based on the information extracted from the incoming packet header (step 2 in Fig. 2). After the update, the program computes the selected features $\boldsymbol{x}$ based on the updated statistics and converts them into the fixed-point representation.

Given the selected features $\boldsymbol{x}$ in fixed-point format, the program applies the standardization only to the inputs of the MLP classifier (step 3 in Fig. 2). The program then performs inference using the classifier (step 4 in Fig. 2). It is noted that the trained parameters of the classifier are stored in BPF maps. If the classifier identifies the flow as malicious, the program drops the packet to prevent potential DDoS attacks (step 5a in Fig. 2). Otherwise, the packet is allowed to proceed to the next network function (step 5b in Fig. 2).

Recall that the MLP classifier requires tail calls to implement layer-wise operations within eBPF programs, as discussed in Section III-A. 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.

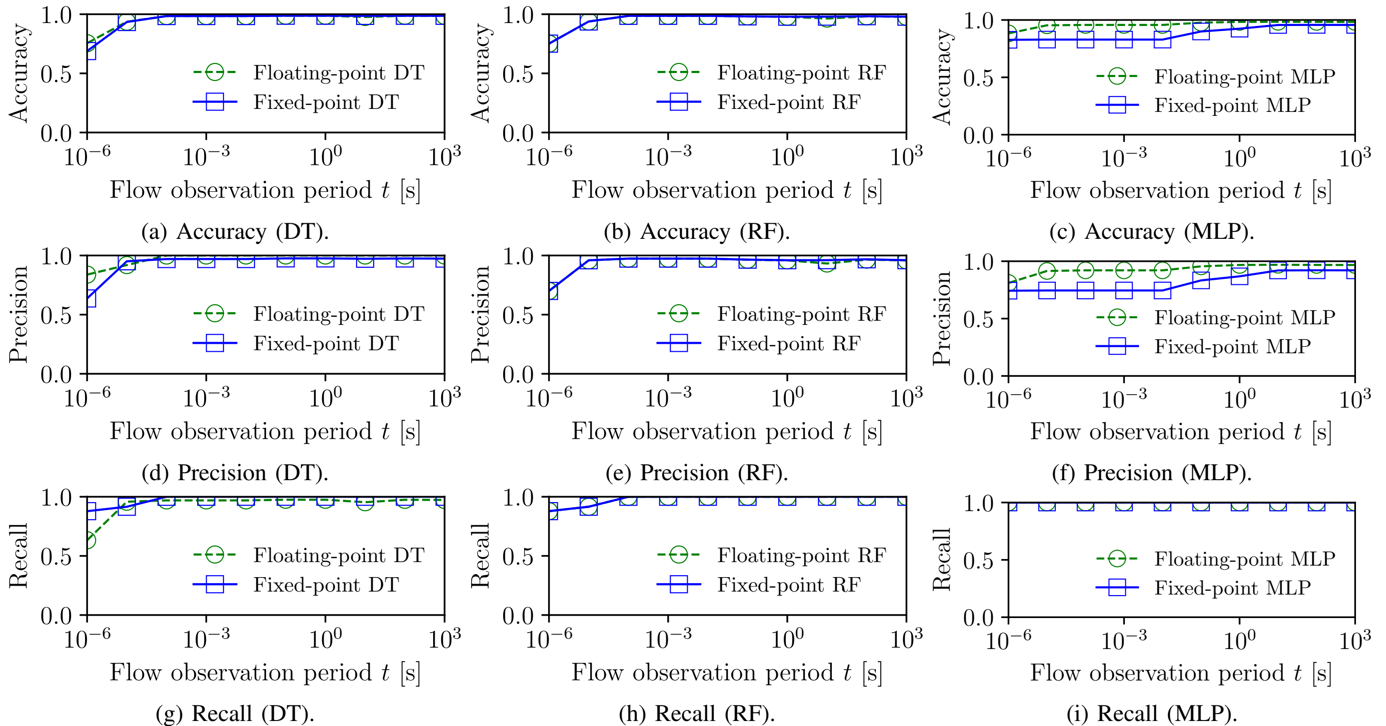
V. Evaluation

A. Detection Performance

1) Evaluation Setup: We evaluate the binary classification performance of the proposed NIDPS using the CIC-DDoS2019 dataset [20]. The dataset captured on Dec. 1 is used for training, while the dataset captured on Nov. 3 is used for testing. The twelve features listed in Table I are extracted from both datasets. For the training and validation datasets, we randomly select 20% flows of the dataset captured on Dec. 1 and split them into the training and validation datasets with an 8:2 ratio. For the testing dataset, we randomly select flows of the dataset captured on Nov. 3 such that the number of benign flows matches the number of malicious flows. The total number of flows in the training, validation, and testing datasets are 2,858,792, 571,758, and 44,544, respectively. 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 IV-A.

In the evaluation metrics, accuracy, precision, and recall are used, each of which is defined as follows: $\mathrm{Accuracy} = (\mathrm{TP}+\mathrm{TN})/(\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN})$, $\mathrm{Precision} = \mathrm{TP}/(\mathrm{TP}+\mathrm{FP})$, and $\mathrm{Recall} = \mathrm{TP}/(\mathrm{TP}+\mathrm{FN})$, 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, and false negatives, respectively.

We implement the fixed-point DT with maximum depth of 10, RF with maximum depth of 10 and 5 estimators, and MLP with a 32×32 hidden layer in 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 [19]. For comparison with the detection



(a) Accuracy (DT). (b) Accuracy (RF). (c) Accuracy (MLP).

(d) Precision (DT). (e) Precision (RF). (f) Precision (MLP).

(g) Recall (DT). (h) Recall (RF). (i) Recall (MLP).

Fig. 3: Detection performance.

