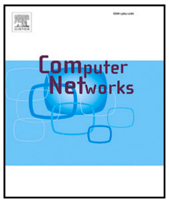
Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



# Machine learning-powered traffic processing in commodity hardware with eBPF

Jorge Gallego-Madrid, Irene Bru-Santa, Alvaro Ruiz-Rodenas, Ramon Sanchez-Iborra *, Antonio Skarmeta

*Department of Information and Communication Engineering, University of Murcia, 30100, Murcia, Spain*

ARTICLE INFO

*Keywords:*
eBPF
Machine learning
Edge computing
Linux
Internet of things

ABSTRACT

Network softwarization is paving the way for the design and development of Next-Generation Networks (NGNs), which are demanding profound improvements to existing communication infrastructures. Two of the fundamental pillars of NGNs are flexibility and intelligence to create elastic network functions capable of managing complex communication systems in an efficient and cost-effective way. In this sense, the extended Berkeley Packet Filter (eBPF) is a state-of-the-art solution that enables low-latency traffic processing within the Linux kernel in commodity hardware. When combined with Machine Learning (ML) algorithms, it becomes a promising enabler to perform smart monitoring and networking tasks at any required place of the fog-edge-cloud continuum. In this work, we present a solution that leverages eBPF to integrate ML-based intelligence with fast packet processing within the Linux kernel, enabling the execution of complex computational tasks in a flexible way, saving resources and reducing processing latencies. A real implementation and a series of experiments have been carried out in an Internet of Things (IoT) scenario to evaluate the performance of the solution to detect attacks in a 6LowPAN system. The performance of the in-kernel implementation shows a considerable reduction in the execution time (-97%) and CPU usage (-6%) of a Multi-Layer Perceptron (MLP) model in comparison with a user space development approach; thus positioning our proposal as a promising solution to embed ML-powered fast packet processing within the Linux kernel.

## 1. Introduction

The advent of Next-Generation Networks (NGNs) will demand significant improvements to communication infrastructures in terms of speed, flexibility, intelligence, and latency. This will enable the further development of novel use cases through all verticals [1]. To cope with this explosion, multiple network and traffic management solutions are currently emerging as the complexity for coordinating the different network segments composing the disaggregated Beyond-5G (B5G) architecture will notably increase [2]. Besides, the transition to this new generation of networks poses a significant challenge due to the necessary investment in new hardware, software, and infrastructure.

Advanced and novel technologies will be needed to provide flexibility and intelligence to NGN infrastructures by adopting a network-softwarization approach. A state-of-the-art technology in this regard is the extended Berkeley Packet Filter (eBPF), which allows efficient traffic processing in commodity hardware by enabling the safe execution of code inside the Linux kernel [3]. This is of prominent importance given the capabilities enabled to equipment not specialized in networking tasks, concretely in scenarios considering the fog-edge-cloud continuum. With the help of Machine Learning (ML) techniques, eBPF is an enabler for NGNs to perform intelligent networking and monitoring tasks at any point of the infrastructure, which is a fundamental pillar to manage the expected high-throughput and low latency traffic that will be generated by new services and applications. Besides, this technology is also useful for security purposes, as it permits fast traffic inspection to detect attacks or intrusions in real-time, also crucial for NGNs as new attack vectors will appear within upcoming infrastructures. It is well-known that ML is one of the key technologies to provide intelligent decision making to the management and orchestration of the network [4]. It enables network devices to automatically adapt to the changing network conditions given that ML-powered mechanisms can detect anomalies, predict network behavior, or anticipate failures and bottlenecks [5]. In this way, this proactivity can be used to automate network management tasks to optimize the networks in real time without human intervention [6].

* Corresponding author.
*E-mail addresses:* jorgegm@um.es (J. Gallego-Madrid), ireneleonor.brus@um.es (I. Bru-Santa), a.ruizrodenas@um.es (A. Ruiz-Rodenas), ramonsanchez@um.es (R. Sanchez-Iborra), skarmeta@um.es (A. Skarmeta).

https://doi.org/10.1016/j.comnet.2024.110295
Received 6 September 2023; Received in revised form 20 February 2024; Accepted 27 February 2024
Available online 28 February 2024

The synergies between eBPF and ML have been explored in the literature in recent years [7,8]. Their convergence makes possible to perform fast packet processing in a proactive and intelligent way, automatically adapting the network functions to the requirements demanded by services and applications, leading to an overall improvement in network performance, flexibility, and reliability. Typically, they are used together in the following way: eBPF is in charge of collecting data from traffic-flows at Linux's kernel level, and ML models in Linux's user plane analyze that information and make predictions or decisions. However, decoupling traffic handling and ML processing is not the best approach as performing computation tasks in the user space presents reduced performance in comparison with a complete in-kernel implementation. Other implementation alternatives such as Data Plane Development Kit (DPDK) and AF_XDP also aim at improving the performance of data traffic processing [9,10]. AF_XDP is an extension of the Linux XDP that permits bypassing the Linux kernel network stack and, therefore, it enables the straight delivery of raw packet data from the NIC to user space avoiding its copy. Nevertheless, to implement this zero-copy mode with Direct Memory Access, the NIC driver must support the *XDP_REDIRECT* action, which might not be available in inexpensive and constrained hardware. In turn, DPDK is a framework for fast packet processing in data plane applications that enables more efficient computing than the traditional interruption scheme available in the Linux kernel. However, the development and maintenance of DPDK applications is not trivial and it needs specific equipment usually not available in commodity hardware.

In this context, as the main contribution of this work, we present a solution that integrates intelligent traffic inspection models within the Linux kernel, leveraging the capabilities provided by eBPF to combine fast packet processing and ML-based intelligent decision-making at the same level. This strategy permits to save resources in the device performing such computation and notably reduces processing latencies which, as aforementioned, is highly relevant in edge-enabled B5G systems. Concretely, to the authors' knowledge, this is the first work that presents a functional implementation of a neural network model (Multi Layer Perceptron, MLP) within the Linux kernel for packet processing purposes. By doing so, we provide intelligence to the packet processing capabilities of eBPF, enhancing the performance and reducing the latency of this task in a flexible and lightweight way. This proposal is validated in an Internet of Things (IoT) scenario, in which a resource-constrained device at the edge is able to efficiently handle a notable traffic load aiming at detecting a cyber-attack through ML processing.

The rest of the paper is organized as follows. Section 2 provides background regarding works exploiting both eBPF and ML algorithms. Section 3 presents the system architecture, as well as the design and implementation details of the proposed solution. In Section 4, we discuss and analyze the results obtained in the conducted experiments. Finally, Section 5 concludes the paper and introduces future research lines.

