Leveraging eBPF/XDP for Real-Time Machine Learning Traffic Classification in 5G User Plane Networks

José Silvestre da Silva Galvão[1], Flávio Geraldo Coelho Rocha[2], Ruan Delgado Gomes[1] and Leandro C. de Almeida[1]

[1]Academic Unit of Informatics – Federal Institute of Paraíba (IFPB)
João Pessoa – PB – Brazil.

[2]Federal University of Goiás (UFG), Goiânia - GO - Brazil

silvestre.galvao@academico.ifpb.edu.br, flaviogcr@ufg.br,

ruan.gomes@ifpb.edu.br, leandro.almeida@ifpb.edu.br

Abstract. *5G networks impose stringent requirements, such as low latency and high throughput, that challenge traditional traffic classification mechanisms within the UPF. These approaches often become bottlenecks due to the overhead and latency of user-space processing. To address the challenges, this work proposes a methodology for real-time traffic classification using In-Kernel Machine Learning with eBPF/XDP. The implementation employs C headers and leverages native eBPF mechanisms to manage verifier constraints. Experimental results demonstrate that In-Kernel ML significantly outperforms user space execution, achieving over 36× faster inference speed, reduced CPU usage from 15.79% to 12.87%, and maintaining 99.91% of accuracy.*

1. Introduction

Fifth Generation (5G) of cellular networks is increasingly being adopted in the industry and in critical environments that require specific requirements, such as high throughput, high device density, driven by the increasing adoption of Internet of Things (IoT) devices, and ultra-low latency applications. To address these demands, 5G networks adopt a service-based architecture (SBA), which is designed to satisfy the diverse and stringent requirements imposed by different services [Chowdhury 2020]. For example, a robotic arm requires ultra-low latency, while a video application demands high throughput. 5G also implements Control and User Plane Separation (CUPS), allowing more efficient and flexible deployment of heterogeneous services. Each part of the SBA consists of Network Functions (NFs), implemented as microservices within the 5G core. In contrast to earlier generations of mobile network architectures, these NFs can be deployed and scaled independently.

Generally, control plane functions handle signaling and resource orchestration, manage user authentication and registration, control mobility, and establish data sessions by defining Quality of Service (QoS) policies. In other words, the control plane makes decisions and establishes network policies, but does not handle user data traffic [European Telecommunications Standards Institute (ETSI) 2024]. On the other hand, the NF responsible for handling user data traffic is the User Plane Function (UPF). The UPF performs packet forwarding, enforcing QoS policies and real-time packet

processing. In addition, it manages data tunneling and routes traffic to the Internet, private networks, or edge computing services.

As new services and applications are developed for 5G (such as robotic arms, augmented reality, and video streaming applications), the demand for enhanced QoS grows, and different policies should be applied for multiple dedicated flows. These scenarios require immediate responses, which need strict latencies, while the UPF must handle increasingly heavy traffic and diverse application profiles. However, the capacity of current UPF implementations to guarantee sufficient QoS remains largely unexplored [Siegmund et al. 2026]. As a result, traditional QoS and traffic classification mechanisms, which rely on static policies, predefined policies or computationally expensive Deep Packet Inspection (DPI) techniques, are becoming slow and insufficient for emerging applications [Fei et al. 2023].

In this scenario, technologies such as extended Berkeley Packet Filter (eBPF) can be used to solve problems related to inefficient real-time traffic classification. eBPF emerges to enable secure execution of programs within the kernel space, without requiring source code modifications or system reboots [Rice 2023]. Combined with that, the eXpress Data Path (XDP) enhances network performance by enabling early packet processing in the network stack, reducing latency and system overhead. It allows packets to be analyzed and acted upon at line rate, making it suitable for latency-sensitive and high-throughput 5G scenarios [Siegmund et al. 2026]. Consequently, the adoption of eBPF/XDP in 5G networks can enable the deployment of customizable and high-performance data-plane functions, leveraging the benefits of programmability, isolation, and efficiency without requiring modifications to existing 5G Core components. As a result, UPF and other user plane elements can be extended with advanced capabilities for specifics tasks, such as real-time traffic classification.

Despite the capabilities introduced by eBPF/XDP, there is a growing need for adaptive traffic classification mechanisms that go beyond the limitations of static heuristics and preconfigured rules. Performing this classification at an early stage allows critical decisions to be made before traffic fully enters the UPF stack, and together with prioritization, it can improve traffic resilience by ensuring predictable service degradation under heavy loads [Siegmund et al. 2026]. This approach can reduce processing overhead, memory copy operations, and end-to-end latency. In this context, intelligent classifiers, based on Machine Learning (ML), have emerged as a solution, capable of identifying complex traffic patterns in real-time [Zhang et al. 2024], analyzing flow behaviors rather than relying solely on packet headers.

