1 2 9 0

UNIVERSIDADE Đ
COIMBRA

João Lopes Teixeira Monteiro

# eBPF-IDS: Dynamic networking and security programming for IDS detection

Dissertation in the context of the Masters in Informatics Security, advised by Professor Bruno Sousa presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2024

DEPARTAMENTO DE
ENGENHARIA INFORMÁTICA
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð
COIMBRA

João Lopes Teixeira Monteiro

# eBPF-IDS: Dynamic networking and security programming for IDS detection

Dissertation in the context of the Masters in Informatics Security, advised by Professor Bruno Sousa and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2024

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð COIMBRA

João Lopes Teixeira Monteiro

# eBPF-IDS: Dynamic networking and security programming for IDS detection

Dissertação no âmbito do Mestrado em Segurança Informática, orientada pelo Professor Bruno Sousa apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Julho 2024

The work presented in this thesis was carried out within the Network, Communication and Security (NCS) group of the Centre for Informatics and Systems of the University of Coimbra (CISUC) in the context of the following projects:

- This work was supported in part by project MH-SDVanet: Multihomed Software Defined Vehicular Networks (reference PTDC/EEI-COM/5284/2020);
- This work has been supported by Project "Agenda Mobilizadora Sines Nexus". ref. No. 7113), supported by the Recovery and Resilience Plan (PRR) and by the European Funds Next Generation EU, following Notice No. 02/C05-i01/2022, Component 5 - Capitalisation and Business Innovation - Mobilising Agendas for Business Innovation

# Acknowledgements

I would like to extend appreciation for the guidance and support offered by my supervisor, Prof. Doutor Bruno Sousa. The availability and insightful guidance provided were essential for the development of this work.

Additionally, I would like to thank my family and friends, for the support that they have provided during these academic years.

## Abstract

An Intrusion detection system (IDS) is an essential component in information security, offering monitoring capabilities across various scenarios and alerting for possible breaches in security. The visibility and processing capabilities of these systems become an essential aspect of their effectiveness.

The extended Berkeley Packet Filter (eBPF) is a technology that allows dynamic kernel modifications, enabling traffic analysis very early in the process through the eXpress Data Path (XDP). The level at which eBPF operates and its performance makes it an excellent addition to an IDS. Furthermore, the deployment of eBPF can extend to the hardware domain.

In the realm of eBPF-IDS solutions, when it comes to the mechanism by which intrusions are detected, the use Machine Learning (ML) emerges as a new area of research. Moreover, the incorporation of ML with eBPF in the hardware domain is relatively unexplored, suggesting, that further investigation should take place.

This thesis explores the integration of eBPF and IDS, developing a proof of concept that uses both technologies. The developed solution incorporates a Random Forest (RF) model with eBPF to differentiate between normal traffic and various types of port scans. Additionally, the proposed method can partially operate in the hardware domain using XDP Offload, enhancing its performance and efficiency.

The evaluation of the final solution presented promising results. The system possessed high packet processing capabilities and effective detection in the domain of port scans. This suggests that real-world scenarios could benefit from implementing this type of solution.

## Keywords

extended Berkeley Packet Filter, Intrusion Detection System, Machine Learning, Port Scanning, eXpress Data Path

## Resumo

Um Sistema de Deteção de Intrusões (IDS) é um componente essencial na segurança da informação, oferecendo capacidades de monitorizar vários cenários e alertar para possíveis intrusões. A visibilidade e as capacidades de processamento destes sistemas tornam-se um aspeto essencial para a sua eficácia.

O Filtro de pacotes Berkeley estendido (eBPF) é uma tecnologia que possibilita modificações dinâmicas no kernel, permitindo analisar o tráfego através do Rota de Dados Expressa (XDP). O nível em que o eBPF opera e seu desempenho fazem dele uma excelente adição a um IDS. Além disso, a implementação do eBPF pode estender-se ao domínio do hardware.

No âmbito das soluções eBPF-IDS, no que diz respeito ao mecanismo pelo qual as intrusões são detetadas, o uso de Aprendizagem de Máquina (ML) surge como uma nova área de estudo. Além disso, a incorporação de ML com eBPF no domínio do hardware é relativamente inexplorada, sugerindo que mais investigação deve ser realizada.

Esta tese explora a integração de eBPF e IDS, desenvolvendo uma prova de conceito que utiliza ambas as tecnologias. A solução desenvolvida incorpora um modelo de Floresta Aleatória (RF) com eBPF para diferenciar entre tráfego normal e vários tipos de varreduras de portas. O método proposto pode operar parcialmente no domínio do hardware com XDP Offload, melhorando o seu desempenho e eficiência.

A avaliação da solução final apresentou resultados promissores. O sistema possuí altas capacidades de processamento de pacotes e deteção eficaz no domínio das varreduras de portas. Isto sugere que cenários reais poderiam beneficiar da implementação deste tipo de solução.

## Palavras-Chave

Filtro de pacotes Berkeley estendido , Sistema de Deteção de Intrusões, Aprendizagem de Máquina, Varredura de Portas, Rota de Dados Expressa

# Contents

# Acronyms

**AC** Aho–Corasick.

**AI** Artificial Intelligence.

**ANN** Artificial Neural Network.

**DMA** Direct Memory Access.

**DT** Decision Trees.

**eBPF** extended Berkeley Packet Filter.

**HIDS** Host-based Intrusion Detection System.

**ICMP** Internet Control Message Protocol.

**IDS** Intrusion detection system.

**IGMP** Internet Group Management Protocol.

**JIT** Just-in-Time.

**KNN** K-Nearest Neighbors.

**LRU** Least Recently Used.

**ML** Machine Learning.

**NFP** Netronome Flow Processor.

**NIC** Network Interface Card.

**NIDS** Network-based Intrusion Detection System.

**RF** Random Forest.

**SCADA** Supervisory Control and Data Acquisition.

**SKB** Socket Kernel Buffers.

**SVM** Support Vector Machine.

**TAP** Test Access Port.

**TCP** Transmission Control Protocol.

**UDP** User Datagram Protocol.

**XDP** eXpress Data Path.

# List of Figures

# List of Tables

# Listings

# List of Algorithms

# Chapter 1

# Introduction

This document serves as a comprehensive presentation of the research and accomplishments achieved in the context of the Dissertation component of the Master's program in Information Security, conducted at the Faculty of Sciences and Technology, University of Coimbra, for the academic year 2023/2024.

The undertaken Dissertation topic relates to Intrusion detection system (IDS) and the extended Berkeley Packet Filter (eBPF) technology. IDSs are a critical component in the domain of Information Security, therefore their place is well established within this realm. eBPF is a tool that allows to change the kernel behaviour dynamically. At this level, one can possess greater visibility over various security affairs, with which, they can make more informed decisions. The combination of these two tools can be very beneficial.

This chapter is divided into four sections, in Section 1.1 we will centre our attention on elucidating the motivation behind the undertaken work. Section 1.2 will delve into the Dissertation objectives as well as the strategies followed to mitigate the risks that can be encountered in trying to achieve them. Section 1.3 presents the contributions made. Finally, Section 1.4 will entail a succinct global structure of the document.

## 1.1 Motivation

Organizations worldwide must allocate resources to implement enhanced security measures and keep their training programs up to date, given the continuous sophistication of threats and the increasing importance of information security. According to [Morgan, 2020], the average cost of a cyber breach was predicted to increase by 15 percent per year, wherein 2015 started at 3 trillion USD and will reach 10.5 trillion USD by 2025. For this reason, the search for better solutions will go on.

At its core, the security of any environment relies heavily on safeguarding the integrity of data transactions taking place within networks. As a consequence, the ongoing demand for faster, more robust methods to manage, monitor, and trace network events has led to the continuous evolution of technology. In this context, an IDS emerges as a pivotal tool in the realm of cyber-security, enabling the oversight of these network activities.

The speed at which an IDS can process packets becomes a critical factor in network security. Software-based IDSs present an attractive solution due to their scalability and cost-effectiveness. However, when it comes to operational environments where performance is crucial, professionals may face the challenging decision of balancing security efficiency against system performance. In contrast, hardware-based IDSs offer a way to maintain system efficiency as they operate on a separate plane, further providing superior packet processing throughput.

In the dynamic landscape of cybersecurity, the emergence of new threats is a constant, therefore countermeasures are also ever-evolving. A technology that has the potential to address these challenges is eBPF. This innovative technology facilitates real-time kernel modifications and is often associated with offloaded solutions, making it a compelling choice for enhancing the capabilities of IDS systems.

Every attack is mostly accompanied by a precursor. This usually entails some form of reconnaissance, which is a method by which each attacker attempts to gather information about a system. Networking probing activities are usually methods associated with this step. When developing an IDS, there should be an attack vector for which the solution is designed. With this in mind, combating probing activities like port scanning is a great domain to fight against.

## 1.2 Objectives and Associated Risks

Our first objective surrounds the understanding of eBPF with an in-depth exploration of the intricacies surrounding it, examining whether it is a viable solution and delving into the most effective strategies for its integration.

One risk associated with this item is the complexity of the topic at hand, resulting in a poor understanding of certain issues and subsequently poor decisions. eBPF is a complex subject, and for that reason, to guarantee that the fundamentals can be well understood, enough time needs to be dedicated to this topic. The best solution is to resort to the documentation on this topic. The documentation covers not only the examination of features but also the reference guides. In addition, books that may provide even greater insight, namely those usually indicated for further reading, can prove very valuable to reach this target objective.

Considering the mentioned strategy, the documentation [eBPF Documentation] was deemed essential for our initial understanding of the issue. A book reference for further reading in this document was [Rice, 2023]. This book allowed for a deep dive into the topic proving itself extremely important for the understanding of the domain. Finally, the reference guide [BPF & XDP Reference Guide] was also established as a fundamental for this goal.

In this domain, one established objective was that the final proposed practical solution encompassed offloading. Taking this into account, to fulfil this objective, we identified, from the previous documents, the domain that entailed this feature. Analysis of technical papers [Kicinski and Viljoen, a], [Kicinski and Viljoen, b] and viewing of the webinars like [Kicinski, 2018].

In the following phase, we will direct our attention towards IDS. Our objective is to identify how these solutions can be deployed and how they can achieve detection among other intricacies.

IDS is an item that has already been addressed extensively throughout the Master Course in Information Security, so background knowledge is firmly established, providing a solid foundation for the upcoming research and analysis. Considering this, the research conducted is more to provide an auxiliary exploration, delving deeper into specific aspects, and contributing nuanced insights to the existing body of knowledge. In addition to some selected papers, the book [Mell, 2003] was the one most heavily used for this auxiliary research.

The issues associated with this topic relate to how broad it is, as a result, not allowing for an in-depth understanding of key features. Therefore, a well-defined, manageable scope is necessary. With this in mind, the research will encompass ML models used in IDS and the attack vector.

ML models become a subsequent objective to understanding IDSs. A risk that may arise is the availability of high-quality data that need to be used for the training of the model. If from the identified attack vector no data can support the intended development, either a change in the attack perspective or, ultimately resorting to the development and creation of data, needs to take place. To ensure that this risk is mitigated, we must define the characteristics of the desired data, and having that in mind, research the literature thoroughly.

Recognizing the importance of identifying an attack vector upon which we can develop our solution, port scanning stands out as a notable concern. Consequently, another key objective of this thesis is to gain a comprehensive understanding of port scanning and explore techniques for its detection.

A risk associated with this objective is concerning if it is relevant and realistic. The selection must fall within the overall topic and be an issue that is important within the IDS solutions. There is also the issue of examining whether there exists enough literature and resources available not only for insight into the topic but also for the development of the mechanisms that will combat and detect this in the practical scenario. To try and address this, the best solution is to first examine the issues IDSs try to combat. From there analyse literature concerning the identified attack for insights on the issue; also how researchers have formulated solutions to fight against them.

Having established the background knowledge needed, we move on to a literature review. Here we attempt to evaluate the existing body of knowledge concerning eBPF-IDS solutions and port scanning detection with ML, with the final goal of intersecting both.

There may be limited literature on the specific intersection of eBPF and IDS, and port scanning and ML. For this reason, to mitigate the risk of developing a poorly informed solution, more elaborate research is required. Conducting a systematic review of the issue may provide the best insights into the topics.

An important objective is the development of a practical proof of concept. This solution must encompass the decisions taken during the previous stages.

There could be issues one may encounter when implementing the solution. It is important to give enough time not only for development but also for subsequent research necessary to solve the problems one may encounter. An established feature of this solution is that it employs an ML algorithm. However, there is the risk that this feature may be out of the reach of the eBPF capabilities. To address this, the review conducted on the state of the art must analyse with great detail the algorithms employed to try and determine if this is possible and if not, what circumvention one can take to allow an ML algorithm to coexist with eBPF.

Once the proof of concept is developed, our final objective is to conduct an assessment of the eBPF-IDS solution, evaluating its performance, efficiency, and security capabilities. This step will allow us to determine the practicality and effectiveness of our proposed intrusion detection system.

There is the risk that our evaluation metrics are improper, resulting in conclusions that do not align with the truth. With that, we must consider proper metrics of evaluation, within this context it is important to examine other proposed solutions metrics so that comparisons can be made to allow placing our work concerning the others.

## 1.3 Contributions

During the development of this thesis, there were some contributions made:

- The early stages of development, encompassing the integration of ML in the kernel via eBPF and related results, were presented in the thirty-fifth instalment of the Mobile Communications Thematic Network (RTCM) seminar [RTCM] on February 9th, 2024;

- The final solution was also presented in the CISUC's NCS workshop [NCS Workshop] on May 15th, 2024. Here, the proposed offload solution and results obtained were discussed, alongside future work that could evolve from the current solution.

- The developed work was also made public. This can be used by others to conduct further research. This is present in the following GitHub repository [eBPF-IDS].

- During development, an error was found in one of the tools that did not allow hardware offloading. The bug correction was proposed in the following merge [BCC pull request 5051], which as been accepted.

- From the developed work a scientific paper was produced and submitted to the IEEE NFV-SDN 2024 conference [NFVSDN]. This paper is still waiting for acceptance.

## 1.4 Thesis Outline

The primary objective of the initial chapter of this thesis is to demonstrate the significance of the research that was undertaken. It will delineate the context and importance of the work.

Moving on to the second chapter, we extensively examine the tools that will be employed to serve as a reference for upcoming chapters. These tools pertain to both eBPF, IDS and port scanning. This comprehensive review plays a vital role in providing valuable insights and knowledge which, in turn, will help in the development of a practical proof of concept.

Entering the third chapter, a literature review concerning the topic was executed. This examination of the current state of eBPF IDS solutions and port scanning can help guide practical development. This examination also helps to put work developed in the context of the existing literature.

The fourth chapter is dedicated to the practical development phase. In here all details surrounding the examination of the Dataset, training the ML model and implementing it in eBPF are discussed.

The fifth chapter will entail the evaluation of the solution. It examined its capabilities and also to put it into perspective to other works. Determining its suitability for real-world applications.

The final chapter, sixth, corresponds to the conclusion of this document. Giving context to what future work can be developed.

Appendix A documents the research conducted with private firmware. This process was carried out to assess its suitability for the final solution.

Appendix B provides a guide on the installation of firmware. This was included to allow this document to be self-contained.

# Chapter 2

# Reference Technologies

This chapter will delve into existing technologies across a range of pertinent topics. By delving into these sources, we aim to enhance our comprehension of the prevailing tools and establish a robust foundation upon which we can construct a compelling proof of concept. This comprehensive examination of the technologies will aid in the development of a concrete and well-informed solution.

Section 2.1 will handle extended Berkeley Packet Filter (eBPF) and contextualize the reader on its importance, how it can be used as a solution and other details that may concern the development of the proof of concept. Section 2.2 will delve into Intrusion detection system (IDS) alongside the details surrounding their characteristics. Finally, Section 2.3 will comprehensively cover the domain of port scanning, as it aligns with the identified attack vector that our practical solution aims to combat. Section 2.4 will provide an overview of the chapter.

## 2.1 extended Berkeley Packet Filter

The Linux kernel serves as the intermediary software layer, bridging the gap between software applications and the underlying hardware on which they operate. Applications do not possess direct access to hardware. The domain on which they operate, user space, is restricted, for that reason an application makes its requests via the system call interface which are then handled by the kernel [Rice, 2023], as depicted in Figure 2.1.

The eBPF is a technology developed to allow one to change the kernel's behaviour. An interesting feature of eBPF is that it allows programs to be loaded dynamically. This means that at any time the process is running, or not, an eBPF program can be attached or removed. eBPF programs operate on an event-driven basis, as shown in Figure 2.1, executing when a hook (system call, network event or other) is triggered. In the case that the hook is not pre-defined it is possible to create a kernel or user probe (*kprobe* and *uprobe* respectively) to attach the eBPF program nearly anywhere in the kernel or application [eBPF Documentation; Rice, 2023].

As mentioned previously, adopting IDSs comes as a necessity to monitor and provide visibility over applications, networks and others. This tool should also not constrain system performance and be reliable on the measured data. Given the properties of eBPF, it can provide this need and it proves to be a good solution for solving this issue [eBPF Documentation; Rice, 2023].

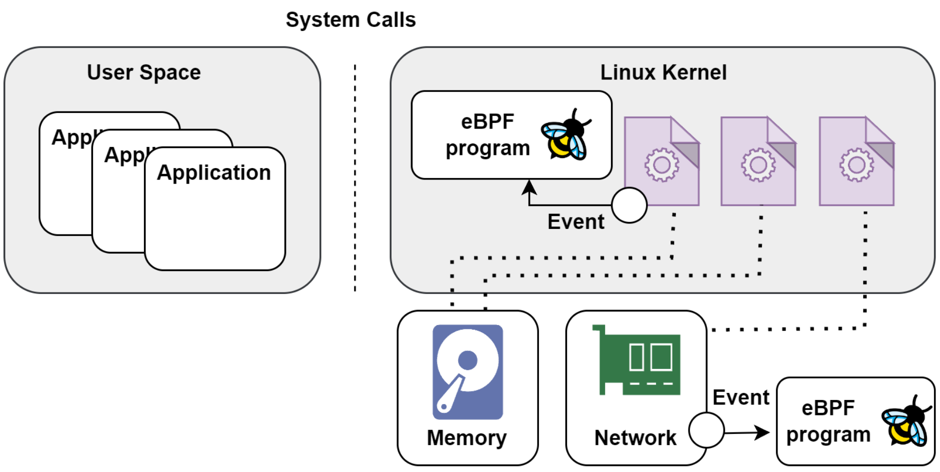


Figure 2.1: Userspace concerning the kernel and insertion of eBPF programs

### 2.1.1 eBPF Verifier and JIT Compiler

Modifying the Linux kernel is a highly intricate task due to its complexity, requiring a profound understanding and a set of skills that would otherwise render it incredibly challenging. The kernel does contain means of loading and unloading modules on demand, known as kernel modules. But again, these still require highly complex kernel programming. Furthermore, these modules must be thoroughly scrutinized to ensure safety. This means that they will not crash, cause if they do, they will take down the machine alongside it, and do not pose any security vulnerabilities. eBPF can be used to assess these challenges [Rice, 2023].

In comparison to kernel modules, eBPF offers an approach to tackle the safety of programs to make sure that they don't crash, compromise data or lock the machine in a hard loop state. For this reason, after having identified the desired hook to which the program will be attached, the program goes through two steps before being loaded into the kernel. Firstly, the verification, where various conditions like the previously mentioned must be met. The verification stage must be considered when choosing and developing the algorithms used in the practical stage, as it will impose limitations on its selection. Secondly, to make sure that the program executes with the same efficiency as kernel code that has been natively compiled or code loaded into the kernel as a module, the Just-in-Time (JIT) compiler is used to achieve just that, this can be seen in Figure 2.2. So, besides natively changing the kernel, which requires a deep knowledge of the codebase, or loading kernel modules, which poses a risk of kernel corruption, there is eBPF, which makes kernel modifications safely [eBPF Documentation; Rice, 2023].

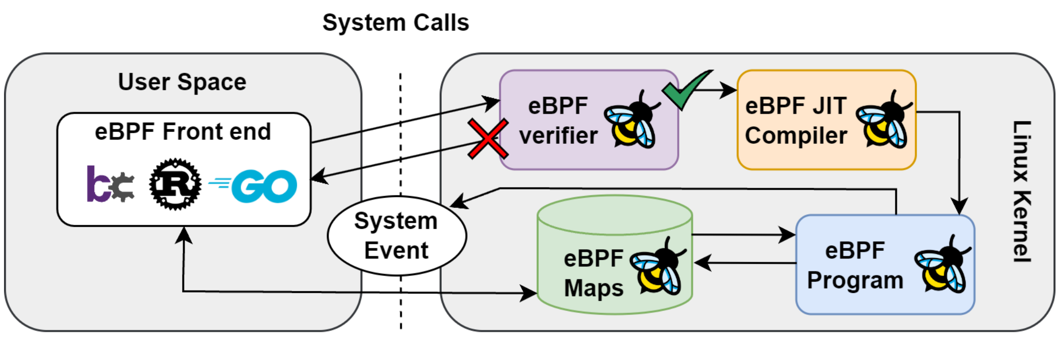


Figure 2.2: High-level visualization of the Verifier and JIT compiler

### 2.1.2 eBPF Maps

As seen in Figure 2.2 an element called "eBPF Maps" is depicted. These maps are not just ordinary structures, rather they play a pivotal role in the eBPF development by allowing data storage and retrieval. These maps are accessible both by eBPF programs running within the kernel and by user-space applications. This dual accessibility allows a bidirectional flow of data, allowing eBPF programs to seamlessly retrieve information from user space while also enabling the exchange of data between various eBPF programs. According to [Linux's uapi/linux/bpf.h, a] there are 30 different map types. The decision behind the use of each is reflected in the desired solution trying to be implemented. One example could be in a situation where the solution being developed requires a queue data structure, for this specific necessity the `BPF_MAP_TYPE_QUEUE` could be employed.

Each Map may possess variants, like Least Recently Used (LRU) and PER_CPU. When elements within LRU maps undergo updates, it can potentially trigger eviction actions once the map's capacity is exceeded. The update algorithm employs a series of steps aimed at preserving the LRU property. PER_CPU maps indicate that each individual CPU has its dedicated copy of the underlying memory.

### 2.1.3 eBPF Programming

As illustrated in Figure 2.2, the front-end frameworks play a crucial role in the eBPF ecosystem. When delving into the realm of eBPF programming, one discovers an array of toolchains and resources designed to enhance the development process. These toolchains offer valuable support, ranging from code compilation to program debugging and performance analysis, making eBPF programming more accessible and efficient [eBPF Documentation; Rice, 2023]. Examples depicted in Figure 2.2 are: BCC, which makes use of Python and compiles eBPF programs into bytecode, loads them into the kernel and enables data collection and display for tracing and profiling tasks; Rust, which allows writing both user space and kernel code; Go which possesses a lot of frameworks around it such as Gobpf, Ebpf-go and Libbpfgo.

### 2.1.4 eBPF and XDP for Networking

In the context of the thesis, the kernel's behaviour to handle is related to network operations. There is a framework that enables the attachment of a program at the instant a network driver accepts a packet. This framework is named eXpress Data Path (XDP). XDP offers raw packet handling at the deepest software layer. This enhances the performance of processing packets. However, it is important to note a big constraint of XDP, which is that it can only examine the incoming traffic [Rice, 2023].

After a packet arrives, it initiates the execution of an XDP program. As mentioned earlier, eBPF programs operate on an event-driven basis. This program will analyze the incoming packet and, based on its analysis, return a code that specifies the appropriate action to be taken with the packet. The return codes are as follows [BPF & XDP Reference Guide; Rice, 2023]:

- `XDP_PASS` informs that the packet will continue to the network stack.
- `XDP_DROP` results in the immediate deletion of the packet.
- `XDP_TX` sends the packet through the same interface at which it arrived.
- `XDP_REDIRECT` sends the packet through a different interface.
- `XDP_ABORTED` also results in the drop of the packet, however, it implies the occurrence of an error and not a conscious decision of its deletion.