## 2. Background

eBPF is a technology integrated into the Linux kernel with the ability to enable sandboxed programs to run in a privileged context. It can safely extend the capabilities of the kernel without losing efficiency or requiring kernel source code changes. eBPF programs can target a wide set of use cases, but are mainly developed for security, observability, and networking [11]. For the latter scenarios, it is usually combined with the Linux eXpress Data Path (XDP), resulting in a powerful tool to implement flexible, efficient, and portable network functions, for example in the form of Virtualized Network Functions (VNFs).

NGNs are expected to handle a vast range of applications, technologies, and devices. This inherent heterogeneity makes it difficult to efficiently cope with changes in the network conditions. In consequence, it is necessary to integrate intelligent functions within the network architecture. eBPF is gaining momentum recently due to its flexibility and portability. For that reason, different works have proposed the use of eBPF programs as the enforcement tool directed by ML-powered frameworks operating at application/control level. In this line, authors of [12] developed an ML-based framework to dynamically select and deploy congestion control algorithms. The solution is based on two eBPF modules, one to collect information about the TCP flows and forward it to a user space framework, and another that implements a congestion control algorithm, which can be reconfigured in run time by the mentioned framework. Experiments performed both in emulated and production networks showed its effectiveness over baseline solutions. Work in [13] developed a prediction model based on eBPF and Long Short-Term Memory (LSTM) to monitor the Linux network stack status. The solution used an eBPF program to track HTTP requests and responses in the kernel network stack. These data were then forwarded to the LSTM model to predict the subsequent network situation. When compared to similar methods, the proposal showed more accuracy to perform real-time predictions. In [14], a monitoring system at the kernel level was introduced. It was composed of a non-intrusive eBPF program that collected application layer traffic. The gathered information was then analyzed using ML methods to obtain a performance diagnosis, hence enabling the localization of network bottlenecks. Work in [15] developed an automatic Redis tuning model based on eBPF and random forest. An eBPF program was used to identify different working scenarios. These data are sent to a random forest module, which sorts the memory parameters to find those with higher efficiency. Then, this information is sent directly to the operating system to optimize the hardware resource usage. Authors in [16] proposed a solution to fingerprint and classify microservices. The fingerprinting is performed using an eBPF module to trace system calls. Then, a combination of Bayesian learning and LSTM autoencoders can fingerprint many real-world microservices with a 99% of accuracy, using only a 1%–2% of additional CPU usage.

As can be seen through existing works, eBPF has emerged as a highly suitable solution to perform monitoring and networking within the Linux kernel. Together with ML algorithms, they provide a powerful tool to automatically detect disruptions in the network and act in consequence in real time. However, there is still no integration of complex ML models within the Linux kernel using eBPF capabilities. We just have found a preliminary work in which a simple decision tree model is implemented by using a series of concatenated *if/else* instructions [7]. However, as aforementioned, it is common to find frameworks that use eBPF programs to collect data from the network and then forward them to the user space, where the ML algorithms are fed and generate the decisions [17]. This incurs in extra overhead, as the ML-based processing is taken out of the kernel space. Therefore, to our knowledge, this is the first integration within the Linux kernel of a complex ML algorithm, namely, a MLP neural network, embedded in an eBPF program to enable low-latency and intelligent traffic processing.

## 3. Use case

One of the areas where eBPF has enormous potential is IoT. Typical IoT devices are notably constrained, presenting limited processing power, memory, and energy resources. These restrictions pose a challenge for implementing effective cybersecurity functions, as traditional security solutions are resource-intensive and may not be feasible for IoT devices. This is especially notorious considering the new wave of ML-based cybersecurity schemes that, although they are showing great performance for detecting many types of attacks [18,19], their integration in constrained end-devices remains a challenge [20].

In IoT deployments, every end node is a possible entry point towards the entire network infrastructure, hence improving their robustness against attacks is crucial for the overall system security. As a result, it is critical to develop defense mechanisms specifically designed for IoT devices, taking into account their limitations and the particular threats they face in their operational environments. Since eBPF allows the efficient execution of code within the Linux kernel, it has the potential to improve the security of IoT elements by enabling the development of sophisticated and lightweight security solutions that can be fully integrated within the operative system, i.e., in its kernel. Therefore, eBPF can be used to enforce security policies at the kernel level, providing an additional layer of self-protection against attacks on the end device. Thus, it is possible to build self-protection mechanisms that are tailored to the specific needs of IoT nodes, allowing them to autonomously detect and respond to security threats in a timely and efficient manner, thus reducing their dependence on fixed infrastructure.

IoT devices have countless applications and it is common to have several devices working together in what is called a Wireless Sensor Network (WSN). One of the most employed technologies for enabling these systems is 6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks), which is an open standard defined by the IETF to enable IPv6 packets to be carried on top of low-power wireless networks such as the ones defined by IEEE 802.15.4 [21]. 6LoWPAN was designed to make IPv6 usable by highly constrained devices, hence enabling their direct connection to the Internet [22]. Highly related to 6LoWPAN, RPL (Routing Protocol for Low-Power and Lossy Networks) [23] is a routing protocol for wireless networks with low power consumption and generally susceptible to packet loss. It is a proactive protocol based on distance vectors and operates on top of IEEE 802.15.4 and 6LoWPAN. This protocol can quickly create network routes, share routing information, and adapt to topology changes in an efficient way, making it very suitable for WSN with constrained devices, thus being one the most employed routing protocols in these kinds of systems [24].

Considering these aspects, the goal of this use case is to demonstrate and validate the potential of eBPF to enable the self-protection of IoT devices by permitting the execution within the Linux kernel of attack-detection algorithms powered by complex ML algorithms such as neural networks, namely, MLP models. Aiming at allowing experiment reproducibility, the steps followed to implement the use case under consideration are presented in the next subsections.

### 3.1. Dataset generation

In order to generate the ML models that will be embedded within the end-devices kernel, a dataset is needed for their training. To generate that dataset, a controlled test scenario has been configured. This testbed provides the ability to gather reliable data without having to compromise the security of any in-production WSN. This environment has been deployed using Cooja [25], which is a tool that allows the emulation of networks of devices running Contiki, an open-source operating system specifically designed for IoT devices that implements both 6LoWPAN and RPL.

The studied scenario focuses on the detection of the “Hello Flood” attack on the RPL protocol. To implement this attack, the *RPL-Attack-Framework* [26] has been used. This includes the modules for root, malicious, and benign nodes within the RPL-managed WSN. In this use case, the “Hello Flood” attack involves one or more malicious nodes periodically generating routing information requests to make the other nodes respond to those messages, intending to waste their resources such as their battery power.