performance of the proposed NIDPS, we also evaluate the detection performance of the classifiers implemented in Python language using floating-point representations of the twelve features.

2) Binary Classification Results: Fig. 3 illustrates the impact of the flow observation period t on the detection performance among the DT, RF, and MLP classifiers, where t denotes elapsed time since a new flow was generated. We observe from Fig. 3 that all the classifiers exhibit similar trends: (1) accuracy, precision, and recall improve as the flow observation period t increases and (2) the floating-point classifiers outperform the fixed-point classifiers due to the fixed-point arithmetic errors, except in the recall of the DT classifier. In the DT and RF classifiers, the performance gap between the fixed-point and floating-point classifiers appears when the flow observation period is short. This is because the fixed-point arithmetic errors significantly affect the threshold values of the DT and RF classifiers when the number of incoming packets is small. In the MLP classifier, the fixed-point arithmetic errors have a large impact on the performance because they accumulate across all layer operations. As a result, the fixed-point MLP classifier achieves an accuracy of 95.6% and a precision of 92.0% when $t = 10$, which are 2.77% and 4.95% lower than those of the floating-point MLP classifier, respectively. All fixed-point classifiers achieve high recall values exceeding 95.3% when $t = 10$ while exhibiting high accuracy and precision. These results indicate that the proposed NIDPS can achieve high DDoS detection performance comparable to the existing userspace NIDPSs.

B. Packet Processing Capabilities

1) Evaluation Setup: We use two computation servers: a sender equipped with 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 the NIDPS and L2 forwarding. To fully utilize a single core on the receiver, we merge multiple RX queues into a single queue. We send 64-byte UDP packets from the sender to the receiver over a 100-second interval using `pktgen` tool [23] with 8 CPU cores.

As the evaluation metric, we measure the packet processing rate, defined as the number of packets processed by the receiver per second. To explore distinct packet processing mechanisms, we implement three types of NIDPSs: (1) native XDP, (2) uBPF over AF_XDP, and (3) uBPF over DPDK. We use the uBPF implementation provided in [13] and implement eBPF programs equipped with either the three types of classifiers or the flow extraction. The uBPF runtime is configured to use an LLVM-based backend [12]. For the configurations of the kernel-bypassing frameworks, AF_XDP enables zero-copy mode, `SO_PREFER_BUSY_POLL`, and `XDP_USE_NEED_WAKEUP` and uses 64-packet batches while DPDK is configured with a memory buffer pool of 512 MB.