This capability could be particularly relevant in 5G scenarios, where traffic heterogeneity and stringent latency requirements limit the effectiveness of static classification mechanisms. However, the traditional ML-based classification process is slower because it runs in user space. Figure 1 shows the main bottleneck of traditional user space classification, where packets must traverse multiple layers, incurring switch context and copy data operations. In scenarios such as 5G networks, characterized by increasing traffic loads, this architecture can introduce latency and overhead, resulting in slow classification and slow decisions. In this context, our aim is to answer the following question:

"*Is it possible to execute a lightweight classification model within a privileged eBPF/XDP environment, enabling a form of in-kernel ML in the 5G UPF*?".

The embedding of intelligence directly into the datapath can facilitate new forms of in-network ML. Furthermore, eBPF and ML enable novel strategies not foreseen by 3GPP specifications, such as behavior-driven dynamic QoS.

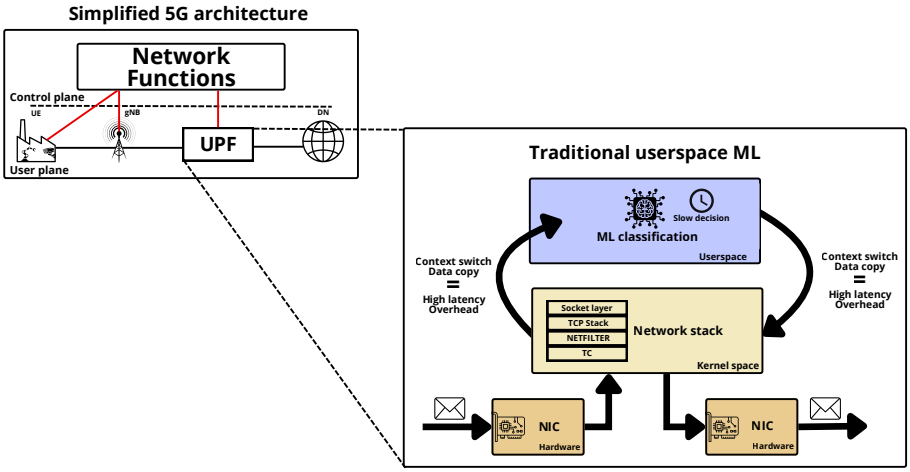


Figure 1. Traditional user space classification in the 5G UPF forces network traffic to traverse between kernel and user space, introducing CPU overhead, and high latency due to repetitive context switching and data copying required by network stack .

Given the aforementioned possibilities of integrating eBPF with ML, this work uses a methodology for training a model using data extracted with eBPF in a 5G environment, featuring traffic patterns typical of Enhanced Mobile Broadband (eMBB) and Ultra-Reliable Low Latency Communication (URLLC) demands. Preliminary results indicate that in-kernel classification outperforms user space classification, showing an absolute CPU usage of 12.87%, compared to 15.79% in user space, and achieving a classification speed more than 36$\times$ faster. The contributions of this study are as follows: i) we design and implement a real-time ML traffic classification for 5G UPF leveraging eBPF/XDP; ii) we adapted the inference logic to satisfy the eBPF verifier, implementing a majority voting scheme and tail calls and iii) we conduct an in-depth experimental evaluation comparing in-kernel and user-space traffic classification.

The remainder of this article is organized as follows: Section 2 reviews related works. Sections 3 and 4 describe the proposal and evaluation, respectively. Section 5 discusses possible use cases and deployment perspectives of the proposed approach. Finally, Section 6 concludes the study.

2. Related Work

In this Section, we describe related works focused on developing solutions for 5G networks, traffic classification, and the use of machine learning models within the Linux kernel with eBPF.

The work described in [Gallego-Madrid et al. 2024] proposes an in-kernel traffic classification approach using eBPF/XDP and TinyML neural networks to secure resource-constrained IoT devices. To enable Multilayer Perceptron (MLP) execution, the authors

use the `emlearn` library to port the model to eBPF, bypassing verifier restrictions through fixed-point arithmetic and loop unrolling. Using eBPF maps for result aggregation, the system reduces inference time by over 95% compared to user space execution, validating TinyML's efficiency in eBPF. While their primary focus was demonstrating feasibility and performance impact rather than model accuracy, our study aims to achieve high classification accuracy while operating under eBPF constraints.

In a 5G context, the work described in [Kawasaki et al. 2023] investigates the use of eBPF to predict failures caused by packet losses and CPU overhead within the 5G Core.The authors developed a Long Short-Term Memory (LSTM)-based model trained on kernel metrics extracted via eBPF tools and compared its performance against models trained using container metrics captured by `cAdvisor` and application logs from `5G-exporter`. The study showed better results with the model trained on eBPF captured data and the use of fine-grained observability provided by eBPF can enhance the performance of machine learning models. Although that work uses eBPF for data collection, it relies on pre-existing BPF Compiler Collection (`BCC`) tools. In contrast, our work features a custom code to extract the specific data required for our study.