As shown in Fig. 1, the simulation consists of one root node (green device), ten sensor nodes (yellow devices), and one malicious node within the range of communication of the root node (purple device). The *Radio Messages Tool* of Cooja is used to capture all the traffic generated during a ten-minute simulation, generating a .pcap file from which the dataset is generated. In our experiments we have focused on enabling the self-protection of the root node, as it is the most important element of the RPL network. To this end, the raw .pcap file is processed to extract the RPL packets exclusively received by the root node, with each row representing the number of packets (segregated

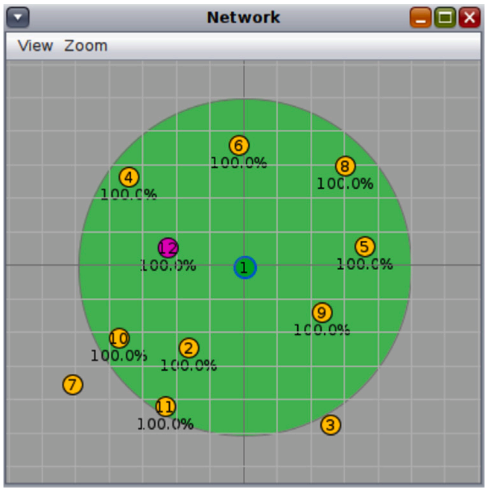


**Fig. 1.** Cooja scenario.

by RPL message type,[1] namely, DODAG Information Solicitation (DIS), DODAG Information Object (DIO), Destination Advertisement Object (DAO), and other) generated by the same source and received by the root node within a given time window. Tests have been conducted with time windows of 1 s, 5 s, and 10 s to evaluate the most adequate data aggregation to maximize the accuracy of the ML models and the performance of the end-device. Therefore, three datasets of 1919, 1047, and 572 rows, respectively, are obtained. In these datasets, each row also includes the address of the corresponding source node, although this field is just informative and has not been employed for training the ML models, and a label indicating whether the source is a malicious node or not. With this process, note that we are generating datasets that make use of data that can be perfectly collected by each node (the root node in this case) in real-time during its operation, i.e., the number of DIS, DIO, DAO, and other messages (input features for the ML model) received for each neighbor within a time frame (1 s, 5 s, and 10 s). Therefore, the training datasets and, consequently, the generated models, are completely realistic and can be used in a real WSN scenario. The message handling and data aggregation in the IoT node during the normal operation of the device is later explained, when the eBPF implementation is detailed.

### 3.2. ML model development

In this step, we make use of the generated datasets that gather the RPL protocol messages received by the root node from its neighboring nodes over certain time windows. As aforementioned, the ultimate objective is to implement an in-kernel ML-based solution powered by eBPF capable of identifying certain attacks (“Hello Flood” attack in this case). While different types of ML algorithms could have been considered for this purpose, we have considered the use of MLP models giving the flexibility and good accuracy provided by neural networks for analyzing traffic and detecting anomalies [27]. As previously mentioned, recent research has successfully developed a decision tree implementation in eBPF by means of a number of *if/else* statements [7]. However, we go one step further by implementing a much more complex ML algorithm like MLP. In the context of detecting cyber-attacks, MLP neural networks have been historically used due to their ability to handle non-linear relationships between input features and output labels, which is a common characteristic of network traffic data. MLPs can learn complex patterns in the data, making them suitable for detecting subtle

[1] https://datatracker.ietf.org/doc/html/rfc6550

attack behaviors that may not be immediately apparent. Moreover, MLPs can achieve high accuracy and speed, making them ideal for real-time attack detection in online security systems [28].

For this particular use case, we have made use of a Python's Scikit-Learn-based MLP implementation, which has shown good performance results with our generated datasets when using the Rectified Linear Unit (ReLU) activation function. This function is useful for binary classification tasks, such as detecting whether the traffic received by a node from a neighbor constitutes an attack or not. Furthermore, it is a suitable choice for eBPF, as it is effective in pattern detection, but computationally efficient for applications that require near real-time processing. Regarding the training process, the generated dataset was split with an 80% allocation of samples for training and a 20% allocation for testing. The choice of an 80/20 split strikes a balance between providing sufficient data for model training and maintaining a sizable test set for meaningful performance evaluation. In addition, we included stratification in the split to ensure that the original class distribution is maintained in both the training and test sets. This decision is particularly important due to the dataset is quite unbalanced as it contains more negatives than positives samples. This procedure helps to avoid bias in the model performance evaluation by ensuring proportional representation of both classes in both training and test sets.

After several tests with different MLP configurations (hidden layers and neurons per layer), a neural network architecture with a limited number of layers and neurons is sufficient. This is because loss of linearity is not necessary to obtain good results. Adding too many layers or neurons potentially leads to overfitting, which would reduce the model's generalization performance. Concretely, for our tests, we have finally adopted an MLP model consisting of two hidden layers, precisely composed of three and two perceptrons. In each layer, the ReLU activation function is utilized, while the sigmoid function is employed in the final layer. It is worth noting that, as detailed in the next section, the adopted development procedure enables the implementation of solutions with more complex neural networks and other activation functions. With the MLP selected configuration, we attain an accuracy of 0.99, 1, and 1, with the models trained using the aggregation windows of 1 s, 5 s, and 10 s, respectively. Recall that, although the accuracy achieved is very high, the focus of this work is on the challenge of running neural network models within the Linux kernel.

Finally, in order to port the obtained MLP model from Scikit-Learn 1.2.0 to eBPF, we have made use of the TinyML's *emlearn* 0.14.0 library [29], which transforms the Python model into C code, which notably eases the integration of the model within an eBPF program. This library provides conversion utilities to generate float arrays that contain the weights and biases of each neural network's layer, as well as a complete structure that represents the trained model. The next section gives more details about the adopted approach considering the final step: the eBPF implementation of the attack-detection solution.

### 3.3. eBPF-based implementation

The integration of the C-coded MLP model within an eBPF program is not a trivial process given the great restrictions imposed by the eBPF verifier [11]. In the following, we detail the development process and all the challenges faced during this process and how we have overcome them. The resulting code has been made available in a public repository.[2]

As a first step, we have made use of the header files provided by the *emlearn* library, which contain functions that implement the logic of the neural network. Specifically, for each layer of the neural network, a linear combination is made between the outputs of the previous layer with the weights and biases of the current layer, followed by the application of a non-linear activation function (ReLU activation function in our case). Despite the benefits provided by the *emlearn* library, integrating the MLP model into eBPF is not feasible in a straightforward way due to the aforementioned constraints in eBPF programs. Therefore, to address these limitations, several modifications have been introduced in the *emlearn* library to adapt its generated code to a valid one accepted by the eBPF verifier. The main implementation challenges are described in the following.

Firstly, eBPF programs cannot make use of floating-point operations [30]. This is a great limitation given that neural network's weights and biases are non-integer numbers. To tackle this challenge, we have introduced a fixed-point representation with 16 bits designated for the integer part, 15 bits for the fractional part, and 1 bit for the number's sign. In order to facilitate this transformation, we have also incorporated tailored conversion functions to systematically convert all floating-point values into 32-bit fixed-point integers using this implicit representation.