The application of XDP is extensive. Various solutions can be implemented using these return codes, from DDoS and firewall protection to forwarding and load balancing. However, one application that stood out was the mitigation of packet-of-death vulnerabilities. This is a class of kernel vulnerabilities where a received packet, which is maliciously crafted, results in unsafe processing. Exploitation of this issue can cause the kernel to crash. In the presence of this vulnerability, one has to resort to installing a new kernel with the appropriate remediation, resulting in machine downtime. With the eBPF properties, a solution that detects these packets and uses the return codes can be dynamically deployed [BPF & XDP Reference Guide; Rice, 2023].

A mentioned constraint of traditional IDSs lies in their inability to analyse encrypted packets, particularly those safeguarded by encryption protocols like TLS. However, with eBPF, it becomes feasible to attach custom programs at precise junctures in the network traffic flow, either right before encryption or immediately after decryption, providing visibility over the communications in user space. With this, an IDS could then have more visibility over the traffic being exchanged. Considering SSL/TLS, if one desires to observe the data, hooking an eBPF program with *uprobes* to functions like `SSL_write()` or `SSL_read()`, functions used by OpenSSL, would allow doing so. No need for keys or certificates would be required as these are already provided by the application [Rice, 2023]. This IDS needed to be deployed on the same host it is trying to monitor.

### 2.1.5 XDP Operation Modes

The operation modes used for the XDP programs will, consequently, determine their performance, as these modes of operation are inserted at different parts of the network handling infrastructure. These modes are native, generic, and offloaded [BPF & XDP Reference Guide; Karlsson and Brouer, 2019]:

- The default mode, and the one typically implied when discussing XDP, is native, also referred to as driver mode. In this mode, the eBPF program will run in the earliest path of the network driver [BPF & XDP Reference Guide]. A driver hook becomes accessible immediately after Direct Memory Access (DMA) packet transfer from the NIC to the NIC driver. It handles packets before allocating Socket Kernel Buffers (SKB), which are the data structures used by the kernel to represent and manage network packets [Karlsson and Brouer, 2019]. Not all drivers support this mode, however, according to [BPF & XDP Reference Guide], many solutions are already available.
- Generic or SKB is a mode used by developers who wish to test programs that make use of the kernel's XDP API. This mode is also used for drivers that do not support the native or offloaded modes. Generic mode will not operate at near performance levels compared to native and offloaded [BPF & XDP Reference Guide]. It provides a hook called from the function: `netif_receive_skb()`. This function takes a received packet in the form of an SKB, which means that the packet is only processed after the packet DMA transfer to SKB [Karlsson and Brouer, 2019].
- The offloaded mode is the one that can provide the best performance out of the three. The reason for this is that in this mode the host machine does not spend CPU cycles on handling the packet, instead, all processing is done on the Network Interface Card (NIC) itself. This means that the packet gets processed, dropped and so on, before reaching the kernel network stack. However, to deploy such a mode, specific hardware is required. Not all NICs support offloading, this task is typically employed by SmartNICs. According to [BPF & XDP Reference Guide], only one vendor supports XDP offloading drivers, the NFP driver owned by Netronome [1] [Netronome].

Figure 2.3 offers a concise high-level visualization that illustrates the operational modes concerning the hardware, kernel, and userspace. This depiction provides a quick reference for understanding the positioning of these modes within the system architecture.

Taking into account these inherent characteristics, the optimal choice to maximize the performance of an IDS is undoubtedly the offload mode. Therefore, in developing the proof of concept, special attention will be dedicated to addressing and accommodating these feature and hardware requirements. In doing so, we aim to ensure that the IDS operates at its peak efficiency.

[1]The validity of the list of drivers presented by [BPF & XDP Reference Guide] is supported by [Rice, 2023]

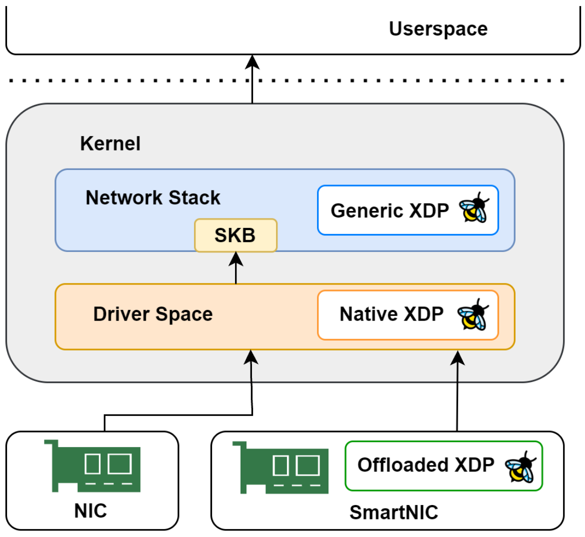


Figure 2.3: High-level visualization of each XDP mode, native, generic and offloaded

### 2.1.6 XDP Offloading Netronome NFP

The use of XDP for security applications, whether it involves comprehensive network monitoring or the implementation of effective security solutions, represents an excellent choice due to its speed and efficiency. As data transfer rates increase, the importance of CPU utilization becomes more pronounced. In shared CPU environments, this can lead to a compromise in effectiveness, where neither applications nor XDP solutions can operate at their peak efficiency. With an offloaded infrastructure in the kernel, this issue could be addressed.

Previous endeavours aimed at promoting broad networking offloads within the Linux kernel have encountered limitations. Historically, only initiatives with narrow scopes have achieved success, primarily due to a convergence of factors. One significant factor has been the constrained capabilities of NICs, which prioritize performance over flexibility and predominantly offer rudimentary, stateless, and task-specific offload functionalities. Furthermore, the scalability of CPU architectures like x86 and other general-purpose processors has accommodated the evolving demands of networking, effectively supporting comprehensive stateful networking tasks. Another contributing element has been the prevalence of vendor-specific solutions. To allow universal offloading capabilities, it is imperative to establish an infrastructure that caters to diverse hardware platforms [Kicinski and Viljoen, b]. This type of solution is provided by Netronome with the Netronome Flow Processors (NFPs) driver.

According to technical papers provided by Netronome, [Kicinski and Viljoen, b] and [Kicinski and Viljoen, a], there is an ongoing development of the model, it is unclear if the last proposal has been integrated or rejected. However, the overall structure of their solution can be observed.

## Hardware

The NFP architecture comprises a sequence of hardware modules arranged into islands, designed for specialized tasks, consisting of two instances of a networking block which processes all basic networking functions, like check checksums of packets and provides some parsing, two memory units, internal and external, in the middle, the flow processing cores which are the foundation of the chip where advanced packet processing is performed, the PCIe interfaces, the security island, as the name suggests, is used for security, where cryptographic operations like encryption and authentication can be done by using the hardware accelerators, the ARM subsystem for management, and finally, the Interlaken LA used to connect between various chips [Kicinski, 2018].

These elements are also depicted in Figure 2.4, which shows the overall structure of the modules and their elements, along with the identification of where eBPF Maps and Programs are stored and accompanied by the packet flow through the hardware.

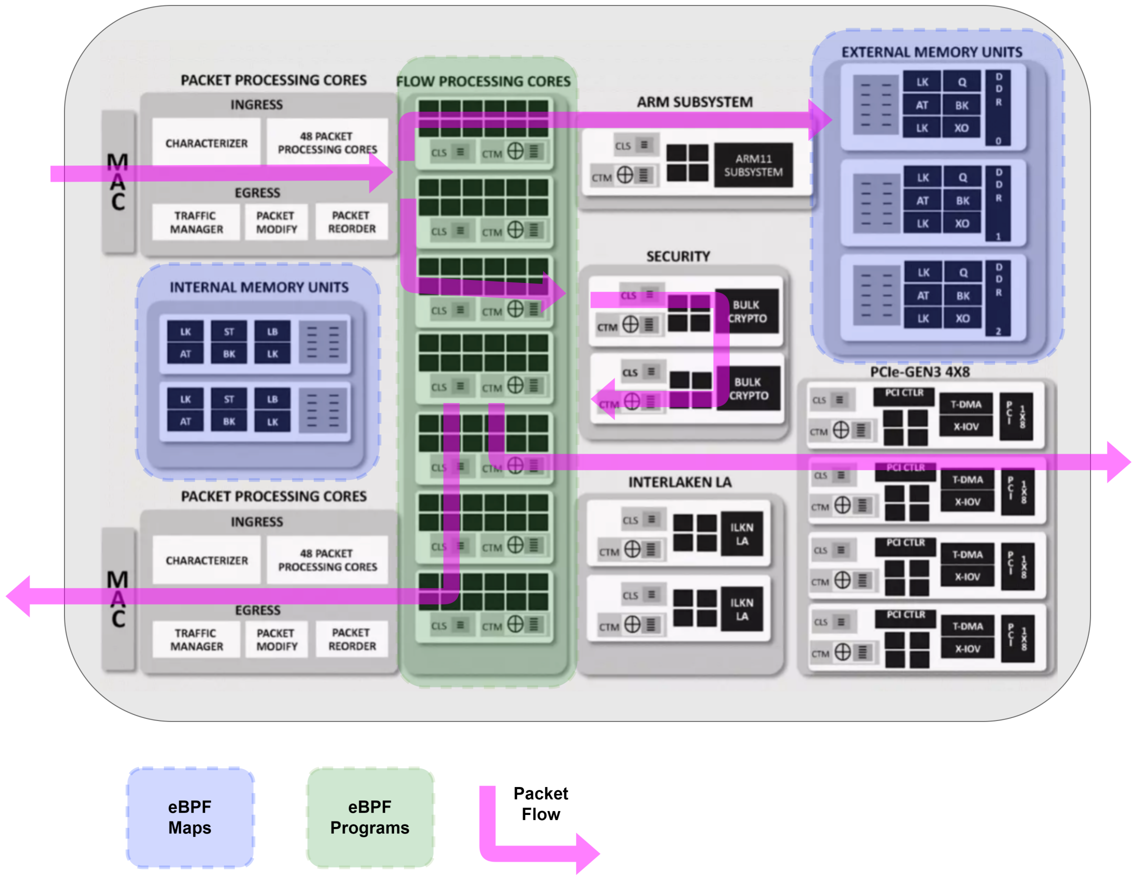


Figure 2.4: NFP SoC Architecture [2]

[2]This image was edited from [Kicinski, 2018]. As can be seen, Maps reside in the device's memory entirely, for that reason the Host does not have access to these offloaded maps and vice versa. Within the userspace perspective, the operations go through a system call path and that system call is redirected to the device, but as previously mentioned other kernel programs do not have access to these maps.

### Software

The offload model was introduced with the primary goal of improving transparency and simplification. The design of the offload mechanism was aimed at ensuring that existing programs could seamlessly operate with minimal adjustments compared to their operation on the host. This was achieved through the implementation outlined in Figure 2.5, which involved the incorporation of an NFP JIT component into the driver. As mentioned in section 2.1.1 to make eBPF programs operate with the same efficiency as kernel modules or native kernel modification it requires the JIT to convert the program bytecode to machine code, in this case, the `nfp_bpf_jit.c` is the solution used to make the conversion from bytecode to NFP machine code. This innovation empowers users to compile the same program whether it is executed on the NFP or on the host [Kicinski and Viljoen, a].

Certain operations, which align with the host architecture, may be accepted by the Verifier but not supported by the offload device. In response to the need for a more comprehensive verification per instruction, a callback into the kernel eBPF Verifier was integrated [Kicinski and Viljoen, a].

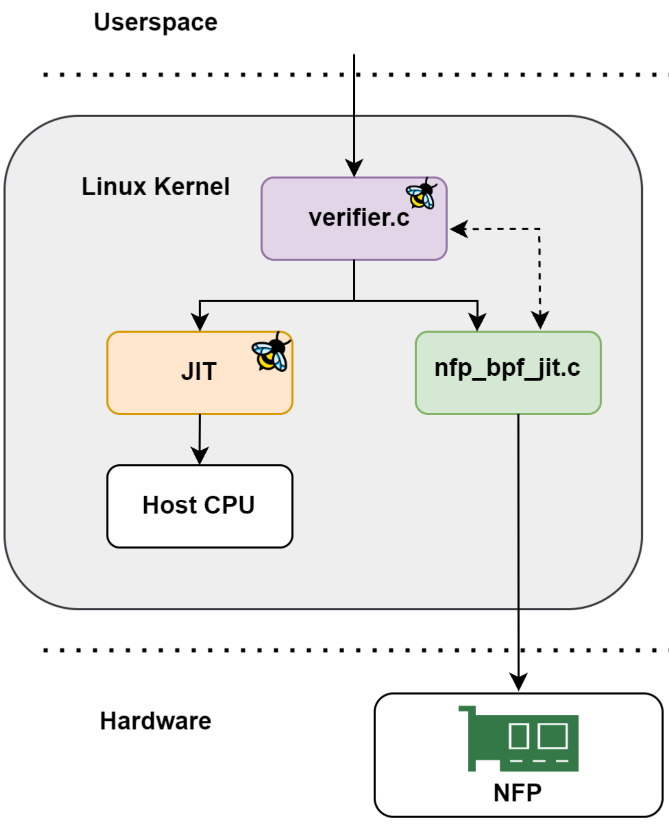


Figure 2.5: XDP offload high-level model

## 2.2 Intrusion Detection System

An intrusion detection system (IDS) holds a significant position within cybersecurity. Considering that the thesis is dedicated to crafting such a tool, a profound understanding of the inner workings of these systems is necessary. This comprehension is crucial to ensure that well-informed decisions guide the development of the proof of concept.

An IDS can be categorized as a hardware or software solution with the capability to discern unauthorized actions. An intrusion, characterized by an illicit entry into a system, has the potential to wreak havoc. Such actions can inflict substantial harm upon the confidentiality, integrity, and/or availability of information.

The main reason for the acquisition of an IDS revolves around the issues of detecting unpreventable attacks, preventing network probing, or documenting threats. In many legacy systems, updating may not be an option, either due to a lack of resources to replace the systems, compatibility issues resulting from deep integration with other applications, or even the vendor no longer providing support [Mell, 2003]. Considering the scenario of a Supervisory Control and Data Acquisition (SCADA) system reliant on the unencrypted Modbus protocol, a challenge arises when the manufacturers of the system's components cease to provide support for these outdated devices. Consequently, to maintain operational continuity, the system is forced to maintain this vulnerability. For that reason, an IDS could be a great solution, given that an attack cannot be prevented but can be detected, this would allow one to perform damage control and recovery. In a network without an IDS, an attacker can freely explore its weaknesses. If a vulnerability is present, the attacker will eventually find it and exploit it. However, if within the presence of an IDS, the probing activities performed by an attacker can be detected. These attempts can be blocked, diverted to a honeypot or used to alert security personnel responsible for the network security. An IDS allows documentation of threats that can then be used to analyse the frequency and characteristics of attacks. With this information, appropriate security solutions and measures can be taken [Mell, 2003].

The operational characteristics of an IDS are influenced by its deployment context and the specific threat detection solutions it employs. Furthermore, the nuanced aspects of its functionality contribute to its overall effectiveness. In the depiction represented in Figure 2.6, a fundamental taxonomy of IDS attributes is illustrated. This taxonomy serves as a conceptual framework to better understand how IDS systems function, taking into account the diverse factors that shape their operation [Mell, 2003].

[3]This depiction draws from references in various academic papers which include [Axelsson, 2000; Liao et al., 2013]

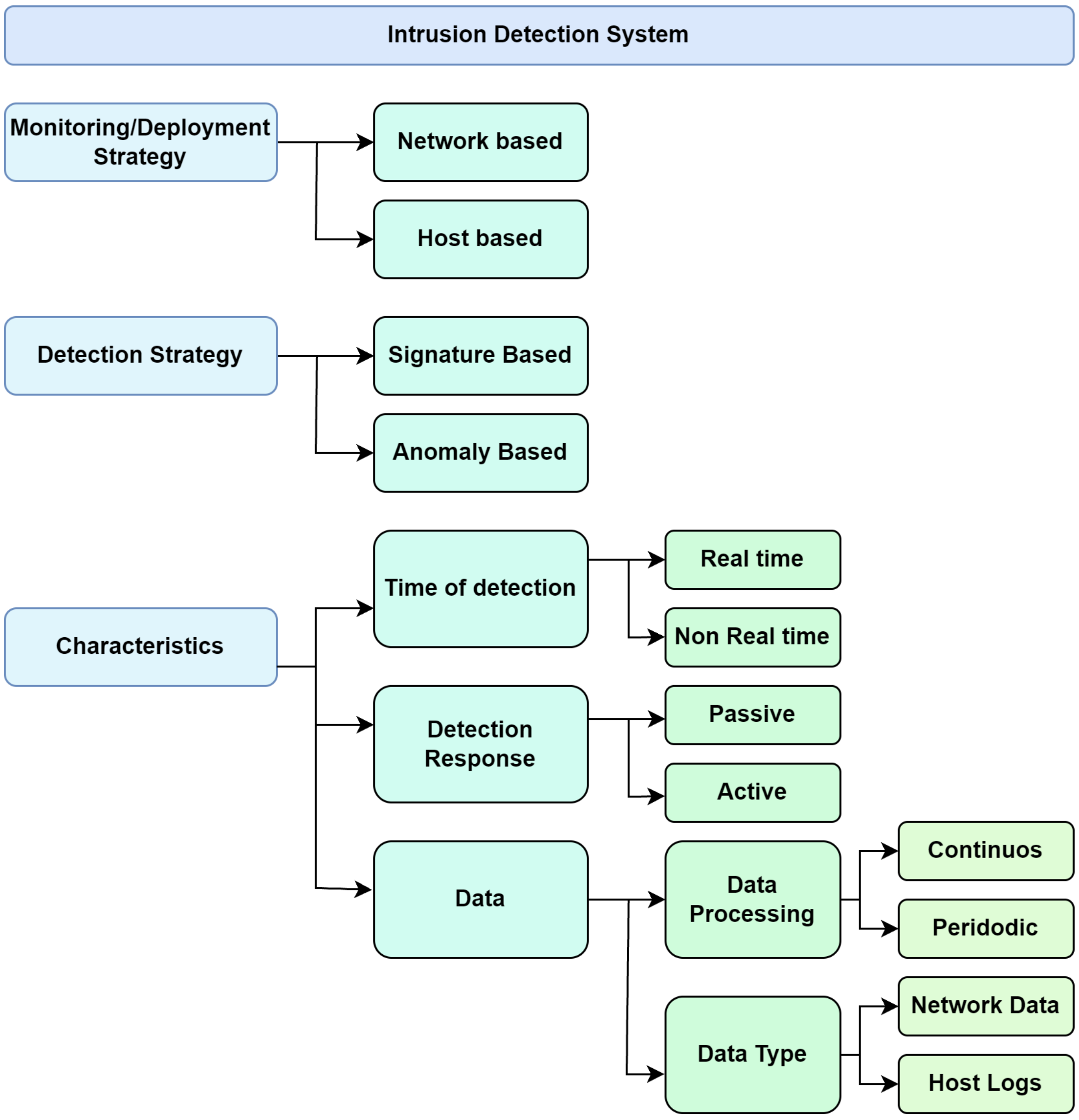


Figure 2.6: Taxonomy of an Intrusion Detection System[3]

### 2.2.1 IDS Monitoring Strategy

Categorizing an IDS hinges on the approach it employs for threat monitoring. Two fundamental categories for IDS are host-based and network-based solutions. Host-based Intrusion Detection System (HIDS) focusses on individual devices and their internal activities, while Network-based Intrusion Detection System (NIDS) monitors network traffic and patterns.

A deeper dive into these categories allows us to gain a clearer perspective on how each one deals with various security threats. As we carefully examine and compare the strengths and weaknesses of each type, we can more easily identify the most suitable option for our practical proof of concept.

### Network Based IDS

In the context of network-based IDSs, these devices determine intrusions from analysing captured packets. These devices consist of a dedicated host or multiple sensors, that may be located at different parts of the network, and they then report to a single control console. Network-based IDSs can effectively monitor a large network. Installing them does not significantly affect the network's performance, and they operate without disrupting regular network activities. A network previously designed without an IDS solution in it can easily adopt a network-based IDS. However, this may often have limitations in the face of a large, busy network with high traffic, as the processing mechanisms may not recognize an attack [Mell, 2003].

Typically, this form of deployment often necessitates the use of multiple sensors. These sensors can come in two varieties: appliance-based, employing dedicated hardware solutions, requiring specialized NICs and their corresponding drivers; or software-based, where software is installed on hosts. In the case of software-based sensors, the host environment must meet specific system requirements to ensure the effective operation of the software components [Babatope et al., 2014].

Another aspect of this solution is the location at which the IDSs are inserted. The location can be inline. This type of deployment involves a monitoring strategy upon which the IDS is positioned directly in the network traffic path. This is usually employed to allow immediate action, such as blocking or diverting malicious traffic. Its location can also be in a passive mode. In this solution, a copy of the traffic is taken, and no actual traffic passes through it. A common example of this solution is through the use of a network Test Access Port (TAP), where all traffic is copied from a physical network device like a router [Babatope et al., 2014].

### Host Based IDS

Host-based IDSs focus on analyzing the activity that occurs on a specific computer, requiring them to collect data directly from the host. This level of access allows these IDSs to scrutinize host activities with great precision. In the context of networking, the amount of data to be analysed is much smaller and for that reason attacks that may have gone unnoticed have a higher probability of being detected given this localized perspective. As mentioned previously, through the use of eBPF, encrypted traffic can now be analysed, this is only one of the many new solutions that host-based IDSs can possess. However, there are disadvantages to consider. The maintenance of this solution is also not trivial given that attention must be given to every host that is monitoring intrusions. Moreover, not only is the host allocating resources for the execution of this solution which may lead to performance loss, but because it resides within the host itself, if this falls victim to an attack, the IDS may also be hijacked [Mell, 2003].

In the context of developing a proof of concept, the host-based implementation of eBPF stands out as the most logical and effective approach. This is due to several key factors. One of the primary reasons for favouring a host-based eBPF approach is its ability to provide deep insight into network traffic at the host level, gaining visibility and control that is difficult to achieve using alternative methods. Furthermore, the performance impact of eBPF in a host-based configuration is minimal, effectively addressing the issue of resource allocation.

### 2.2.2 IDS Event Examination Approach

In the realm of event analysis and processing methods, two primary categories emerge: signature-based and anomaly-based. While most commercial and open-source IDSs rely on a signature-based approach, anomaly-based IDSs are less commonly utilized and remain a subject of ongoing research. This subsection will delve into the reasoning behind this by analysing how each of these approaches examines data.

#### Signature Based IDS

In this approach, detection is achieved by matching activities with a predefined set of events that represent well-known attacks. These events are formulated on a model of how an intrusion typically works and the expected traces it should leave in the system. In simple terms, we can define what is considered abnormal behaviour and then check if the observed behaviour matches these definitions. It is important to note that in this system the detection method has no conceptual knowledge of what normal behaviour of the environment equates to, and it simply examines patterns that could indicate an intrusion. These systems excel at detecting such attacks, resulting in a minimal rate of false positive identifications. However, due to their reliance on a fixed dataset of known attacks, they may fail in the face of new attack methods and require periodic updates to remain effective [Axelsson, 2000; Mell, 2003].

