Journal of the Brazilian Computer Society, 2026, 32:1, doi: 10.5753/jbcs.2026.5551

Sapo-boi: Bypassing Linux Kernel Network Stack in the Implementation of an XDP-based NIDS

Raphael Kaviak Machnicki [**Federal University of Paraná** | ***rkmachnicki@inf.ufpr.br***]
João Ribeiro Andreotti [**Federal University of Paraná** | ***jrandreotti@inf.ufpr.br***]
Ulisses Penteado [**Bluepex Cybersecurity** | ***ulisses@bluepex.com***]
Jorge Pires Correia [**Federal University of Paraná** | ***jpcorreia@inf.ufpr.br***]
Vinicius Fulber-Garcia [**Federal University of Paraná** | ***vinicius@inf.ufpr.br***]
André Grégio [**Federal University of Paraná** | ***gregio@inf.ufpr.br***]

Department of Informatics, Federal University of Paraná, R. Evaristo F. Ferreira da Costa, 383-391, Jardim das Américas, Curitiba, PR, 82590-300, Brazil

Received: 24 February 2025 • **Accepted:** 11 July 2025 • **Published:** 02 March 2026

Abstract. Network intrusion detection systems (NIDS) must inspect multiple parts of a packet to detect patterns of known attacks. With the advent of XDP, it has become feasible to implement such a system within the kernel's own network stack for the evaluation of ingress traffic. In this work, we propose Sapo-boi, an NIDS solution consisting of two modules: (i) the Suspicion Module, an XDP program capable of processing packets in parallel, discarding packets considered safe, and redirecting suspicious packets for verdict in user space through XDP sockets (AF_XDP); and (ii) the Evaluation Module, a user-level process capable of finding the rule to which the suspicious packet should be analyzed in constant time and triggering notifications if the suspicion is confirmed. The system demonstrated superior results in terms of packet analysis rates and CPU usage compared to traditional NIDS alternatives (Snort and Suricata).

Keywords: Network Intrusion Detection Systems, BPF, XDP, XDP sockets, AF_XDP

1 Introduction

The number of potential threats to connected systems grows in proportion to the increasing use of computer networks for enabling efficient and effective communication in the modern world. In this context, security solutions play a crucial role in detecting, preventing, and mitigating these threats, ensuring the confidentiality, integrity, and availability of processes and data.

In particular, Network Intrusion Detection Systems (NIDS) are a class of tools designed to detect attack patterns in network traffic. The primary goal of a NIDS is to inform relevant entities by triggering alerts when attack patterns are identified. It's worth noting that, due to their typical use of Deep Packet Inspection (DPI), these systems— even in well-known implementations like Snort [1] and Suricata[2]— often present low performance in packet processing and high packet drop rates Erlacher and Dressler [2018].

As a strategy to address the performance challenges associated with NIDS, technologies for accelerating network traffic processing can be employed. Among these technologies, batch processing tools such as the Data Plane Development Kit (DPDK) Conole *et al*. [2024] and PF_RING Biscosi *et al*. [2024] stand out. However, these tools typically impose high demands on computational resources, especially CPU, and can significantly increase packet processing latency Du *et al*. [2023].

Other prominent technologies, such as Extended Berkeley Packet Filter (eBPF) and eXpress Data Path (XDP), have gained popularity for various networking applications Sundberg *et al*. [2023]; Abranches *et al*. [2021], yet they remain relatively underexplored for implementing IDS solutions.

Considering the potential advantages and features of eBPF/XDP, and intending to create an effective NIDS solution that outperforms traditional alternatives, this work introduces Sapo-boi: the Evaluation and Traffic Processing System using BPF and XDP for Intrusion Detection (in Portuguese *Sistema de Avaliação e Processamento de tráfegO usando BPF e XDP para Observação de Intrusões*). Although we are not focused on directly increasing the number of detected malicious packets (as it would be achieved by a refined ruleset), the performance optimizations have the potential to maximize the amount of handled data, thus improving the probability of attack detections.

Sapo-boi is a NIDS consisting of two modules: one operating in user space (Evaluation Module) and the other in kernel space (Suspicion Module). The Suspicion Module inspects network traffic in parallel, detecting threat-related patterns at the first layer of the kernel's network stack. It forwards suspicious packets to the Evaluation Module via XDP family sockets. The Evaluation Module then conducts a deep inspection to confirm attacks, triggering alerts as needed.

The main contributions are: (i) the design, development, and deployment of a robust NIDS that processes packets via XDP and forwards traffic through XDP sockets, sending metadata along with packets to user space to aid in deep inspection tasks; and (ii) the experiments conducted to compare the computational performance of the proposed solution with three

[1] https://www.snort.org
[2] https://suricata.io

existing NIDS solutions: two entirely running in user space (Snort and Suricata) and one operating in both kernel and user space (Wang and Chang [2022]).

The results demonstrated the efficiency and effectiveness of the Sapo-boi solution. Despite every compared solution's capacity to fully perform as a signature-based NIDS, CPU usage was significantly lower compared to the user-space-only solutions and was very similar to the kernel-based solution. Furthermore, Sapo-boi achieved the highest alert-triggering rate in attack scenarios, outperforming all the tested alternatives.

The remainder of this work is organized as follows: Section 2 presents background concepts; Section 3 presents the related works; Section 4 describes the architecture and implementation of the proposed solution; Section 5 details the experimental setup and methodology; Section 6 presents and discusses the experimental results; Section 7 outlines limitations and future works; and, finally, Section 8 concludes the paper.

2 Background

In this section, the concepts, technologies, and algorithms used in this article are introduced. The discussion will encompass BPF (Berkeley Packet Filter) and XDP (eXpress Data Path) technologies, Network Intrusion Detection Systems (NIDS), the pattern detection algorithm Aho-Corasick Aho and Corasick [1975], the AF_XDP socket family Kernel [2024], and concepts related to BPF Type Format (BTF) and softirqs.

2.1 Intrusion Detection

Intrusion detection refers to the process of monitoring a computer system for signs of attacks, whether through analyzing events within the operating system and its applications or inspecting network traffic Liao *et al*. [2013]. To automate this process, mechanisms known as Intrusion Detection Systems (IDS) are employed. IDS can be classified based on the source of information they analyze—Host-based IDS (HIDS) for host-based monitoring and Network-based IDS (NIDS) for network-based monitoring—or based on the approach used for detection—signature-based IDS, which detects known attack patterns/signatures, or anomaly-based IDS, which identifies abnormal behavior Bace and Mell [2001]. This work focuses specifically on network-based intrusion detection using the signature-based approach.

2.1.1 NIDS (*Network Intrusion Detection System*)

Intrusion Detection Systems (IDS) in networks are programs designed to examine network packets to detect attack patterns. Often, Network IDS (NIDS) must perform Deep Packet Inspection (DPI) Lin *et al*. [2008] on each packet, which significantly impacts their performance.

As indicated in Figure 1, port mirroring is employed when NIDS are in execution. This means that the NIDS process copies of packets and, therefore, is non-blocking, allowing normal packet reception to continue uninterrupted. Figure 1 also illustrates that NIDS takes as input a configuration file and a rules file. The configuration file allows for defining variables and behaviors that the NIDS will adopt, such as specifying the local network address or instructing the system not to perform DPI on encrypted packets. The rules file serves the purpose of defining attack signatures, specifying which patterns the system should detect through DPI to indicate in its logs that there has been an intrusion attempt. Figure 2 illustrates an example of a rule supported by NIDS like Snort and Suricata.

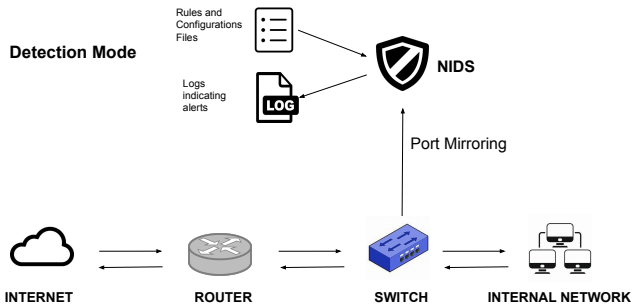

Figure 1. NIDS Operating with Mirrored Traffic

The first field specifies the action to be taken if the rule is matched. The second and third fields denote the source IP address and source port, respectively. Following the arrow, we have the destination IP address and port. The part enclosed in parentheses is known as the rule options. In this case, this rule will generate an alert only if, when the packet is processed: i) the source port is 13; ii) the destination address is within the network 192.168.1.0 (after NAT) and the destination port is 99; iii) the string 'pattern' is found anywhere in the packet payload. The sid (signature id) refers to the unique identifier of the rule (signature) and is not related to the DPI process.

```
alert tcp any 13 -> 192.168.1.0/24 99 (content: "pattern"; sid:156)
```

Figure 2. Example of an IDS Rule

The trigger for a rule to generate an alert is that all patterns contained within it must be present in the packet. If any of the loaded rules generate an alert, the system stops analyzing the current packet. In other words, for each packet, the system will test each loaded rule until one of them matches. The loaded rules are evaluated from top to bottom, and once one rule generates an alert, no further rules are evaluated. This means the rules operate under a logical OR condition. Since all patterns within a rule must be present in the packet, patterns within a rule form a logical AND condition.

2.1.2 Contents and Fast Patterns

Rules in an IDS can include an option named *content*. The *contents* within a rule represent the patterns that the system should find in the packet for that rule to generate an alert. A rule can contain zero or more *contents*.

Every rule in an IDS (which contains one or more *contents*) has a single *fast pattern*, which can be specified in the rule using the keyword *fast_pattern*, followed by a *content*. If no *fast pattern* is specified, the IDS will use the longest *content* in the rule as the *fast pattern*. Figure 3 illustrates the use of *fast*

patterns, showing that the string ”pattern” is the *fast pattern* in this rule. Note that if the keyword “fast_pattern” were not present, the *fast pattern* of the rule would be “otherPattern”, since it is the longest *content* in the rule.

```
alert tcp any 13 -> 192.168.1.0/24 99 (content: “pattern”;
fast_pattern: “otherPattern”; sid: 1;)
```

Figure 3. Explicit *Fast Pattern* in an IDS Rule

This mechanism exists for performance reasons. In order to conserve computational resources during the processing of a single packet, the system attempts to match the payload of the analyzed packet only against the *fast patterns* of the loaded rules. If none of these *fast patterns* are found in the packet, there is no need to evaluate it further against the remaining *contents*. A *fast pattern* should be chosen such that, if found, there is a high likelihood that the represented rule will be matched. In other words, the *fast pattern* should be the most representative pattern, one that best defines the attack. If the *fast patterns* are common and/or small strings, the system’s performance will be severely impacted because all other patterns in the rule will be checked once the *fast pattern* is present.

2.2 BPF

BPF (Berkeley Packet Filter) was proposed in 1993 with the aim of optimizing the performance of *tcpdump* Roesch *et al.* [2024b]. The classic BPF, as it became known, could compile a *tcpdump* filtering expression into BPF instructions, which were then executed in a virtual machine within the Linux kernel.