2) Packet Processing Rate: Fig. 4 depicts a comparison of the packet processing rate among the three packet processing mechanisms and the four NFs (i.e., flow extraction only, DT,

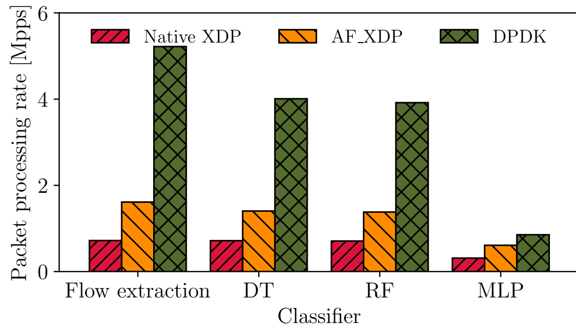


Fig. 4: A comparison of packet processing rate.

RF, and MLP). Their instruction counts are 449, 474, 739, and 5,769 for flow extraction, DT, RF, and MLP, respectively. We observe from Fig. 4 that the uBPF-based NIDPSs have the potential to improve the packet processing rate. The uBPF-based NIDPSs over the AF_XDP data plane achieve packet processing rates of 1.61 Mpps, 1.40 Mpps, 1.38 Mpps, and 0.61 Mpps for flow extraction, DT, RF, and MLP, respectively, which are 2.24×, 1.96×, 1.95×, and 1.95× higher than those of the native XDP-based NIDPSs. The uBPF-based NIDPSs over DPDK data plane further improve the packet processing rates to 5.22 Mpps, 4.01 Mpps, 3.92 Mpps, and 0.85 Mpps for flow extraction, DT, RF, and MLP, respectively, thanks to fully bypassing the kernel network stack.

We also observe that the packet processing rate decreases as the instruction count of NFs increases, regardless of the packet processing mechanisms. Furthermore, we find that the improvement in packet processing rate by using the kernel-bypassing frameworks becomes less significant as the instruction count of NFs increases. More specifically, the uBPF-based NIDPSs over the DPDK data plane exhibit 7.25×, 5.60×, 5.53×, and 2.74× higher packet processing rates for flow extraction, DT, RF, and MLP, respectively, compared to the native XDP-based NIDPSs. These results indicate that the effect of the network I/O overhead reduction becomes less significant as the instruction count of NFs increases. In other words, the ML execution becomes the bottleneck of packet processing.

VI. Conclusion

In this paper, we proposed an ML-based NIDPS, leveraging eBPF and uBPF, to achieve high portability across diverse runtime environments, including native XDP, AF_XDP, and DPDK. Representative results demonstrated that (1) the three fixed-point classifiers achieved high DDoS detection performance comparable to the existing floating-point classifiers while mitigating the calculation errors and (2) the uBPF-based NIDPSs over the AF_XDP and DPDK data planes achieve much higher packet processing rates than the native XDP-based NIDPSs. In future work, we plan to conduct the analysis for feature engineering effects, the multi-class classification evaluation, and the comparative evaluation with signature-based NIDPSs.

References

[1] Y. Ma, L. Liu, Z. Liu, F. Li, Q. Xie, K. Chen, C. Lv, Y. He, and F. Li, "A Survey of DDoS Attack and Defense Technologies in Multiaccess Edge Computing," *IEEE Internet of Things Journal*, vol. 12, no. 2, pp. 1428–1452, Jan. 2025.

[2] N. S. Musa, N. M. Mirza, S. H. Rafique, A. M. Abdallah, and T. Murugan, "Machine Learning and Deep Learning Techniques for Distributed Denial of Service Anomaly Detection in Software Defined Networks—Current Research Solutions," *IEEE Access*, vol. 12, pp. 17 982–18 011, 2024.

[3] 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 EIDWT*, 2020, pp. 458–468.