Within this realm, there are, however, subcategories. This is by no means an exhaustive listing of every method and only serves as a mean of elucidating the various possible solutions that one may employ to satisfy this type of detection. As mentioned by [Liao et al., 2013] researchers often only study the detection approaches from the two methods already mentioned and further subcategorization often lacks a more detailed view. Having this in mind some subcategories are state-based, which employs a finite state machine based on network behaviours to recognize attacks, string matching, where it analyses substrings in the data transmitted, and Rule-based specify a set of rules which describe an intrusion [Axelsson, 2000; Liao et al., 2013].

#### Anomaly Based IDS

Anomaly-based approaches operate by identifying activities within a host or network that derive from normal operations. It is rooted in the notion that malicious actors often exhibit behaviour distinct from that of regular users. However, they are notorious for generating numerous false alarms, as typical user and network behaviours can exhibit considerable variability. Despite this drawback, researchers argue that anomaly-based IDSs excel in detecting new attacks, in contrast to signature-based systems that rely on historical attack data. While some commercial IDSs incorporate limited anomaly detection, few rely exclusively on this technology, but ongoing research in this field persists [Mell, 2003].

Like the previous method, there can be sub-categories of this approach. This approach can be based on statistics where a profile of the network, which describes normal behaviour, is built from several parameters. If, for example, the result of these parameters exceeds an established threshold, an intrusion is inferred. Another common category is Heuristic-based, it involves defining a set of heuristics or rules that describe what is considered normal or suspicious behaviour. When the IDS detects behaviour that violates these predefined heuristics, it raises an alert [Axelsson, 2000; Liao et al., 2013].

One of the objectives of this thesis is to propose a solution that uses ML. This solution aligns itself with the realm of anomaly-based IDS, emphasizing its capacity to discriminate between normal and abnormal network activities. This approach, as previously mentioned, distinctly differs from the signature-based paradigm, as it doesn't rely on predefined attack patterns but instead operates by identifying deviations from established norms within the network environment.

### 2.2.3 IDS Non-Detection Traits

The scope of an IDS extends well beyond its mere detection methodology. It demands consideration of many principles and factors to guarantee its efficacy in safeguarding network and system security. Some of these factors relate to the response time, the type of response, and how and what data are processed.

#### Timeliness

This characteristic of IDS refers to the ability to respond to events concerning the time in which they occur. There are two main categories, real-time and non-real-time. An IDS that offers a real-time response is an IDS that responds to events immediately or near immediately. The timeliness aspect in a real-time IDS is employed to detect and respond to an attack as it happens, to minimize any possible damages, and to reduce their impact. On the other hand, non-real-time IDS are usually employed for forensic-like purposes, to recognize breaches that happened in the past. This category operates with historical data or logs [Axelsson, 2000].

#### Response

Another aspect to consider is the response mechanism itself. This response can be passive or active. In a passive response, no direct action is taken against the threat, instead, relying on human intervention to provide a decision on how to handle the intrusion. This type of response includes the generation of alerts and/or notifications and logs. In an active response, direct actions are taken against the intrusion, for example by blocking or diverting the traffic. This response is applied either to the system under attack or to the attacking system [Axelsson, 2000]. While it is the case that this form of response is the most effective in combating a threat in real-time, before damage can occur, there are risks associated. When the detection mechanism erroneously classifies legitimate traffic as malicious, it results in a false positive detection, and, consequently, the intended actions destined to be applied to intrusions would be applied to normal activities [Axelsson, 2000].

#### Data

Finally, there are concerns about how data is processed and what type of data is collected. The processing of data can be done in two approaches: continuous and periodic. Continuous data processing involves analysing the data continually; this means that as data arrives it is analysed and classified. Periodic data processing, on the other hand, involves analyzing data at predefined intervals. As noted by [Axelsson, 2000] this is associated with the timeliness of response, however, these two are not linked one-to-one since a system can continuously process the data but perhaps with delay or process batches of data in real-time. Concerning the data used to identify attacks, information can be collected from the network traffic or activities logged from a system [Axelsson, 2000].

### 2.2.4 IDS Machine Learning Models

Machine Learning (ML) is a component of Artificial Intelligence (AI) in which systems acquire the knowledge and ability to improve their performance without the need for explicit, rule-based programming. ML can be classified into four categories [4]: supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning.

#### Supervised Learning

Supervised Learning is a paradigm in which an algorithm is trained on a labelled dataset consisting of input-output pairs. The relation between input and output can be established through classification or prediction. The input is first divided into two datasets, train and test. The model must then learn patterns from the training dataset and apply them to the test dataset. Figure 2.7 is meant to represent the overall workflow of this type of learning [Mahesh, 2020; Saranya et al., 2020].

[4]This categorization was conceived from the categorizations employed in the papers [Mahesh, 2020] and [Saranya et al., 2020].

[5]This Figure has been recreated based on supervised learning workflow illustration put forth by [Mahesh, 2020]

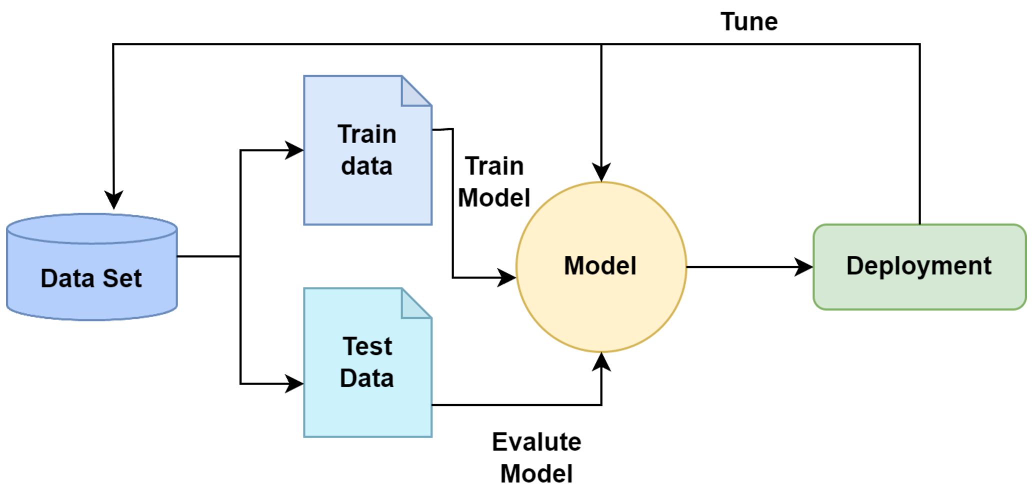


Figure 2.7: Supervised Learning workflow [5]

Numerous algorithms embrace this form of learning, and while there are a multitude of options, certain algorithms stand out as the most frequently discussed and widely used in the literature. These prominent algorithms encompass Decision Trees, Naïve Bayes, Support Vector Machines, Neural Networks, and Random Forests.

- Decision Trees (DT) function by traversing a sequence of decisions to arrive at an outcome. This algorithm can be visualized as a structured graph, where the nodes signify decision points, and edges denote the conditions or criteria that lead to subsequent choices. This is a popular solution given its simplicity and ease of implementation [Haq et al., 2015; Mahesh, 2020].

- Naïve Bayes is a probabilistic classification algorithm based on Bayes' theorem. It operates under the assumption of conditional independence among features, put simply, it assumes that the presence of a feature is unrelated to the presence of others. It is more commonly employed for text classification [Haq et al., 2015; Mahesh, 2020].

- Support Vector Machine (SVM) operate by identifying a hyperplane in a high-dimensional feature space that maximally separates different classes. In summary, it draws boundaries between classes, making these boundaries distance the maximum to minimize errors in classification [Haq et al., 2015; Saranya et al., 2020].

- Artificial Neural Network (ANN) is an approach that was inspired by how human brains function. They consist of various interconnected layers of artificial neurons that process data. Each neuron takes input, performs a computation, and produces an output [Haq et al., 2015].

- Random Forest (RF) work similarly to DT, however, they employ a model which predicts an outcome by constructing many decision trees and pooling them together to determine a result. These types of algorithms, where multiple learners are combined to create improved classifiers, can be categorized into ensemble classifiers [Saranya et al., 2020].

### Unsupervised Learning

Compared to the previous form of learning, this approach does not accompany training or labelled data. Instead, it seeks to discover inherent patterns, structures, or relationships within the data. Its main use is within the scope of clustering. Figure 2.8 depicts the workflow of such an approach.

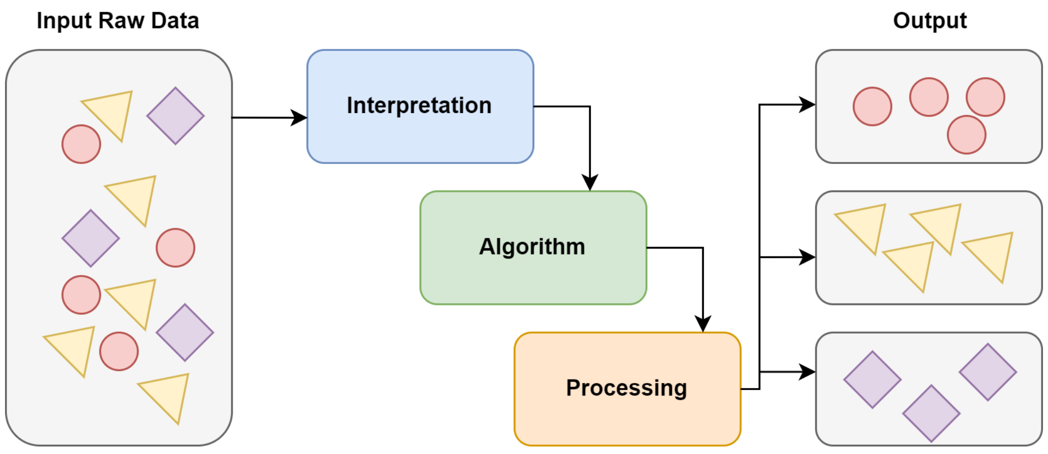


Figure 2.8: Unsupervised Learning workflow

An algorithm that follows this approach is K-means. This algorithm follows a procedure which begins by defining k centroids and then iteratively refines them until convergence. It assigns data points to the nearest centroid, recalculates the centroids, and repeats this process until the centroids no longer change significantly [Mahesh, 2020].

### Semi-supervised Learning

This learning method sits between the two previous methods, combining elements of both supervised and unsupervised learning. They make use of both unlabelled and labelled data. This approach is most effective in situations where acquiring labelled data is costly or time-consuming [Mahesh, 2020].

### Reinforcement Learning

Unlike supervised learning, where algorithms learn from labelled data, or unsupervised learning, which discovers patterns in unlabelled data, reinforcement learning centres around the idea of learning by interaction and feedback. An agent takes action within an environment to achieve a goal. Through these interactions, the agent receives feedback that comes in the form of a reward or a penalty. Over time, after multiple interactions, the agent learns a method on which its decisions result in the highest rewards.

**Performance evaluation**

In addition to the task of selecting the most suitable learning method for the deployment of an IDS, several other important considerations require attention. Among these considerations, the evaluation of the performance of the chosen approach stands out as a pivotal step. The effectiveness and performance of machine learning algorithms can be rigorously assessed through the use of various metrics that provide valuable insights into their capabilities. Metrics such as accuracy, precision, recall, F-Score and many others. The following enumerations correspond to these metrics and terminology [6].

Terminology:

- True Positive (TP): number of instances correctly classified as positive.
- True Negative (TN): number of instances correctly classified as negative.
- False Positive (FP): number of instances incorrectly classified as positive.
- False Negative (FN): number of instances incorrectly classified as negative.

Formulas:

- Accuracy = $\frac{TN+TP}{TP+TN+FP+FN}$ ; proportion of correctly predicted instances.
- Precision (P) = $\frac{TP}{TP+FP}$ ; metric that quantifies the proportion of instances identified as positive by the model that are, in fact, true positives.
- Recall (R) = $\frac{TP}{TP+FN}$ ; metric that quantifies the proportion of actual positive instances that the model correctly identifies.
- F-Score = $\frac{2*(R*P)}{R+P}$; metric that balances the trade-off between precision and recall, close to 1 meaning that it is well balanced, close to 0 meaning the model excels and is one of the aspects and is therefore unbalanced.
- Detection Rate = $\frac{TP}{TP+FN}$ ; for IDS most researchers employ the detection rate metric. This metric is calculated in the same manner as recall. The mention of this metric is only for the cohesion of the different terminologies.
- False Positive Rate = $\frac{FP}{FP+TP}$ ; metric that indicates the proportion of instances identified as positive by the model that are not true positives.
- Area Under Precision-Recall Curve (AUPR) = $\int_0^1 (\frac{TP}{TP+FP}) d(\frac{TP}{P})$ ; metric to establish, at various thresholds, the tradeoff between the recall and the precision.

[6]This information was obtained from the paper [Saranya et al., 2020] which employs a chapter on the strategies used to evaluate an ML algorithm in the context of IDSs.

## 2.3 Port Scanning

When embarking on the implementation of an IDS, a critical aspect to bear in mind is the analysis of the threats the solution intends to mitigate. Recognizing that diverse threats necessitate distinct IDS approaches, it is imperative to tailor the deployment to specific threat profiles, thereby enhancing the system's detection capabilities. As mentioned in the subsection 2.2, one of the motivations behind the deployment of such a tool is to recognize and mitigate probing activities such as port scanning. Identifying port scanning as the attack vector we aim to combat, allows us to develop a solution which is tailored to this issue and therefore much more effective. Hence, this particular subsection serves the purpose of delving into the intricacies of such threats. This level of detailed understanding ensures that our solution is thoroughly informed.

The initial step an attacker often undertakes when preparing to launch an attack is reconnaissance. Reconnaissance is the process of gathering information and intelligence about a target to understand its vulnerabilities and potential weaknesses. In the context of networking, this can come in the form of scanning devices to determine the open ports and services available in each host. This is known as port scanning. Because port scanning can come as a precursor to an attack, detecting such activities can be pivotal in alerting or preventing potential attacks.

### 2.3.1 Port Scanning Methods

There are many ways a port scan can be carried out, however, they all have the same objective. The classification of a port scan has two perspectives, the attacker perspective and the target(s) perspective. The attacker's perspective corresponds to how the source of the attack is represented. The target(s) perspective corresponds to how the attacker scans the hosts.

When an attacker performs the port scan, the source of the attack can be represented in two categories, single source and distributed scan [Bhuyan et al., 2011]. In a single-source scan, the attacker performs the probing from a single host. This single host may scan in a one-to-one configuration or a one-to-many configuration. In a distributed scan, the attack is scattered through multiple hosts, put simply, they are multiple single-sourced scans; however, they are coordinated, meaning that these individual scanning activities are synchronized and orchestrated to work together as a collective force. These hosts may scan in a many-to-one configuration or in a many-to-many configuration. Figure 2.9 is meant to represent these methods.

The target ports can also be scanned using different approaches. The attacker may employ a vertical, horizontal, or block scan technique. In a vertical scan, the attack focuses on scanning several ports of a single host. This approach is considered somewhat unsophisticated, if carried out in a single source manner, because of its simplicity, making it relatively easy to detect. A horizontal scan operates by scanning the same port on a range of hosts. This is usually employed when the attacker is aware of a vulnerability associated with that specific service running on that port number. A block scan is a combination of the previous two approaches, which results in a wide range of port identification [Bhuyan et al., 2011; Lee et al., 2003].

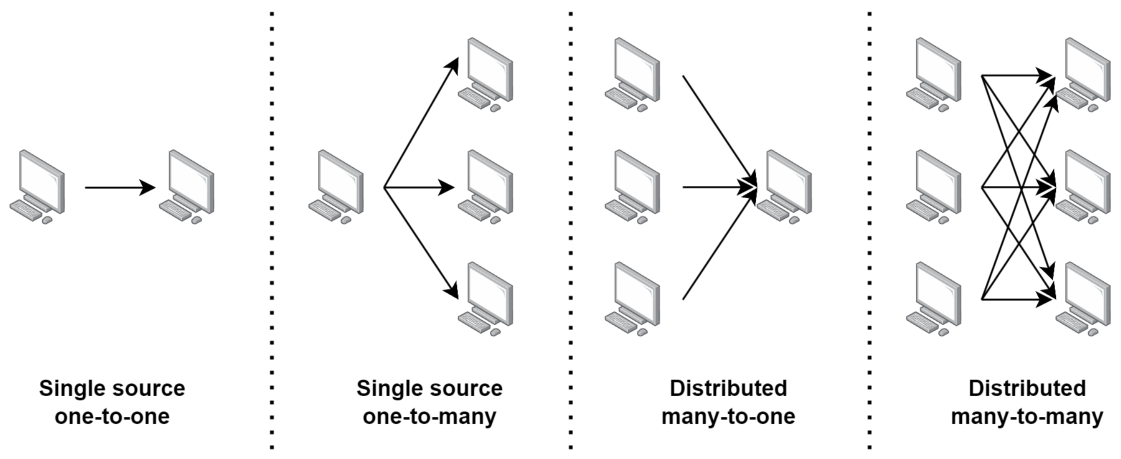


Figure 2.9: Port scan methods

## 2.3.2 Port Scanning Types

Having established how a port scan can be conducted, it is also important to understand the various types of scans. These scanning types offer distinct approaches to probing a target's open ports and services. Each of these scan types has unique advantages and limitations. By analyzing and understanding the mechanism by which the techniques gather information, one can develop a detection approach that is tailored to identify and mitigate these specific scanning methods effectively.

There exist several types of port scans, the most extensively researched ones being Transmission Control Protocol (TCP) scans like SYN, ACK, FIN, NULL, and User Datagram Protocol (UDP) scans. It is important to note that delving into these types in great detail may not always be necessary, as they tend to exhibit similar behaviours and share common features. However, it becomes evident that more targeted and specialized scans can possess distinct characteristics. If one aims to detect these more specialized scans, further examination may be necessary. The categorization that follows results from the examination of more than a dozen port scans present in the documentation of Nmap [Nmap Reference Guide]. Other tools can be used to perform scans, however, Nmap stands out in the literature. Therefore, their reference guide will be used to assess the intricacies of many scan types. To enhance the organization of each scan type, they have been classified into distinct categories: Standard, Stealthy, and Specialized [7].

[7]To avoid continuous citation of Nmap [Nmap Reference Guide], consider that the following details originate from that reference.

### Standard Scans

This type of scan was classified as such because it is commonly used and does not fall within the other categories. In this domain reside the TCP Connect and UDP scans.

- A TCP Connect scan operates by conducting full complete connections to open ports, unlike other approaches which rely on half-open connections. Taking this into consideration, this scan will in consequence take more time and be more prone to be logged.
- A UDP scan targets UDP services. In this approach, the payload is empty, unless further specification is provided or the destined port is common, but if not, empty payloads may give away the scan. To determine if these are open, closed, or filtered it analyses the received packet. If it receives back a UDP packet, then the port is open, if it receives an ICMP port unreachable error, it will classify it closed or filtered, depending on the error type.

### Stealthy Scans

The scans that fall in this category are the ones that are intended to be used discreetly or subtly so that their presence is hidden from the rest of the traffic. In this domain reside the TCP SYN, NULL, FIN, ACK, Windows, and Xmas scans.

- A TCP SYN scan, also known as a half-open scan, operates by sending a SYN packet, as if one desired to establish a connection and then waiting for the response. A "SYN/ACK" response typically signifies that the port is open, whereas the presence of a "RST" response usually indicates that the port is not actively listening or open. If no response is received after several retransmissions or ICMP errors are retrieved, then, is marked as filtered.
- TCP NULL, FIN, and Xmas scans exhibit identical behaviour, with their distinction being solely based on the specific flags they manipulate. NULL does not set any bits, FIN sets the TCP FIN bit and Xmas sets FIN, PSH, and URG flags. They are used to exploit the TCP RFC to determine the status of ports. In systems adhering to this RFC specification, if a packet lacks the SYN, RST, or ACK bits, it will trigger an RST response when the port is closed, and no response when the port is open. With the FIN, PSH, and URG flags, information about the ports can then be obtained.
- TCP ACK scan operates distinctly to the previous ones, as it does have the objective of determining if ports are open or closed, but to determine if a firewall or filtering device is present. In this scan, a TCP packet is sent with the ACK flag, if the target responds with an RST it indicates that the target is reachable, however, it is inconclusive on its status; if there is no response or the response is an ICMP error, it could indicate that the port is filtered, and the firewall or filtering device is dropping the packets.

- TCP Window scan operates in the same domain as ACK, however, it attempts to make the distinction between opened and closed ports. The TCP Window from the RST packet returned will indicate this information. If the window size is positive, the port is opened, if the size is zero, then it is closed. There are, however, some nuances and this result is not always reliable.

#### Specialized Scans

Finally, some scans are more advanced and targeted. The scans that fall into this category are the TCP Maimon, SCTP COOKIE ECHO, Idle and FTP bounce. Certain scans within this group are also discreet, but due to their more targeted methodology, they were classified within this class.

- The TCP Maimon has the same objectives as NULL, FIN, and Xmas scans. The only difference is that its probe relies on a FIN/ACK flag.
- SCTP COOKIE ECHO scan makes use of the behaviour of SCTP implementations, which quietly discard packets containing COOKIE ECHO chunks when the target port is open while issuing an ABORT signal when the port is closed.
- Idle scan relies on fundamental principles: a TCP port's status can be determined by sending a SYN packet, eliciting a SYN/ACK response for an open port and an RST for a closed one. Unsolicited SYN/ACK triggers an RST, and monitoring the IP packet's ID helps determine the number of sent packets. By combining these elements, an attacker can stealthily scan a network while appearing to be an innocent zombie machine performing the scan.
- FTP bounce exploits vulnerable FTP servers to indirectly probe other hosts by redirecting the scan through the FTP server, potentially revealing open ports.

### 2.3.3 Port Scan Detection

Having presented the traits associated with port scanning, we can now formulate a strategic approach to detect these scanning activities by considering how they are executed. In this section, we refrain from delving into the specifics of implementation; instead, our focus lies on evaluating these discerned characteristics to identify potential methods for achieving effective detection. From the preceding sections, two distinct domains emerge to explain the methodology behind the execution of port scans and the scans themselves.

In the realm of an attacker's scanning methodology, a detection strategy might entail a comprehensive analysis of network flows, aimed at identifying patterns. For instance, a straightforward example involves observing a host connecting to a multitude of ports on another host within a specific time interval.

When focusing on the scans themselves, a more targeted approach becomes imperative, involving an analysis of packet characteristics to discern the specific attributes associated with the scanning activity. An example would be to analyse the flags present in the packet and the size of certain fields.

If we consider Snort or Suricata, which are open-source IDSs, both function in a signature-based approach making their detection based on rules. These rules can extended to packet examination or connections characteristics evaluation. If we take, for example, the port scan inspector by Snort, which identifies port scans, Portsweeps, Decoy port scans (scans with spoofed IPs) and Distributed Scans, the detection is mostly concerned with connections to closed services. So the examination is more focused on network flow analysis [README.sfportscan; Snort 3 Inspector Reference].

While it is possible to utilize either of these approaches individually, the most robust detection strategy can be achieved by combining both. Incorporating both data flow analysis and packet attribute examination enhances the effectiveness of the detection process.

## 2.4 Chapter Wrap-Up

This chapter provides a comprehensive overview of several key topics while delving into preliminary research, thus allowing the establishment of some initial decisions. By exploring these subjects, we aim to lay the foundational knowledge needed for the rest of the document. The topics discussed are concerning eBPF, IDSs and port scanning.

In Section 2.1, a comprehensive exploration of the eBPF technology was carried out, highlighting its suitability as a solution for IDS. This examination delved into the fundamental components of eBPF, including the Verifier, JIT Compiler, Maps, and User Space platforms. Furthermore, the section details the integration of XDP within this context, shedding light on the diverse deployment modes available and their respective limitations. Having established the XDP offloaded mode as the most desirable option, the section culminated in an in-depth analysis of the dedicated hardware and software infrastructure that harnesses this solution.