Executing this conversion process requires a series of adjustments to our neural network implementation. Specifically, we have developed new arithmetical functions (addition, subtraction, multiplication, and division) that work with this type of representation. While adaptation for addition proved to be straightforward, multiplication and division of fixed-point integers required the adoption of 64-bit integers in order not to lose precision and scaling the result using powers of two. Moreover, adaptations have been made to the activation functions within the neural network. In our implementation, the ReLU function is exclusively employed between layers, while the sigmoid function is utilized for the final classification. Regarding ReLU, additional adjustments have been implemented to facilitate its acceptance of fixed-point integers as input and to validate whether they surpass zero. Simultaneously, optimizations have been introduced to the sigmoid function to streamline the final classification process during inference. It is important to note that in order to incorporate any desired activation function, careful consideration of a fixed-point implementation is imperative. This may involve direct use of the newly introduced arithmetical functions, approximation using numerical methods, or the optimization of the function, especially in the context of final classification tasks.

Besides, the eBPF verifier impose limitations when working with loops, as it has to guarantee the termination of all programs loaded into the kernel, for example, to avoid Denial of Service (DoS) attacks. At first, their use was not allowed, but the 5.3 version of the Linux kernel introduced support for bounded loops,[3] although within the boundaries dictated by the maximum number of eBPF instructions permitted per program (1 million from version 5.2). Following this enhancement, version 5.17 added the *bpf_loop()* helper function,[4] which trades some execution time for a much faster verification process and allows the use of bigger bounded loops (up to 8 millions of iterations), as it is not restricted by the eBPF instruction limit. However, it is still necessary to comply with the verifier stack limit to avoid programs that run for too long time due to nested loops. In our implementation, given that we do not need loops with a large number of iterations, we have adjusted the loops to make them bounded following the first approach, so that the number of iterations is known at compile time. This allows loop unrolling directives from the compiler to transform them into a sequence of independent instructions. To accomplish this, variables that represented the number of layers, the lengths of auxiliary buffers, and neural network outputs were replaced with MACROS, and *for* loops were modified accordingly. This approach not only enables loop unrolling but also improves the code's readability and maintainability, making it easier to reuse for other types of neural networks. Additionally, constant static arrays that contain the number of inputs

[2] https://github.com/irenebru/eBPF-neural-network

[3] https://cdn.kernel.org/pub/linux/kernel/v5.x/ChangeLog-5.3

[4] https://cdn.kernel.org/pub/linux/kernel/v5.x/ChangeLog-5.17

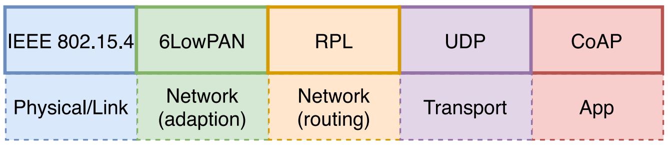


**Fig. 2.** Packet headers structure.

and outputs for each neural network layer are used to properly apply the linear combination of weights and biases for each layer. Considering this, our implementation has been designed to permit the use of neural networks with different numbers of layers, neurons, and activation functions, hence proving generalization capabilities.

In order to use the MLP model for detecting "Hello Flood" attacks to the RPL protocol, a header parsing procedure has also been implemented in eBPF to analyze incoming network packets. The parsing process in eBPF requires bound checking to ensure that memory accesses are within the expected boundaries. Otherwise, the eBPF verifier does not permit the program to run. To successfully implement the parsing process, it is important to consider that the link level header is IEEE 802.15.4, followed by the 6LoWPAN header, which contains the compressed IPv6 address (see Fig. 2). Note that the 6LoWPAN header may not include the entire address, as it usually omits the prefix or includes only the last few bytes. The RPL message type is determined by a byte in its header with a value of 0 for DIS, 1 for DIO, and 2 for DAO messages.

### 3.4. Experiments

After parsing the headers, the collected data need to be stored persistently between eBPF program executions as there is no maintained state between them. To accomplish this, pinned eBPF maps have been utilized, which enables the tracking of the messages sent by each IPv6 address to the root node over different time windows (1 s, 5 s, and 10 s). Concretely, the eBPF maps used are of type BPF_MAP_TYPE_HASH, which provides hash map storage for general purposes. We use them as a hash map where the key is the IP address of the source device, and the value is an array of 4 integers. Each of these integers represents the number of packets of each type received from that IP address, namely, DIS (0), DIO (1), DAO (2), and other types of packets (3). Regarding the computation of time windows, we make use of the *bpf_ktime_get_ns()* function, which returns the machine time in ns. To implement the time windows, we use again an eBPF map of type BPF_MAP_TYPE_HASH. Thus, each time the neural network is executed, the time from the machine is taken and stored in the map. At each execution of the eBPF program, this saved time is compared with the current time, so it is checked whether the time-window has expired. Once the window arrives to end, the MLP model is applied to analyze the collected data and determine whether an attack has occurred. This process is repeated over time to guarantee the continuous protection of the device (the WSN root node in our case). It is worth mentioning that all the described behavior has been implemented in a single eBPF program, therefore no tail calls have been used in our solution.

Once the implementation is ready, the eBPF program is then compiled, verified and loaded to the generic XDP eBPF hook. XDP permits the attachment of programs to the NIC using different models, being the offloaded and native alternatives the most efficient. However, these two models need the NIC driver to support this kind of operation [31]. Commodity hardware is often equipped with more limited NICs with no support for these drivers. In this case, XDP allows the attachment of eBPF programs to generic hooks that run after the device driver. This permits the execution of eBPF programs on constrained commodity hardware, enabling advanced packet processing in these devices.

As a result of this process, we obtained our eBPF program ready to be validated through the experimentation phase. To better understand and summarize the functioning of this tool, Fig. 3 depicts the state diagram of the program. Note that it runs in an event-based manner, thus the whole process will be repeated per each packet received in the eBPF/XDP hook, just after its reception by the NIC. When a packet arrives, its headers are parsed and the RPL packet type is identified, the packet type count and its source are then stored in an eBPF hash map as explained before, so these data can be accessed from subsequent executions. Then, the program checks whether the time window has expired or not. If that is not the case, the execution ends. Otherwise, the ML model gathers the packet count information from the persistent map and infers if an attack has happened or not. Finally, noting that considering advanced countermeasures to the attack is out of the scope of this paper, hence in our implementation, when an attacker is detected a new rule is instantiated, so the root node directly drops the packets coming from this device, identified by its IPv6 address, to avoid wasting unnecessary resources.