In [Bachl et al. 2021], the authors demonstrate the feasibility of implementing a flow-based intrusion detection system in-kernel using eBPF, employing decision tree models to classify malicious traffic. To circumvent eBPF limitations, specifically the lack of support for floating-point operations, the authors utilized 64-bit fixed-point arithmetic. In addition, they used hash tables to maintain state and calculate flow statistics, such as average packet size. The experimental results showed that this approach outperformed an equivalent user space implementation by more than 20%. In contrast to the approach used in our work, the authors neither employed *emlearn* to export their models nor included an evaluation of a Random Forest model.

The work described in [Zhang et al. 2024] proposes a real-time intrusion detection architecture using in-kernel Deep Neural Networks (DNNs) via eBPF/XDP. To overcome eBPF verifier constraints, such as instruction limits and the lack of floating-point support, the authors employ 32-bit integer (int32) and chained tail calls. Their model achieves inference times of 3000–5000 ns per flow and F1-scores up to 0.992. In contrast, our work distinguishes itself by using a lightweight Random Forest (RF) model for traffic classification, utilizing the `emlearn` library alongside chained tail calls for efficient execution within the privileged kernel context.

The authors in [Monteiro and Sousa 2024] propose an Intrusion Detection System (IDS) for port scanning that leverages XDP hardware offload in SmartNICs. Using the `emlearn` library, they implement a RF model that operates partially in hardware, achieving 605 kpps in offload mode,significantly outperforming traditional solutions like Snort (51 kpps). While [Monteiro and Sousa 2024] relies on a hybrid architecture to distribute the processing load, our work focuses entirely on in-kernel execution using standard CPUs.

Based on the conducted literature review, all presented studies propose solutions to address the limitations of eBPF when deploying traffic classification models, focusing mainly on failure and anomaly detection. Various approaches have been introduced, employing different models and leveraging specific eBPF features. However, this work

extends beyond the scope of failure and anomaly detection by proposing the classification of traffic from diverse services in a 5G network.

Table 1 summarizes the conducted literature review and highlights the key differences compared to the work described in this article.

Table 1. Related work

Article	In-kernel classification	Random Forest	`emlearn`	eBPF data collection	5G traffic
[Gallego-Madrid et al. 2024]	●	○	●	○	○
[Bachl et al. 2021]	●	○	○	○	○
[Zhang et al. 2024]	●	○	○	●	○
[Monteiro and Sousa 2024]	●	●	●	○	○
[Kawasaki et al. 2023]	○	○	○	●	●
This article	●	●	●	●	●

3. Proposal

This section presents the proposed methodology, detailing the adopted experimental topology, the data extraction process using eBPF/XDP, and the training and integration of the machine learning model for traffic classification in both kernel and user space, as illustrated in Figure 2. Furthermore, the limitations imposed by these technologies are discussed, along with the strategies developed to overcome these constraints.

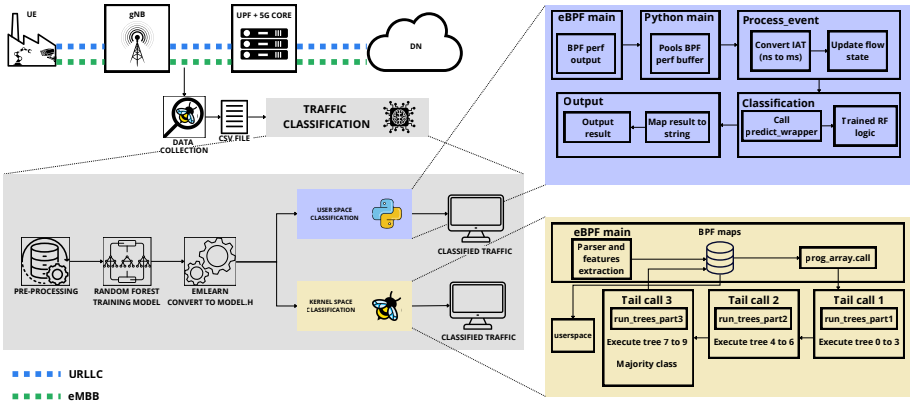


Figure 2. Diagram of the proposed model implementation pipeline. The data is captured, an RF model is trained and then converted with `emlearn`. For comparision, both traditional user space and in-kernel classification are deployed

3.1. Data Collection

As depicted in the Figure 2, traffic data collection was implemented using eBPF, with the program attached to the XDP hook to ensure high-performance packet processing before packets enter the operating system network stack. The `BCC` framework was used for the development of the monitoring code and also for all eBPF codes in this work. The developed code enabled the extraction of the features such as source/destination IP, source/destination port, protocol, Inter-Arrival Time (IAT), and other TCP header fields.

For the IAT feature, a specific logic was implemented within the eBPF code to perform the calculation and transmit the result, along with other features, to the user space. Performing this calculation in kernel space minimizes resource overhead and ensures high-speed execution, as the timestamp does not need context switching. Figure 3 illustrates the IAT calculation process and the general structure of the code.