Recent versions of this technology, called eBPF (Enhanced BPF), have become more versatile. BPF programs are now used as metrics and/or tools to measure/process events within the kernel. By utilizing hooks present in the kernel, it is possible to instrument syscalls, kernel functions, interrupts, and more. For example, counting how many times the ‘fork()’ syscall has been executed using a BPF program that acts as a callback, updating a counter each time the syscall is invoked. In other words, the BPF program has been attached to a kernel hook that calls it every time ‘fork()‘ is invoked.

With the advent of this technology, it became possible to define user-level programs to run in the BPF virtual machine (VM) within the kernel. The BPF VM acts as a sandbox, meaning that before a BPF program is loaded, a verifier statically analyzes the code and ensures that a set of restrictions is met. Among these, notable restrictions include the maximum number of instructions in a BPF program, which cannot exceed 1 million, and the verification of each memory access (not in the stack section) to ensure it is valid (not out of bounds).

2.2.1 BPF Maps

BPF also facilitates inter-process communication (IPC) between kernel space and user space through key-value data structures known as maps. To create maps in kernel space, it is possible to define a section in the BPF file, specifying the map type, as well as the key and value types, the maximum number of entries, and the map’s name. Maps provide IPC because both the user and kernel programs can write to/read from them. In other words, they allow the exchange of program data through shared memory.

There are currently 33 different types of BPF maps. A few of them represent generic data structures; for example, BPF_MAP_TYPE_HASH, BPF_MAP_TYPE_ARRAY, BPF_MAP_TYPE_QUEUE, and BPF_MAP_TYPE_STACK, which allow the storage of user-defined structures. On the other hand, there are specific-purpose BPF maps, for example, BPF_MAP_TYPE_XSKMAP, which is filled with descriptors to XDP sockets, allowing packet redirection from the kernel to user spaces; and BPF_MAP_TYPE_PROG_ARRAY, which allows a BPF program to call another BPF program by storing a reference to the program in the map.

Sapo-boi uses mainly 3 types of maps: Hash maps, to represent Aho-Corasick automata; Array of Maps, a special type that allows for storing maps in an array; and XSK maps, to perform XDP sockets operations.

2.2.2 BPF helper functions

Functions responsible for interacting with the system or execution context called from BPF programs are known as helper functions. These functions can be used, for example, to print debug messages, return the time elapsed since the system boot, and manipulate network packets, among other tasks.

2.3 XDP

XDP (eXpress Data Path) is a hook located in the kernel’s network stack. Figure 4 illustrates this structure.

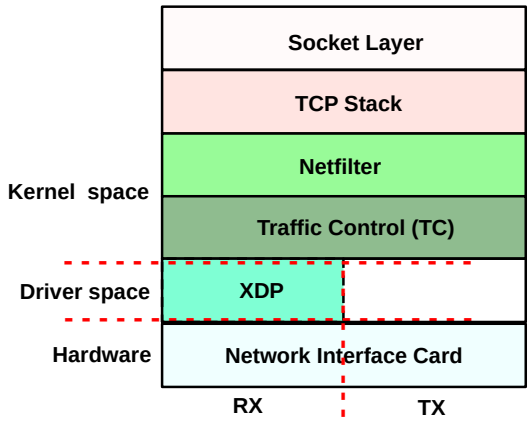

Figure 4. Linux Kernel Network Stack (Adapted from Vieira *et al.* [2020])

Since XDP (eXpress Data Path) is the first layer after the network interface for incoming packets, a BPF program running at this layer represents the earliest possible point at which a received packet can be processed by a user-defined program.

An XDP program, which is what a BPF program running on the XDP hook is called, can return 5 types of actions: 1) XDP_PASS: The packet continues to the network stack. 2) XDP_DROP: The packet is dropped. 3) XDP_TX: The packet is transmitted back out the same network interface it was received on. 4) XDP_ABORT: Indicates an error during

processing; the packet may be dropped. 5) XDP_REDIRECT: The packet is redirected to another network interface, to another CPU, or to a user space program.

Note that before taking any of these actions, the packet can be processed and modified by the XDP program, provided it adheres to the restrictions imposed by the BPF verifier.

2.4 AF_XDP

BPF programs are limited in the number of instructions they can execute. If more processing is required, the packet must pass through the network stack to reach a user space program.

XDP address family sockets (AF_XDP or XSK) are designed to redirect incoming packets from the kernel to user space. In this regard, they are similar to sockets in the AF_PACKET family, but they offer superior performance because no copy of the packet is sent to the network stack. This allows for a partial bypass of the kernel's network stack (partial because there still needs to be an XDP program that redirects the packet, which is part of the network stack).

AF_XDP sockets are directly tied to a user space memory region called UMEM (User Mode Memory). This region is divided into buffers of equal size, where packets transferred from or destined to the kernel are stored. The size and number of these buffers can be configured when the user program allocates memory for UMEM. In addition to the buffer set, UMEM also includes two rings: the fill ring and the completion ring.

The fill ring stores the relative addresses of packets whose ownership has been transferred from user space to the kernel, while the completion ring stores the relative addresses of packets that now belong to user space. Here, the relative address denotes the offset of the packet from the beginning buffer of UMEM to which that position in the ring refers, as depicted in Figure 5.

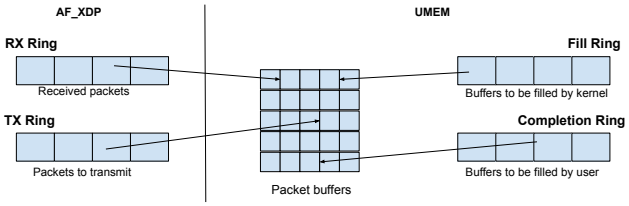

Figure 5. AF_XDP Sockets Architecture (Adapted from Karlsson and Töpel [2018])

An XDP socket, also allocated by the user, has 2 rings: RX (receive) and TX (transmit), for received and transmitted packets, respectively. When a packet is received, the kernel populates a descriptor in the RX ring. This descriptor points to a packet buffer in UMEM where the packet is stored, indicates its offset within that buffer, and specifies the size of the packet.

To receive packets through XDP sockets, a user program must first write relative addresses into the fill ring and then submit these addresses to the kernel. The kernel subsequently reads an address (the first one added by the application, following FIFO policy) and writes the packet data to the indicated address in UMEM. Afterward, the kernel writes this address to the RX ring of the socket that received the packet. The application then reads the address from the RX ring (which is the first address it placed in the fill ring).

The next steps are as follows: (i) The application reads the packet by reading from the specified address in UMEM. (ii) After processing the packet, the application recycles the address by placing it back into the fill ring for future packet reception.

Each socket is attached to a queue on the network card. When initializing a socket, the user-space process needs to specify which network card and which queue on that card the socket will be associated with. Sockets associated with different queues are unaware of traffic redirected by others.

Each socket is connected to a UMEM, but a single UMEM can be associated with multiple sockets as long as they are on the same queue of the network card. In other words, the UMEM is indirectly associated with a single queue on the network card.

All four types of rings described are FIFO (First-In-First-Out) data structures. The fill and TX rings are considered producers (written by the application and read by the kernel), while the RX and completion rings are consumers (written by the kernel and read by the application).

For the producers (fill and TX rings), the application must reserve slots in the ring. Once these slots are filled with relative addresses from UMEM, they should be submitted to the kernel.

For consumers, the user program needs to peek into the ring: if packets are available, they can be consumed. Once the packets from a slot are read, the slot should be released so that the kernel can fill it again.

2.5 Aho-Corasick Algorithm

The Aho-Corasick algorithm is used for pattern matching in a long text. This algorithm is efficient in finding an arbitrary number of substrings within a given string. Therefore, it aligns well with the objectives of an IDS, as these systems aim to identify malware signatures in the payload of a packet. For this reason, the Aho-Corasick algorithm is utilized by Snort and Suricata Waleed *et al*. [2022].

The output of the Aho-Corasick algorithm is an automaton capable of recognizing any of the patterns provided as input within any given string. Table 1 illustrates an example of a state transition table for an automaton with patterns "ar" and "ara". The table shows the current state, the transition character, the new state, and which pattern was matched. It abstractly represents the automaton. When iterating over a string, each transition can be executed in a single line of code using a key-value data structure, such as BPF maps, specifically BPF_MAP_TYPE_HASH.

Table 1. Output of Aho-Corasick Algorithm for Strings "ar" e "ara"

State	Transition	New State	Pattern id
0	a	1	-
1	r	2	1 (ar)
2	a	3	2 (ara)
1	a	1	-
3	r	2	1 (ar)

The algorithm has a time complexity of $O(n + m + z)$,

where n is the size of the string in which patterns are being searched, m is the sum of the sizes of all patterns, and z is the number of patterns found in that string. Therefore, it can be concluded that the smaller the size of the automaton, the better the performance of this algorithm.

2.6 BTF

The BPF Type Format (BTF) emerged with the goal of enabling sections with debugging information in BPF object code, similar to DWARF, which allows the creation of such sections in object files generated from C and C++ code. An evolution of BTF has allowed writing in a metadata information section, which can be sent along with the packet through the XDP program to the user application via AF_XDP sockets.

2.7 Softirq

Every time a packet arrives through the network interface card, it triggers a hardware interrupt (IRQ - Interrupt ReQuest). When this happens, the kernel takes over from a process executing on a CPU core and handles the interrupt (forced preemption). Subsequently, the kernel's network stack needs to process the packet. This processing is carried out by softirqs, which are software tasks executed immediately after a hardware interrupt.

The entire kernel network stack operates within the context of softirqs. For incoming packets, the NET_RX_SOFTIRQ softirq handles the processing. Since XDP programs reside within the network stack, they execute in the context of softirqs as well. Therefore, to measure the CPU time of an XDP program, one should observe the time that the processor spends in softirq context.

2.8 Perf events

Perf events represent a generic way to send data from a BPF program to a user-space process. To use them, it is necessary to set up a BPF map of type *BPF_MAP_TYPE_PERF_EVENT_ARRAY*, which enables Inter-Process Communication (IPC) between programs in different spaces.

3 Related Work

Network attacks occur frequently for various reasons, including politics, economics, society, racism, sports, games, and attempted fraud Abhishta *et al*. [2020]. Therefore, effective ways to detect such abuses become crucial. Snort Roesch *et al*. [2024a] emerged in 1998 as an evolution of *tcpdump*, capable of pattern detection in the payload of network packets. Starting from 2021, with the release of version 3, Snort transitioned into a multi-threaded IDS.

Suricata emerged in 2010, intending to be more efficient than Snort, using a multi-threaded approach from its inception. There is a wide range of studies comparing the performance of these two systems, most of which used single-threaded Snort Park and Ahn [2017] White *et al*. [2013] Albin and Rowe [2012] Murphy [2019], all showing Suricata's significant superiority in terms of CPU and memory usage and packet loss rates. However, a more recent study Waleed *et al*. [2022] shows that there is now a closer proximity between the IDS, even though Suricata remains more efficient. The study also demonstrates that Snort's packet loss rate became insignificant at transmission rates lower than 1 Gbps, and the performance difference using different approaches for pattern recognition and packet capture. A 2020 study Hu *et al*. [2020] also compares Snort 3 with Suricata, this time using higher transmission rates, from 10 to 100 Gbps. As expected, running those solutions in conjunction with the host destination means the latter does not receive packets in their entirety, as the presence of the systems increases CPU time in softirq context, consequently resulting in packet loss.