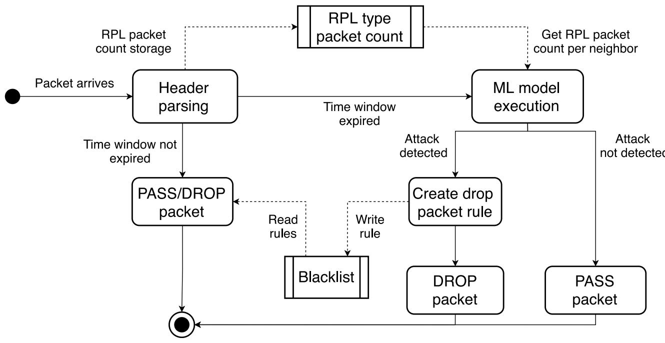


**Fig. 3.** State diagram of the eBPF program.

The experiments were conducted using two elements, namely, a regular computer to generate traffic and an IoT device running the eBPF program, which processes the received packets and executes the embedded neural network. The first one was a desktop computer powered by an Intel i5-3470 CPU with four cores and 8 GB of RAM, while the latter was a Raspberry Pi 3 Model B V1.2, with a four-core CPU and 1 GB of RAM, using Ubuntu 22.04 LTS and the GNU/Linux 5.15.0-1034-raspi kernel. Both were directly connected via an Ethernet cable, with a maximum theoretical throughput speed of 100 Mbps, as the network interface of the Raspberry Pi is limited to that rate. The traffic was generated and injected using the *tcpreplay* [32] tool, which was used to replicate the captured traffic of the raw dataset presented previously. In this way, the traffic extracted from the original .pcap file (see Section 3.1) left the desktop computer at a desired speed and arrived at the Raspberry Pi's Ethernet port, to whose interface was attached the eBPF program.

Besides the eBPF implementation of the neural network, another version of our program has also been developed to decouple the neural network from the parsing mechanism and execute it in Linux's user space. Hence, we can compare the performance of the ML model operating in both kernel and user spaces (Fig. 4). In this way, two scenarios were prepared for the experiments, one in which the parsing of the packets and the neural network execution were performed in a privileged context inside the kernel, and another one in which the neural network calculations were moved to the user space. The neural network implementation in user space uses the same C code that the in-kernel implementation, with 125 Lines of Code (LoC). The eBPF program has 176 LoC, and the hybrid solution is divided in the eBPF parsing program (141 LoC) and the ML model handling in user space with 63 LoC. Besides, the binary sizes of the eBPF program with the whole implementation, the eBPF program only with the parser, and the user space program are 26 KB, 16 KB, and 546 KB, respectively. The time that the Kernel needs to load the eBPF program with the ML

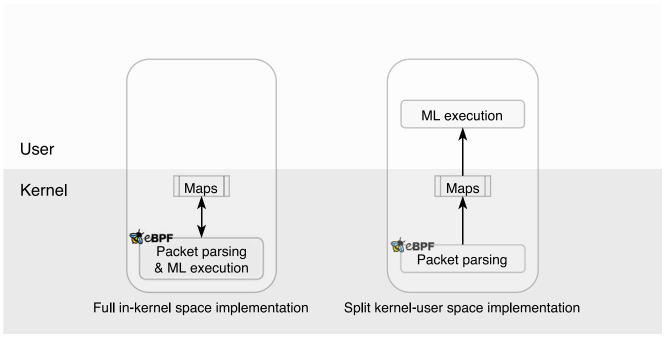

User
Kernel
Maps
eBPF
Packet parsing & ML execution
Full in-kernel space implementation
ML execution
Maps
eBPF
Packet parsing
Split kernel-user space implementation

**Fig. 4.** Conceptual diagram of the two implementations.

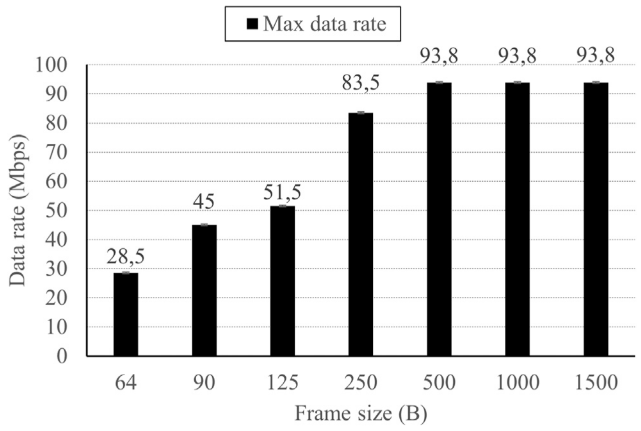


**Fig. 5.** Maximum data rate supported by the physical link between both ends.

model is 830,2 ±53, 4 μs, and it takes 788,8 ±57, 9 μs to load the eBPF program that only contains the packet parsing mechanism.

As explained previously, three different time windows (1 s, 5 s, and 10 s) have been chosen to assess the best data aggregation in terms of ML model's accuracy and to inspect the impact on the end-device performance. Each experiment was executed 10 times during 60 s, to avoid singularities in the attained outcomes and to obtain statistical wealth.

## 4. Results

This section presents the performance of the developed solution in the testbed described above. The first conducted test aimed at evaluating the maximum data rate supported by the physical link between both devices without executing the eBPF program (baseline performance). This consisted in measuring the maximum bandwidth achieved with multiple packet sizes (TCP traffic) by using the iPerf tool. As can be seen in Fig. 5, with lower packet sizes, the data rate decreases as can be expected. This is because the IoT device's NIC has to process a huge number of packets per second, exceeding its capabilities. The maximum attained rate is above 90 Mbps with big packets, which decreases down to 28,5 Mbps with packets of 64 B. Note that in this experiment and the following ones, we made use of high data rates to stress both the IoT device and the eBPF function in order to study their performance with demanding conditions. Nevertheless, this is not the usual behavior of typical IoT systems, which usually present more sporadic and limited traffic.

Once the eBPF program is embedded and executed in the IoT node, Table 1 presents the maximum data rates achieved per configuration

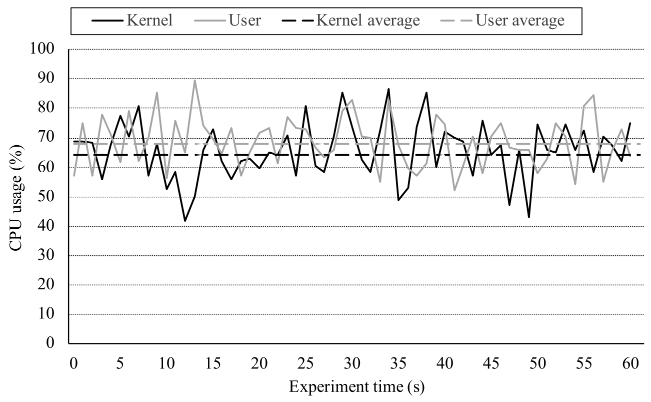


**Fig. 6.** CPU usage.

**Table 1**
Maximum data rate per configuration.

| ML execution space | Window (s) | Data rate (Mbps) |
|---|---|---|
| Kernel | 1 | 43,1 |
| Kernel | 5 | 43,6 |
| Kernel | 10 | 44,5 |
| User | 1 | 45,5 |
| User | 5 | 45,5 |
| User | 10 | 45,5 |