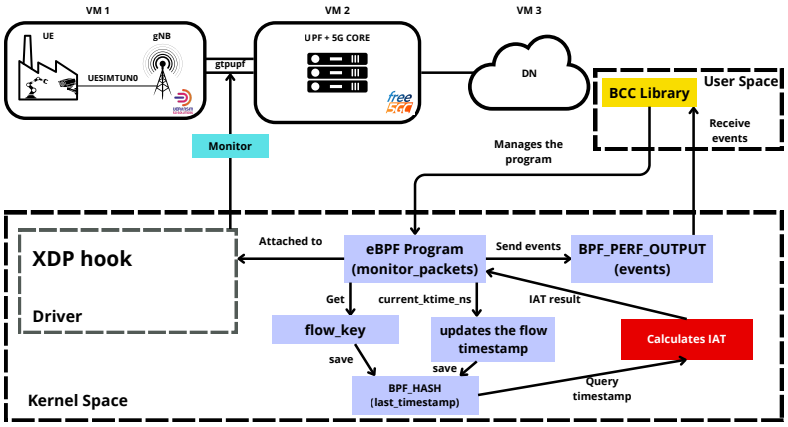


Figure 3. In the eBPF/XDP monitoring logic, the traffic traversing the UPF interface is captured at the driver layer with XDP hook, where the eBPF program identifies flows, updates informations in a BPF hash map and send the data to the user space.

The code is executed on the 5G UPF, monitoring the communication interface between UPF and gNB. The eBPF program is attached to the XDP hook, intercepting packets at the lowest possible level, within the network driver, before the operating system allocates standard kernel data structures[Vieira et al. 2020]. The main function of the program (`monitor_packets`) analyzes the incoming packets, extracts the `flow_key_ts` (source/destination IP, source/destination port, and protocol) to identify the flow, and captures the packet arrival time. These details are stored in a key/value structure (`BPF_HASH` map), which is required to retrieve the timestamp of the last packet in the same flow, allowing the calculation.

The processed data are submitted to `BPF_PERF_OUTPUT`, a ring buffer that transfers data from the kernel to the user space. The user space component manages the eBPF cycle, compiling the C code and loading it into the kernel. In addition, it consumes data from the output buffer, structuring the collected information, and saving it to a file for subsequent model training.

3.2. Machine Learning Model Implementation

The generated dataset was used to train a Random Forest (RF) model using `scikit-learn`. The classifier was implemented using a majority voting scheme, since a probabilistic approach requires floating-point numbers that are not supported in the kernel [Vieira et al. 2020]. The model was trained with the following hyperparameters: a `max_depth` of 10, `n_estimators` of 10, and `min_sample_leaf` of 5, using an 80/20 train/test split. These hyperparameters were specifically selected to limit the structural complexity of the resulting decision trees. Deeper trees or larger forests would increase the number of conditional instructions, making the code impossible to load due to strict complexity limits imposed by the eBPF verifier.

Data labeled 0 corresponds to URLLC, while label 1 corresponds to eMBB traffic. The features used for training were derived to characterize the statistical pattern of the IAT in a flow. Initially, the mean and variance of the IAT were calculated to capture the central tendency and dispersion of packet arrivals. After that, these same metrics were computed for the preceding 500 ms window. The current and past statistics allow the model to analyze the temporal evolution of the flow. These four features and the label were used for the training.

Executing ML models within the Linux kernel with eBPF has significant restrictions: i) there is no access to standard C external libraries; ii) memory and instruction counts are strictly limited; iii) eBPF verifier prohibits infinite loops or excessive complexity; iv) finally, the lack of support for floating-point arithmetic [Gallego-Madrid et al. 2024]. `Emlearn` is a Python library that acts as a bridge, converting the Python-trained model into a C header file (.h) that implements the tree logic using static data structures and conditional logic (`if-else`). Using this library, the problems iii and iv can be addressed.

Despite the simplicity of logic, some modifications were necessary. First, the standard inclusion of `#include <stdint.h>` had to be replaced with `#include <linux/types.h>`, along with the following definitions of data types compatible with the eBPF kernel environment:

```
typedef s64 int64_t;
typedef s32 int32_t;
typedef s16 int16_t;
typedef s8 int8_t;
```

Second, within the main function (`model_predict`), the array initialization `int32_t votes[3] = {0,};` was replaced by explicit element-by-element assignment:

```
int32_t votes[3];
votes[0] = 0;
votes[1] = 0;
votes[2] = 0;
```

This modification was necessary because aggregate initialization was causing the compiler to generate implicit calls to external functions (restriction i) such as `memset` to clear the memory. As previously mentioned, standard library functions are unavailable in the restricted eBPF environment, which leads to verification failures. Thus, explicit assignment was the solution to this problem. The `model_predict_proba` function was entirely removed, as it caused compilation errors due to the use of floating-point arithmetic. This removal does not impact the functionality of the system, as the classification process relies solely on the `model_predict` function.