Section 2.2 details the characteristics that one must consider when developing an IDS solution. This examination delved into the monitoring methods, host or network-based, detection method, anomaly or signature-based, and other non-detection traits an IDS uses in its functionality. Finally, we have a study of the various ML methods one can employ when developing an ML IDS solution.

Having established the attack vector as port scanning, Section 2.3 provides research on the topic. Firstly, it examines the various modes an attacker may configure the attack machines, single or distributed, and the scanning method on the target, vertical, horizontal, or block. It then delves into the examination of the operations executed by various types of scans. Finally, an examination of various detection approaches may be used.

# Chapter 3

# Literature Review

This chapter is dedicated to the research and documentation of findings related to the topics at hand. Within the context defined previously, examining the literature concerning eBPF-IDS solutions and port scanning is essential. The research of the state-of-the-art of eBPF-IDS approaches can help guide decisions, understand constraints and means of avoiding them, and finally allow for the placement of the work developed concerning other works. Given the defined offensive approach, it is also necessary to understand how this issue is fought. Making the intersection between these two subjects can allow us to delineate the best decisions to make.

Section 3.1 is dedicated to the research of eBPF-IDS solutions. The Section that follows, 3.2 focuses on the research in port scanning and ML. From these two sections, Section 3.3 attempts to intersect the previous two domains. Finally, Section 3.4 provides a summary of the chapter.

## 3.1 eBPF-IDS State of the art

Given that this is the core of the work being developed, more elaborate research is imperative. This not only ensures a comprehensive grasp of the issue but also facilitates an evaluation of the literature, particularly in the realm of eBPF and IDS intersection. Whether the existing literature is limited or abundant, this approach enables a discerning selection of relevant works.

### 3.1.1 Research Strategy and Works Identified

To search relevant papers regarding the topic, a logical keyword combination must be used. The search terms are in conjunction with boolean operators. These searches were done in two databases, Scopus and Google Scholar. The first search was conducted as "extended Berkeley packet filter" AND "intrusion detection system" which wielded 0 results in Scopus and 89 in Google Scholar, respectively. However, we must consider that most researchers fall back to the acronym of some terms, in this case eBPF and IDS. So combinations of previous terms and acronyms were used like: "eBPF" AND "intrusion detection system", "extended Berkeley packet filter" AND "IDS" and finally “eBPF” AND “IDS”. These results produced in Scopus and Google Scholar the respective results: 4 and 195, 0 and 213, 4 and 520.

From the Scopus results, a manual examination of each paper can be easily done to determine whether they fit the objectives of this review. From it, two papers related to the topic at hand were retrieved: [Wang and Chang, 2022] and [Pacífico et al., 2022].

Using Google Scholar, a wide array of documents on eBPF and IDS topics was explored. However, it became apparent that the term "IDS" was frequently used to reference "Identifiers" rather than "Intrusion Detection Systems." In light of this, the complete "Intrusion Detection System" keyword was employed to refine the search results and identify pertinent literature. Given the several papers available, it quickly became clear that many addressed the topics but did not delve into the implementation of Intrusion Detection Systems using eBPF. So the keyword “implementation” was added along with the full “extended Berkeley packet filter”. The year was also used as an exclusion criterion, removing papers before 2019 which yielded 73 results. To select the relevant papers, a thorough analysis of their titles was carried out to determine whether they aligned with the desired scope. In cases of uncertainty, abstracts were consulted for further clarification. In the end, eight papers were retrieved: [Pradhan and Mannepalli, 2021], [ANAND et al., 2023], [Ognibene, 2021], [Bachl et al., 2021], [Carvalho et al., 2023], [de Carvalho Bertoli et al., 2020], [Sadiq et al., 2023] and [Wieren, 2019].

In summary, from both databases, a total of 10 papers were withdrawn. The table 3.1 depicts the summary of the search conducted and the results obtained. Important to note that the last search was only conducted in the Google Scholar domain due to the reason previously mentioned.

| **Keywords** | **Scopus Results** | **Google Scholar Results** |
|---|---|---|
| "extended Berkeley packet filter" AND "intrusion detection system" | 0 | 89 |
| "eBPF" AND "intrusion detection system" | 4 | 159 |
| "extended Berkeley packet filter" AND "IDS" | 0 | 213 |
| “eBPF” AND “IDS” | 4 | 520 |
| "extended Berkeley packet filter" AND "intrusion detection system” AND “implementation” | - | 73 |

Table 3.1: Search Strategy for eBPF-IDS papers

### 3.1.2 Synthesis of Identified Works

Having selected the previous papers an overview of each one is due. With that, we can assess the common practices, challenges, and limitations we may face and how researchers address them as well as details surrounding their solutions, namely algorithms and implementation strategy.

Authors [Wang and Chang, 2022] proposed an IDS implementation using eBPF composed of two interconnected parts. The first part operates within the Linux kernel and uses eBPF to rapidly identify and drop a substantial portion of packets that do not align with any predefined rules, the algorithm used to perform this analysis is the Aho–Corasick (AC) algorithm. The second part functions in user space and assesses the remaining packets from the first part to identify matching rules utilizing a modified version of Snort's rule set. This subset was created due to eBPF constraints. During the development process, other issues arose. Due to the Verifier constraints, all execution paths must have fewer than one million instructions, and an eBPF program should have fewer than 4096 instructions. To address this, they used tail calls and bpf-to-bpf function calls to reduce code duplication. The eBPF Verifier also requires memory access validation, which was performed manually throughout the code. In older Linux versions, backward jumps were disallowed, which meant no loops in eBPF programs. This issue was resolved in Linux 5.3 with bounded loops, allowing fixed iterations. To successfully implement their solution, they adopted the bounded loop method.

Authors [Pacífico et al., 2022] present a comprehensive system for the implementation of an IDS that takes advantage of SmartNICs and eBPF technology. The system architecture consists of serverless components and hardware devices, where users can interact with the system to create, execute, and manage packet processing filters. A specific focus on SmartNICs, such as the Netronome CX 2x10 GbE, is given. This is done to back up the chosen hardware used, as this possesses high processing power and the ability to offload eBPF programs as mentioned in subsection 2.1.5. The system's implementation revolves around an interface, *ebpfaas-cli*, designed to create, update, remove, and execute eBPF/XDP filters. Users use this interface to specify filter filenames, triggering the creation of containers based on custom templates. Within these containers, an index program communicates filter execution times to a handler program, responsible for compiling and generating eBPF instructions in the C language. The transmission component forwards these instructions via TCP/IP sockets to the filter queue, which organizes the instructions using a first-in, first-out scheduling approach. The SmartNIC executes the first loaded filter from the queue with status updates provided to the user. One of the limitations faced by the authors was that the eBPF Verifier only allows for a limited amount of loops. The Clang compiler's "loop unroll" directive must be used to work around this limitation. This is because the eBPF code verifier has restrictions on the complexity of loops it can handle. However, using this directive, depending on the size of the regular expression (RegEx) used in the filter, the number of instructions can exceed the limit supported by Netronome, which is 131,072 instructions as indicated by the authors. This issue does not occur when filters check packet headers or apply RegEx to specific, defined parts of the packet.

Both authors [Pradhan and Mannepalli, 2021] and [Bachl et al., 2021] documented an approach very similar to each other but very distinct from the previous authors. In their approach, they employed Machine Learning (ML) algorithms to make the distinction between normal network traffic and malicious traffic. The ML was elected on the basis of the limitations of eBPF. Both authors decided to employ Decision Trees (DT) for its simplicity and effectiveness. In both methods, they employ a solution on which a network flow is tracked.

Authors [ANAND et al., 2023] proposed a solution similar to [Pradhan and Mannepalli, 2021] and [Bachl et al., 2021]. Their intrusion detection approach is made through the use of ML to make a distinction between normal and abnormal network traffic. In their solution not only DT is used, but they also present Random Forest (RF), Support Vector Machine (SVM) and Twin SVM. Electing RF as the one with the best performance.

The paper [Ognibene, 2021] proposed a DDos detection system using an algorithm named LUCID, which employs Deep Learning techniques. eBPF would be used to obtain the information necessary for the execution of the algorithm. Additionally, Polycube and DeChainy, eBPF frameworks used for network monitoring, were employed. Similarly to [Wang and Chang, 2022], the detection is not performed in the kernel, but instead in userspace. Besides the issues already documented by the previous authors about the eBPF Verifier, some limitations related to the XDP mode are presented. When it comes to driver mode, it requires a specific memory model and operations like `XDP_REDIRECT`, are not possible between different drivers.

The work by [Carvalho et al., 2023] proposed an architecture to perform classification directly on network devices. The approach involves optimizing machine learning models taking into account eBPF constraints. In their solution, the K-Nearest Neighbors (KNN) make the distinction between data.

The work [de Carvalho Bertoli et al., 2020] presents a solution using eBPF and XDP to block TCP flag-based probing attacks employing a signature-based approach. By examining the probing characteristics of the Nmap tool, the attributes deemed relevant were used to build a filter to drop these connections.

Authors [Sadiq et al., 2023] also developed a solution target for DDoS detection. Their solution also focused on the use of filters. In their approach, the rules for DDoS mitigation are meant to be automatically generated and incorporated into a filter. They assume that a network operator is in charge of being able to alter filter rules. A limitation presented by the authors is that as eBPF maps grow in size the loading time also increases. However, in their exploration, the maximum size needed for their solution was measured and in consequence, the loading speed in their proposal was deemed acceptable.

Finally, [Wieren, 2019], also presents a solution dedicated to DDoS detection, however, their approach is developed for a Kubernetes environment. Although not classified as such, we have extrapolated their detection approach into a filter-like mechanism as their approach uses an unnamed algorithm that examines the characteristics of connections to determine if it is malicious or not. One identified limitation is that there are no established standards in the eBPF environment, making it difficult for organizations to adopt.

The table referenced as 3.2 provides a succinct enumeration, offering a comprehensive overview of each identified work in the analysis.

| **Author** | **Year** | **Cited** | **Algorithm** | **Implementation Strategy** |
|---|---|---|---|---|
| [Wang and Chang, 2022] | 2022 | 12 | AC | Kernel and Userspace |
| [Pacífico et al., 2022] | 2022 | 0 | Filter | Hardware Offloaded |
| [Pradhan and Mannepalli, 2021] | 2021 | 1 | DT | Kernel |
| [ANAND et al., 2023] | 2023 | 0 | DT, RF, SVM, Twin SVM | Kernel |
| [Ognibene, 2021] | 2021 | 0 | LUCID | Kernel and Userspace |
| [Bachl et al., 2021] | 2021 | 14 | DT | Kernel |
| [Carvalho et al., 2023] | 2023 | 0 | KNN | Kernel |
| [de Carvalho Bertoli et al., 2020] | 2020 | 2 | Filter | Kernel |
| [Sadiq et al., 2023] | 2023 | 0 | Filter | Kernel |
| [Wieren, 2019] | 2019 | 9 | Filter | Kernel |

Table 3.2: eBPF-IDS Identified Works Summary

### 3.1.3 Analysis and Observations

The authors of the identified articles did not follow the same strategy to measure their solution. Having different metrics makes comparing solutions difficult. For that reason, we will examine the work firstly in a more high-level approach.

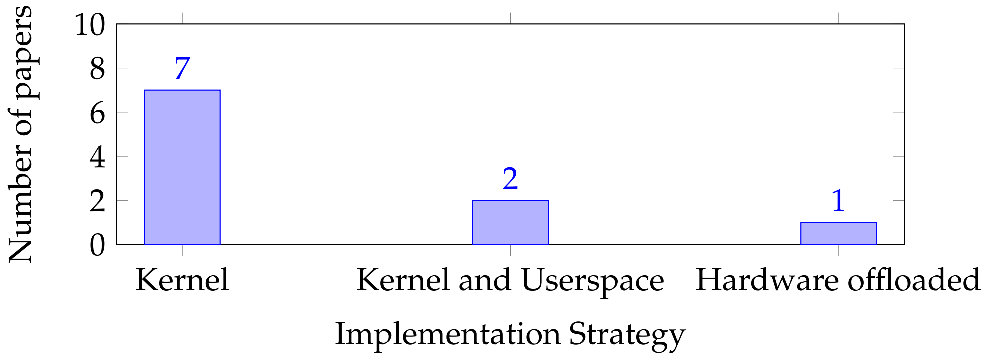


Figure 3.1: Implemention Strategies of the authors

There is a clear reason for the trend depicted in Figure 3.1. The solutions that combined Userspace did so due to the algorithms that they desired to implement, having only proposed eBPF as an initial auxiliary and not the core of the solution. Offloaded solutions required specific hardware. This limitation, in consequence, leads to a decline in its adoption. This leaves fully kernel solutions as the most utilized. The offloaded solution, in terms of implementation strategy, is the one that best aligns with our goals, however when it comes to the algorithms, the kernel solutions are the ones that best relate to the desired objectives. Considering that only one solution utilizes an offloaded strategy and its metrics are not directly comparable to other works, we find that kernel-based solutions align more closely with our goals. Furthermore, in theory, these kernel solutions could be extrapolated to an offloaded approach, making them a more promising avenue for our investigation. Consequently, we will conduct a more in-depth exploration of these kernel-based solutions.

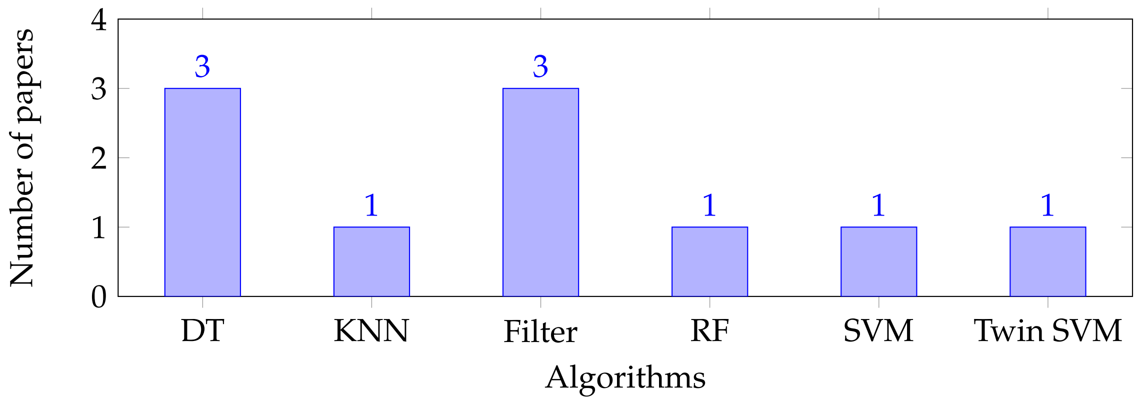


Figure 3.2: Implemented Algorithms by the authors

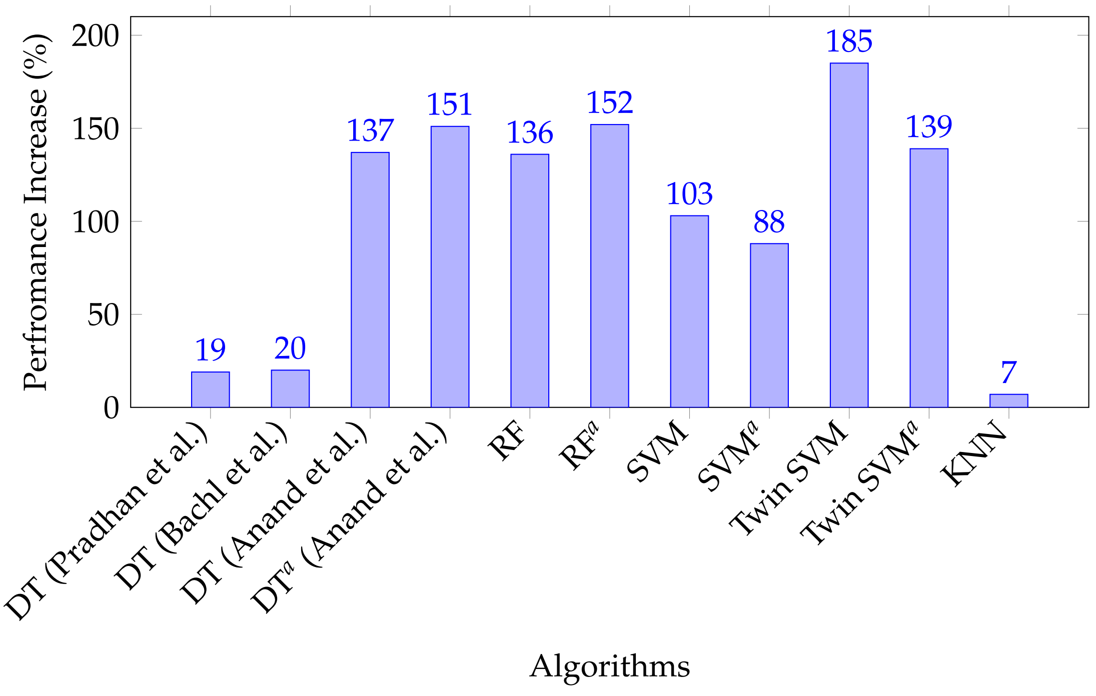
By examining Figure 3.2, depicting the distribution of algorithms within the Kernel domain, one can draw more conclusions. While DT and Filters may appear to be the most desired, it is crucial to recognize that this comparison isn't entirely fair. It is essential to note that the Filters domain encompasses numerous algorithms, many of which are unnamed. To establish a fairer comparison, it would be more appropriate to categorize all machine learning algorithms under a unified umbrella, allowing for a more aligned assessment. Having this in mind, it then becomes clear that ML algorithms greatly exceed the ones of filters.

Fortunately, this broad use of ML in this domain aligns with our objectives. In addition, by making this further categorization patterns emerge and metrics comparison between solutions become possible. Table 3.3 depicts the metrics measured by each author that could be compared across solutions. These metrics encompass the packets processed per second in the kernel as well as the same solution in userspace, along with the performance increases in percentage.

| **Author** | **Dataset** | **ML** | **Userspace pps** | **eBPF pps** | **Increase in %** |
|---|---|---|---|---|---|
| [Pradhan and Mannepalli, 2021] | CICIDS2017 | DT | 125 320 | 152 174 | 19 |
| [Bachl et al., 2021] | CICIDS2017 | DT | 125 420 | 152 274 | 20 |
| [ANAND et al., 2023] | CICIDS2017 | DT | 46 239 | 109 691 | 137 |
| | DoS/DDoS of CICIDS2017 | DT | 42 463 | 106 421 | 151 |
| | CICIDS2017 | RF | 45 978 | 108 534 | 136 |
| | DoS/DDoS of CICIDS2017 | RF | 41 632 | 105 245 | 152 |
| | CICIDS2017 | SVM | 45 590 | 92 978 | 103 |
| | DoS/DDoS of CICIDS2017 | SVM | 49 376 | 92 581 | 88 |
| | CICIDS2017 | Twin SVM | 38 430 | 109 865 | 185 |
| | DoS/DDoS of CICIDS2017 | Twin SVM | 49 376 | 117 536 | 139 |
| [Carvalho et al., 2023] | NSL KDD | KNN | 1 976 | 2 110 | 7 |

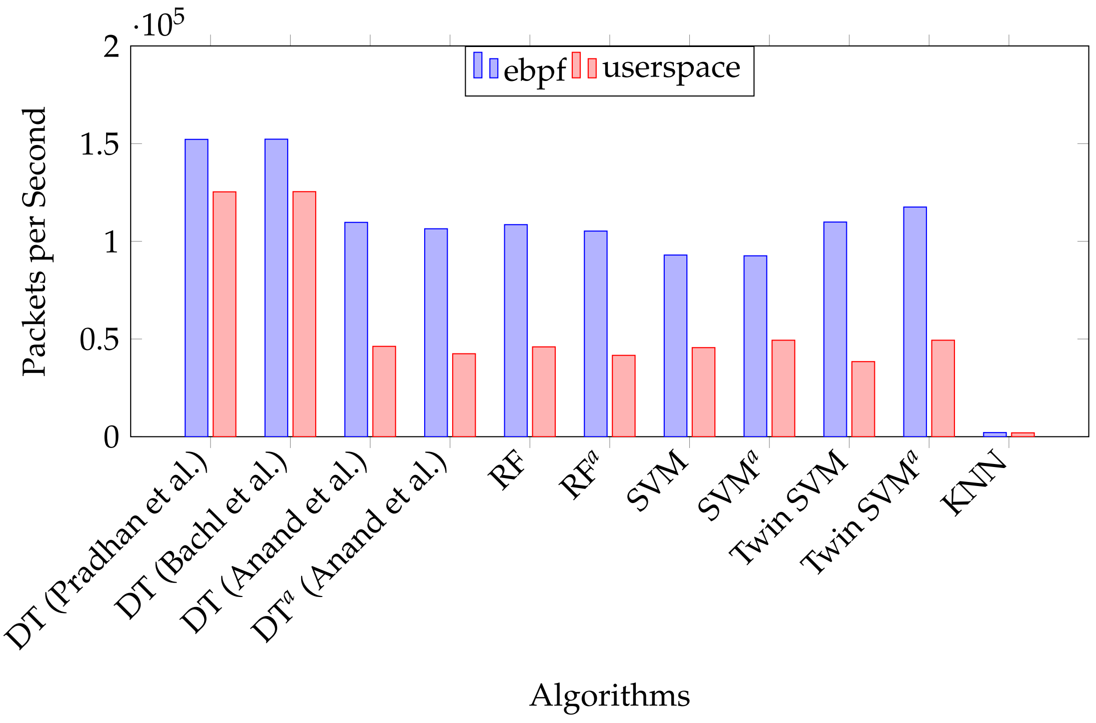
Table 3.3: Identified Works Algorithm Details

For a clear visualization of the trends, graphs were created. Figures 3.3 and 3.4 depict the trends in performance increase between algorithms and the packet processed in the userspace in contrast to in kernel. In case of doubt, the values represented in each chart correspond to the same order that in Table 3.3.



[a] DoS/DDoS of CICIDS2017 Dataset

Figure 3.3: Performance Increase of Implemented Algorithms by the Authors



[a] DoS/DDoS of CICIDS2017 Dataset

Figure 3.4: Packets per Second of eBPF and Userspace solutions

We cannot draw immediate conclusions from these graphs, for example, even though Twin SVM seems to exceed in performance improvement, we have to consider that in terms of actual packet processing, it is averaging the same results in the kernel as the other solutions by the same author but comparably the lowes in userspace. We also have to consider that different datasets are being used.

- Pradhan et al. and Bachl et al. approaches are similar and therefore their results also align. Both employed DT which shows the most packet processing capabilities, but a low-performance increase compared to other authors.
- Anand et al. solutions show that across Datasets DT and RF both present stable and good results concerning the authors' other solutions. When it came to SVM it presented the lowest results. Twin SVM as explained possess in the kernel, a speed close to DT and RF, but the lowes in userspace and that is the reason for the performance values, the other Twin SVM solution is a little bit better than DT and RF but we have to consider that is uses a subsection of the entire dataset and in terms of performance increase is lower than DT and RF in that same dataset.
- Carvalho et al. KNN scored the lowest in all metrics.

From this, considering all factors, DT seems to be the most favourable solution followed by RF. The Dataset most predominantly used is the CICIDS2017.

### 3.1.4 Additional eBPF-IDS Literature

The previous analysis was more targeted at specific solutions that used eBPF to develop an IDS and detailed their findings. In this way, we could possess information to lead our proof of concept. However, an additional document was identified as of interest. It was placed in this separate section as it does not align with any of the other works.