without packet loss. The variables of the configurations are whether the neural network is executed in kernel or user space (first column), and the duration of the time window to aggregate data and execute the neural network (second column). Firstly, considering that the average packet length employed in our experiments (extracted from the raw traffic captured during the dataset generation) is 90 B, there is not an appreciable decrease in the attained data rate in comparison with the baseline scenario (Fig. 5). Therefore, it can be inferred that the eBPF program is not acting as a bottleneck for the IoT device. Comparing both implementation approaches, the highest speeds are obtained when the neural network execution is performed in the user space, that is, when the parsing of the packets is executed by the eBPF program, and the ML model is executed in the user space, gathering its input data from a shared map in a decoupled way (Fig. 4). When the neural network is executed in the Linux kernel, the data rate decreases in comparison with the other configuration, with a difference of around 5% in the worst case (aggregation window of 1 s). This is due to the requirement by the eBPF program to execute the neural network each time the time window expires, a period during which the program remains "busy" and unable to receive additional packets. Likewise, this is the reason why the data rate increases with the increment of the time window in the in-kernel scenarios. This outcome is expected, as the ML model is executed less frequently with longer time windows, therefore the eBPF program is capable of processing more packets as it is not busy with the other task. This can be understood as a drawback of the in-kernel implementation although, as aforementioned, the performance decrease with highly demanding traffic conditions is just of 5%, and this scenario does not seem likely in IoT environments.

Considering computational performance, Fig. 6 shows a real-time graph of CPU consumption during the execution of an experiment in terms of software interruptions in both scenarios with an aggregation time window of 5 s. This result aims at showing the stability of both implementations and compare their performances at runtime. It can be seen that, although the range in which both graphs oscillate is similar (60%–80%), on average the CPU consumption using the in-kernel implementation is lower than the user space one. Concretely,

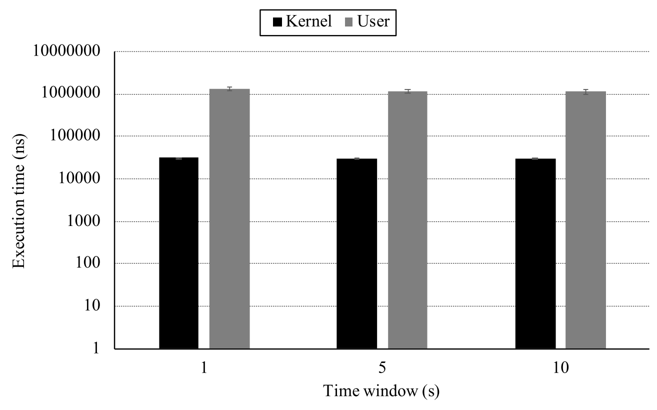


**Fig. 7.** Execution time comparison.

by running the neural network in the kernel, a reduction of 6% of CPU usage is attained. Besides, there are notable low peaks that do not appear when it is run from the user space. A reduced use of CPU is of high importance in IoT environments given the notable computation constraints of end nodes. Note that the experiments showed that the CPU consumption does not reach 100% usage of the core resources. This is because the experiments were performed at the maximum lossless data rate possible in each configuration, which means that this graph represents the stable behavior of the CPU just before reaching a data rate where packet drops start to occur due to CPU saturation (CPU usage of 100%).

Finally, Fig. 7 shows the average execution time of the ML model in both implementations and depending on the chosen time window. These measurements have been obtained in real-time using OS's tools (*bpf_ktime_get_ns()* and *clock()*) that permit to compute the time machine in nanoseconds before and after the call to the ML model. This call encompasses both the handling of inputs from the map and the execution of the neural network itself. It can be seen the notable decrease in the execution time of this process when it is run in kernel space in comparison with the case of running it in user space (reduction of around 97% for the three time windows under consideration). This remarkable difference is because the execution is faster in the privileged context where eBPF can run sandboxed programs in a very efficient way. Other OS activities when running the program in user-space add extra overhead in comparison with its execution in the Kernel's environment, e.g., context switches, I/O operations, syscalls, etc. As expected, the execution time does not hardly vary with the length of the window, as it is not affected by the frequency of data aggregation. We consider this outcome of relevant importance, as it evidences the advantages of executing complex tasks within the kernel in order to achieve a clear performance improvement in comparison with their execution in the user space.

Therefore, the attained results demonstrate the advantages eBPF provides to run programs in kernel space to enhance packet processing time and computing resource usage. In this way, this kind of solution can be interesting in fog/edge scenarios where performing any kind of ML-based processing is relevant such as the considered case of self-detection of cyberattacks. Besides, it also arises as a cost-effective workaround to handle traffic in an intelligent way without needing expensive hardware, as it has been showcased that the performance in commodity hardware with notable resource constraints is more than acceptable for certain kinds of scenarios.

## 5. Conclusions

NGNs are calling upon network and traffic management solutions able to enhance the intelligence and flexibility of the underlying infrastructures. With the softwarization of the network, multiples technologies are emerging to address the demanding requirements of B5G architectures in this regard. Consequently, a flexible and portable technology arises as an enabler for low-latency traffic processing in unexpensive hardware: eBPF. It permits the execution of programs within a privileged context in the Linux kernel, permitting the treatment of traffic flows in a quick and flexible way at any point of the network infrastructure (fog-edge-cloud). So far, eBPF has been used in combination with ML models by employing eBPF programs as mere data collectors at kernel level and feeding a ML algorithm running in user space. As the main contribution of this work, we presented a solution that integrates ML processing within the Linux kernel to provide intelligent fast packet inspection. Concretely, we have implemented a complex model such as MLP. By leveraging eBPF, we achieve high-performance and low-latency processing within the network data plane, enabling real-time traffic classification tasks without relying on a powerful external support. Considering an IoT cybersecurity use case, the attained results showed a clear improvement in comparison with a classical user space implementation approach. Concretely, we attained considerable reductions of 97% and 6% in the execution time and CPU usage of the ML model, respectively, when operating within the Linux kernel, rather than in user space. This paves the way for the development of novel network functions with high efficiency and portability able to run in a plethora of devices.

Several directions can be explored to enhance and extend our work. In first place, the integration in eBPF of more complex neural network models, such as convolutional neural networks, may increment the classification capabilities of the solution. Besides, alternative implementation methodologies may be adopted aiming at increasing the achieved data-rate and improve the fixed-point arithmetic in eBPF to boost the accuracy of the implemented neural networks. Additionally, the potential of hardware acceleration for eBPF-based neural networks is a promising line for future research. Specific hardware, such as FPGAs or SmartNICs can potentially increase the performance of eBPF-based classifiers, enabling its application in more demanding use cases.