Both Snort and Suricata allow the use of BPF for traffic filtering. These filters work the same way as those in tcpdump, even having the same syntax. However, neither of them allows pattern matching in kernel space. A BPF system capable of performing DPI in a video streaming and Internet of Things context is described in Baidya *et al*. [2018], but the system only works for specifically formatted packets, which does not suit IDS general purposes.

In the context of networking with BPF, it's demonstrated in Viljoen and Kicinski [2018] and Xhonneux *et al*. [2018] that it is possible to perform packet switching and routing using BPF, and even offload these functionalities to smart network interface cards. *In-Kev* Ahmed *et al*. [2018] allows the construction of Service Function Chains (SFCs) using consecutive calls to BPF programs. The system provides the capability to build Network Functions Virtualization (NFV) functions directly in the kernel, at any layer of the network stack.

Scholz *et al*. [2018] provides detailed insights on the costs associated with applications regarding function instrumentation and performance analysis of BPF programs aimed at firewall systems. To do so, they compare XDP programs with iptables and nftables and, besides being published in 2018 when the restrictions imposed by the BPF verifier were more strict than today, they showed that XDP programs exhibited superior performance related to packet loss and CPU usage.

Shuai and Li [2021] introduces a modification in Snort to enable packet capture using the Data Plane Development Kit (DPDK), showing that the rate of analyzed packets (and consequently the rate of generated alerts) reaches rates close to 100%. However, CPU usage tends to be at considerably higher rates.

Kostopoulos [2024] details an anomaly detection system based on BPF whose idea is to demonstrate that it is possible to extract features from known attacks and train a specific machine learning model developed for the task. The structures representing the trained model are then stored in BPF maps. Using the payload of the incoming packet along with the weights and biases of the model, it is possible to calculate a value that the author referred to as the "Malicious Indicator Number", which determines the verdict for that packet. As this work's goal is to compare the computational performance of signature-based solutions, Kostopoulos [2024] is out of scope, due to proposing an anomaly IDS.

The work most similar to this is proposed by Wang and

Chang [2022]. In their study, the authors detail a system architecture that, similar to this work, consists of two programs: one in user space and another in XDP, in which only suspicious packets transition from kernel to user space. The main difference lies in how communication occurs between these two processes. While the authors of Wang and Chang [2022] chose to send packets and data through perf events, this work utilizes XDP sockets, designed precisely for this purpose. Regarding the results, the maximum throughput, which is the maximum transmission rate possible before the overhead generated by the system results in packet loss, is three times higher than that of Snort. Suricata and other systems were not compared. Additionally, the study shows that the CPU time spent in softirq context (processing the kernel network stack) in their system is lower than that used by Snort.

Table 2 shows the scope of the related work discussed in this section, which is the most similar to the current. The second left-most column shows if authors were able to perform DPI in kernel context, followed by a column that indicates if there is a comparison with/between Snort and Suricata. Next, a column representing whether the work proposes a system, or simply compares existing work, and lastly, if the authors evaluated different approaches to capture packets for posterior analyses.

Table 2. Scope of Related Work

Work	DPI with BPF	Comparison (Snort3 - Suricata)	System Proposed	Capture Method Evaluation
Baidya *et al*. [2018]	Yes	No	Yes	No
Waleed *et al*. [2022]	No	Yes	No	Yes
Hu *et al*. [2020]	No	Yes	No	Yes
Kostopoulos [2024]	Yes	No	Yes	No
Wang and Chang [2022]	Yes	No	Yes	No
This work	Yes	Yes	Yes	Yes

4 Project and Implementation

The proposed solution in this work, named **Sapo-boi**, consists of two main modules: (i) Suspicion Module, implemented as an XDP program running at the kernel level, responsible for identifying fast patterns from loaded rules; and (ii) Evaluation Module, a program executed as a user-level process, whose objective is to determine if other patterns (*contents*) related to the rule referenced by the fast pattern detected in the Suspicion Module are present in the suspected packet.

Figure 6 depicts the high-level architecture of Sapo-boi. On the left side, the system's initialization phase is represented. Initially, the Evaluation Module is responsible for loading the Suspicion Module (XDP program) within the kernel, as well as the BPF maps that the component will use. Consequently, the Evaluation Module must process the rules and populate two structures: (i) BPF maps, which represent Aho-Corasick automata constructed from the fast patterns of the rules; and (ii) contents automata, graph structures residing in user space until the end of the NIDS execution, responsible for pattern detection in packets deemed suspicious.

Once all the BPF maps and user-space structures are populated, the execution phase begins. In this phase, incoming packets are processed by the Suspicion Module. If a packet is considered suspicious, it is redirected to user space via an XDP socket for further processing in the Evaluation Module. If the latter determines that the packet is indeed malicious, it will write to the log file, indicating which rule triggered the alert.

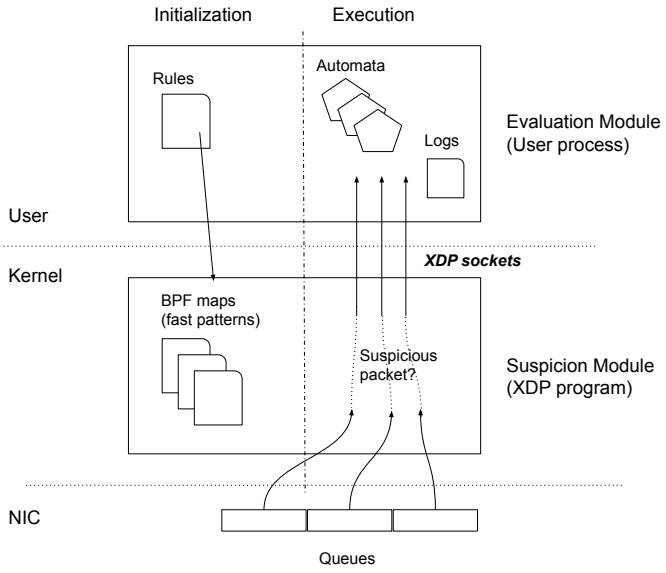

Figure 6. Sapo-boi's Architecture

It is worth noting that, like traditional NIDS (Figure 1), the proposed solution processes copies of the packets; that is, in the event of an XDP_DROP action (packet discard), the original traffic is not affected in any way. Thus, if no fast pattern is found by the Suspicion Module, the copy of the packet under analysis is immediately discarded. In this situation, the packet discard action indicates that it was considered safe and does not require further processing by the Evaluation Module.

On the other hand, in the presence of a fast pattern in a packet under analysis, the Suspicion Module sends the latter to the Evaluation Module via XDP sockets. To achieve this, the solution relies on BPF maps (*BPF_MAP_TYPE_XSKMAP*), which must be created and initialized from the Evaluation Module, allowing communication between the two modules. Once the BPF maps are prepared, the Evaluation Module enters a standby mode, awaiting the arrival of suspicious packets.

The following subsections detail the operations performed by each of the Sapo-boi modules.

4.1 Suspicion Module

The Suspicion Module receives network traffic directly from the interface, without any prior processing. Technically, this means that the interpretation of the headers of the traffic units (frames and packets) is performed by the module itself. For example, when considering a TCP flow, the module must initially identify the Ethernet header, followed by the IP and TCP headers, to finally be able to inspect the payload.

It should be noted that incoming traffic may present encapsulation with VLAN headers. Thus, the Suspicion Module will discard these headers until the corresponding Ethernet header is detected.

Once the Ethernet header is obtained, the `ethertype` option is located, which indicates the next protocol related to the network layer. If the observed protocol is not IP (neither IPv4 nor IPv6), the packet is discarded. Otherwise, if an IP packet (either IPv4 or IPv6) is detected, the module will consider two

possible cases for the transport layer: TCP and UDP. Other possible protocols result in the packet being discarded.

Also, for the Suspicion Module to execute efficiently, multiple CPU cores must be used concurrently. This is only possible if the network card supports multiple queues for Receive Side Scaling (RSS). Specifically, RSS is responsible for allowing traffic to be distributed among processors through indirection tables Woo and Park [2012]. The processing performed by the Suspicion Module will be detailed in the following subsections.

4.1.1 Kernel space pattern detection

Once the received traffic is identified, the Suspicion Module checks for the existence of fast patterns using BPF maps of type hash (BPF_MAP_TYPE_HASH). These maps allow for the definition of abstract key/value structures, representing the states of an Aho-Corasick automaton. Specifically, the keys represent a state of the automaton (integer) and a transition (character/byte), while the values define the resulting state of the transition. Table 1 accurately shows the structure of the map, where the keys are represented by the current state and the transition character, and the value is the resulting state, as well as a flag indicating if this is a final state (meaning that there is a match).

Note that the pattern matching is performed by the Suspicion Module by iterating over every byte of the analyzed packet payload, and a state transition in the Aho-Corasick automaton is performed by checking if a key belongs to the map, being a key defined by the evaluated byte and the current state of the automaton. Since hash maps are optimized for search, the transition is done in constant time.

4.1.2 TCP and UDP ports

As discussed in Section 2.5, the Aho-Corasick algorithm becomes more efficient for smaller automata, as both the total size of the patterns and the potential number of pattern combinations are reduced. Therefore, to create smaller automata, we divided the rules by the tuple composed of the source and destination ports of the packets. We also divided the rules applied to TCP and UDP packets, composing even smaller automata. Each source/destination pair is referred to as a port pair.

This form of segmentation provides a way to separate IDS rules so that the generated automaton contains only the fast patterns of rules related to a specific port pair. Thus, the fast patterns of the rules on each port pair are mapped to a single automaton. Figure 7 illustrates an example of separating rules into port pair groups. Note that there are two types of port pair groups, one for UDP and one for TCP. It can be observed that, for UDP, the rule with signature ID (sid) 4 is present in all UDP port pair groups, and for TCP, the same happens for the rule with sid 8. Therefore, the patterns will be inserted in all automata, since specific source and destination ports are not specified. This phenomenon can impact the IDS performance, as the fast pattern corresponding to sid 4 will be tested for all incoming traffic.

When the Suspicion Module verifies that a packet contains TCP or UDP headers, it retrieves the source and destination

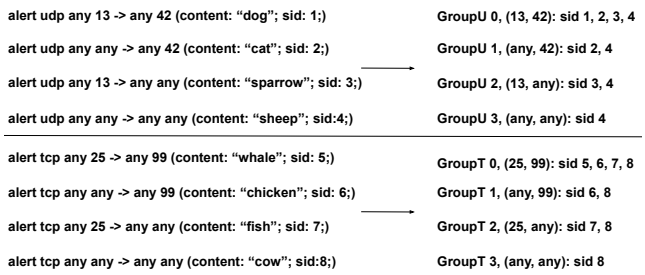

Figure 7. Rule Grouping by Port Pairs

ports from the header and then analyzes only the automaton relevant to that port pair. If no automaton is found, the packet is discarded because there is no loaded rule that could consider the packet malicious.