Upon adapting the C header, it was integrated into the eBPF code. The code originally used for data collection was modified to implement the random forest structure. One of the constraints of eBPF is the strict limit on the number of instructions per program. Despite constraining the model size with hyperparameters, the eBPF verifier still rejected the structure due to the excessive instruction count. The algorithm 1

Algorithm 1 In-kernel RF classifier

Require: Features Array F
Require: Global Scratchpad S (Persists across tail calls)
Ensure: Predicted Class C_{final}
Initialization
$S.Features \leftarrow F$
$S.Votes \leftarrow [0, 0, 0]$ ▷ Reset vote count for the 3 classes
TAILCALL(Model_Part1)
Part 1, 2 and 3
function MODEL_PART(tree_indices, next_program)
    $F \leftarrow S.Features$
    $V \leftarrow S.Votes$
    for $tree\_id \in \{tree\_indices\}$ **do**
        $prediction \leftarrow Tree_{tree\_id}(F)$
        $V[prediction] \leftarrow V[prediction] + 1$
    end for
    $S.Votes \leftarrow V$
    TAILCALL(next_program)
end function
Part 3: Final voting ▷ The previous function is performed without tail call other programs
function MODEL_PART3
    $F \leftarrow S.Features$
    $V \leftarrow S.Votes$
    $max\_votes \leftarrow -1$
    $C_{final} \leftarrow -1$
    for $class\_id \in \{0, 1, 2\}$ **do**
        if $V[class\_id] > max\_votes$ **then**
            $max\_votes \leftarrow V[class\_id]$
            $C_{final} \leftarrow class\_id$
        end if
    end for
    return C_{final}
end function

describes the classification process and how the eBPF restriction ii were addressed using the tail calls to divide the trees structure.

Tail calls are eBPF features that allow the chaining of multiple programs, allowing execution to be transferred from one program to another [Rice 2023]. Through this mechanism, the RF classifier was divided into three different eBPF programs, each responsible for evaluating a subset of the model's trees, as shown in Algorithm 1. The process begins with initialization, in which the features are loaded onto the scratchpad. Then the execution is passed to `Model_Part1`, which evaluates trees 0 to 3 and updates the global vote count. It then calls `Model_Part2`, which evaluates trees 4 to 6. Finally, `Model_Part3` evaluates trees 7 to 9. Instead of calculating probabilities, Part 3 performs the integer majority check to determine `C_final`. This approach reduces

the complexity of each individual program, complies with the verifier constraints, and maintains classification entirely within kernel space, thereby avoiding unnecessary data copies to user space and preserving real-time performance.

Figure 4 illustrates the ML execution flow using eBPF maps. Initially, a key-value `BPF_HASH` map stores flow statistics indexed by the 5-tuple, from which the model features are derived. Next, the `PROG_ARRAY` map acts as a chaining table for tail calls, enabling the sequential execution of multiple eBPF programs. This mechanism uses an integer index to dynamically select the next program, decomposing the inference pipeline into smaller stages. Finally, a `PERCPU_ARRAY` map, indexed by a fixed integer, serves as per-CPU context storage. It maintains intermediate inference data, including the feature vector and tree vote accumulator, which are shared among programs throughout the execution chain.

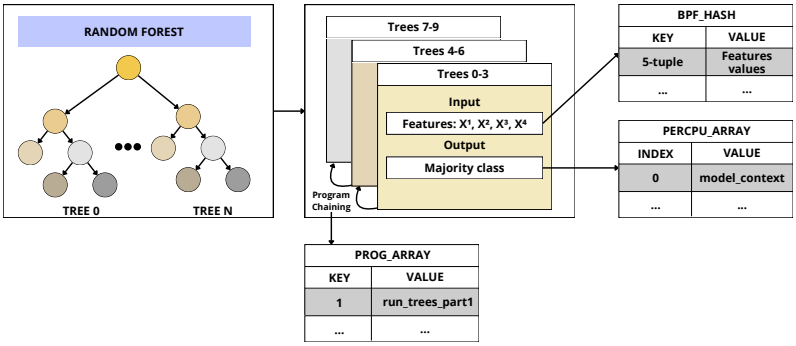


Figure 4. Representation of how random forest model is executed using eBPF maps.

For the user space model application, the entire classification complexity was shifted to the Python environment. The eBPF code functions solely as a metric collector, as shown in Figure 3, gathering packet metadata, calculating the IAT, and transmitting the data to the user space. The Python component handles the pipeline management, reconstructing flow states based on received events.