As mentioned in Chapter 2 Sub-section 2.3.3 Suricata is an open-source, signature-based IDS. In Suricata's 4.1, released in November of 2018, support for eBPF and XDP was added [Eric Leblond, 2019]. From the development carried out, eBPF can be used for three solutions:

- eBPF filters. A filter can be created to drop or only accept packets with a determined characteristic;
- eBPF load balancing. To distribute a packet among all sockets for example;
- XDP programs that can, for example, drop packets before they reach the network stack.

Adopting this type of solution could be an interesting area of study. However, this strategy seems to be more aligned with the use of eBPF for an initial preprocessing and not the main mechanisms of detection as we envision. Additionally, Suricata focuses on the utilization of filters, implying a signature-based approach. Given that our goals fall in the domain of anomaly-based detection through the use of ML, this is another motive to not follow this method.

### 3.1.5 Research Alignment with the Literature

Having presented and evaluated all the works that are associated with our specific research, this section will now attempt to place our work in comparison to the others. With this, we can see how it relates and what it offers different from the other solutions. This comparison is depicted in Table 3.4.

It is possible to clearly see the new domains our solution tackles, offering an offloaded solution, rarely adopted, and not intersected with ML in this domain; the attack vector focused on identifying different types of port scans, with the ML model hard-coded, different from the other authors. Some of the elements depicted in this table, related to our solution, have not yet been discussed. However, the context of this section, demanded that such a comparison take place.

| **Author** | **Implementation Strategy** | **Algorithm(s)** | **Attack Vector** [1] | **ML Storage** [2] |
|---|---|---|---|---|
| João Monteiro | Hardware Offloaded | RF | Port Scan | eBPF Maps |
| | Kernel | | | Hardcoded |
| [Wang and Chang, 2022] | Kernel and Userspace | AC | - | - |
| [Pacífico et al., 2022] | Hardware Offloade | Filter | - | - |
| [Pradhan and Mannepalli, 2021] | Kernel | DT | - | eBPF Maps |
| [ANAND et al., 2023] | Kernel | DT, RF, SVM, Twin SVM | - | eBPF Maps |
| [Ognibene, 2021] | Kernel and Userspace | LUCID | DDoS | - |
| [Bachl et al., 2021] | Kernel | DT | - | eBPF Maps |
| [Carvalho et al., 2023] | Kernel | KNN | - | eBPF Maps |
| [de Carvalho Bertoli et al., 2020] | Kernel | Filter | Flag-Based Probing | - |
| [Sadiq et al., 2023] | Kernel | Filter | DDoS | - |
| [Wieren, 2019] | Kernel | Filter | DDoS | - |

[1] May not be clearly defined ( using the attack vectors present in a Dataset, using community Rulesets ) [2] If applicable

Table 3.4: eBPF-IDS Identified Works Summary

## 3.2 Port Scan Detection with ML Work Review

The focus of this section will be to gather information to evaluate whether there are resources to allow the development of the practical scenario concerning the identified attack vector. Within this domain, the main focus is on the datasets that can be used to train the practical model alongside the most widely used ML algorithms employed for this purpose.

### 3.2.1 Research Strategy and Works Identified

The same set of databases, Scopus and Google Scholar, were utilized in this phase of the study. Initially, a search query in Scopus combining "port scan detection" AND "machine learning" yielded two results; however, neither aligned with the research objectives. Subsequently, an adjustment was made to the keywords, changing "port scan" to "portscan," which produced two new results. From these, one relevant paper was identified [Jasim et al., 2023]. Another search query, "detect port scan" AND "machine learning" on Scopus resulted in two papers, one of them aligned with our research goals, [Aksu and Aydin, 2018].

Moving on to Google Scholar the keywords "portscan detection" and "machine learning" resulted in 67 results, noting that works before 2019 were omitted to narrow down the scope. Firstly, two articles were identified, [Mir and Sandhu, 2019] and [Algaolahi et al., 2021]. However, one additional paper was considered extremely useful [Pittman, 2023], this paper encompasses a systematic review of port scans utilizing machine learning and was published recently. This paper can be used in a snowball-like effect, in the sense that enables us to uncover further relevant studies and insights in this domain. This specific paper analyses 15 papers on the issue. A cross-examination was done to evaluate if identified papers were not included, however, the work by [Algaolahi et al., 2021] was already contained within this review. For that reason we discarded it. With this, we can deem that we possess enough information to satisfy our defined goal. Selecting 2 papers from Scopus, and 2 from Google Scholar where one corresponds to an analysis of 15 papers. Table 3.5 depicts a summary of the search.

| **Keywords** | **Scopus Results** | **Google Scholar Results** | **Scopus Selected Paper** | **Google Scholar Selected Paper** |
|---|---|---|---|---|
| "port scan detection" AND "machine learning" | 2 | - | 0 | - |
| "detect port scan" AND "machine learning" | 2 | - | 1 | - |
| "portscan detection" AND "machine learning" | 2 | 67 | 1 | 2 [a] |

[a] Note that one of these papers is a systematic review, analysing 15 papers

Table 3.5: Search Strategy for eBPF-IDS papers

### 3.2.2 Analysis of Identified Works

The authors of [Jasim et al., 2023] proposed a solution to detect port scan attacks. In their solution, a decision table and OneR classification [OneR Algorithm] were utilized to make the distinction between normal and malicious data. The dataset employed was the CICIDS2017.

The paper [Aksu and Aydin, 2018] demonstrates a solution that also uses the CICIDS2017 dataset. In this approach, two algorithms are used, SVM and Deep Learning. In their research, SVM performed better than Deep Learning.

In the work [Mir and Sandhu, 2019] the CICIDS2017 dataset is used once again. The solution, however, employs new algorithms which encompass SVM, RF and ANN. From the conducted study, RF and ANN performed better than SVM.

Finally, the systematic review [Pittman, 2023] evaluated 15 papers on the topic at hand. Throughout the papers, various algorithms were utilized: RF, SVM, Regression, Naive Bayes and ANN. Also, a lot of datasets were employed: CICIDS2017, NSLKDD, MAWILab, PSA-2017, NMAP-A, NMAP-S, GEN, TCP, NMAP-Y, NMAP-F, and Bonafide. From their analysis, ANN presented itself as being the best across all papers, followed by RF. From the paper 34 algorithms were employed and 11 datasets were used, 47% chose the CICIDS2017 dataset.

Table 3.6 is meant to give a brief overview of each document. The chart 3.5 is meant to depict the distribution of the algorithms used.

| **Author** | **Year** | **Cited** | **Algorithm** | **Dataset** | **Favoured Algorithm** |
|---|---|---|---|---|---|
| [Jasim et al., 2023] | 2023 | 0 | Decision Table and OneR | CICIDS2017 | -[a] |
| [Aksu and Aydin, 2018] | 2018 | 98 | SVM, Deep Learning | CICIDS2017 | SVM |
| [Mir and Sandhu, 2019] | 2019 | 0 | SVM, RF, ANN | CICIDS2017 | RF, ANN |
| [Pittman, 2023] | 2023 | 3 | RF, SVM, Regression, Naive Bayes, ANN | CICIDS2017, NSLKDD, MAWILab, PSA-2017, NMAP-A, NMAP-S, GEN, TCP, NMAP-Y, NMAP-F, and Bonafide | ANN and RF |

[a]This author did not apply different algorithms and compared between them, and therefore not identifying one has the best candidate

Table 3.6: Port Scanning and ML Identified Works Summary

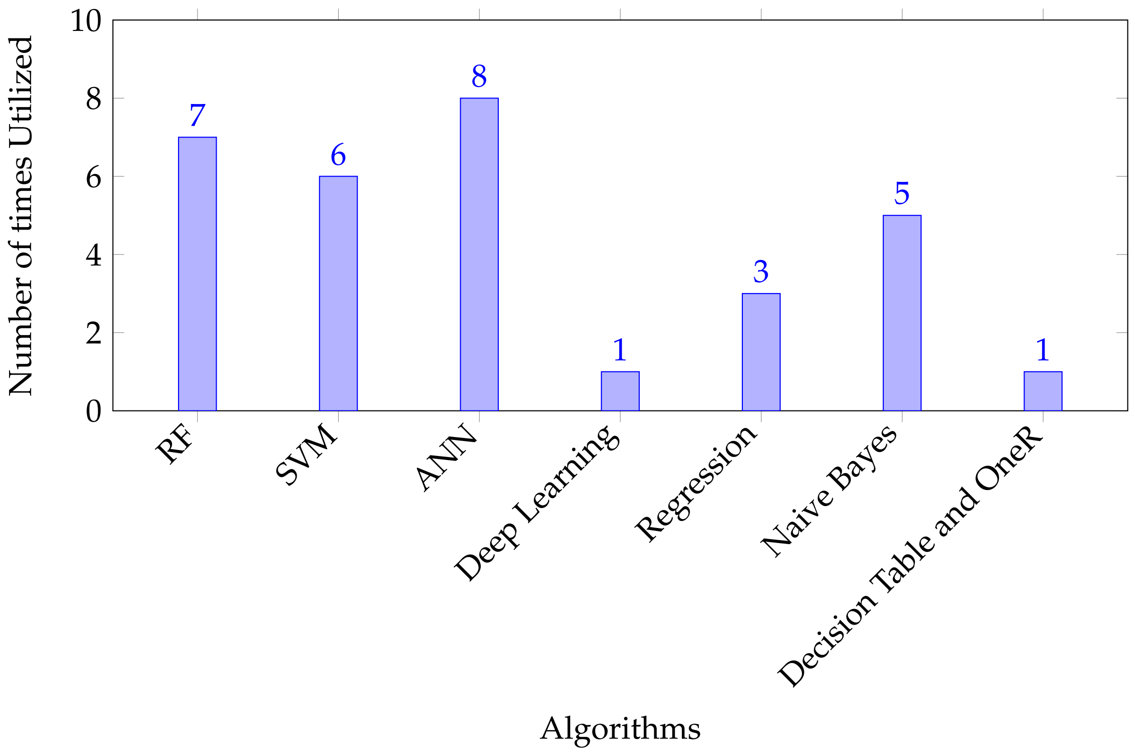


Figure 3.5: Algorithms used by the Authors

In conclusion, there seems to be a clear trend. Note that in chart 3.5 we took into consideration the other 15 papers contained in the work [Pittman, 2023]. We can see a clear trend in the algorithms most adopted. We also have to consider the algorithms elected as the best, which is mentioned in Table 3.6. From this information, we can clearly state that the most popular dataset is the CICIDS2017 accompanied by the ANN and RF algorithms.

### 3.2.3 Additional Port scan Literature

The work previously conducted was highly targeted on the domain of the intersection between port scanning, IDSs and ML. These approaches were more concerned with the characteristics of packets associated with network flows. However, as we have mentioned in Section 2.3.3, it would be of interest to take into account the analysis of the flows themselves.

Some research was done in this domain, and some papers were deemed of interest. These are not focused on the realm of ML but rather on the development of solutions that can discern patterns that identify threats that, for example, possess a slow probing approach or a distributed offensive strategy. Table 3.7 gives a brief overview of the selected papers.

| Author | Year | Cited | Topic |
|---|---|---|---|
| [Gates] | 2009 | 49 | Coordinated/Distributed Scan detection |
| [Dabbagh et al., 2011] | 2011 | 18 | Slow detection of port scans |
| [Tatsch, 2017] | 2017 | 4 | Vertical Scan detection |

Table 3.7: Additional Port scan Identified Works Summary

**Identified work Synthesis**

The work [Gates] involved the development of a solution that, from identified vertical scans, can then make the association to determine if they are organized. In the domain of our work, our strategy encompasses the use of machine learning. However, these authors' strategy focuses on adversary modelling, inspired by insights into agents' incentives and efficiency criteria. For each adversary, a footprint is generated considering the number of targeted ports, the count of targeted IP addresses, the algorithm used for the selection of IP addresses, and the use of camouflage, if any, to obscure the actual target. This footprint will be used to detect coordinated scans.

In the paper [Dabbagh et al., 2011] the objective is to address port scans that attempt to circumvent detection by employing a slow network reconnaissance. In their solution, they take into account the scan characteristics, as those mentioned in Section 2.3.2. From this, they classify IPs as legitimate, suspicious, or as a scanner. If an IP is classified as suspicious multiple times in a given interval it is then determined that a slow port scan is taking place.

Finally, the work [Tatsch, 2017] was again focused on the analysis of the characteristics of packets in flows. Based on the number of connections in a determined time frame to a single or multiple targets, scans could be inferred.

This analysis serves the purpose of demonstrating that, even though ML is a great solution to the problem, there are clear nuances that emerge from these other papers that tell us that to develop a more complete solution, our final proposal would benefit from taking into account other measures of detection outside of ML.

## 3.3 Unifying eBPF-IDS and Port Scan Detection

From our comprehensive review of the literature, discernible patterns emerge. In the realm of eBPF-IDS solutions, the primary algorithms in prevalence are DT, followed by RF. Concurrently, the dataset of choice for these solutions is predominantly aligned with CICIDS2017. When it comes to port scanning, the prevalent algorithms include ANN, with RF as a noteworthy contender. Here too, the favored dataset is consistently CICIDS2017. It becomes evident, by intersecting these two domains, that the RF algorithm, coupled with the CICIDS2017 dataset, emerges as the most suitable for the issue at hand.

### 3.3.1 Identified Issues

At first glance, the conclusion formulated presents itself as being suitable for the practical proof of concept we wish to develop. However, this is not the case, we must consider the limitations that arise from the already-established decisions.

When considering the CICIDS2017 dataset an imidate issue is identified, this dataset represents a bidirectional flow of traffic in each row. However, we have already established that our solution will be host-based, and considering that XDP, as mentioned in Section 2.1.5, can only read incoming traffic, means that the traffic flows will be unidirectional. Some of the features of this dataset are differentiated between forward and backwards traffic, and this could be used for our goal, however, it does not seem appropriate to use a dataset designed bidirectionally to be used unidirectionally. But this is not the main drawback, this dataset, even though it possesses data concerning port scanning, it does not make the distinction between each type of port scan, which is something that we wish to make possible in the final proposed practical solution.

### 3.3.2 Criteria-Driven Dataset Search

Having in mind the mentioned constraints, a dive into the literature for a dataset that satisfies our requirements was conducted. The dataset we seek to find must align with the following:

- Traffic flow be Unidirectional;
- Port scan as an Attack vector;
- Distinguish each port scan attack;
- Have it access open;

Our research did not follow a specific keyword search strategy. Instead, our emphasis was on scrutinizing papers that conducted reviews or surveys of datasets for IDSs. Although several documents listed datasets, they often lacked the specification listing of specific requirements we had defined, which made this investigation difficult. However, one paper stood out from the rest: [Ring et al., 2019]. Examining the identified datasets, one was precisely aligned with our requirements. This dataset is the CIDDS-002 dataset. A further study of this dataset will be conducted in the development phase.

## 3.4 Chapter Wrap-Up

The research conducted, concerning eBPF-IDS solutions, revealed that the issue at hand is a recent area of study and exploration. From the gathered information, it was possible to conclude that researchers follow one of three implementation strategies, fully in kernel, kernel combined with userspace and offloaded to hardware. From these, the one that presents itself as the most used is the fully in kernel. The reason for this stems from the fact that solutions that utilized the Userspace, were aimed to use eBPF for an initial computation of the packets. The algorithms used by these authors were also designed to work within the Userspace domain. Offloaded solutions require specific hardware, and this imposes limitations on their adoption, this being evident in the number of researchers that adopted this solution. Within fully in-kernel solutions, ML algorithms are commonly used, aligning with our objectives. Concerning these algorithms, DT followed by RF presents itself as the most favourable and the CICIDS2017 as the most adopted Dataset.

In the section that followed, port scan detection approaches that considered the use of ML in their solutions were the topic of research. In this research, a systematic review of the topic was found, which was the main source of information on the topic. From it and the other identified papers, it was concluded that ANN followed by RF are the most suitable algorithms, having the CICIDS2017 as the dataset most used.

In the culmination of both research endeavours, an attempt was made to intersect the domains. It became evident that the RF algorithm and the CICIDS2017 dataset were common elements in both areas. However, further investigation of the dataset allowed us to determine that it was not aligned with our already established decisions. Further research was conducted to try and find a dataset that aligned with our requirements. From this research, the CIDDS-002 dataset was chosen.

# Chapter 4

# Development and Implementation

This chapter encompasses the practical development and implementation of the eBPF-IDS solution tailored for port scan detection through ML. It was important to first analyse and consider the most important aspects of the proof of concept before moving on to the more detailed aspects of it. Having this in mind, the initial development encompassed the parsing of packets, the study of the dataset and subsequent ML analysis and finally the implementation of the ML algorithm in eBPF. In the initial stage, we did not attempt to distinguish between each type of scan but instead focused on implementing an ML algorithm that can discern between normal traffic and port scans. After having this well established we updated the solution for it to be able to make that distinction.

Let us first consider all decisions that have been taken until this point, and summarize it, as this will help in the details that will follow. In the domain of eBPF, we will use XDP as the framework to monitor network traffic. Given the various modes we have elected offloading as the most promising, however, given that this solution requires specific hardware, in the initial development we employed employ the generic/skb mode, refer to Section 2.1.5. In the realm of IDS, we have elected the monitoring strategy as host-based. Because we are trying to monitor an attack and stop it as early as possible, host-based is the one that makes the most sense. Because our strategy must employ an ML algorithm like the ones entailed in Section 2.2.4, anomaly-based detection is the most appropriate. Given these characteristics, our IDS will work in real-time using network data, the response can be passive or active. Regarding the attack vector, the port scan was the one that was selected. The CIDDS-002 dataset and the RF algorithm are the ones elected as the most suitable for our goal.

Note that BCC was the elected eBPF framework to develop our eBPF program, refer to Section 2.1.3 for clarification. All user space code was developed in Python.

The overall strategy established encompasses parsing packets that are then classified into a flow table. Once a flow is created or updated, it is passed through the classification algorithm which will classify it as normal or as a port scan. If the flow is deemed malicious it is inserted into the port scan table, which aggregates various flows classified as malicious. Once a threshold is hit a port scan is inferred and an alert is generated into user space. Refer to figure 4.1.

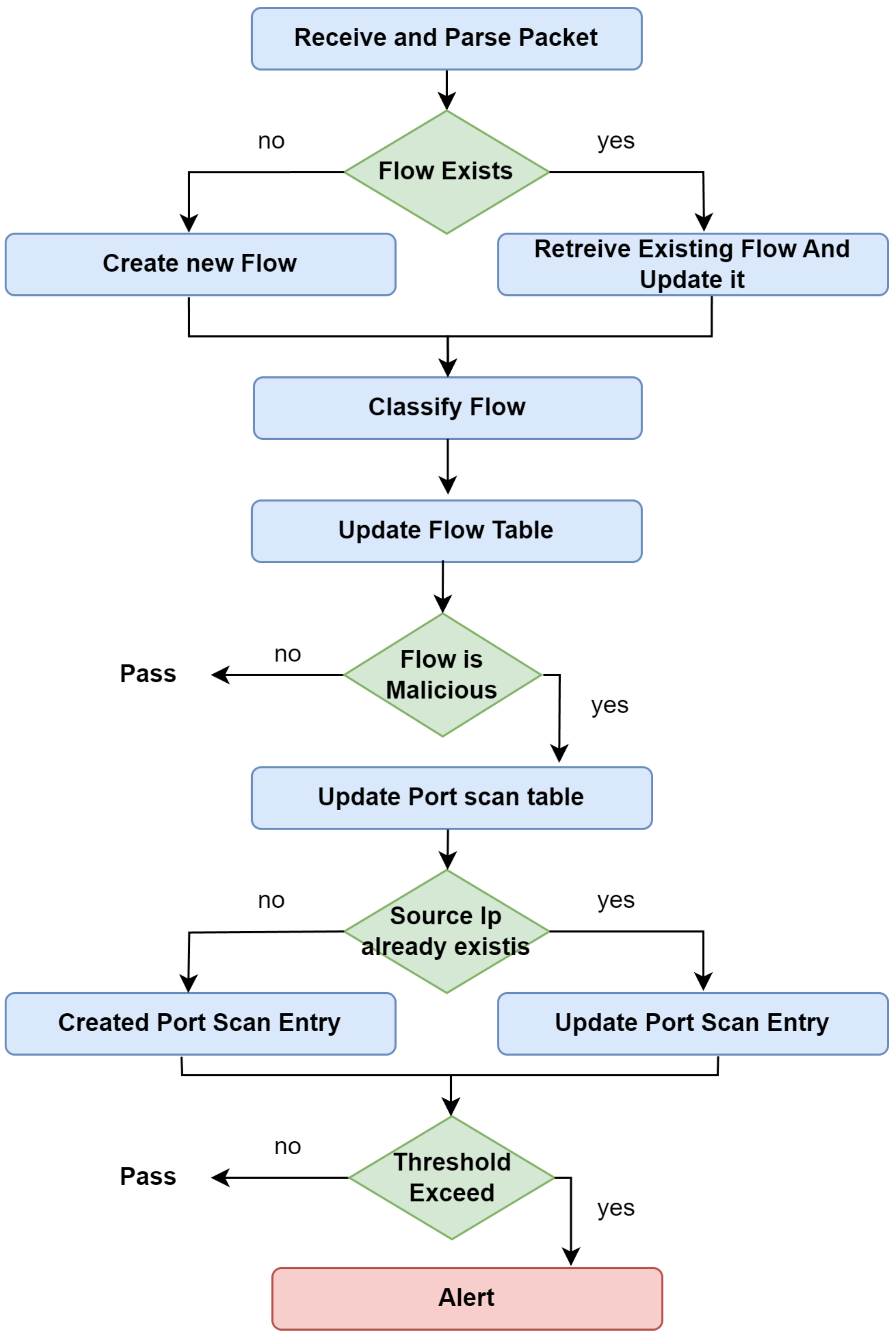


Figure 4.1: Overall solution strategy

Section 4.1 evaluates and processes the chosen dataset and ML algorithm. Section 4.2, explains how packets are handled. The following Section, 4.3, explains how ML was integrated into eBPF. Section 4.4 explains the port scan detection logic. Section 4.5 dives into how alerts are generated. Section 4.6 details all work associated with the offloaded component. Finally, Section 4.7 summarizes the chapter.

## 4.1 Dataset and ML Analysis

As we have identified in Chapter 3 Section 3.3.2 the dataset CIDDS-002 is the one that best aligns with our goals. From the study carried out Random Forest (RF) is the algorithm that we have selected. As we have identified in Chapter 2 Section 2.2.4 the workflow of supervised ML algorithms is iterative. Firstly, we will examine the elected dataset and make the necessary processing for it to fit our goal. From there, train the model with the dataset and determine the best parameters for it.

### 4.1.1 CIDDS-002 Analysis and Processing

The CIDDS-002 is a labelled dataset that contains flow data from normal traffic and port scans. These flow data are unidirectional and contain various attributes. According to the dataset documentation and technical report [Ring et al., 2017, to appear], there are 14 features, however, by analysing the dataset it is clear that there are 16. As per our analysis, this dataset contains the following port scans: SYN, ACK, FIN, UDP and Ping scans. Table 4.1 depicts the attributes of the dataset according to the documentation and the review undertaken.

| Nr | Name | Description |
|---|---|---|
| 1 | Src IP | Source IP Address |
| 2 | Src Port | Source Port |
| 3 | Dest IP | Destination IP Address |
| 4 | Dest Port | Destination Port |
| 5 | Proto | Transport Protocol (e.g., ICMP, TCP, or UDP) |
| 6 | Date first seen | Start time flow first seen |
| 7 | Duration | Duration of the flow |
| 8 | Bytes | Number of transmitted bytes |
| 9 | Packets | Number of transmitted packets |
| 10 | Flags | OR concatenation of all TCP Flags |
| 11 | Class | Class label (normal, attacker, victim, suspicious, or unknown) |
| 12 | AttackType | Type of Attack (portScan, dos, bruteForce, —) |
| 13 | AttackID | Unique attack id. All flows which belong to the same attack carry the same attack id. |
| 14 | AttackDescription | Provides additional information about the set attack parameters (e.g., the number of attempted password guesses for SSH-Brute-Force attacks) |
| - | Flows | It may refer to how many times that flow reappeared |
| - | ToS | Given its name, one can infer that it refers to "Type of Service". |

Table 4.1: CIDDS-002 Attributes

For our goals, there are clear attributes that we can discard immediately, as they do not align. These attributes are the "Date first seen", "Src IP" and "Dest IP", as these are related to the test bed used and cannot be transposed to other environments. Another intuitive modification is on the attribute "Class". In this domain, there can be five labels: normal, attacker, victim, suspicious, or unknown. From our analysis only three are present: normal, attacker and victim. As explained in previous chapters, our solution is host-based, and due to the limitations of XDP, it can only analyse incoming traffic, making the victim label pointless. For this reason, all entries with this label were removed. From there we analysed the distribution of this label, which is depicted in Figure 4.2.