The automaton related to the defined port pair is made available to the Suspicion Module through a BPF map, as demonstrated in Figure 8. Each entry in the map has a key composed of two 16-bit integers representing the source and destination ports. The value stored for this key is an index pointing to a map that informs the Suspicion Module where the automaton to be analyzed is located. If there is no entry in the port pair map, it means that no rule has been loaded for the packet’s port pair, resulting in processing interruption and packet discard.

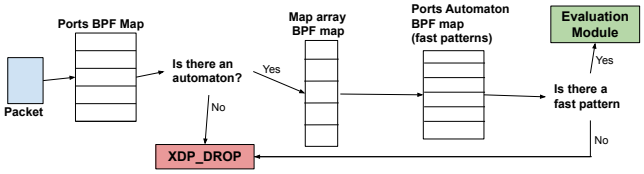

Figure 8. Packet Path in Suspicion Module

Once the relevant index is determined, the program accesses the specific automaton for the port pair of the packet under analysis. Then, from this automaton, if a fast pattern is found in the packet’s payload, it is sent to the Evaluation Module; otherwise, the packet is discarded.

4.1.3 Metadata

When a fast pattern is found in a packet, the Suspicion Module redirects the entire packet, along with a set of metadata, to the Evaluation Module. In the XDP context, metadata is defined by an abstract data structure added before the start of the packet. To add such metadata, the program resets the pointer to the beginning of the packet and then calculates the size of the structure, thereby increasing the data space as needed. This operation is performed using the `bpf_xdp_adjust_meta` helper function.

In general, the Suspicion Module sends three pieces of information as metadata: (i) the index in the map array, obtained through the port pairs map; (ii) the index of the identified rule determined by the found fast pattern; and (iii) an integer representing the offset between the start of the packet and the start of the packet’s payload. These pieces of information allow the Evaluation Module to efficiently identify the rule that needs to be evaluated, as it can tell beforehand which port pair the rule belongs to and the index of the rule within that port pair, as well as where to look in the packet.

4.1.4 Packet redirecting to Evaluation Module

Once all the steps depicted in Figure 8 are completed, the suspicious packet and its metadata are forwarded to the Evaluation Module via an XDP socket. For this purpose, there needs to be an XSK type BPF map, where a key corresponds to the index of the network card queue from which the packet was received, and the value designates the socket descriptor responsible for handling the redirect. The redirecting process is carried out using the `bpf_redirect_map` helper function.

4.1.5 Suspicion Module Overview

As one could see in the previous sections, the Suspicion Module is an XDP program able to fetch a packet's port pair and an Aho-Corasick automaton map associated with that port pair. It is also responsible for redirecting packets to the Evaluation Module if a malicious pattern is found. Algorithm 1 shows how port pair and automaton map fetching occur. Note that the XDP program receives a pointer to the beginning of the packet, parses every header until it finds the TCP/UDP port pair, and then uses this information to look up the appropriate automaton map in the BPF map arrays discussed in Section 4.1.2. Note that the packets themselves do not need to be modified to serve as input to the Aho-Corasick algorithm; it is only necessary to parse them until reaching the payload area. If no automaton map is found in the specific array of maps, the packet is discarded, meaning the port pair in the packet does not have any automata to be analyzed.

On the other hand, Algorithm 2 describes automaton lookup via BPF maps. As input, there is the `port_pair_index`, the `offset` until reaching the packet's payload, and the `automaton_map`, all of them obtained as described in Algorithm 1. Pattern matching is done by iterating over every byte of the payload of the packet. If there is a match between the current state and the evaluated byte, the state should be updated. If a final state is reached, the program allocates space for packet metadata, fills the information described in Section 4.1.3 (received as input), and sends the packet to userspace via XDP sockets. Note that the `bpf_redirect_map` helper function receives a descriptor to an XDP socket, obtained from the queue id on which the packet was received. It is also important to highlight that if no pattern is found when looping through the whole payload, the packet is dropped.

Algorithm 1 Suspicion Module Action when a packet is received:

Require: pkt_start: pointer to beginning of packet
Require: TCP_map_array: TCP map array of maps
Require: UDP_map_array: UDP map array of maps
$eth\_type \leftarrow parse\_eth\_hdr(pkt\_start)$
$pkt\_start \leftarrow pkt\_start + sizeof(eth\_hdr)$
if $eth\_type = IPv4$ **then**
  $ip\_type \leftarrow parse\_ipv4\_hdr(pkt\_start)$
  $pkt\_start \leftarrow pkt\_start + sizeof(IPv4\_hdr)$
else if $eth\_type = IPv6$ **then**
  $ip\_type \leftarrow parse\_ipv6\_hdr(pkt\_start)$
  $pkt\_start \leftarrow pkt\_start + sizeof(IPv6\_hdr)$
else
  return XDP_DROP
end if
if $ip\_type = TCP$ **then**
  $is\_TCP \leftarrow$ TRUE
  $src\_port, dst\_port \leftarrow parse\_tcp\_hdr(pkt\_start)$
  $pkt\_start \leftarrow pkt\_start + sizeof(TCP\_hdr)$
else if $ip\_type = UDP$ **then**
  $is\_TCP \leftarrow$ FALSE
  $src\_port, dst\_port \leftarrow parse\_udp\_hdr(pkt\_start)$
  $pkt\_start \leftarrow pkt\_start + sizeof(UDP\_hdr)$
else
  return XDP_DROP
end if
$ports.src\_port \leftarrow src\_port$
$ports.dst\_port \leftarrow dst\_port$
if is_TCP **then**
  $automaton\_map, port\_pair\_index \leftarrow lookup(TCP\_map\_array, ports)$
else
  $automaton\_map, port\_pair\_index \leftarrow lookup(UDP\_map\_array, ports)$
end if
if $automaton\_map = NULL$ **then**
  return XDP_DROP
end if

Algorithm 2 Suspicion Module Action when looking for a pattern:

Require: $port\_pair\_index$: port pair map index fetched in Algorithm 1
Require: $offset$: packet offset until reach payload
Require: $automaton\_map$: automaton map fetched in Algorithm 1
$xsk\_desc \leftarrow$ XDP socket descriptor for this NIC queue
$state \leftarrow 0$
for $byte \in packet\_payload$ **do**
  $state \leftarrow lookup(automaton\_map, state, byte)$
  if $state.is\_final$ **then**
    $meta \leftarrow bpf\_xdp\_adjust\_meta()$
    $meta.rule\_index \leftarrow state.rule\_index$
    $meta.port\_pair\_index \leftarrow port\_pair\_index$
    $meta.offset \leftarrow offset$
    return $bpf\_redirect\_map(xsk\_desc)$
  end if
end for
return XDP_DROP

4.2 Evaluation Module

The Evaluation Module is not limited to processing rules based on suspicions raised by the Suspicion Module; it performs a series of operations even before the latter starts functioning. These operations are essential for the correct and logical operation of the proposed NIDS and include program loading, map creation, metadata structure registration, rule processing and loading, and initialization of XDP socket maps and packet handling. Below, the complete set of operations performed by the Evaluation Module is detailed.

4.2.1 Suspicion Module loading and Maps creation

The first operation performed by the Evaluation Module is loading the BPF object file and attaching the XDP program to

a specific network interface. This operation not only loads the program that runs the Suspicion Module but also constructs all the maps present in the object file. In other words, the maps described in Section 4.1 are created at this moment, although they are not yet initialized.

4.2.2 Metadata registration

The second operation involves registering the BTF metadata structures. These structures enable the transfer of (meta)data from the kernel space to the user space along with the redirected packets, provided that all fields are properly registered with the kernel and alignment constraints are met.

4.2.3 Rule Processing

After creating the maps and registering metadata, the next step is processing the rules. The operations performed based on the loaded rules determine the port pairs maps, map array, and per-pair automata, as seen in Figure 8. The first step is to determine the port pairs, similar to what is shown in Figure 7. These pairs are structures that contain, among other elements, an array of rules. The pairs themselves are also stored in an array, as indicated in Figure 9. It is important to note that there is an array for TCP ports and another for UDP ports (differentiated according to the transport layer protocol).

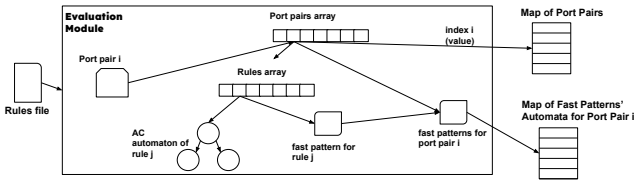

Figure 9. Structures in the Evaluation Module

Each rule within a port pair is handled distinctly regarding its fast pattern and other contents. The fast pattern of each rule is stored in a list, which will later be converted into an automaton and then into a specific port pair's BPF map. Figure 9 shows the fast patterns of port pair i being inserted into the same automaton map. In addition to state and transition information, this BPF map will have entries representing the final states of the automaton, with a field indicating which rule was matched. More specifically, the index of the rule in the pair's rule array.

Therefore, when the Suspicion Module finds a fast pattern, it can send to the Evaluation Module the index of the port pair array, as well as the index of the rule array within that pair, indicating the rule that needs to be evaluated. Thus, the Evaluation Module can locate the desired rule in constant time. The next step is to add the ports and the pair index to the port pair's BPF map (note that this map has a key of a port pair and a value of the index in the map array, as seen in Figures 8 and 9). This way, the same index:

- In the kernel map array, represents the map that contains the fast patterns for that port pair.
- In the user space port pairs array, represents the port pair whose rule has the same fast pattern that was found by the Suspicion Module (XDP program).

The contents, on the other hand, are treated individually for each rule. For each rule in the pair, an Aho-Corasick automaton will be created (unlike fast patterns, where one automaton is created per port pair). It is important to note that since this evaluation takes place in user space, individual rule automata should no longer be represented by BPF maps, but represented as graph structures instead. This way, there is no need to call BPF map helpers when analyzing the packet (in user space).

4.2.4 AF_XDP operations

After preparing the rules and patterns, the socket and UMEM (user memory region where the suspicious packets will lay) structures are initialized, along with the socket map. Initially, memory is allocated for the UMEM region where packet buffers will reside. In this proposal, 4096 frames of 4096 bytes each were allocated, resulting in a 16MB area per UMEM. This size was chosen to ensure that even with high transmission rates and sending by the Suspicion Module, the UMEM still has space to receive new packets. Next, the UMEM is registered in the kernel using the `xsk_umem__create` libbpf function. Note that each socket has an associated UMEM. In other words, the number of sockets and UMEMs used by the system is the same.

Therefore, the next step is to create the XDP socket associated with the allocated UMEM. This operation is performed by the `xsk_socket__create` function. After creating the socket and associating it with a UMEM, the next step involves adding relative addresses (offsets in a buffer, as described in Section 2.4) to the UMEM's ring fill. These addresses indicate to the kernel where to write the suspicious packets. Failing to complete this step would result in the inability to forward traffic to the Evaluation Module because the kernel would have no place to write the packets.