Unlike the kernel implementation, no modification to the original `emlearn` model was necessary for user space classification, as kernel restrictions do not apply in this context. However, since Python cannot directly execute a raw C header, an additional compilation step was required. The header containing the model was encapsulated within a C wrapper and compiled into a shared library. This library is dynamically loaded into Python via the `ctypes` interface, enabling the classifier to execute as native code while the remainder of the pipeline operates in user space. This approach facilitates efficient integration between eBPF, C and Python, ensuring fidelity to the original model and serving as a robust baseline for direct comparison with the kernel space implementation. All codes used in this work are available on GitHub[1].

3.3. Experimental Environment

The proposed experimental environment shown in Figure 5 consists of three Virtual Machines (VMs) running Ubuntu 20.04.6 LTS (kernel 5.4.0-216-generic, important to

[1]`https://github.com/jose-galvao/eBPF-Classification`

run GTP5G kernel module). Each VM uses 4 vCPUs and 6 GB of vRAM. The topology is divided into the following roles: VM 1 executes UERANSIM[2] to simulate both User Equipment (UE) and gNB. VM 2 executes Free5GC[3], running the 5G Core components, including the UPF, which is a central component of this study. Finally, VM 3 acts as the Data Network (DN), serving as the final destination for traffic generated by the UE.

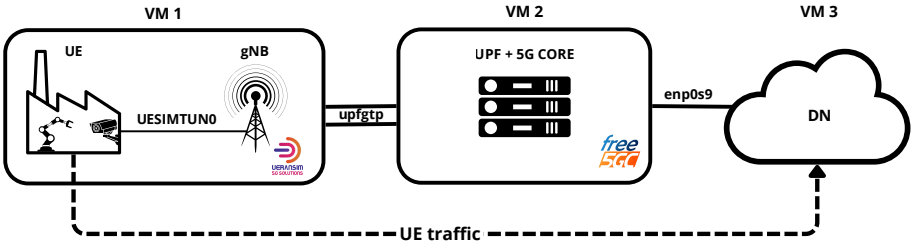


Figure 5. The experimental 5G environment illustrates the data plane path. UE traffic generated in VM 1 traverses the RAN and 5G CORE (VM 2) before reaching the DN in VM 3, establishing an end-to-end communication for analysis.

Communication between VM 1 and VM 2 occurs via GTP tunnel established over the logical interface `upfgtp`. For traffic generation, an uplink scenario was configured. VM 1 hosts an adaptive video streaming representing an eMBB application and an URLLC traffic generator simulating robotic arm control traffic [Kalør et al. 2018]. The destination acts like a client, consuming the eMBB and receiving the URLLC data transmitted across the simulated 5G infrastructure.

4. Evaluation

The experimental validation methodology was conducted following the same scenario that was depicted in Figure 2. On the left side of the diagram, the 5G UEs generate two distinct traffic classes: URLLC (blue), characterized by low latency and high reliability, and eMBB (green), characterized by high data rates. In the uplink, these flows traverse the gNB and the UPF towards the DN. The classification logic was deployed in two distinct modes, as detailed in Section 3.

With the exported model ready for deployment, two scenarios were defined to evaluate classification performance in terms of accuracy, per-flow window classification time, CPU usage, and context switching. In the first scenario, the model operates within the kernel space; in the second, classification is performed in the user space.

Following training with scikit-learn, the model achieved an accuracy of 99.97%. Subsequently, the model was converted and ported to the eBPF environment using `emlearn` as described in Section 3. When deployed in the eBPF environment, the model achieved an accuracy of 99.60%, representing only a 0.37% reduction compared to the original training. This difference arises because the eBPF environment lacks support for floating-point operations. Consequently, the model must be converted to use only integer values, which results in a loss of precision.

The average per-flow window classification time within eBPF was 0.9046µs, demonstrating the XDP capacity to perform inference with extremely low latency. This

[2]`https://github.com/aligungr/UERANSIM`
[3]`https://free5gc.org`

result is an advantage for URLLC traffic, which demands low latency. In such scenarios, any optimization that minimizes processing overhead is important for the objectives imposed by the 3GPP standards.

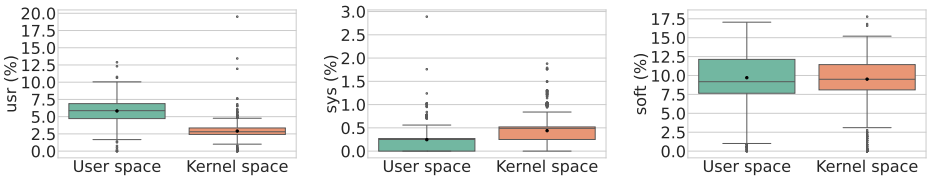
The user space classifier achieved better accuracy, but with a minimal difference compared to the kernel space implementation, because the user space does not restrict the use of floating-point arithmetic. However, an analysis of classification time and resource consumption shows higher values compared to those observed in kernel space. The classification yielded an accuracy of 99.91%, representing a 0.31% increase over the previous scenario.