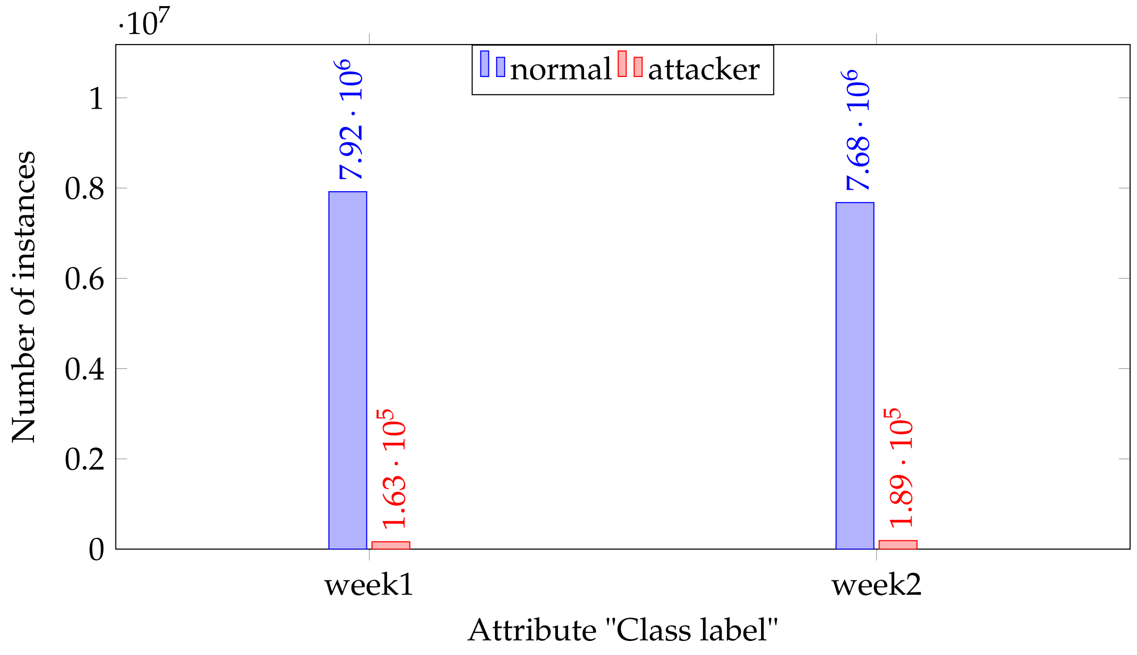


Figure 4.2: CIDDS-002 Class Distribution Week 1 and Week 2

From this, we can see an unbalanced distribution. The "normal" label is about 98% of the data. This may lead the model to become biased toward the majority class. This is because, during the tree-building process, each tree in the forest is likely to be exposed to more instances from the majority class. To overcome this, the "normal" traffic needs to be under-sampled to match the "attacker" traffic. The data is split between two datasets, one for each week. Given the massive under-sampling that will take place, concatenating both datasets will be done to increase the size of the data. Under-sampling will only take place as the final step, however for the remaining examination we will consider the concatenation of both datasets, week1 and week2.

Our focus shifted to analyzing the attributes of the attackers, revealing discrepancies. Initially, we examined the attack flows based on the protocol. However, when delving into the breakdown of port scan types given by the "AttackDescription" attribute, we encountered a noteworthy issue. This challenge became more apparent as we sorted each port scan type according to its corresponding protocol: SYN, ACK, FIN for TCP; UDP for UDP; and Ping for Internet Control Message Protocol (ICMP). The distributions of these protocols are illustrated in Figure 4.3.

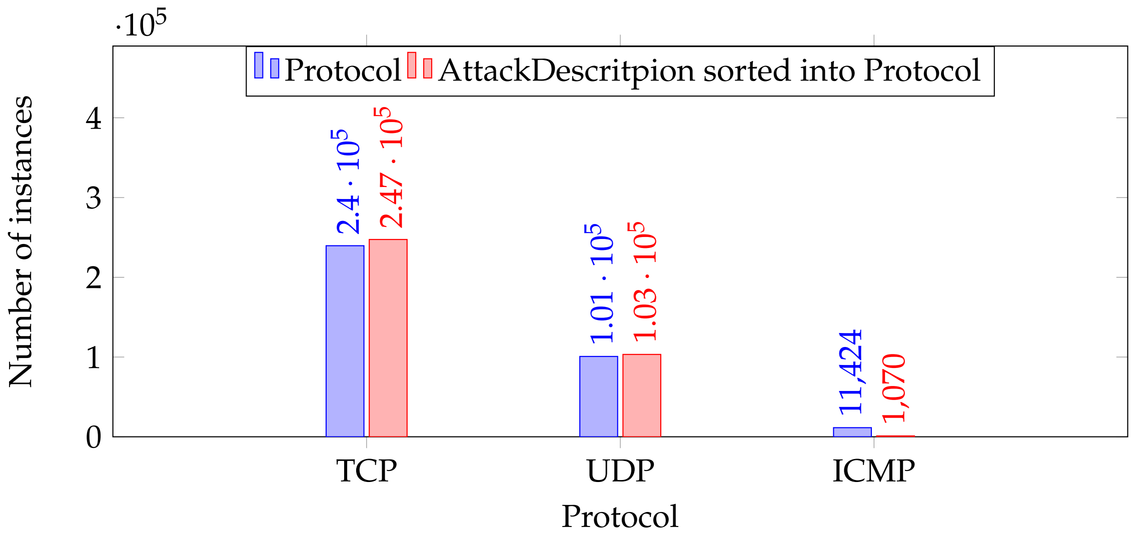


Figure 4.3: CIDDS-002 Attacker Protocol Distribution

The attack descriptions do not match the protocol. In the presence of this observation, we decided to examine these misidentified flows to see what the distribution of protocols where the "AttackDescription" did not match, how many of each, and to what attack they were associated. Figure 4.4 represents the distribution of attacks in which the description has been associated with the wrong protocol.

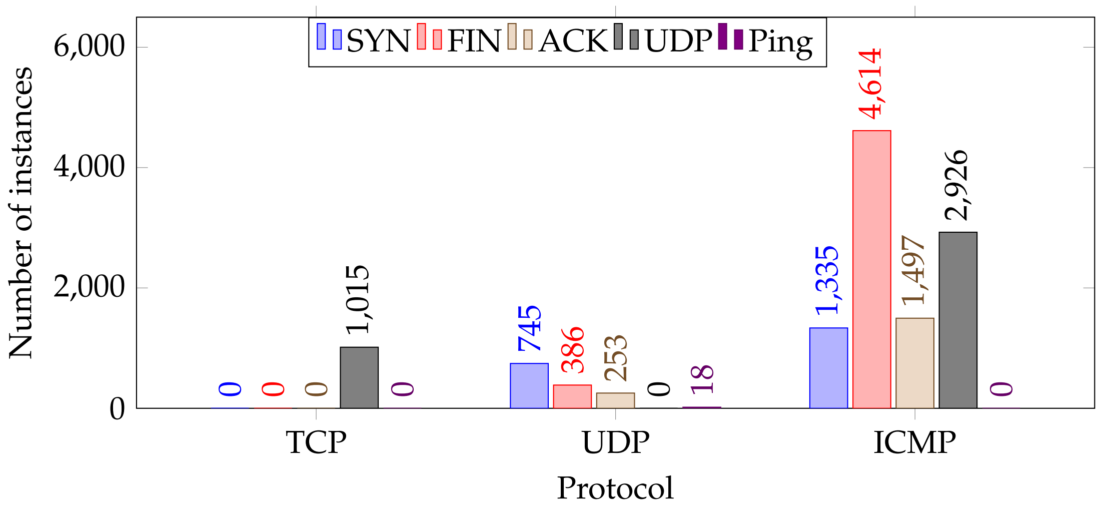


Figure 4.4: CIDDS-002 Misidentified Attacks Distribution

It is not clear why these classifications exist. When examining ICMP, there are more misidentified than correctly identified. ICMP packets do occur when performing those attacks, however, they are sent by the victim as a response and not by the attacker. As noted in Nmap Documentation related to Host Discovery, [Nmap Reference Guide - Host Discovery], Ping scans can be accompanied by other protocol scans, this could be the reason behind those classifications. We can also argue whether ping scans fall within the domain of port scanning, as they are not intended to examine ports but instead hosts. Given this reason and the ambiguity of the results observed, for our development, we decided to remove all ICMP flows. Additionally, the other TCP and UDP classifications that do not align were also discarded. This leaves us with only TCP and UDP flows in which the attack descriptions match the protocol.

It is important to note that during this protocol examination, a fourth protocol was identified, Internet Group Management Protocol (IGMP). However, this protocol is only used in normal flows. Given this distribution, we also decided to discard it as it would not provide any additional benefit.

Examination of the dataset continued and the unidentified attributes "Flows" and "ToS" were addressed. The "Flow" label always had the value 1, and "ToS" possessed the values 0, 192 and 16, where 0 is 99% of the values. Given the lack of distribution of these values and that we do not possess concrete descriptions of what they represent, we also discard these attributes.

From the remaining attributes, "Dest Port" and "Src Port" were evaluated to see if they fit our goal. Because we don't want our algorithm to be biased against only the ports scanned on the dataset, we ultimately decided not to include them.

This leaves "Proto", "Bytes", "Flags", "Duration" and "Packets" as the features used to train our model. Attributes from 11 to 14 were added later as labels for each flow. Because in this initial analysis, we only want to make the distinction between normal and attacker traffic, we elected the attribute "Class" as our target.

For our elected features, some additional processing was required. The "Proto" string values were converted to their assigned corresponding Internet protocol numbers 6 and 17 for TCP and UDP respectively. "Bytes" possessed some values with "M" at the end probably referring to Megabytes, so all values were converted to bytes. "Flags" were depicted as a string, placing "." if the flag was off, and the respective flag letter if it was on. Note that they follow this order: U, A, P, R, S and F. Given that each position in the string corresponds to a unique flag, we decided to represent them in a binary format, placing 1 for "." and 2 for a letter. "Duration" values were all converted from seconds to nanoseconds.

This is all that is needed for the binary classification of flows between normal and a port scan. However, to discern between scans, some alterations are needed. We must consider the distribution of each scan type. Scans with lower instances were oversampled to match the others. The normal class, instead of under-sampled to the same size of the aggregation of all port scans, is under-sampled to match the size of a single scan type.

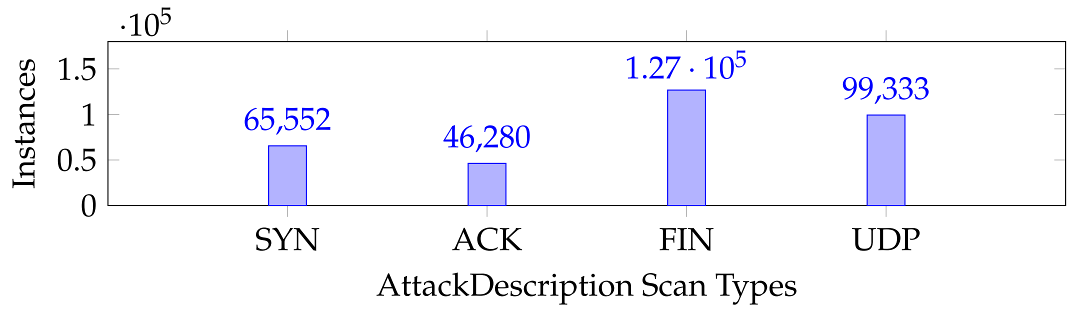


Figure 4.5: CIDDS-002 Scan Types Distribution

### 4.1.2 Random Forest Model Development

As decided, the chosen algorithm is RF. To train the model we employed Python and the *scikit-learn* library [Pedregosa et al., 2011]. For the initial development, "Class" was the target label, making the distinction between normal traffic and port scans. It was then changed to "AttackDescription", to allow the subcategorization of each type of scan. The features are "Proto", "Bytes", "Flags", "Duration", and "Packets". The distribution of training and testing was chosen to be an 80:20 ratio.

As further detailed in the thesis, eBPF has limitations that impact the way IDS can be built. One in particular is related to the number of instructions. For this reason, we have to carefully evaluate the model for the maximum number of trees in the forest and the maximum depth that each tree can have. With this in mind, the model's performance was first examined concerning the maximum number of trees. We capped the `max_leaf_nodes` to 1000 just to have a limit baseline for each tree to guarantee some level of coherence throughout each iteration as there is no maximum depth yet defined. Figure 4.6 shows the results of the various performance metrics when varying the number of trees.

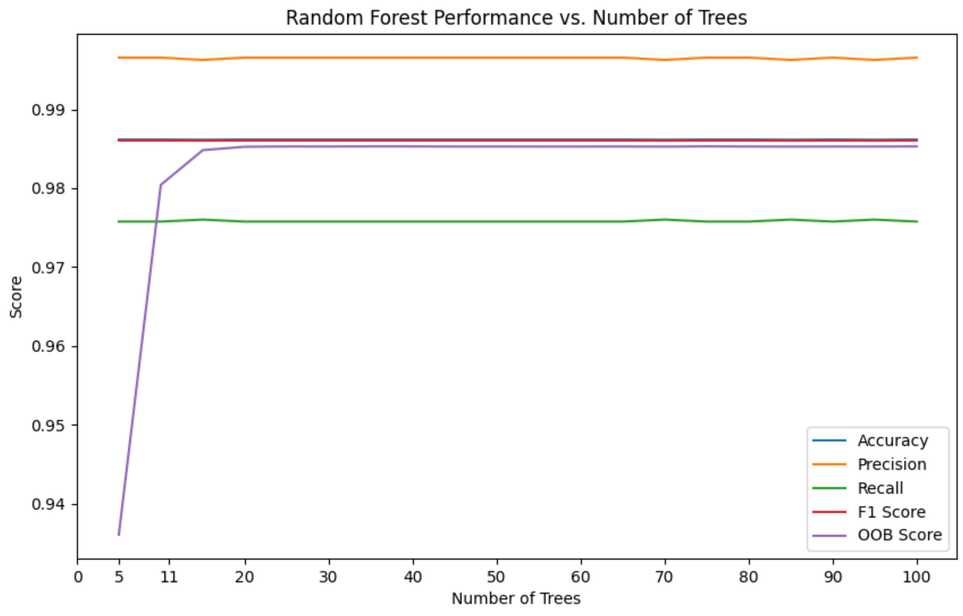


Figure 4.6: Performance Variation with number of trees for Binary Model

With these results, we determined that 11 would be an appropriate number for the number of trees to not risk overfitting. From this, we examined the variation of these performance metrics when the depth was changed. Figure 4.7 depicts the results of the various performance metrics when the number of trees is 11.

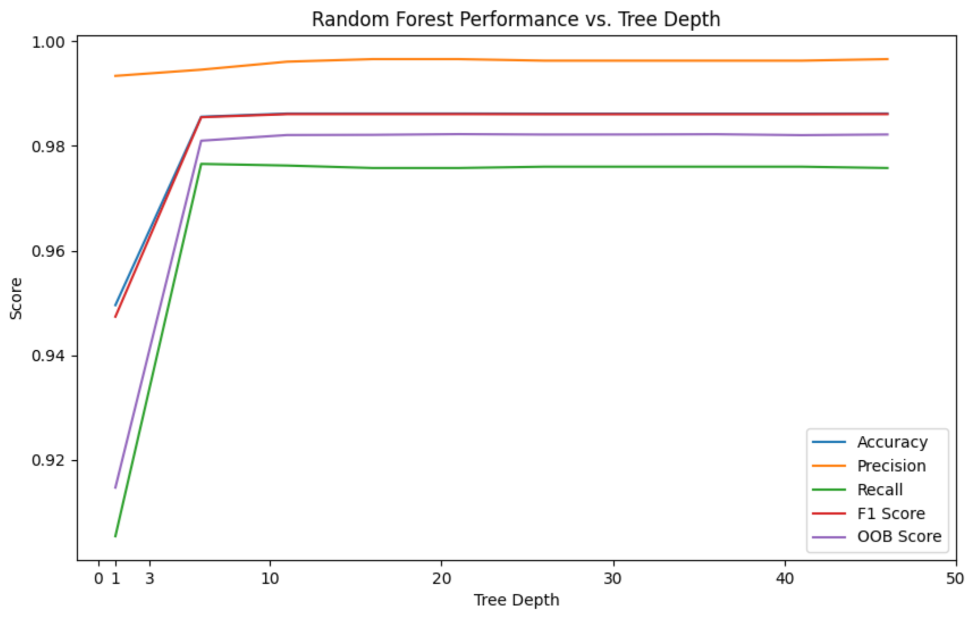


Figure 4.7: Performance Variation with the number of depth for Binary Model

From these results, the maximum depth of 3 was deemed reasonable, as seen in the results we risk overfitting the model. With these variables, training the model gives the confusion matrix in Figure 4.8; additionally, the feature importance is depicted in Figure 4.9.

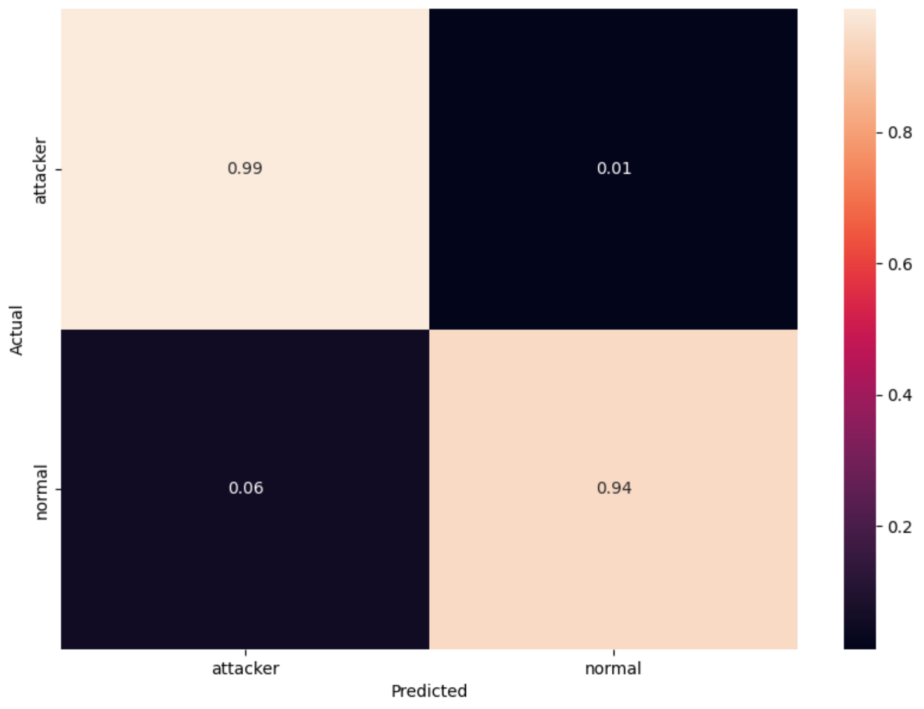


Figure 4.8: Confusion Matrix for Binary Model

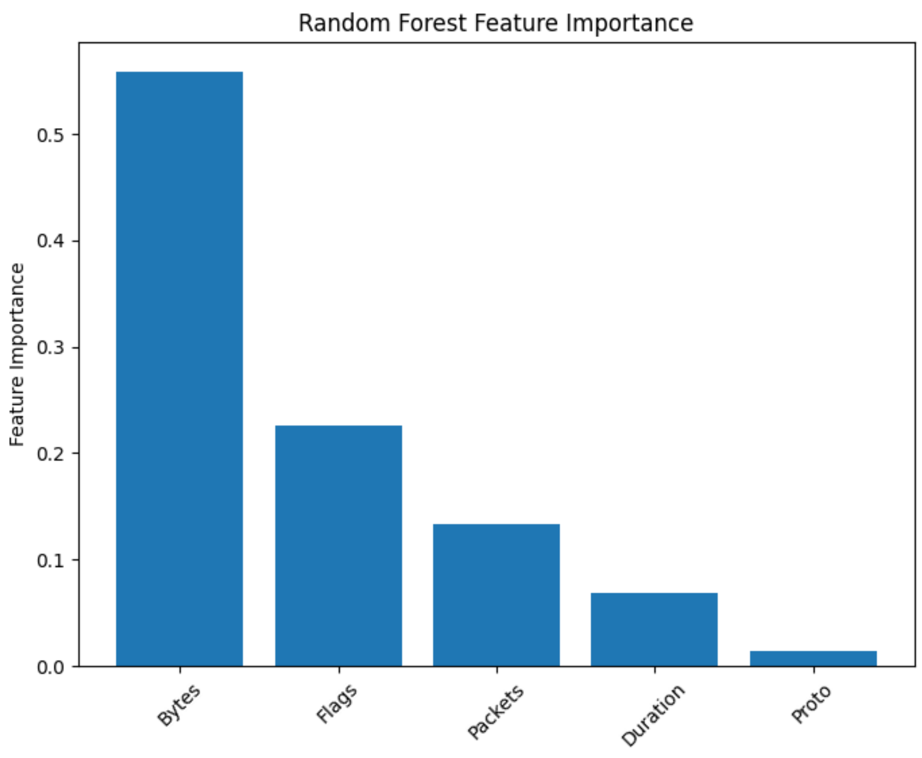


Figure 4.9: Feature Importance for Binary Model

This distribution is somewhat expected. The features "Bytes" possess the most distribution allowing to discern flows the best. "Flags" comes right after, however, in normal and attack flows they possess similar properties. "Packets" and "Duration" do contribute but again, this is even similar between both classes. Finally, "Proto" only varies between TCP and UDP, resulting in the least important.

As we moved on, to attempt to distinguish between each type of scan, similar tests were conducted. Measuring performance by changing the number of trees of the model and once a value is selected, observe variations with changes in depth. For metrics requiring average, the *macro* keyword was used.

When it came to the number of trees, the value of 11 was once again deemed appropriate. Like previously, higher values seem to lead to an overfitting of the model. The chosen depth was initially selected to be 6. However, it was later observed, in the deployment of the model, that as the depth increased, its capacity to distinguish between normal traffic and Ack scans, became harder. The depth was then decreased until this property was no longer present. Additionally, by examining the number of paths in the model for each label, some were much less represented including the Ack label. To address this, weights were attributed to each class. We ended up at the same value as in the binary model, 3. The metrics are depicted in Figures 4.10 and 4.11.

From these values, the confusion matrix and feature importance were again calculated. These are depicted in Figures 4.12 and 4.13 respectively. Compared to the previous model, the "Flags" attribute rises to the most important for obvious reasons. This attribute is the metric that allows us to distinguish between scans.