With the UMEMs and sockets initialized, they are finally added to the XDP socket map. To achieve this, a positive integer index is added as a key in the socket map. This index represents the queue index of the network card where the socket will operate. The value associated with the key is the socket file descriptor, obtained using the `xsk_socket__fd` function.

4.2.5 Packet managing and processing

After initialization of the AF_XDP actions, the Evaluation Module can start waiting for suspicious packets. At this point, a thread is associated with each XDP socket created in the previous step. Through the technologies used, there are two ways to receive packets: (i) continuously executing the `recvfrom` syscall for all socket descriptors; and (ii) using the `poll` function, which takes a list of socket descriptors and returns the number of descriptors where there is an event. For the proposed NIDS, the second option was chosen due to the ease of managing multiple sockets (and queues) through a unified function. When the `poll` function returns, the threads on which the associated XDP socket has an event are executed.

Thus, when events of interest occur, it is checked whether there has been any new write operation in the RX ring of the socket where an event was detected. During this write,

the next available memory space in the UMEM's ring fill is occupied. This space is then utilized by the Evaluation Module, aided by the `xsk_ring_cons__rx_desc` function, to obtain the relative address (offset from the beginning of the buffer) and the size of the received packet. After this process, the used memory space is returned to the UMEM's ring fill as available for reuse, ensuring address recycling.

With the relative address and size information, the `xsk_umem__get_data` function is used to obtain the absolute address, and consequently, the packet and metadata redirected by the Suspicion Module. Finally, the packets and metadata are processed according to the following pipeline:

- The metadata is processed and interpreted. This means the application knows which port pair is of interest, which automaton should be analyzed, and the offset within the packet that represents the start of the payload.
- If the Evaluation Module determines that the rule has no additional contents beyond the previously detected fast pattern, the rule is accepted, an alert is generated, and processing is terminated.
- If there are patterns to be analyzed, the Evaluation Module converts the packet payload into a character array and analyzes it using an automaton associated with the rule's previously detected fast pattern.
- If the total number of patterns found equals the number of patterns registered in the rule being analyzed, the rule is accepted, an alert is generated, and processing is terminated. Otherwise, the packet is considered falsely suspicious, no alert is yet triggered, and the Evaluation Module evaluates the payload again for possible fast patterns different from the one matched by the Suspicion Module. If there are fast patterns, the remaining rule contents for each matched pattern will be evaluated. Otherwise, processing is terminated.

5 Experimental Setup

The experiments focus on analyzing four NIDS solutions: (i) Sapo-boi, developed as part of this work; (ii) PF IDS, described in Wang and Chang [2022], but reimplemented in this work due to the unavailability of the source code; (iii) Snort, version 3.81.84; and (iv) Suricata, version 7.0.5.

PF IDS is divided into an XDP program and a user-space process, enabling fast pattern detection within the kernel and discarding packets deemed safe. It is important to highlight that we referred to the solution proposed in Wang and Chang [2022] as "PF IDS" because it utilizes **p**er**f** events to transfer packets from kernel space to user space, whereas Sapo-boi redirects them using XDP sockets.

The analyzes are primarily based on three factors: (i) the rate of non-analyzed packets; (ii) CPU usage across kernel, user, and softirq contexts; and (iii) the number of potential alerts missed during communication between modules for Sapo-boi and PF IDS.

Additionally, this section describes the network topology used in the tests, along with the specifications of its components. It is important to note that during the experiments, each NIDS was executed individually under the same set of rules, using identical hardware and the same underlying software. In other words, every effort was made to ensure the same execution conditions as much as possible for all four solutions.

Table 3. Description of the hosts executing the tests

Operating System	*Ubuntu 24.04 LTS; Kernel 6.8.0-31-generic; x86_64*
Network Interface Card	*Mellanox Technologies MT27500 [ConnectX-3] 10Gbps*
CPU	*12th Gen Intel(R) Core(TM) i7-12700; 16 cores*
RAM	16GB
Link	SPF+ E124936-D copper bidirectional

Regarding XDP in-kernel solutions (Sapo-boi and PF IDS), the tests were conducted by adding a memory copy operation to the driver space, immediately before the BPF program is executed. This adjustment was necessary to emulate the packet acquisition process performed by user-space solutions, which operate on copied traffic. In other words, the kernel used for testing Sapo-boi and PF IDS differs slightly from the standard kernel used for testing Snort and Suricata. The primary difference is a call to the `memcpy` function in the mlx4 driver, before invoking the XDP program.

The following subsections provide details about the experimental testbed and the auxiliary software used in the tests.

5.1 *Hardware* and *Software*

The network topology used for the experiments is described as follows: a host runs the evaluated NIDS, while a sender generates traffic to the host at various transmission rates (bandwidth). Table 3 presents the specifications of the two components mentioned (both have identical specifications). The network interface cards of the two machines are connected via a bidirectional Small Form-factor Pluggable (SFP+) cable.

5.2 Rules and Configuration

The rules used in the experiments are a modified subset of the Snort registered ruleset Roesch *et al*. [2024a]. As described in Wang and Chang [2022], two types of rules were removed to form the subset used. First, rules without a content option were excluded. Next, rules requiring Snort plugins and/or modules were also removed. In addition to that, Snort supports rules with content and port declarations negated; those rules were also excluded. The resulting subset was used to test Sapo-boi, PF IDS, and Snort. For Suricata, slight modifications to the ruleset were necessary because Snort and Suricata do not share the same syntax for rule construction.

Regarding the configurations, all preprocessors, plugins, and modules of both Snort and Suricata not directly involved in pattern detection were disabled, except for those that display processing statistics at the end of the NIDS execution. Snort and Suricata were tested using their default packet capture models. Specifically, Snort was tested with the libdaq library, which provides an abstraction layer for libpcap, while Suricata captured packets using AF_PACKET sockets.

5.3 Generating and Sending Traffic

In order to generate traffic, a program that analyzes the rules and forges malicious packets using the Scapy Python library was used. Additionally, the `iperf3` utility was used to gener-

ate non-malicious traffic. The traffic generated by these tools was stored in `pcap` files.

The traffic is sent from the sender to the host running the evaluated NIDS using the `tcpreplay` tool, which allows for setting the link's bandwidth, thereby enabling the evaluated NIDS solution to be stressed with higher or lower reception rates over time intervals.

6 Results Evaluation

This section presents and discusses the results obtained from the conducted experiments, considering the experimental setup outlined in Section 5.

6.1 Non-analyzed Packet Rate

This section details two approaches used to verify the rate of packets not analyzed by the systems. Unanalyzed packets are defined as those that reach the destination host where the NIDS solution is located, but due to high transmission rates, the number of rules, and the nature of the traffic, the solution is unable to capture the entire set of incoming packets for evaluation. For simplicity, the rate of unanalyzed packets will be referred to as the packet loss rate.

The first approach refers to the number of flows present in the transmitted traffic. In this context, the number of flows determines how many different source and destination IP-port pairs are in the traffic. It is important to note that the number of flows in the traffic may affect the NIDS solutions' ability to process packets, especially if the evaluated traffic is segregated by flow for load balance among available CPUs.

The second approach concerns the number of rules loaded by the NIDS. As expected, the higher the number of rules, the higher the rate of packets that the solutions fail to analyze. However, it is observed that in-kernel solutions experience a smaller variation in this rate, whereas Snort and Suricata present a significant increase in their packet loss rates as the number of rules grows.

Snort and Suricata provide the number of packets analyzed at the end of their respective executions, allowing for the calculation of how many packets were not evaluated, given the total number of packets sent. For Sapo-boi and PF IDS, a packet counter was added to the Suspicion Module (XDP program) via a BPF map, enabling the determination of the number of packets analyzed by these systems. The following subsections detail the experiments conducted for both approaches.

6.1.1 Number of Flows

Before diving into the results presented by the NIDS solutions regarding the number of flows present in the network traffic, it is important to understand how incoming packets are segregated for processing. In modern NICs, this is done via the RSS (Receive Side Scaling) algorithm. RSS detects IP-Port pairs and, based on a hash table, determines which CPU will process the incoming packet Woo and Park [2012].

Figure 10 shows the processed packet rate by CPU for 20- and 500-flow traffic. Both traffic scenarios contained the same number of packets of the same size. Note that for 500 flows, every CPU receives packets to process; however, for 20 flows, CPUs 0, 1, 2, 3, 8, and 13 remain idle. As a result, each active CPU must process a higher number of packets in order to evaluate all of them.

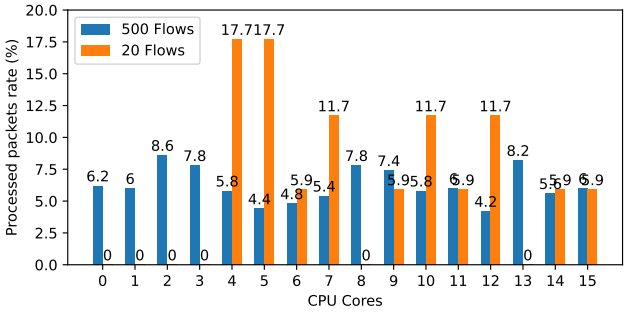

Figure 10. Packet Processing Rate

Figure 11 shows that Suricata exhibits lower packet loss rates as the number of flows the system analyzes increases. Suricata's strong performance is attributed to its ability to use RSS as a load balancer, which distributes the flows among the available CPUs. This ensures a balanced distribution of packets across CPUs, thereby maximizing the total number of packets analyzed.

However, when the number of flows is low, Suricata suffers from having all traffic redirected to a smaller number of CPUs for processing, which increases packet loss rates. Snort is unaffected by this issue, as the system acquires packets on a single core, meaning that packet processing occurs across multiple cores, but traffic capture is limited to one core. Figure 11 shows a comparison of the packet loss rates of the four analyzed solutions when loaded with 13,000 rules. All traffic (3 million packets, with 1,000 bytes of payload) sent to the host is free of malicious packets. The first graph shows traffic with 20 flows, while the second shows traffic with 500 flows.

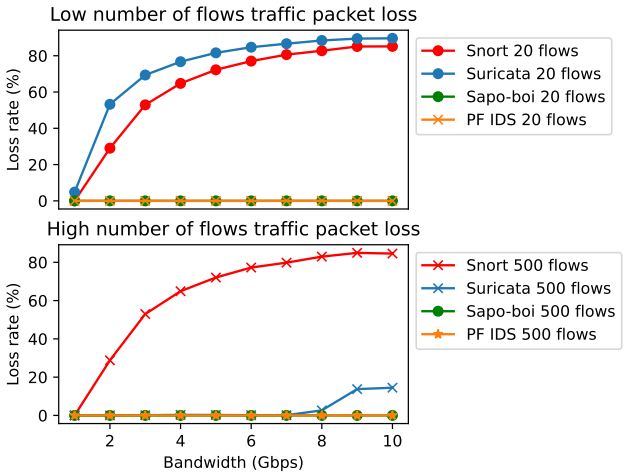

Figure 11. Packet Loss Rate

For kernel-based systems (Sapo-boi and PF IDS), packet loss remained at zero for all bandwidths, regardless of the number of flows. This is because there is no malicious traffic, so packets are considered safe as soon as the system recognizes that the source and destination ports will not trigger any alerts. Since XDP is capable of processing 24 million

packets per second on a 6-core machine Høiland-Jørgensen *et al*. [2018], these results are expected.