In contrast, the classification latency increased 36.35×, resulting in an average time of 32.88 µs. An increased classification latency can be unacceptable in time-critical applications that demand real-time responsiveness. In applications with ultra-low latency requirements, such as in URLLC scenarios, a high delay in the classification process can significantly increase end-to-end network latency, compromising the ability to meet reliability and response time requirements.

Another important metric to be analyzed is the CPU usage. Figure 6 shows the CPU usage during in-kernel and user space classification. Total usage is categorized into `%usr`, `%sys`, and `%soft` metrics, which represents different types of processor load:

- **`%usr` (user CPU time)**: Time spent executing process in user space, such as applications, scripts, and services when running outside the kernel;
- **`%sys` (system CPU time)**: Time spent executing system calls and kernel routines, including network management, socket handling, interrupts, and internal scheduling;
- **`%soft` (softirq time)**: Time dedicated to soft interrupts, which are lightweight interrupt routines used particularly by network subsystems.

These three metrics indicate how the system distributes the workload across execution layers.



(a) CPU usage comparison: `%usr` (b) CPU usage comparison: `%sys` (c) CPU usage comparison: `%soft`

Figure 6. CPU usage for in-kernel and user space classification

Figure 6(a) indicates that the `%usr` metric shows a significantly higher processing load for the user space approach, exhibiting a mean consumption of 5.83% with a distribution concentrated between 5% and 7%. In contrast, the kernel space implementation reduces this demand, showing a mean of 2.91%, indicating greater efficiency by processing tasks in the lower layers of the operating system. However, it should be noted that the kernel space scenario presents a higher dispersion of outliers in the `%usr` metric, suggesting peaks of user activity, possibly related to `BCC` control and management tasks.

Regarding the `%sys` metric, the opposite relation is observed. The user space scenario maintains lower system consumption compared to the kernel space scenario, with averages of 0.25% and 0.44%, respectively. In the kernel space scenario, a rise in both the mean value and the associated variability is observed. Although the percentage rise is small, it confirms that the workload has been effectively transferred to the operating system kernel, a typical characteristic of eBPF/XDP-based solutions.

Finally, Figure 6(c) shows the `%soft` metric, evidencing that network packet processing is the most resource-intensive component in both scenarios. Both approaches present high averages: 9.71% for user space and 9.52% for kernel space. Additionally, total CPU usage for user space was higher, reaching 15.79%, while kernel space reached 12.87%. These increases in average classification time and CPU usage are attributed to the requirement for packets to traverse the full kernel network stack pipeline.

Another important point to observe is the context switching, shown in Figure 7. The user space scenario shows a higher volume of context switching, with a mean value of 5736.07. Furthermore, the distribution is characterized by high variability, evidenced by a wide Interquartile Range (IQR) and outliers reaching up to 12000 context switches. This behavior shows that continuous transfers of packets between the kernel and the Python classifier in user space create an overhead that could significantly affect performance. In the kernel space scenario, the average number of context switches was 3452.09. The distribution is compact with a narrow IQR, indicating a stable behavior. This reduction shows that performing classification within the kernel reduces the costly overhead associated with traversing packets from the kernel to the user space, which also reduces CPU consumption, as shown in Figure 6.

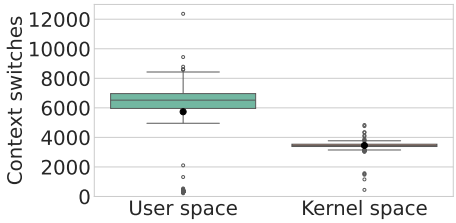


Figure 7. Context switching during classification tasks.

5. Discussion about possible use cases

According to the results described in the previous section, it is evident that kernel space classification demonstrates performance advantages compared to the user space approach. Therefore, it is possible to envision potential applications for deploying intelligent classification mechanisms directly within the network data plane. Specifically, we conjecture that implementing this approach within the 5G UPF could be an alternative to enhance packet processing efficiency, minimize latency, and optimize resource utilization, particularly in scenarios demanding high performance and low latency.

The standard kernel network stack involves multiple processing stages. In virtualized UPF solutions, such as Free5GC or Open5GS[4], packets follow a strictly defined path: The packet is received at user plane level, goes through GTP-U

[4]`https://open5gs.org`

decapsulation, and is subsequently matched against Packet Detection Rules (PDR), Forwarding Action Rules (FAR), and QoS Enforcement Rules (QER). Based on these rules, the UPF makes forwarding decisions, applies QoS policies, and finally executes packet transmissions. Although functional, this pipeline incurs significant computational overhead, particularly in scenarios characterized by high traffic volumes or latency-sensitive applications. Related work, such as [Zhou et al. 2023] and [do Amaral et al. 2021] used eBPF to reduce overhead and accelerate the UPF. The introduction of an early classification mechanism within the kernel with eBPF/XDP enables the shifting of specific logic to a stage preceding standard UPF processing. This architectural change can yield direct performance enhancements.