## CRediT authorship contribution statement

**Jorge Gallego-Madrid:** Writing – review & editing, Writing – original draft, Software, Methodology, Investigation, Data curation, Conceptualization. **Irene Bru-Santa:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Formal analysis. **Alvaro Ruiz-Rodenas:** Writing – original draft, Investigation, Formal analysis, Data curation. **Ramon Sanchez-Iborra:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Methodology, Investigation, Conceptualization. **Antonio Skarmeta:** Writing – review & editing, Writing – original draft, Validation, Supervision, Resources, Project administration, Methodology, Funding acquisition, Conceptualization.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

Data will be made available on request.

## Acknowledgments

This work has been supported by the Spanish Ministry of Science and Innovation (MCIN) and the Spanish Research Agency (AEI), under the project ONOFRE-3 (Grant PID2020-112675RB-C44 funded by MCIN/AEI/10.13039/501100011033); by the Fundación Séneca—Agencia de Ciencia y Tecnología de la Región de Murcia—under the FPI Grant 21429/FPI/20, and co-funded by Odin Solutions S.L., Región de Murcia (Spain); by the Fulbright Commission in Spain under the Fulbright grant 00003/ FLB/21; by the European Commission under the 5GASP (Grant No. 101016448) and NANCY (Grant No. 101096456) projects; and by the Spanish Ministry of Economy and Digital Transformation, under the project CERBERUS-HADES (Grant No. TSI-063000-2021-62).

## References

[1] C.-X. Wang, X. You, X. Gao, X. Zhu, Z. Li, C. Zhang, H. Wang, Y. Huang, Y. Chen, H. Haas, J.S. Thompson, E.G. Larsson, M.D. Renzo, W. Tong, P. Zhu, X. Shen, H.V. Poor, L. Hanzo, On the road to 6G: Visions, requirements, key technologies, and testbeds, IEEE Commun. Surv. Tutor. 25 (2) (2023) 905–974, http://dx.doi.org/10.1109/COMST.2023.3249835, URL https://ieeexplore.ieee.org/document/10054381/.

[2] C. Yeh, G.D. Jo, Y.-J. Ko, H.K. Chung, Perspectives on 6G wireless communications, ICT Expr. 9 (1) (2023) 82–91, http://dx.doi.org/10.1016/j.icte.2021.12.017, URL https://linkinghub.elsevier.com/retrieve/pii/S240595952100182X.

[3] M.A.M. Vieira, M.S. Castanho, R.D.G. Pacífico, E.R.S. Santos, E.P.M.C. Júnior, L.F.M. Vieira, Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications, ACM Comput. Surv. 53 (1) (2020) 1–36, http://dx.doi.org/10.1145/3371038, URL https://dl.acm.org/doi/10.1145/3371038.

[4] C. Hardegen, B. Pfulb, S. Rieger, A. Gepperth, Predicting network flow characteristics using deep learning and real-world network traffic, IEEE Trans. Netw. Serv. Manag. 17 (4) (2020) 2662–2676, http://dx.doi.org/10.1109/TNSM.2020.3025131, URL https://ieeexplore.ieee.org/document/9201025/.

[5] A. Shahraki, T. Ohlenforst, F. Krey, When machine learning meets network management and orchestration in edge-based networking paradigms, J. Netw. Comput. Appl. 212 (2023) 103558, http://dx.doi.org/10.1016/j.jnca.2022.103558, URL https://www.sciencedirect.com/science/article/pii/S1084804522001990.

[6] J. Gallego-Madrid, R. Sanchez-Iborra, P.M. Ruiz, A.F. Skarmeta, Machine learning-based zero-touch network and service management: a survey, Digit. Commun. Netw. 8 (2) (2022) 105–123, http://dx.doi.org/10.1016/j.dcan.2021.09.001, URL https://linkinghub.elsevier.com/retrieve/pii/S2352864821000614.

[7] J.F. Maximilian Bachl, T. Zseby, A flow-based IDS using machine learning in eBPF, 2022, arxiv URL https://arxiv.org/abs/2102.09980.

[8] M.R.S. Sedghpour, P. Townend, Service Mesh and eBPF-Powered Microservices: A Survey and Future Directions, in: 2022 IEEE International Conference on Service-Oriented System Engineering, SOSE, IEEE, 2022, pp. 176–184, http://dx.doi.org/10.1109/SOSE55356.2022.00027, URL https://ieeexplore.ieee.org/document/9912629/.

[9] M. Karlsson, B. Töpel, The Path to DPDK Speeds for AF_XDP, in: Linux Plumbers Conference, 2018.

[10] F. Parola, R. Procopio, R. Querio, F. Risso, Comparing user space and in-kernel packet processing for edge data centers, SIGCOMM Comput. Commun. Rev. 53 (1) (2023) 14–29, http://dx.doi.org/10.1145/3594255.3594257.

[11] S. Miano, M. Bertrone, F. Risso, M. Tumolo, M.V. Bernal, Creating Complex Network Services with eBPF: Experience and Lessons Learned, in: IEEE 19th International Conference on High Performance Switching and Routing, HPSR, IEEE, 2018, pp. 1–8, http://dx.doi.org/10.1109/HPSR.2018.8850758, URL https://ieeexplore.ieee.org/document/8850758/.

[12] J. Zhou, X. Qiu, Z. Li, G. Tyson, Q. Li, J. Duan, Y. Wang, Antelope: A framework for dynamic selection of congestion control algorithms, in: 2021 IEEE 29th International Conference on Network Protocols, ICNP, 2021, pp. 1–11, http://dx.doi.org/10.1109/ICNP52444.2021.9651912.

[13] X. Zhang, Z. Liu, J. Bai, Linux network situation prediction model based on eBPF and LSTM, in: 2021 16th International Conference on Intelligent Systems and Knowledge Engineering, ISKE, 2021, pp. 551–556, http://dx.doi.org/10.1109/ISKE54062.2021.9755426.

[14] C. Liu, Z. Cai, B. Wang, Z. Tang, J. Liu, A protocol-independent container network observability analysis system based on eBPF, in: 2020 IEEE 26th International Conference on Parallel and Distributed Systems, ICPADS, 2020, pp. 697–702, http://dx.doi.org/10.1109/ICPADS51040.2020.00099.

[15] J. Yang, L. Chen, J. Bai, Redis automatic performance tuning based on eBPF, in: 2022 14th International Conference on Measuring Technology and Mechatronics Automation, ICMTMA, 2022, pp. 671–676, http://dx.doi.org/10.1109/ICMTMA54903.2022.00139.

[16] H. Chang, M. Kodialam, T. Lakshman, S. Mukherjee, Microservice fingerprinting and classification using machine learning, in: 2019 IEEE 27th International Conference on Network Protocols, ICNP, 2019, pp. 1–11, http://dx.doi.org/10.1109/ICNP.2019.8888077.