Regarding Suricata, it is noteworthy that high-flow traffic benefits the system. At 10 Gbps with low-flow traffic, Suricata experienced a packet loss rate of 89.61%. However, when the traffic type changed to high-flow, this rate dropped significantly to 14.68%. Additionally, the loss rates remain close to zero up to 7 Gbps, increasing to 2.65% at 8 Gbps, then 13.74% at 9 Gbps, and 14.68% at 10 Gbps. This experiment highlights the importance of user-space IDS solutions being able to detect and leverage the RSS technology available in modern NICs for packet acquisition. It also confirms the expected trend: the higher the bandwidth, the higher the packet loss rates.

Snort exhibited very similar loss rates in both scenarios, showing behavior opposite to that of Suricata and indicating that the number of flows does not affect the number of packets analyzed by the system. For example, at 8 Gbps with low-flow traffic, the loss rate was 82.76%, while with high-flow traffic, it slightly increased to 82.97%. At 10 Gbps, the loss rates were 85.15% and 84.57%, respectively. These results underscore that Snort lacks effective load-balancing mechanisms for data acquisition among processors.

6.1.2 Number of Rules

The number of rules loaded by a NIDS, as well as the bandwidth at which traffic is received, are the most critical factors for analyzing packet loss in traditional NIDS solutions. However, kernel-based systems, due to their implementation, tend to exhibit significant variations in loss rates primarily in response to changes in bandwidth, while showing minimal variation relative to the number of loaded rules.

Table 4 presents the results obtained for different systems, with varying transmission rates and numbers of rules. The high-flow traffic transmitted consists of 3 million packets, each with a 1000-byte payload size, of which 5% were malicious. Of the total packets, 90% were TCP, and 10% were UDP. Each data point in Table 4 represents the mean and standard deviation from 10 executions of the analyzed NIDS for each number of rules and bandwidth.

It can be observed that Sapo-boi and PF IDS have a low loss rate. This phenomenon can be attributed to the integration of the IDS into the kernel's network stack, ensuring that if a packet is processed by the stack, it has already been evaluated by the IDS. The differences in loss rates between Sapo-boi and PF IDS stem from the methods used by each system to send suspicious packets to the Evaluation Module and, primarily, from the significant variance in execution data across runs. However, when considering the standard deviation, the loss rates for both systems are effectively equivalent.

Another important observation is that both in-kernel solutions do not fail to evaluate a single packet when loaded with a single rule. This can be attributed to the fact that the loaded rule did not match any port pair in the traffic used, which allows the system to terminate processing before executing any pattern-matching routine, as explained in Section 4.

Snort exhibits high loss rates, failing to analyze over 50% of packets when loaded with 8000 rules at 2 Gbps, and over 90% at 8 Gbps. Notably, there is a significant increase in packet loss between 1 Gbps and 2 Gbps for all rule sets starting from 2000 rules, indicating that the packet arrival rate heavily impacts Snort's ability to analyze packets. Although not shown in the table, Snort's loss rate at 2 Gbps was 2.31% with 200 rules. However, this rate increases to 26.71% with 500 rules at the same bandwidth, highlighting the system's sensitivity to increases in the number of loaded rules.

Table 4. Packet Loss Rates for the Evaluated NIDS

Num. Rules	Bandwidth	Sapo-boi	PF IDS	Snort	Suricata
1	1 Gbps	0% ± 0	0% ± 0	0% ± 0	0% ± 0
	2 Gbps	0% ± 0	0% ± 0	0% ± 0	0% ± 0
	4 Gbps	0% ± 0	0% ± 0	0,07% ± 0,009	0% ± 0
	8 Gbps	0% ± 0	0% ± 0	0,13% ± 0,03	2,31% ± 0,03
	9 Gbps	0% ± 0	0% ± 0	0,2% ± 0,01	2,39% ± 0,04
	10 Gbps	0% ± 0	0% ± 0	0,2% ± 0,09	2,41% ± 0,05
2000	1 Gbps	1,84% ± 0,03	1,83% ± 0,01	0% ± 0	0% ± 0
	2 Gbps	2,19% ± 0,002	2,19% ± 0,30	44,3% ± 0,004	0% ± 0
	4 Gbps	2,18% ± 0,003	2,19% ± 0,09	74,42% ± 0,005	0,28% ± 0,01
	8 Gbps	2,20% ± 0,02	2,20% ± 0,08	88,95% ± 0,03	2,99% ± 0,8
	9 Gbps	2,19% ± 0,02	2,19% ± 0,02	89,66% ± 0,05	3,55% ± 0,47
	10 Gbps	2,20% ± 0,02	2,20% ± 0,02	89.47% ± 0,06	3,83% ± 0,93
8000	1 Gbps	1,84% ± 0,04	1,84% ± 0,01	3,74% ± 0,009	0% ± 0
	2 Gbps	2,20% ± 0,03	2,19% ± 0,03	53,55% ± 0,01	0% ± 0
	4 Gbps	2,20% ± 0,03	2,23% ± 0,07	78,70% ± 0,005	0,49% ± 0,01
	8 Gbps	2,19% ± 0,02	2,19% ± 0,15	90,81% ± 0,02	4.17% ± 0,03
	9 Gbps	2,20% ± 0,02	2,25% ± 0,05	90,02% ± 0,05	5,24% ± 0,07
	10 Gbps	2,19% ± 0,01	2,22% ± 0,04	91,39% ± 0,04	6,22% ± 0,08
13000	1 Gbps	1,84% ± 0,03	1,84% ± 0,01	8,55% ± 0,01	0% ± 0
	2 Gbps	2,18% ± 0,02	2,22% ± 0,03	56,24% ± 0,009	0,002% ± 0,006
	4 Gbps	2,20% ± 0,01	2,20% ± 0,04	79,92% ± 0,004	0,61% ± 0,05
	8 Gbps	2,20% ± 0,02	2,20% ± 0,18	91,36% ± 0,02	5.49% ± 0,09
	9 Gbps	2,19% ± 0,01	2,18% ± 0,02	91,63% ± 0,05	6,77% ± 0,8
	10 Gbps	2,19% ± 0,03	2,22% ± 0,56	91,55% ± 0,05	7,88% ± 2,52
16000	1 Gbps	1,84% ± 0,03	1,85% ± 0,01	8,41% ± 0,01	0% ± 0
	2 Gbps	2,19% ± 0,01	2,18% ± 0,03	56,20% ± 0,01	0,012% ± 0,008
	4 Gbps	2,21% ± 0,03	2,18% ± 0,03	79,91% ± 0,004	0,73% ± 0,09
	8 Gbps	2,19% ± 0,01	2,22% ± 0,14	91,26% ± 0,02	7,88% ± 0,06
	9 Gbps	2,19% ± 0,01	2,20% ± 0,05	91,63% ± 0,05	8,60% ± 1,26
	10 Gbps	2,23% ± 0,01	2,22% ± 0,02	91,70% ± 0,05	9,61% ± 1,01

Snort analyzes more packets than Suricata when loaded with a single rule. However, as the number of rules increases, it becomes evident that Suricata maintains low loss rates, whereas Snort begins to fail to analyze a progressively larger number of packets. For instance, Suricata loses only 4.17% of packets at 8 Gbps with 8000 loaded rules, while Snort loses over 90% under the same conditions. These results demonstrate that Suricata scales more effectively than Snort.

For comparison, Sapo-boi and Suricata were tested with 16,000 rules, across bandwidths ranging from 7 to 10 Gbps. The results are shown in Figure 12. It is observed that Suricata's loss rates increase significantly, indicating that both the number of rules and the bandwidth indeed affect the system's ability to evaluate packets. In contrast, for Sapo-boi, the loss rate remains relatively stable up to 10 Gbps.

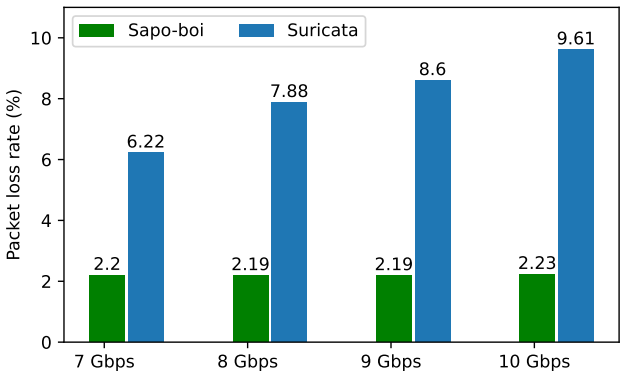

Figure 12. Sapo-boi and Suricata with 16,000 Rules

Figure 13 illustrates the point at which the loss rates between Sapo-boi and Suricata reverse. We excluded PF IDS from the analysis for visualization purposes since its loss ra-

tio is similar to those of Sapo-boi (considering the standard deviation). We also excluded Snort because of its high loss rate. The graphs for 1, 4000, and 7000 rules show Suricata's superiority, with only a small discrepancy between the rates compared to Sapo-boi. With 8000 rules, the loss rates are similar for both systems up to 5 Gbps. However, after 8 Gbps, Suricata experiences higher loss rates. The remaining graphs emphasize the growing disparity in loss rates beyond 8 Gbps, once again highlighting the stabilization trend in Sapo-boi and the increase in Suricata's loss rates. Considering that a NIDS needs to handle the entire traffic of a network, the packet rates can easily exceed 8 Gbps. Therefore, Sapo-boi is suitable for this topology.

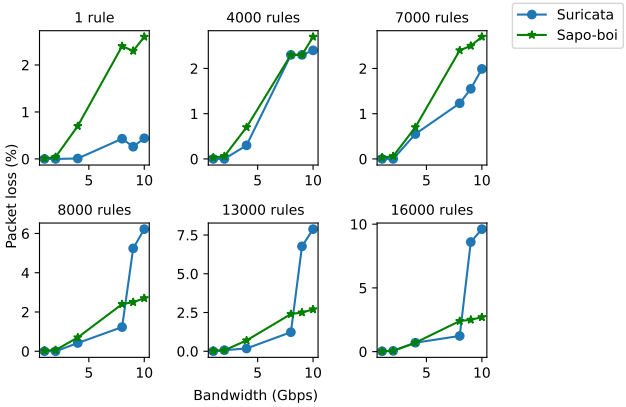

Figure 13. Packet Loss Rate Reverse in Sapo-boi and Suricata

In summary, four main findings emerged from the analysis:

- Snort and Suricata exhibit different packet loss rates. While Snort fails to analyze 5% of packets at 1 Gbps with 8000 rules, Suricata shows a similar loss rate at 9 Gbps with the same number of rules, highlighting Suricata's superior performance among user-space systems.
- Packet loss rates in user-space systems are heavily influenced by the number of loaded rules, whereas for kernel-based solutions, this influence is reduced. Sapo-boi does not overcome PF IDS when considering the increase in the number of rules. However, Section 6.3 will show that when considering module communication (packet forwarding to user-space module for completing evaluation), Sapo-boi does outperform PF IDS.
- Suricata performs excellently with high-flow traffic and a low number of rules. However, as the number of rules increases, packet loss rates also rise. As shown in Figure 13, there is always an inflection point where Sapo-boi outperforms Suricata when both solutions operate with 8000 or more loaded rules.