In addition, it becomes possible to redirect specific packets, allowing them to bypass the entire UPF pipeline. For instance, flows associated with latency-sensitive applications, such as URLLC, AR/VR, or industrial applications, can be steered directly to a local processing point or specific interfaces to be forwarded, reducing processing overhead. This approach minimizes memory copy operations, reduces the time packets spend within the system, and contributes to a lower overall latency.

In addition to early packet redirection, the eBPF/XDP classification enables the separation of flows into distinct priority classes while still within the kernel, before entering the full UPF logic. Consequently, sensitive packets can be directed to high-priority queues; otherwise, traffic is handled with lower priority.

In other scenarios, in-network classification can serve as a mechanism to support the dynamic definition of QoS parameters. Automatic identification of video streams, gaming traffic, or any other critical application could facilitate the dynamic association of QoS Flow Identifiers (QFI) or even trigger resource allocation across other NFs.

6. Conclusion

The main objective of this study was to determine if it is possible to execute a lightweight classification model within a privileged eBPF/XDP environment, enabling a form of in-kernel ML in the 5G UPF. The results indicate that in-kernel classification is possible and significantly outperforms user space execution, achieving an inference speed more than 36$\times$ faster while reducing total CPU consumption to 12.87% compared to 15.79% in user space. The use of tail calls with `emlearn` library proved to be an effective strategy for overcoming eBPF verifier constraints, enabling intelligent processing with minimal impact on model accuracy.

As a future perspective, an evolving classification mechanism is proposed, integrated with an active traffic management system capable of identifying and prioritizing flows in real-time. By integrating the classifier with XDP redirection actions, it will be possible to steer critical traffic, such as URLLC, to high-priority queues before standard kernel processing occurs. We conjecture that this enables the implementation of dynamic and autonomous QoS in distributed UPFs, ensuring the strict latency requirements necessary for industrial applications in private 5G networks.

Acknowledgments

This work is supported by EMBRAPII (BFA 2301.0001) and the companies Cisco, Prysmian, and MPT Cable. The authors also thank CPQD, Inatel, Taggen, Data Machina,

CNPq (307108/2025-2, 404509/2025-8), IFPB, and the IFPB Innovation Hub.

References

Bachl, M., Fabini, J., and Zseby, T. (2021). A flow-based ids using machine learning in ebpf.

Chowdhury, A. (2020). The impact of 5g network on industry 4.0.

do Amaral, T. A. N., Rosa, R. V., Moura, D. F. C., and Rothenberg, C. E. (2021). An in-kernel solution based on xdp for 5g upf: Design, prototype and performance evaluation. In *2021 17th International Conference on Network and Service Management (CNSM)*, pages 146–152.

European Telecommunications Standards Institute (ETSI) (2024). 5g; system architecture for the 5g system (5gs) (3gpp ts 23.501 version 18.5.0 release 18).

Fei, X., Martins, P., and Lu, J. (2023). Real-time traffic classification for 5g nsa encrypted data flows with physical channel records. In *IEEE VTC 2023-Fall*.

Gallego-Madrid, J., Bru-Santa, I., Ruiz-Rodenas, A., Sanchez-Iborra, R., and Skarmeta, A. (2024). Machine learning-powered traffic processing in commodity hardware with ebpf. *Computer Networks*, 243:110295.

Kalør, A. E., Guillaume, R., Nielsen, J. J., Mueller, A., and Popovski, P. (2018). Network slicing in industry 4.0 applications: Abstraction methods and end-to-end analysis. *IEEE Transactions on Industrial Informatics*, 14(12):5419–5427.

Kawasaki, J., Koyama, D., Miyasaka, T., and Otani, T. (2023). Failure prediction in cloud native 5g core with ebpf-based observability. In *2023 IEEE 97th Vehicular Technology Conference (VTC2023-Spring)*, pages 1–6.

Monteiro, J. and Sousa, B. (2024). ebpf intrusion detection system with xdp offload support. In *2024 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–6.

Rice, L. (2023). *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security*. O'Reilly Media, 1 edition.

Siegmund, F., Kundel, R., Meuser, T., and Steinmetz, R. (2026). User plane performance in beyond 5g networks: Comprehensive analysis and evaluation. *Computer Communications*, 247:108397.

Vieira, M. A. M., Castanho, M. S., Pacífico, R. D. G., Santos, E. R. S., Júnior, E. P. M. C., and Vieira, L. F. M. (2020). Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv.*, 53(1).

Zhang, J., Chen, P., He, Z., Chen, H., and Li, X. (2024). Real-time intrusion detection and prevention with neural network in kernel using ebpf. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 416–428.

Zhou, J., Ma, Z., Tu, W., Qiu, X., Duan, J., Li, Z., Li, Q., Zhang, X., and Li, W. (2023). Cable: A framework for accelerating 5g upf based on ebpf. *Computer Networks*, 222:109535.