[17] S.-Y. Wang, J.-C. Chang, Design and implementation of an intrusion detection system by using extended BPF in the linux kernel, J. Netw. Comput. Appl. 198 (2022) http://dx.doi.org/10.1016/j.jnca.2021.103283, https://www.scopus.com/inward/record.uri?eid=2-s2.0-85120624023&doi=10.1016%2fj.jnca.2021.103283&partnerID=40.

[18] A. Boukerche, R.W.L. Coutinho, Design guidelines for machine learning-based cybersecurity in internet of things, IEEE Netw. 35 (1) (2021) 393–399, http://dx.doi.org/10.1109/MNET.011.2000396.

[19] H. Wang, L. Barriga, A. Vahidi, S. Raza, Machine learning for security at the IoT edge - a feasibility study, in: 2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems Workshops, MASSW, 2019, pp. 7–12, http://dx.doi.org/10.1109/MASSW.2019.00009.

[20] R. Sanchez-Iborra, A.F. Skarmeta, Tinyml-enabled frugal smart objects: Challenges and opportunities, IEEE Circuits Syst. Mag. 20 (3) (2020) 4–18, http://dx.doi.org/10.1109/MCAS.2020.3005467.

[21] G. Montenegro, J. Hui, D. Culler, N. Kushalnagar, RFC FT-ietf-6lowpan-format: Transmission of ipv6 packets over ieee 802.15.4 networks, IETF Datatracker (2007) URL https://datatracker.ietf.org/doc/html/rfc4944.

[22] A. Minaburo, L. Toutain, C. Gomez, D. Barthel, J.-C. Zuniga, RFC FT-IETF-LPWAN-ipv6-static-context-HC: SCHC: Generic framework for static context header compression and fragmentation, IETF Datatracker (2020) URL https://datatracker.ietf.org/doc/html/rfc8724.

[23] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, R. Alexander, et al., RPL: IPv6 routing protocol for low-power and lossy networks, 2012, URL https://www.rfc-editor.org/rfc/rfc6550.

[24] K.A. Darabkh, M. Al-Akhras, J.N. Zomot, M. Atiquzzaman, RPL routing protocol over IoT: A comprehensive survey, recent advances, insights, bibliometric analysis, recommendations, and future directions, J. Netw. Comput. Appl. 207 (2022) 103476, http://dx.doi.org/10.1016/j.jnca.2022.103476, URL https://www.sciencedirect.com/science/article/pii/S1084804522001242.

[25] An Introduction to Cooja, URL https://github.com/contiki-os/contiki/wiki/An-Introduction-to-Cooja,

[26] Introduction - RPL Attacks Framework, URL https://rpl-attacks.readthedocs.io/en/latest/.

[27] J.E.D. Albuquerque Filho, L.C.P. Brandao, B.J.T. Fernandes, A.M.A. Maciel, A review of neural networks for anomaly detection, IEEE Access 10 (2022) 112342–112367, http://dx.doi.org/10.1109/ACCESS.2022.3216007.

[28] P. Shettar, A.V. Kachavimath, M.M. Mulla, G.D. Narayan, G. Hanchinmani, Intrusion Detection System using MLP and Chaotic Neural Networks, in: 2021 International Conference on Computer Communication and Informatics, ICCCI, IEEE, 2021, pp. 1–4, http://dx.doi.org/10.1109/ICCCI50826.2021.9457024, URL https://ieeexplore.ieee.org/document/9457024/.

[29] Machine Learning inference engine for Microcontrollers and Embedded devices, URL https://github.com/emlearn/emlearn.

[30] M. Jadin, Q. De Coninck, L. Navarre, M. Schapira, O. Bonaventure, Leveraging eBPF to make TCP path-aware, IEEE Trans. Netw. Serv. Manag. 19 (3) (2022) 2827–2838, http://dx.doi.org/10.1109/TNSM.2022.3174138, URL https://ieeexplore.ieee.org/document/9772044/.

[31] 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, CoNEXT '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 54–66, http://dx.doi.org/10.1145/3281411.3281443.

[32] Tcpreplay - Pcap editing and replaying utilities, URL https://tcpreplay.appneta.com/.

**Jorge Gallego-Madrid** is a full-time researcher at the University of Murcia (Spain). He received the B.Sc. degree in Computer Engineering and the M.Sc. on New Technologies in Computer Science from the same university in 2018 and 2019, respectively, where he is also a predoctoral researcher at the Department of Information and Communication Engineering under the Fundacion Seneca-Agencia de Ciencia y Tecnologia de la Region de Murcia FPI grant, in collaboration with Odin Solutions S.L. He is also a Fulbright Visiting Graduate Scholar at Whiting School of Engineering of Johns Hopkins University, under the Fulbright Spanish program. He has participated in international projects such as 5GASP, 5G-MOBIX or USE-IT: CHISTERA, and in national projects such as SCOOTER. His research interests include Internet of Things, intelligent transportation systems, 5G and network slicing techniques.

**Irene Bru-Santa** received the B.Sc. degree in Computer Engineering from the University of Murcia (Spain) in 2023. Currently, she is in the final stages of her studies in the B.Sc. degree in Mathematics at the same university. In 2022, she was awarded a Collaborative Scholarship by the Spanish Ministry of Education to initiate research in the Department of Information and Communication Engineering at the Faculty of Computer Science, University of Murcia. Her academic pursuit has been complemented by internship experiences as a Solutions Architect at Amazon Web Services and as a Software Developer at Activision Blizzard King. Her primary research interests encompass the areas of embedded machine learning, cybersecurity, and fast packet processing.

**Alvaro Ruiz-Rodenas** is a graduate researcher at the Information and Communications Engineering Department in the University of Murcia, Spain. From this institution, he received the B.Sc. degrees in Computer Engineering and Mathematics in 2023. He is continuing his education with an M.Sc. in Cybersecurity from the same university. His main research interest are cyber–physical systems, and cybersecurity in IoT and 5G architectures.

**Ramon Sanchez-Iborra** is an Associate Professor at the Information and Communications Engineering Department in the University of Murcia. He received the B.Sc. degree in telecommunication engineering in 2007 and the M.Sc. and Ph.D. degrees in information and communication technologies in 2013 and 2016, respectively, from the Technical University of Cartagena. His main research interests are evaluation of QoE in multimedia services, management of wireless mobile networks, green networking techniques, and IoT/M2M architectures.

**Antonio F. Skarmeta** received the B.S. degree (Hons.) from the University of Murcia, Spain, the M.S. degree from the University of Granada, and the Ph.D. degree from the University of Murcia, all in computer science. He has been a Full Professor with the University of Murcia, since 2009. He has been part of many EU FP projects and even coordinated some of them. He has published more than 200 international articles. His main interests include the integration of security services, identity, the IoT, 5G, and smart cities.