6.2 CPU Usage

Table 5 shows the CPU usage of the four evaluated systems for 1, 500, 8000, and 16000 rules. The data represents the mean of 10 executions for each NIDS. The standard deviation is not provided to maintain the table's readability, and in addition to that, the main goal of the table is to show the differences in CPU usage among the solutions, not the variation between executions of the same NIDS. Each cell in the table should be interpreted as follows: 'user' refers to CPU usage in the user context, 'kernel' represents CPU time in the kernel context, and 'softirq' denotes usage in the softirq context.

```
<user>-<kernel>**<sofirq>
```

The data in Table 5 represents the total CPU usage across each processor core. Since the host running the systems has 16 cores, the rates range from 0% to 1600%. CPU usage in user and kernel spaces was collected using the *pidstat* utility, while CPU time in the softirq context was gathered using the *mpstat* tool. The traffic used in the tests is the same as described in Subsection 6.1.2.

Table 5. CPU usage used by evaluated NIDS.

Num. Rules	Bandwidth	Sapo-boi	PF IDS	Snort	Suricata
1	1 Gbps	0-0**3	0-0**2	480-582**514	143-75**117
	2 Gbps	0-0**2	0-0**2	436-636**634	272-172**270
	4 Gbps	0-0**7	0-0**9	604-784**725	230-168**256
	8 Gbps	0-0**27	0-0**26	606-940**897	325-240**351
	9 Gbps	0-0**28	0-0**26	584-978**939	393-280**390
	10 Gbps	0-0**29	0-0**32	572-1000**971	421-316**424
500	1 Gbps	2-9**24	1-6**6	1372-226**225	256-82**121
	2 Gbps	1-8**56	1-7**16	1278-303**328	312-128**192
	4 Gbps	2-13**74	2-6**35	1296-304**305	299-150**208
	8 Gbps	3-9**118	3-8**58	1010-528**532	565-263**344
	9 Gbps	4-10**121	4-8**66	1013-585**588	624-303**381
	10 Gbps	4-10**177	4-8**81	978-618**625	670-339**404
8000	1 Gbps	4-18**43	7-9**11	1432-168**177	376-88**122
	2 Gbps	4-21**70	11-12**25	1346-252**279	421-109**154
	4 Gbps	5-24**103	12-13**33	1327-271**275	388-131**178
	8 Gbps	5-32**230	13-11**110	1121-490**495	876-303**337
	9 Gbps	6-33**236	13-13**157	1057-541**544	952-354**377
	10 Gbps	8-46**240	16-20**160	1019-581**584	1019-385**396
16000	1 Gbps	2-17**45	6-8**11	1391-228**228	500-67**88
	2 Gbps	6-26**84	10-12**25	1274-322**301	477-87**109
	4 Gbps	6-25**107	10-12**42	1315-284**285	925-167**158
	8 Gbps	7-32**208	11-14**177	1088-515**521	1234-314**338
	9 Gbps	7-32**291	15-15**181	1021-577**574	1199-404**400
	10 Gbps	7-32**300	16-12**220	976-612**614	1146-379**389

The first notable point in the table is the significant discrepancy between the values for the kernel-based IDS solutions (Sapo-boi and PF IDS) and the user-space IDS solutions (Snort and Suricata). This disparity can be explained by the fact that kernel-based systems discard non-suspicious packets immediately after evaluation. Since only 5% of the traffic in this test is malicious, these systems can discard 95% of incoming packets, thereby avoiding the complex processing typically performed in the kernel network stack, which is executed in the softirq context.

It's also important to highlight the zero CPU usage in both user and kernel contexts when Sapo-boi and PF IDS are loaded with a single rule. This behavior occurs because no packets are sent to the Evaluation Module of these systems. In other words, no operations involving XDP sockets or perf events are performed, resulting in 0% kernel space usage. Furthermore, as no packets are delivered to the Evaluation Module, user space usage also remains at 0%.

The third point worth highlighting is that Sapo-boi utilizes more CPU than PF IDS in both kernel and softirq contexts, while consuming less in user space. This occurs because additional processing is required in kernel and softirq contexts to perform operations with XDP sockets compared to perf events. While perf events only involve the creation and dispatch of events, the steps required for the proper usage and operation of XDP sockets are more complex. As detailed in Subsections 2.4 and 4.2.4, for each packet received by the Evaluation Module, several BPF helpers must be invoked to

maintain the correct relative address lists used by the modules for packet writing and reading.

The slightly higher processing time in user space for PF IDS, when more rules are loaded, can be attributed to the fact that perf events are received in a manner that requires extracting packet data from the event. This process is more resource-intensive than directly receiving the packet through XDP socket redirection.

Lastly, it is noteworthy that Suricata consistently outperforms Snort in the comparison between user-space systems. Additionally, it is interesting to observe that for these systems, CPU usage rates in the kernel and softirq contexts are similar. This similarity arises because packet capture and copying, which occur in the kernel, happen concurrently with network stack processing, also handled within the softirq context. In other words, for Snort and Suricata, kernel usage represents packet copying during network stack processing, while softirq measures the overall network stack processing, including packet copying.

6.3 Loss Rate Between Modules Communication

As the evaluated kernel-based IDS solutions must forward suspicious packets to the Evaluation Module (in user space) for verdict, it is important to assess the actual number of packets delivered to the user process. Figure 14 illustrates the number of non-received packets in user space across bandwidths ranging from 1 to 10 Gbps. Both solutions were loaded with a single rule, in a way that every packet received by the kernel module must be sent to user space. When in user space, only one pattern should be sought. The tests were conducted for 1 million packets, each of them with 1000 bytes of payload. The results presented in this section are the mean of 10 executions for each bandwidth.

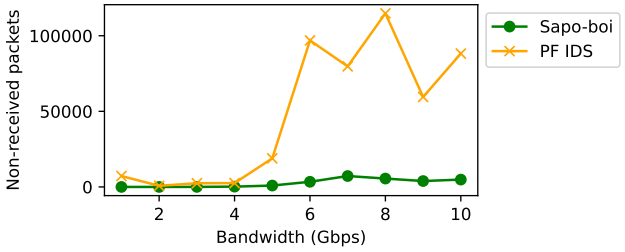

Figure 14. Packet Loss Rate for Kernel-based IDS Modules Communication

Figure 14 shows that SAPO-BOI is not heavily affected by increasing bandwidths when analyzing the number of non-received packets. On the contrary, PF IDS varies the loss rates from 5% to 11% after 6 Gbps. More specifically, Table 4 shows the mean and standard deviation of 10 executions for each of the evaluated bandwidths. One can see that the mean of non-received packets is larger for PF IDS in every bandwidth. Another important detail is the high standard deviation for PF IDS. This can be explained by the fact that PF IDS generates and sends perf events, which may not be successfully received by the user application, particularly at high receive rates, as perf events represent a generic method for sending packets (or any kind of user-defined events) to user space. Sapo-boi presents a lower number of non-received packets in the Evaluation Module because the proposed solution fully redirects the packets using XDP sockets, which represent an expert method for forwarding packets from the kernel to the user space.

Table 6. Number of non-evaluated packets in user space due to communication failure

Bandwidth	Sapo-boi	PF-IDS
1 Gbps	0 ± 0	7218.3 ± 21148.45
2 Gbps	0 ± 0	744.0 ± 276.5
3 Gbps	28.9 ± 107.3	2446.5 ± 1451.8
4 Gbps	201.4 ± 115.3	2540.5 ± 354.2
5 Gbps	915.4 ± 314.1	18912.8 ± 9605.9
6 Gbps	3405.8 ± 1785.7	96907.8 ± 57453.6
7 Gbps	7214.5 ± 10207.1	79741.0 ± 78023.2
8 Gbps	5541.4 ± 5289.8	114803.2 ± 144410.1
9 Gbps	3895.4 ± 1084.3	59499.0 ± 51424.5
10 Gbps	4860.0 ± 3386.6	88207.1 ± 104650.8

For both the solutions, but especially PF IDS, a higher bandwidth does not necessarily mean a higher number of non-received packets. One can see it in Table 6: with a 10 Gbps bandwidth, the loss was expected to be the greatest, but that was not the case. This can be explained mainly by the high standard deviation values, which can themselves be explained by the fact that both the solutions operate by executing *poll* functions. If the user process is too busy processing a packet to look for the remaining contents, the polling operation to fetch new packets may be impaired. Sapo-boi's superior performance can be explained for a *poll* call for events in XDP sockets is faster than a *poll* call for events in a perf buffer, which is the structure used as shared memory for IPC between kernel and user space used by PF IDS. Also, parallelism in communication contributes: Sapo-boi uses one XDP socket per NIC queue, each of them containing a specific UMEM area, whereas PF IDS uses a single perf buffer for the whole traffic.

Despite presenting a smaller number of non-evaluated packets, Sapo-boi still loses packets in module communication, even using XDP sockets. Karlsson and Töpel [2018] lists the main problems with this type of socket, exposing the reasons why user mode driver packages such as DPDK still can perform better. In our case, the environment used in tests did not support *ZERO_COPY* XDP. This means that, for every packet, there is a memory copy from the DMA region to the UMEM area, which negatively impacts performance. When using ZERO_COPY mode, the NIC can write the packet directly to UMEM, so the kernel would only need to fill some descriptors, as explained in Subsection 2.4. However, note that, when using an XDP program, XDP sockets are still the best choice to forward packets to user space, as corroborated in Table 6.

In technical analysis, it can be stated that Sapo-boi is more robust and reliable than PF IDS, as more packets deemed suspicious by the Suspicion Module are guaranteed to be evaluated by the Evaluation Module. This also implies that the amount of malicious behavior alerted by PF IDS may be lower than that reported by Sapo-boi, due to the potential loss of events during module communication.

7 Limitations and Future Work

XDP/BPF programs must tackle several restrictions imposed by the BPF verifier before they can be loaded into the kernel. The main restriction regards the maximum number of instructions a BPF program may contain: a million bytecode instructions. Furthermore, to minimize XDP program size and prevent extensive BPF-to-BPF tail calls, Sapo-boi focuses on detecting attacks solely through pattern-matching operations.

In this way, the experiments were conducted only for signature-based NIDS, allowing equality among the solutions, and assuring that only the subset of rules supported by Sapo-boi were used in the tests. This means Snort and Suricata did not have to perform complex functionality other than pattern matching. But, it's important to keep in mind that, despite Sapo-boi and PF IDS outperforming those user-space solutions in the context of the experiments, they can only execute a subset of the consolidated solutions' functionality.

Another limitation regards the experiments. Well-known datasets that faithfully represent real-life traffic, such as CIC-IDS, could have been used for testing. However, as this work focuses on evaluating the computational performance of the solutions, it was not necessary to have traffic in these terms. Therefore, we gave preference to well-behaved traffic, in which we could control the rate of malicious traffic and packet size and header information, in order to learn how those metrics influence the solutions' behavior. It is also important to highlight that, as shown in Section 6, the performance of a user space signature-based NIDS mainly depends on the number of rules Lin and Lee [2013] when evaluated for the same set of traffic.