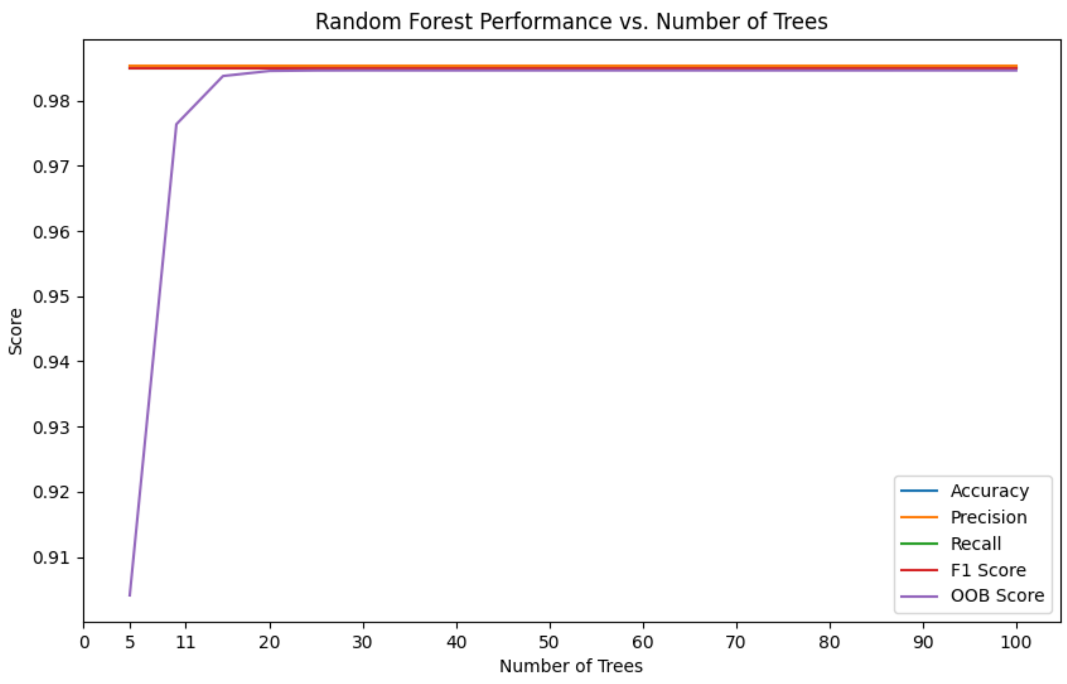


Figure 4.10: Number of trees for Multi-class Model

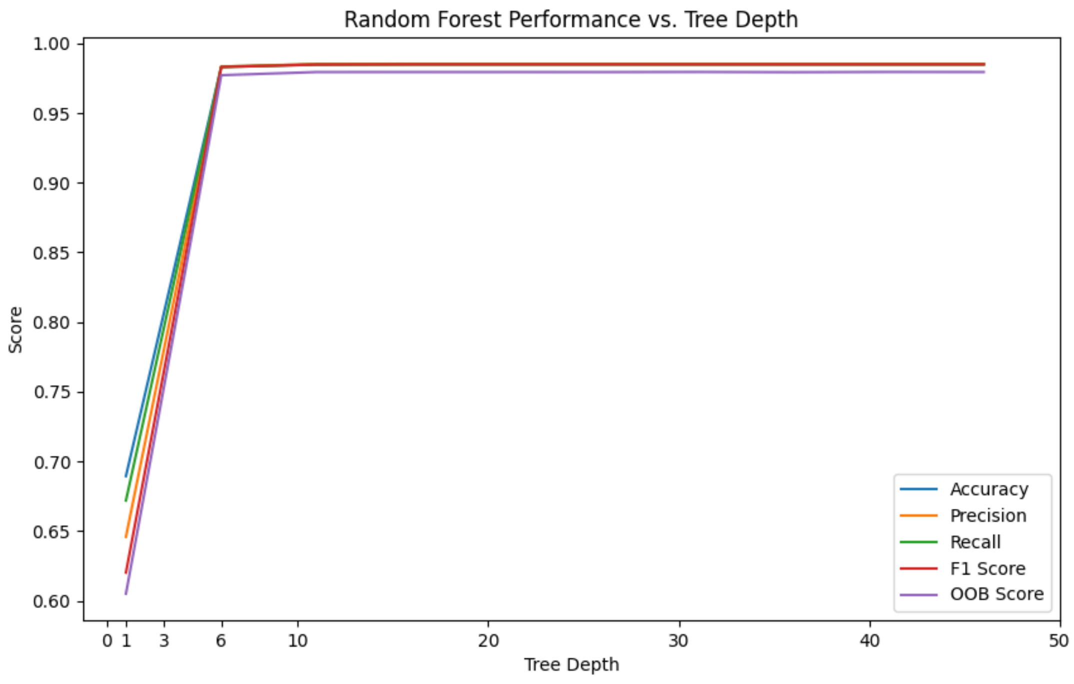


Figure 4.11: Number of depth for Multi-class Model

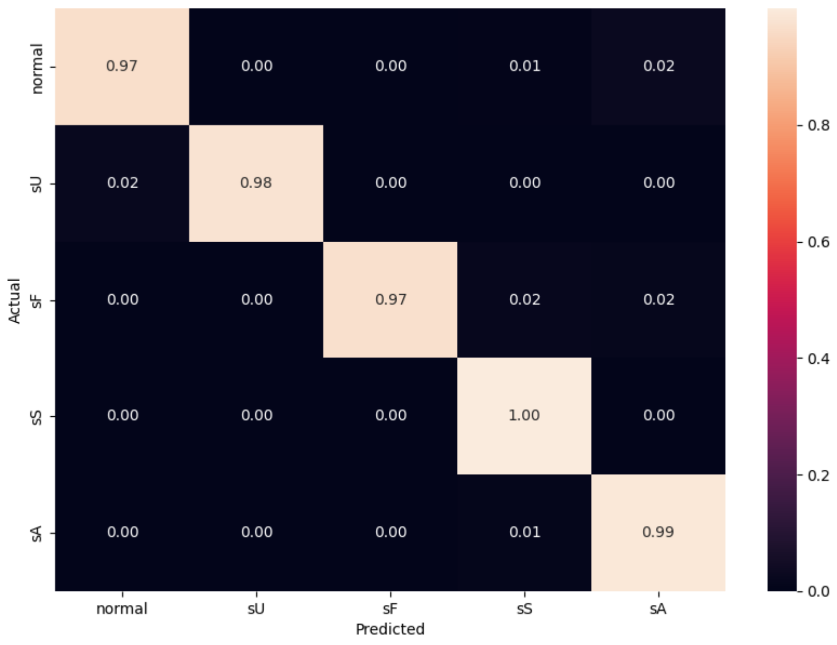


Figure 4.12: Confusion Matrix for Multi-class Model

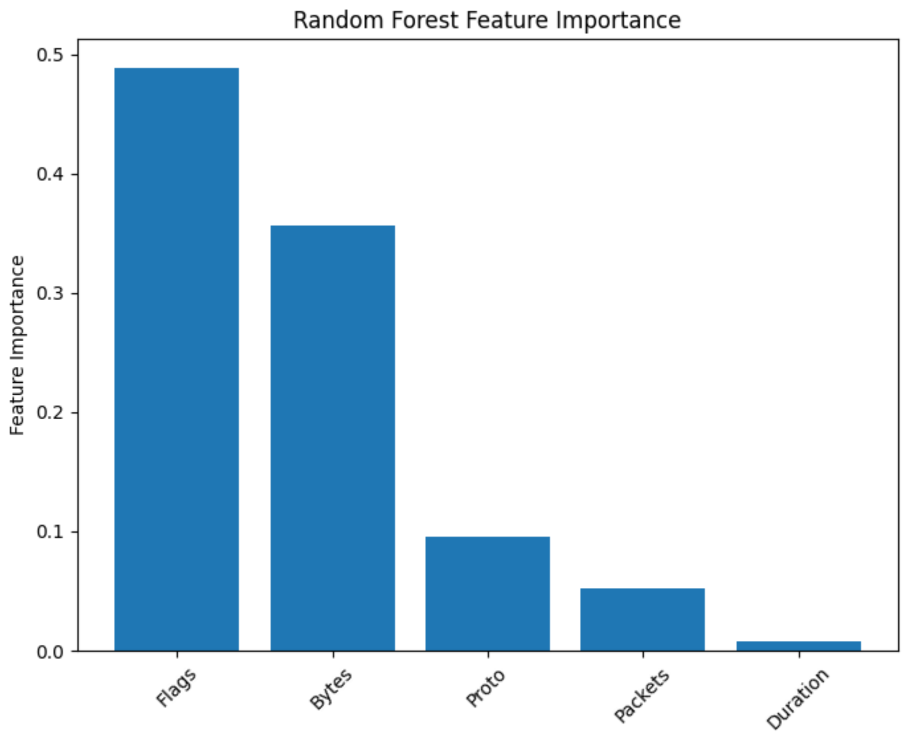


Figure 4.13: Feature Importance Multi-class Model

## 4.2 Packet parsing and processing

An XDP program is triggered at the moment a network packet enters the machine. The incoming metadata comes as a pointer to an `xdp_md` structure [Linux's uapi/linux/bpf.h, b]. This structure is referenced in the Listing 4.1.

```
struct xdp_md {
	__u32 data;
	__u32 data_end;
	__u32 data_meta;
	/* Below access go through struct xdp_rxq_info */
	__u32 ingress_ifindex; /* rxq->dev->ifindex */
	__u32 rx_queue_index; /* rxq->queue_index */
	__u32 egress_ifindex; /* txq->dev->ifindex */
};
```

Listing 4.1: xdp_md Structure

The two first fields indicate the memory location referring to the start and end, respectively. Although they state the `__u32` type, these are pointers. With these, we can parse the contents of the packet. Figure 4.14 depicts the layout of an IP packet, which has the elements of interest highlighted for our development. These elements correspond to the features selected for the ML model or are needed for subsequent parsing.

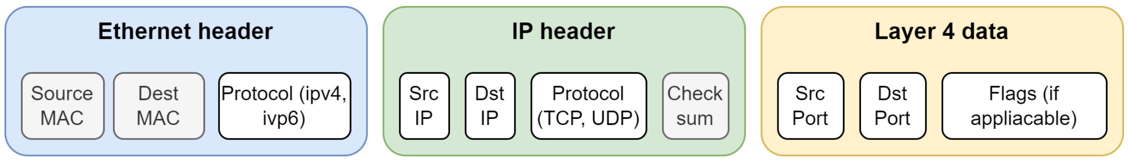


Figure 4.14: IP packet general layout [1]

As we parse through the packet, we must verify each memory access to accommodate the Verifier constraints. To do this, as each element is cast to its designated structure, an offset with the size of the current structure size is incremented. If the pointer to the start of data in addition to this offset exceeds the end of the data, it means that we are operating outside the memory boundaries.

Given the nature of our solution, our model requires the examination of certain features in a network flow to determine if it is normal activity or a port scan. For this, we must keep track of the flows of the network. As each packet arrives, a new flow is created or an existing one is updated. Our solution encompasses the use of a Least Recently Used (LRU) hash map. We chose the LRU variant as it would allow for the management of the flows, deleting the ones that have been inactive the longest, and making sure the number of flows cannot grow out of bounds. Each hash key comprises the source and destination IP and source and

[1]This layout was made taking into reference the IP packet layout presented in the book [Rice, 2023]

destination port, accompanied by the layer 4 protocol (e.g. TCP, UDP, etc.). Even though not used for training the IPs are essential to pinpoint the attack domain. The value associated with the key will possess the features, model classification prediction, and additional information either used to calculate the features or for metrics analysis. For better clarity, the structures of these elements are depicted in Listing 4.2:

```c
typedef struct flow_key {
    __u32 src_ip;
    __u32 dst_ip;
    __u16 src_port;
    __u16 dst_port;
    __u16 protocol;
}flow_key;

typedef struct flow_value {
    __u64 packet_counter;
    __u64 duration;
    __u64 transmited_bytes;
    __u64 flags[6];
    // end of features
    __u64 timestamp; // last timestamp (used to calculate flow duration)
    __u64 scan; // RF model prediction - 0 normal, [1-4] scan
    __u64 scan_counter; // times it has been classified as a scan
    __u64 suspicious; // 0 normal, 1 scan-normal, 2 trigger
}flow_value;
```

Listing 4.2: Flow Key and Flow Value Structures

## 4.3 Implementing ML Models in eBPF for IDS

In the delineation of the thesis objectives, the incorporation of an ML algorithm into the IDS solution was established. Nevertheless, in the quest for the most suitable solution, it is imperative to assess the algorithm's efficiency and consider the constraints imposed by using eBPF technology within the IDS. As mentioned, the eBPF deployment flow passes through an element known as the Verifier. This Verifier imposes limitations on the code that is being deployed. In the context of the algorithm to be chosen, the metrics which we must take into account are the program limit size and unbounded loops. The size of an eBPF program was limited to a maximum of 4096 instructions by the Verifier, however, that limit has grown to 1 million. In case this limit is reached, there are ways to avoid this issue, by using tail calls to other eBPF programs [Rice, 2023]. According to [Miano et al., 2018], which analyzed the limits and possible circumventions of eBPF limitations, this approach of tailing does not add much overhead, however, we should still attempt to keep the complexity of the algorithm low to improve the usage of resources. Another limitation is that unbounded loops are not allowed, in the presence of an algorithm that iterates until something is reached, it needs to be changed to a bounded loop solution.

From the findings presented in [Miano et al., 2018], one stood out as a possible avenue for investigation. The authors evaluated the performance of eBPF and concluded that a solution using hard-coded parameters can have a big positive impact on performance instead of direct memory accesses. With this in mind, our ML implementation will follow two approaches. One possesses the model parameters loaded in maps and the other with them hardcoded. We will then evaluate each one.

Given that our model was developed in Python and eBPF requires it to be in C, we have utilised the Python library *emlearn* [Nordby et al., 2019] to generate the Python model to C to be compiled. The *emlearn* library is specifically crafted to extend machine learning capabilities to microcontrollers and embedded systems. It ensures compatibility by allowing Python models to seamlessly transition to C code, which can be compiled with a C99 compiler. Notably, it eliminates the need for extra libraries and dynamic allocations. The entire code is encapsulated in a single header file, enhancing simplicity and making it well-suited for integration with eBPF. However, the generated code requires alterations to be accepted by the Verifier, this will be addressed in the upcoming sections. With this solution, we can attempt to have the model hard-coded in the program.

### 4.3.1 Random Forest Model in eBPF via Maps

When trying to implement ML in eBPF the most straightforward solution is to retrieve the model values from the *scikit-learn* model, and load them into an eBPF map. From the kernel side, write the RF classification/prediction algorithm that accesses these values to make decisions.

We can build a program that, from the generated RF model, retrieves the nodes and roots and parses them to a structure that can be loaded into the map. However, *emlearn*, already does this. Given that we are using it for the compiled version, we decided to use this feature in this stage for convenience. Note that because our user space program is in Python, a slight modification to the C structures created needs to be done to be loaded via our program.

We require two maps for loading elements from two distinct domains: the tree's roots and forest nodes. Both maps are implemented as array maps. The array map to store forest nodes uses an integer as the key, representing an index, and the corresponding value is the structure of the nodes. This structure encompasses details such as the associated feature, its values, and the indices of the nodes to its right and left. The structure is outlined in Listing 4.3 for reference.

```c
typedef struct tree_node {
    int feature;
    int value;
    int left;
    int right;
} tree_node;
```

Listing 4.3: Tree Node Structure

It is important to note that the item "value" in this structure generated from *em-learn* comes in float. Given that our features are all `Int64`, the decimal values are 0 or 5. Because eBPF can´t do pointer arithmetic we shifted these values by one to the left, effectively multiplying each by 10.

The array map for storing tree roots utilizes an integer as the array key, representing the tree number, and the corresponding value is also an integer denoting the index where it is stored in the array map of forest nodes. Refer to Figure 4.15 for a visual representation of this relationship between maps, providing further clarification.

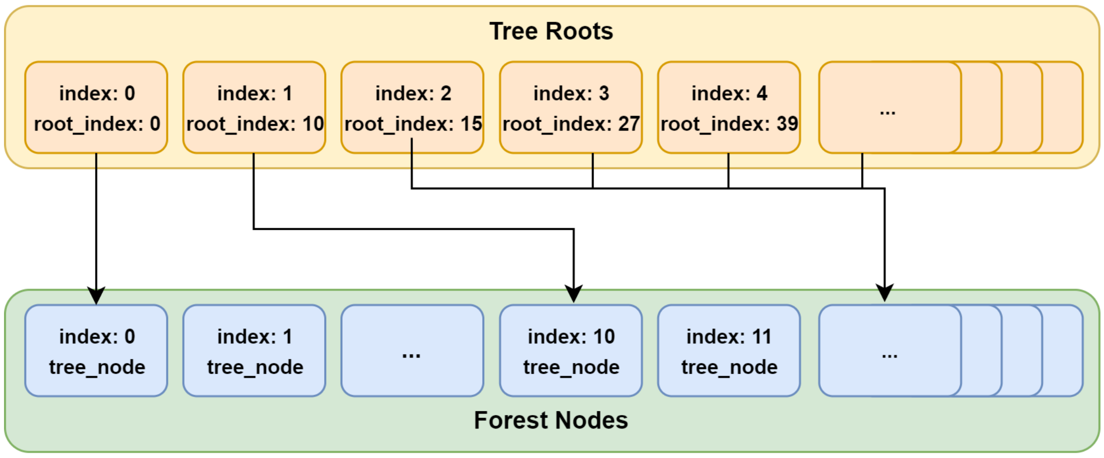


Figure 4.15: Tree roots and Forest Nodes Storage

Having established how we retrieve and load the RF model values, we can move on to developing the prediction algorithm. Given the model is RF the algorithm was developed accordingly. Starting at each root, the node's value is compared to a certain feature, the outcome will decide if the next node to visit it at the left or right of the current node. Once the last node is reached, this will indicate the prediction of the current tree. The last node in a tree possesses its feature value equal to -1, given that -1 is outside the boundaries of the features array.

It is important to note, however, some modifications that were required for the acceptance by the Verifier. Because the node values were shifted, we are required to multiply each feature by 10. To accommodate the bounded-loop criteria, we chose not to iterate through each tree until reaching the last node. Instead, we perform a loop limited by the maximum depth that each tree can attain. This ensures that we can reach the last element. If the element is found before the maximum depth is reached, we exit the loop. To retrieve each tree root, the Verifier prohibited using the same value for both iterating through the loop and retrieving the root via the map. So an auxiliary variable *j* was created. When accessing the Maps, an additional condition was necessary to verify if something was retrieved. When accessing arrays, such as *votes* or *features*, a validation check was implemented to ensure that our access remained within the limits.

Ultimately, all predictions are considered, and the one that has garnered the highest number of votes is selected as the final choice. The algorithm 1 depicts the pseudocode of our solution for further clarification. The workflow of this particular solution is represented in Figure 4.16 to facilitate its comprehension.

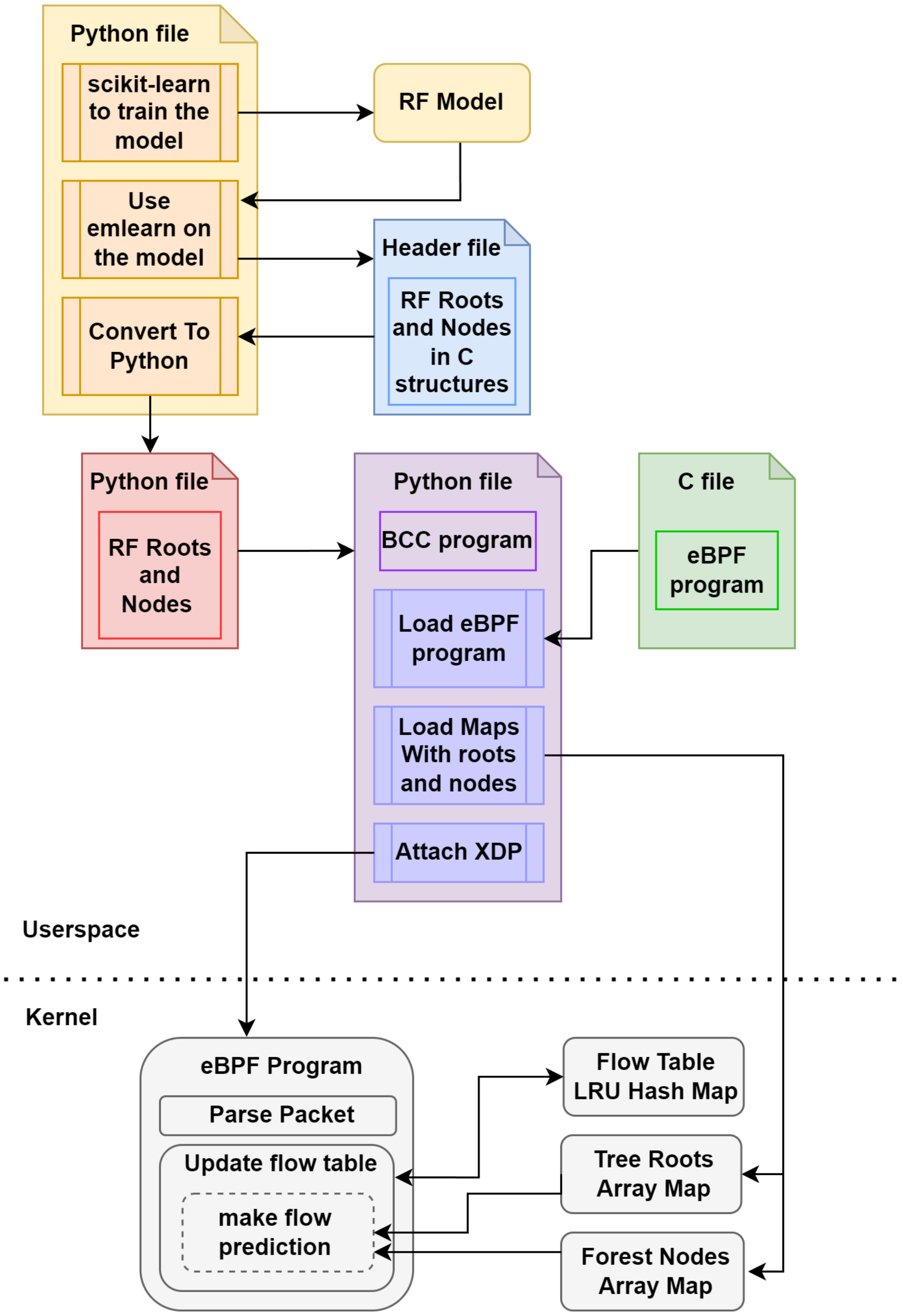


Figure 4.16: RF Model via Maps workflow

**Algorithm 1** RF Model Prediction via Maps

```
1: function PREDICT_MAP(duration, protocol, packet_counter, transmitted_bytes, current_flags)
2:
3:     $features \leftarrow [duration \times 10, protocol \times 10, packet\_counter \times 10, transmitted\_bytes \times 10, current\_flags \times 10]$
4:     $votes \leftarrow [0, 0, 0, 0, 0]$
5:     $j \leftarrow 0$
6:
7:     for $i \leftarrow 0$ to $N\_TREES$ do
8:         $current\_root \leftarrow tree\_roots.lookup(j)$
9:         $j \leftarrow j + 1$
10:        if $current\_root$ then
11:            $current\_node \leftarrow *current\_root$
12:            for $d \leftarrow 0$ to $MAX\_TREE\_DEPTH$ do
13:                $node \leftarrow tree\_nodes.lookup(current\_node)$
14:                if $node$ then
15:                    if $node.feature <$ len$(features)$ and $node.feature \geq 0$ then
16:                        if $features[node.feature] < node.value$ then
17:                            $current\_node \leftarrow node.left$
18:                        else
19:                            $current\_node \leftarrow node.right$
20:                        end if
21:                    else
22:                        if $node.value <$ len$(votes)$ and $node.value \geq 0$ then
23:                            $votes[node.value] += 1$
24:                            break
25:                        end if
26:                    end if
27:                end if
28:            end for
29:        end if
30:    end for
31:
32:    $most\_voted\_class \leftarrow -1$
33:    $most\_voted\_votes \leftarrow 0$
34:    for $i \leftarrow 0$ to len$(votes)$ do
35:        if $votes[i] > most\_voted\_votes$ then
36:            $most\_voted\_class \leftarrow i$
37:            $most\_voted\_votes \leftarrow votes[i]$
38:        end if
39:    end for
40:
41:    return $most\_voted\_class$
42: end function
```

### 4.3.2 Compilation of Random Forest Models in eBPF

Based on the outlined strategy, we proceeded with its development. As we mentioned we will employ *emlearn* to generate the model into C code. We could have followed the same strategy as the one with Maps, having the tree nodes and roots hardcoded in the program with arrays. However, the Verifier did not allow the previous algorithm, with the appropriate changes, to be used with arrays instead of maps. In the face of this, given that *emlearn* also generates each full unrolled tree with if and else statements we decided to employ them for our prediction. Below is an example of a tree generated by *emlearn*.

```
static inline int32_t rf_predict_tree_5(const float *features, int32_t
    features_length) {
          if (features[4] < 116166.0) {
              if (features[3] < 56.0) {
                  if (features[3] < 51.5)
                      return 1;
                  else
                      return 2;
              }else {
                  if (features[1] < 11.5)
                      return 3;
                  else
                      return 0;
              }
          } else {
              if (features[3] < 54.5) {
                  if (features[4] < 121111.5)
                      return 4;
                  else
                      return 0;
              } else
                  return 0;
          }
        }
```

Listing 4.4: Tree from Random Forest generated from emlearn

For this function to be accepted, it would need to shift every value to the left, effectively multiplying by 10. This is the same approach used with Maps. Additionally, the parameter "features" would require to be changed to `int`, and "features_length" could be removed. Furthermore, the type of function would also need to be changed to `int`. At the end of these trees, a function, calling each tree and establishing the most returned prediction, was also generated. To address Verifier issues related to pointer arithmetic, we opted for a streamlined solution. We consolidated the if-else conditions from each tree into a single function, as depicted in the pseudocode of algorithm 2.