[4] M. Bachl, J. Fabini, and T. Zseby, "A Flow-Based IDS Using Machine Learning in eBPF," *arXiv preprint*, no. arXiv:2102.09980, Mar. 2022.

[5] H. Taguchi, T. Hara, and S. Kasahara, "Unsupervised Real-Time In-Kernel Intrusion Detection System Using Autoencoders and eBPF," *IEICE Transactions on Communications*, pp. 1–12, 2025.

[6] T. Hara and M. Sasabe, "On Practicality of Kernel Packet Processing Empowered by Lightweight Neural Network and Decision Tree," in *Proc. of International Conference on NoF*, Oct. 2023, pp. 89–97.

[7] S. Kostopoulos, D. Papatsaroucha, I. Kefaloukos, and E. K. Markakis, "eIDPS: A real-time eBPF-based and Machine Learning-powered Network Intrusion Detection and Prevention Solution," in *Proc. of EEITE*, Jun. 2025, pp. 1–8.

[8] A. Nakryiko, "BPF CO-RE reference guide," https://nakryiko.com/posts/bpf-core-reference-guide/, 2018, Accessed 18 Dec. 2025.

[9] Linux Foundation, "Data Plane Development Kit," https://www.dpdk.org/, 2018, Accessed 18 Dec. 2025.

[10] The Kernel Development Community, "AF_XDP — The Linux Kernel documentation," https://docs.kernel.org/networking/af_xdp.html, 2023, Accessed 18 Dec. 2025.

[11] iovisor, "iovisor/ubpf," https://github.com/iovisor/ubpf, 2025, Accessed 18 Dec. 2025.

[12] Y. Zheng, T. Yu, Y. Yang, Y. Hu, X. Lai, D. Williams, and A. Quinn, "Extending Applications Safely and Efficiently," in *Proc. of USENIX Symposium on OSDI*, 2025, pp. 557–574.

[13] Y. Zheng, P. Gavriil, and M. Kogias, "uXDP: Frictionless XDP Deployments in Userspace," in *Proc. of the 3rd Workshop on eBPF and Kernel Extensions* , Sep. 2025, pp. 8–14.

[14] 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 CoNEXT*, Dec. 2018, pp. 54–66.

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

[16] J. E. Varghese and B. Muniyal, "An Efficient IDS Framework for DDoS Attacks in SDN Environment," *IEEE Access*, vol. 9, pp. 69 680–69 699, 2021.

[17] T. Hara and M. Sasabe, "Practicality of In-Kernel/User-Space Packet Processing Empowered by Lightweight Neural Network and Decision Tree," *Computer Networks*, vol. 240, pp. 110 188:1–110 188:18, Feb. 2024.

[18] V. Bode, C. Trinitis, M. Schulz, D. Buettner, and T. Preclik, "Advancing User-Space Networking for DDS Message-Oriented Middleware: Further Extensions," *Pervasive and Mobile Computing*, vol. 107, pp. 102 013:1–102 013:19, Feb. 2025.

[19] H. Benmaghnia, M. Martel, and Y. Seladji, "Code Generation For Neural Networks Based On Fixed-Point Arithmetic ," *ACM Transactions on Embedded Computing Systems*, Sep. 2022.

[20] I. Sharafaldin, A. H. Lashkari, S. Hakak, and A. A. Ghorbani, "Developing Realistic Distributed Denial of Service (DDoS) Attack Dataset and Taxonomy," in *Proc. of ICCST*, Oct. 2019, pp. 1–8.

[21] B. P. Welford, "Note on a Method for Calculating Corrected Sums of Squares and Products," *Technometrics*, vol. 4, no. 3, pp. 419–420, Aug. 1962.

[22] A. Zien, N. Krämer, S. Sonnenburg, and G. Rätsch, "The Feature Importance Ranking Measure," in *Proc. of ECML PKDD*, 2009, pp. 694–709.

[23] The kernel development community, "HOWTO for the linux packet generator — The Linux Kernel documentation," https://docs.kernel.org/networking/pktgen.html, 2025, Accessed 18 Dec. 2025.