By knowing its operation, a malicious actor could perform a DOS attack in the Suspicion Module, if he/she knows beforehand which set of rules is loaded. Then, they can forge packets to contain patterns that almost match the installed ones, forcing the module to change the Aho-Corasick automaton state very often, causing processing time waste. For future work, Sapo-boi can be extended to detect this type of behavior, closing the connection as soon as the attack begins.

Encrypted traffic is a known issue in NIDS research. Sapo-boi, like other NIDS (not HIDS), is not able to perform DPI appropriately in this case, because the traffic has not yet been decrypted. However, Sapo-boi can be turned into a HIDS by installing it in the network nodes, instead of operating in the switch's mirrored traffic. For future work, Sapo-boi can be extended by using BPF instrumentation capabilities to fetch the packets after decryption, as well as operate with other HIDS capabilities, such as syscall instrumentation and file changes verification.

8 Conclusion

As the bandwidth for receiving traffic increases, it is of great interest that a NIDS maintains reasonable rates of packets analyzed, CPU usage, and alerts generated. By using two modules in separate parts of the operating system, the IDS solution proposed for this work, Sapo-boi, is capable of experiencing less packet loss and CPU usage than Snort and Suricata. It also does not lose events in communication between modules, unlike another evaluated kernel-based approach.

The aforementioned modules are: the Suspicion Module, an XDP program capable of determining if an incoming packet is suspicious using the Aho-Corasick algorithm at the first layer of the kernel network stack, and sending it to a user-space process for final veredict via XDP sockets, discarding other packets deemed safe (not suspicious); and the Evaluation Module, a user-space process capable of finding the rule that should be analyzed against the suspected packet received from the Suspicion Module in constant time, generating alerts if suspicion is confirmed.

This work demonstrates that, despite facing strong kernel restrictions, XDP programs can be used for complex tasks such as pattern matching within the kernel. The study also highlights the feasibility of packet redirection with metadata at high rates of reception using XDP sockets, a topic that is uncommon in the literature.

The efficiency and effectiveness of NIDS are of great impact in a society increasingly connected and generating massive amounts of network traffic. In this work, Sapo-boi, a BPF signature-based NIDS focused on minimizing packet loss and alerts, was proposed. It outperformed results obtained by market solutions like Snort and Suricata, and showed competitive advantages in specific aspects compared to a state-of-the-art BPF NIDS proposal.

Declarations

Authors' Contributions

RKM studied the state-of-the-art, designed and implemented the proposed solution, as well as performed the experiments. UP provided insights on the solution's design, cases, and scenarios, and discussed the experiments and results. JPC participated in the definition of the solution's architecture and implementation and performed experiments to test it. JRA performed tests and participated in the results evaluation. VFG discussed and contributed to the design of the solution and defined the testing methodology. AG contributed to the investigation of related work, solution's design, and evaluation of results. RKM is the main contributor and writer of this manuscript. All authors wrote, reviewed, and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Acknowledgements

The authors would like to thank CAPES, as this work is supported by the Coordination for the Improvement of Higher Education Personnel (CAPES) - Program of Academic Excellence (PROEX). The authors would also like to thank Bluepex CyberSecurity for the enduring research partnership and innovation fostering.

Funding

This study was financed by the Coordination for the Improvement of Higher Education Personnel (CAPES) - Finance Code 001 and

by Bluepex Cybersecurity through MD/MCTI/FINEP/FNDCT 2022 funding program.

Availability of data and materials

Sapo-boi source code and information to conduct experiments are available at: https://github.com/rkmach/SAPO-BOI.

References

Abhishta, A., van Heeswijk, W., Junger, M., Nieuwenhuis, L. J., and Joosten, R. (2020). Why would we get attacked? an analysis of attacker's aims behind ddos attacks. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, 11(2):3–22. DOI: 10.22667/jowua.2020.06.30.003.

Abranches, M., Michel, O., Keller, E., and Schmid, S. (2021). Efficient network monitoring applications in the kernel with ebpf and xdp. In *IEEE Conference on Network Function Virtualization and Software Defined Networks*, pages 28–34. DOI: 10.1109/nfv-sdn53031.2021.9665095.

Ahmed, Z., Alizai, M. H., and Syed, A. A. (2018). Inkev: In-kernel distributed network virtualization for dcn. *ACM SIGCOMM Computer Communication Review*, 46(3):1–6. DOI: 10.1145/3243157.3243161.

Aho, A. V. and Corasick, M. J. (1975). Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340. DOI: 10.1145/360825.360855.

Albin, E. and Rowe, N. C. (2012). A realistic experimental comparison of the suricata and snort intrusion-detection systems. In *International Conference on Advanced Information Networking and Applications Workshops*, pages 122–127. IEEE. DOI: 10.1109/waina.2012.29.

Bace, R. and Mell, P. (2001). Intrusion detection systems. DOI: 10.6028/nist.sp.800-31.

Baidya, S., Chen, Y., and Levorato, M. (2018). ebpf-based content and computation-aware communication for real-time edge computing. In *IEEE Conference on Computer Communications Workshops (INFOCOM)*, pages 865–870. DOI: 10.1109/infcomw.2018.8407006.

Biscosi, M., Cardigliano, A., *et al*. (2024). Pf_ring - high-speed packet capture, filtering, and analysis. Available at:`https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy` Accessed in 05/25/2024.

Conole, A., Richardson, B., *et al*. (2024). Data plane development kit. Available at:`https://www.dpdk.org` Accessed in 05/25/2024.

Du, Y., Chang, K., Shi, J., Zhou, Y., and Liu, M. (2023). A survey on mechanisms for fast network packet processing. In *International Conference on Computing, Networks and Internet of Things*, pages 57–66. DOI: 10.1145/3603781.3603792.

Erlacher, F. and Dressler, F. (2018). Fixids: A high-speed signature-based flow intrusion detection system. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–8. IEEE. DOI: 10.1109/noms.2018.8406247.

Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., and Miller, D. (2018). The express data path: Fast programmable packet processing in the operating system kernel. In *International Conference on Emerging Networking Experiments and Technologies*, pages 54–66. DOI: 10.1145/3281411.3281443.

Hu, Q., Yu, S.-Y., and Asghar, M. R. (2020). Analysing performance issues of open-source intrusion detection systems in high-speed networks. *Journal of Information Security and Applications*, 51:102426. DOI: 10.1016/j.jisa.2019.102426.

Karlsson, M. and Töpel, B. (2018). The path to dpdk speeds for af xdp. In *Linux Plumbers Conference*, volume 37, page 38. Available at:`https://pdfs.semanticscholar.org/52bb/6eec17be15778c09cb9e7b25cd3095b92d4b.pdf`.

Kernel, L. (2024). Af xdp. Available at:`https://docs.kernel.org/networking/af_xdp.html` Accessed in 05/26/2024.

Kostopoulos, S. (2024). *Machine learning-based near real time intrusion detection and prevention system using eBPF*. Bachelor's thesis, Hellenic Mediterranean University. Available at:`https://apothesis.lib.hmu.gr/handle/20.500.12688/10931`.

Liao, H.-J., Lin, C.-H. R., Lin, Y.-C., and Tung, K.-Y. (2013). Intrusion detection systems: A comprehensive review. *Journal of Network and Comp. Applications*, 36(1):16–24. DOI: 10.1016/j.jnca.2012.09.004.

Lin, P.-C. and Lee, J.-H. (2013). Re-examining the performance bottleneck in a nids with detailed profiling. *Journal of Network and Computer Applications*, 36(2):768–780. DOI: 10.1016/j.jnca.2012.12.009.

Lin, P.-C., Lin, Y.-D., Lai, Y.-C., and Lee, T.-H. (2008). Using string matching for deep packet inspection. *Computer*, 41(4):23–28. DOI: 10.1109/mc.2008.138.

Murphy, B. R. (2019). *Comparing the performance of intrusion detection systems: Snort and Suricata*. PhD thesis, Colorado Technical University. Book.

Park, W. and Ahn, S. (2017). Performance comparison and detection analysis in snort and suricata environment. *Wireless Personal Communications*, 94:241–252. DOI: 10.1007/s11277-016-3209-9.

Roesch, M., Henderson, A., *et al*. (2024a). Snort - open source intrusion prevention system. Available at:`https://www.snort.org` Accessed in 05/16/2024.

Roesch, M., Henderson, A., *et al*. (2024b). tcpdump. Available at:`https://www.tcpdump.org/` Accessed in 05/26/2024.

Scholz, D., Raumer, D., Emmerich, P., Kurtz, A., Lesiak, K., and Carle, G. (2018). Performance implications of packet filtering with linux ebpf. In *International Teletraffic Congress*, pages 209–217. DOI: 10.1109/itc30.2018.00039.

Shuai, L. and Li, S. (2021). Performance optimization of snort based on dpdk and hyperscan. *Procedia Computer Science*, 183:837–843. DOI: 10.1016/j.procs.2021.03.007.

Sundberg, S., Brunstrom, A., Ferlin-Reiter, S., Høiland-Jørgensen, T., and Brouer, J. D. (2023). Efficient continuous latency monitoring with ebpf. In *International Conference on Passive and Active Network Measurement*,

pages 191–208. DOI: 10.1007/978-3-031-28486-1$_9$.

Vieira, M. A., Castanho, M. S., Pacífico, R. D., Santos, E. R., Júnior, E. P. C., and Vieira, L. F. (2020). Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys*, 53(1):1–36. DOI: 10.1145/3371038.

Viljoen, N. and Kicinski, J. (2018). Using ebpf as an abstraction for switching. Available at:`http://vger.kernel.org/lpc_net2018_talks/eBPF_For_Switches.pdf`.

Waleed, A., Jamali, A. F., and Masood, A. (2022). Which open-source ids? snort, suricata or zeek. *Computer Networks*, 213:109116. DOI: 10.1016/j.comnet.2022.109116.

Wang, S.-Y. and Chang, J.-C. (2022). Design and implementation of an intrusion detection system by using extended bpf in the linux kernel. *Journal of Network and Computer Applications*, 198:103283. DOI: 10.1016/j.jnca.2021.103283.

White, J. S., Fitzsimmons, T., and Matthews, J. N. (2013). Quantitative analysis of intrusion detection systems: Snort and suricata. In *Cyber sensing*, volume 8757. DOI: 10.1117/12.2015616.

Woo, S. and Park, K. (2012). Scalable tcp session monitoring with symmetric receive-side scaling. *KAIST, Daejeon, Korea, Tech. Rep*, 144. Available at:`https://www.semanticscholar.org/paper/Scalable-TCP-Session-Monitoring-with-Symmetric-Woo-Park/38b647c56a74d634d7c23fe0b99ea1eb6347b09e`.

Xhonneux, M., Duchene, F., and Bonaventure, O. (2018). Leveraging ebpf for programmable network functions with ipv6 segment routing. In *International Conference on emerging Networking EXperiments and Technologies*, pages 67–72. DOI: 10.1145/3281411.3281426.