Figure 4.17 is meant to depict the workflow of this strategy. Making it easier to compare to the one previously illustrated via Maps.

**Algorithm 2** RF Model Prediction Compiled

```
 1: function PREDICT_C(duration, protocol, packet_counter, transmitted_bytes,
    current_flags)
 2:     $features \leftarrow [duration \times 10, protocol \times 10, packet\_counter \times 10, transmitted\_bytes \times 10, current\_flags \times 10]$
 3:     $votes \leftarrow [0,0,0,0,0]$
 4:     if $features[4] < 1161660$ then
 5:         if $features[3] < 560$ then
 6:             if $features[3] < 515$ then
 7:                 $votes[1] += 1$
 8:             else
 9:                 $votes[2] += 1$
10:             end if
11:         else
12:             if $features[1] < 115$ then
13:                 $votes[3] += 1$
14:             else
15:                 $votes[0] += 1$
16:             end if
17:         end if
18:     else
19:         if $features[3] < 545$ then
20:             if $features[4] < 1211115$ then
21:                 $votes[4] += 1$
22:             else
23:                 $votes[0] += 1$
24:             end if
25:         else
26:             $votes[0] += 1$
27:         end if
28:     end if
          ▷ The rest of the trees follow the same format as the one between lines 4
    and 28. They were omitted for clarity
29:     $most\_voted\_class \leftarrow -1$
30:     $most\_voted\_votes \leftarrow 0$
31:     for $i \leftarrow 0$ to len($votes$) do
32:         if $votes[i] > most\_voted\_votes$ then
33:             $most\_voted\_class \leftarrow i$
34:             $most\_voted\_votes \leftarrow votes[i]$
35:         end if
36:     end for
37:     return most_voted_class
38: end function
```

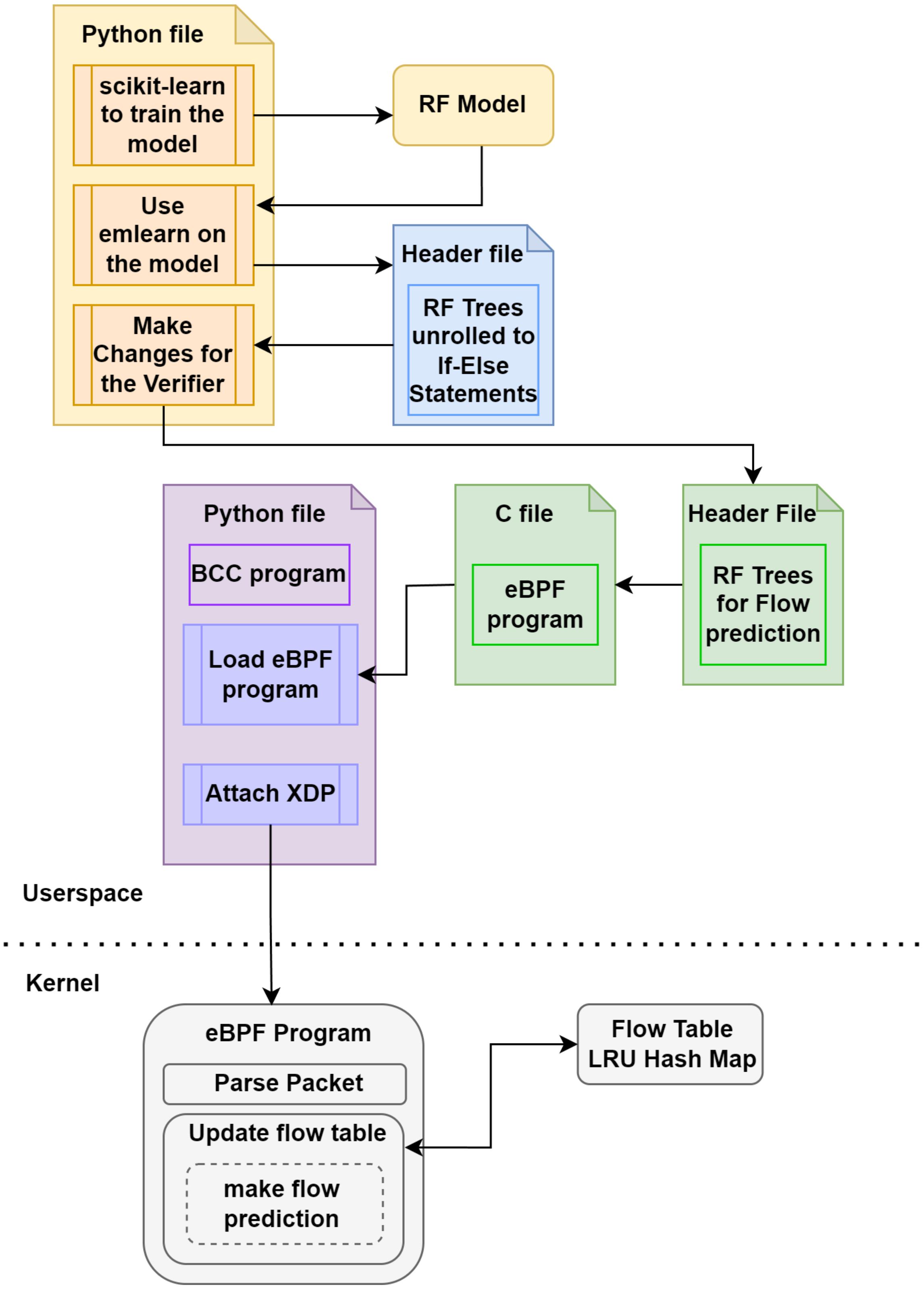


Figure 4.17: RF Model Compiled workflow

## 4.4 Port Scan Detection Mechanism and Response

Even though we can make the distinction between flows, from our analysis it does not seem appropriate to employ a single flow, classified as a scan, to infer that a port scan is being conducted, for various reasons. When discussing port scanning, it is usually inferred that a multitude of ports are being scanned. Additionally, using anomaly-based detection makes the solution prone to False positives. For this reason, it seems more suited to, while keeping this classification of flows, only infer that a port scan is taking place after a certain number of flows, classified as a scan, coming from a determined source, are registered. This would also align with what is stated in the Snort documentation [README.sfportscan]. This reference indicates that a single port scan is not evidence of an attack and a user operating legitimately within the network can employ behaviour similar to an attacker.

To address this issue a new eBPF LRU hash map was created to make this detection, along with two thresholds that define the detection sensitivity. The key of each element is the source IP and the value corresponds to the structure depicted in Listing 4.5. The value contents help in determining the type of scan that is being conducted.

```c
typedef struct ps_value {
    __u32 dst_ip; // (last connected IP) used to calculate ps_method
    __u16 dst_port; // (last connected port) used to calculate ps_method
    __u64 timestamp; // used to compare to PS_DELAY
    int ps_counter; // number of flows classified as scan
    int ps_method; // 0 vertical, 1 horizontal, 2 block
    int ps_type; // type of scan, syn, ack, etc
} ps_value;
```

Listing 4.5: Port Scan value Structure

To infer a port scan a threshold, that represents the maximum number of scan classifications, must be met. The threshold is compared to the variable *ps_counter*. This threshold can be easily changed to make the solution more or less sensible to probing. The value of this threshold is set to 25 by default. This value was determined by consulting the Snort documentation [README.sfportscan].

A second time-related threshold has been incorporated to address the potential false positives in anomaly-based detection. To mitigate this, elements are reset if the time difference between the most recent incoming flow and the previously stored one in the variable *timestamp*, is excessively large. In such cases, the element is reset, and the monitoring restarts with the latest flow. The default threshold for this time difference is set at 30 minutes, however, it can be changed to fit the environment. This choice is based on the consideration that if an attacker is conducting a scan on well-known ports with a 30-minute delay between each attempt, it will take approximately 21 days to complete the scan. A value too large to be feasible. This value was also based on the slow detection strategy used by [Dabbagh et al., 2011], refer to section 3.2.3.

In addition to the time threshold aimed at minimizing false positives, a secondary mechanism, self-proposed, has been introduced to further mitigate them. Each flow is associated with a calculated value that indicates the likelihood of a false positive occurrence. This value is determined by dividing the number of times the flow is classified as malicious by the total number of times the flow is updated. When this value drops significantly, it is interpreted as a false positive event. Subsequently, this triggers a decrement in the *ps_counter* value within the port scan detection map. If the *ps_counter* reaches 0, the corresponding element is then removed from the map.

Regarding the detection process, including the type of scan (SYN, FIN, etc.), scanning methods (vertical, horizontal, and block), and determination of whether the measured values indicate port scan activity, all these are carried out in the kernel. However, when it comes to the displaying of values to a user, a compromise was necessary. It is important to inform the user of the time, target IP and ports of the scanning activities. However, these values cannot be stored in the kernel as it would exceed the BPF stack size. Iterating through the flow table is not an option as by the time a port scan is inferred, flows used to increase the *ps_counter* may no longer be present due to LRU evictions. This information needs to then be stored in user space. How data is sent from the Kernel is explained in Section 4.5.1.

The implemented approach prioritizes minimal utilization of memory and CPU resources. Each flow classified as a scan is stored in user space. If a port scan is inferred all these values are printed and deleted from memory, as subsequent flows related to that source will already imply that they relate to an attack. Additionally, elements from the port scan map that are deleted based on the time threshold or by determining that they are false positives are also deleted from user space. Keep in mind that none of this is necessary if we do not intend to have visibility over the network activities and only want the solution to work in background mode. Figure 4.18 depicts the structure of the stored memory.

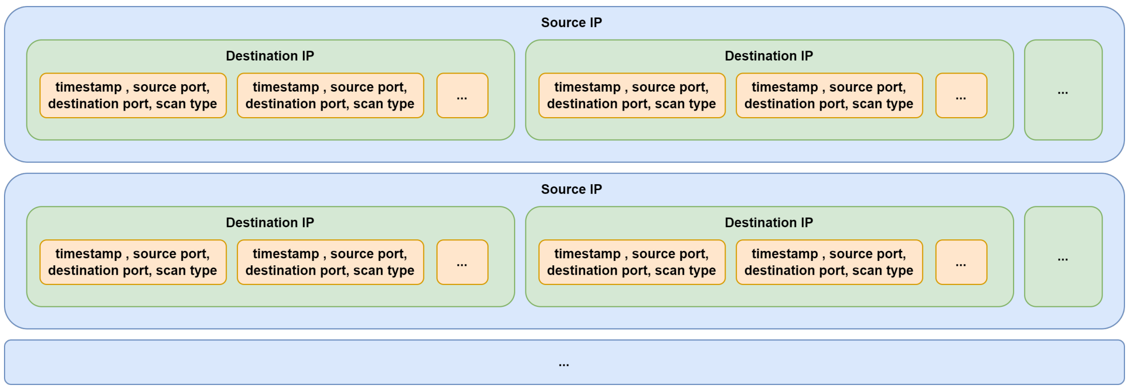

Source IP
Destination IP
timestamp , source port, destination port, scan type
timestamp , source port, destination port, scan type
...
Destination IP
timestamp , source port, destination port, scan type
timestamp , source port, destination port, scan type
...
...
Source IP
Destination IP
timestamp , source port, destination port, scan type
timestamp , source port, destination port, scan type
...
Destination IP
timestamp , source port, destination port, scan type
timestamp , source port, destination port, scan type
...
...
...

Figure 4.18: User Scape Memory Usage

When updating the port scan map, this being with a new flow or by determining a false positive, a return code is given. These return codes are used for determining actions with the user space memory but also to indicate that a port scan has been detected. If the return code corresponds to this measurement and an active response mode has been selected, this and future malicious flows from this source

IP are dropped using the XDP return code XDP_DROP. We still log their future probing attempts, however, they cannot retrieve information from the system. If passive mode is chosen nothing is done and the information is only logged.

To further understand the solution, the Listing 4.6 depicts the section of the program where this takes place. Note that some elements have not yet been addressed. This includes the concept of tailing, the transmission of events to user space and the introduction to XDP offloading. Additionally, the full context of the functions ps_table_add and ps_table_removed, is not depicted but explained previously.

```
int ebpf_main_tail(struct xdp_md *ctx){
    // get the flow information
    int key = 0;
    tail_ctx *value = tail_table.lookup(&key);
    if(value){
        flow_key fk = value->fk;
        flow_value fv = value->fv;
        __u64 rf_pred = value->rf_pred;
        // if the flow is scan, assess thresholds
        if(fv.scan != 0){
            ps_value psv = {};
            int event_type = ps_table_add(fk,fv,&psv);
            event_output(ctx,fk,psv,event_type);
            // drop malicious traffic
            if(event_type == 1 && DETECTION_MODE){
                return XDP_DROP;
            }
        }
        // flow has been deemed a false positive
        else if(fv.suspicious == 2){ // flow is no longer suspicious
            ps_table_remove(fk);
            event_output(ctx,fk,(ps_value){},-1);
        }
        // offload events
        if(OFFLOAD_MODE && (fv.packet_counter==2 ||
            (fv.packet_counter%10==0 && rf_pred == 1))){
            // if the flow is active send event, if after a while
                offloading is still sending market packets, resend event,
                maybe was lost during ring buffer overflow
            event_output(ctx,fk,(ps_value){},-3);
        }
    }

    return XDP_PASS;
}
```

Listing 4.6: Port Scan Detection Logic Function

## 4.5 Generating Alerts and Logs

Log entries need to be generated and presented to the user to allow the visualization of the performed activities in the network. When an attack is detected, the data stored is printed and future events sent by the kernel are also printed in user space. Two approaches were created to present the data to the user, one verbose and one summarized. The most intuitive is also the most verbose. It presents every flow that led to the port scan detection. However, this amount of information can be overwhelming to someone trying to interpret what is occurring. To address this a second approach was developed that aggregates each of these flows and presents this information more concisely. Refer to Listings 4.7 and 4.8.

```
2024-03-06 15:29:20.910753 - ALERT: Vertical Syn Port Scan detected
    from 192.168.1.212:43347 to 192.168.1.216:80
2024-03-06 15:29:20.911063 - ALERT: Vertical Syn Port Scan detected
    from 192.168.1.212:43347 to 192.168.1.216:83
...
2024-03-06 15:29:39.425216 - ALERT: Vertical Ack Port Scan detected
    from 192.168.1.212:40703 to 192.168.1.216:98
2024-03-06 15:29:39.425338 - ALERT: Vertical Ack Port Scan detected
    from 192.168.1.212:40703 to 192.168.1.216:90
```

Listing 4.7: Verbose Port Scan Alert

```
2024-03-06 15:36:19.42 - ALERT . Port Scan detected from 192.168.1.212
                               |___ Scan Method: Vertical
                               | |___Target IP: 192.168.1.216
                               | | |___ Scan Type: ['Syn', 'Ack']
                               | | |___ Scan Attempts: 32
                               | | |___ Target Ports: [80-110]
                               | | |___ Duration Time: 0:00:04.017609
```

Listing 4.8: Simple Port Scan Alert

### 4.5.1 Information from Kernel to User Space

To get information from the kernel there are various means. The most common way is to use the function `bpf_trace_printk`, which is a basic kernel method to use the `printf` to the trace pipe. However, this has various limitations which include a maximum of three arguments, a single %s and the fact that it shares the trace pipe globally resulting in the concurrent output of programs. As mentioned by the BCC documentation [BCC Reference Guide] a better and more elegant approach is to use the `BPF_PERF_OUTPUT`.

The `BPF_PERF_OUTPUT` structure is a BPF table that allows the sending of custom event data to user space via a data structure called perf ring buffer. A ring buffer is a piece of memory organized in a ring. This ring possesses two pointers, one for writing and one for reading. In the case that the read pointer catches the write pointer, it simply indicates that there is no more data to be read. However, if the opposite happens and the write pointer overtakes the read pointer, the data to be read is lost. Listing 4.9 shows the type of event data transmitted.

```c
typedef struct event {
    int type; // 0 store, 1 alert, -1 delete, -2 restart, -3 offload
    int ps_method; // 0 vertical, 1 horizontal, 2 block
    int ps_type; // 1 Udp, 2 Fin, 3 Syn, 4 Ack
    __u32 src_ip;
    __u32 dst_ip;
    __u16 dst_port;
    __u16 src_port;
    __u16 protocol; // used for offload key
    __u16 padding; // padding required to satisfy verifier
} event;
```

Listing 4.9: Event Structure

In the context of our solution, if the speed at which the kernel generates events outperforms the user space capability of reading said events, then the information gets lost. Not only do we lose information, but in the presence of a situation where the events keep getting generated, like in the presence of a DDoS, not allowing for the read pointer to catch up, leads to the inability to stop the execution of the eBPF IDS. The most common solution to fight this issue is to extend the size of the ring buffer's memory or the aggregate events. The first solution does not fix the core of the issue, as it would only delay the situation. Aggregating events were tried, but as explained in the previous section, it was not possible to store this amount of data in the kernel as it would exceed the BPF stack size, which is hard coded and cannot be changed. Additionally, even if this was possible, after the initial alert there is no need to store flows and they would be immediately sent to the ring buffer, leading to the same issue once more.

To address this issue, a simple array map of a single element was created to communicate between user space and kernel, to inform when such a situation is present. With this, every time an event needs to be generated, this map is accessed to determine if information can be sent. When it is determined that information is lost, the value in this map is changed and events are halted temporarily to allow the read pointer to catch up. Information is outputted about how many events have been lost. Within the kernel detection continues as if nothing is happening, it just does not generate events until allowed again by user space, it however increments a counter on how many events it did not send, storing this information in the array map. The following Listing, 4.10, depicts how this type of occurrence is presented, having as an example the values lost in a flood attack which trigger the return of events.

```
EBPF-IDS: ERROR - PERF OUTPUT AS REACHED MAXIMUM CAPACITY, 3330
    POSSIBILITY LOST SAMPLES, STOPPED EVENTS SUBMISSION
EBPF-IDS: LOST 115359 EVENTS
EBPF-IDS: RESTARTING PERF OUTPUT SUBMISSION
```

Listing 4.10: Ring Buffer Alert

## 4.6 Compatibility for Offloaded Mode

In this section, the development and challenges faced in the integration of compatibility for offloaded mode, are presented. As it has been noted numerous times, applying this type of solution requires specific hardware. The NIC acquired is a Netronome SmartNIC Agilio CX 2x10GbE [Netronome].

The use of such a NIC imposes limitations over the ones already imposed by eBPF. Some of these limitations were considered during development. However, it is impossible to account for all possible variables without the component itself. Given that this acquisition only took place late in development, the NIC was incorporated as a last element. Had the development started with this item present, a different path could have been taken, like a limited solution working only in the NIC. Nonetheless, we believe that the proposed solution is still the best approach.

### 4.6.1 Implementation Roadblocks

It is important to address the multiple issues that arose when trying to incorporate this item. With this, one can understand the decisions made.

#### SmartNIC limitations

Firstly, when trying to account for possible limitations, documentation was consulted. However, there was unclear information, that could only be verified with the hardware in possession. One of this information is related to the operations allowed by the firmware on eBPF Maps. The helper function `bpf_update_elem`, is crucial for the functionality of our solution, as it allows us to keep the context of the previously examined information. According to sources like [BPF & XDP Reference Guide] and a blog post by Netronome [Ever Deeper with BPF – An Update on Hardware Offload Support], this helper function was available. However, in Netronome documentation [eBPF Offload Getting Started Guide , Netronome CX SmartNIC], this function is not available in the public firmware, and only support for atomic write operations was available. The `__sync_fetch_and_add` only allows us to add values to existing elements in a Map, which doesn't fit our solution. When we tested the `bpf_update_elem` helper function it was indeed not supported by the firmware. We believe that this inconsistency stems from the fact that map updates are supported but only from user space. However, not all sources are clear about this distinction. Additionally, the helper functions listed as supported, always contained the mentioned function, again, leading to confusion. Later in development we also discovered that the `bpf_delete_elem`, which doesn't possess any ambiguities detailed in documentation was also not supported but listed as so.

Contact with Netronome was established to try and obtain the private version of the firmware. Netronome provided not only the private firmware but also documentation related to this feature which made clear why was kept private. This documentation detailed all available use cases of the function as well as limitations. Due to the way the SmartNIC functions, to address concurrent changes to map value, the use of locks is necessary. This limitation can lead to serious performance degradation. The way `bpf_update_elem` works is most likely identical to `bpf_delete_elem`, and so it being also not public is justified.

An attempt was made to use this private version of the firmware for our solution, to understand the possible impact. A simple implementation that kept track of the network flows, updating their features was developed. The performance degradation from this solution offloaded compared to its deployment in the kernel was drastic. We concluded that following this path was not feasible. Additionally having a solution that only works with firmware on request also seems inappropriate. More on this issue is detailed in Appendix A

From this issue, we pivoted our solution to a partial offload, settling a compromise between what we envisioned and what was possible. More on this matter is discussed in the sections that follow. To pass information from the SmartNIC to the Kernel, access to the *data_meta* parameters was ideal, refer to Section 4.2. Presented in one of the Netronome's webinars [BPF Hardware Offload Deep Dive Webinar] this would be possible, however, the function required to do this, `bpf_xdp_adjust_meta` is not supported by the NFP firmware.

### BCC Errors

Finally, the BCC framework also posed some challenges. Even though stating compatibility with the offloaded mode, when trying to load the program, which is the first step in deployment, results in an error:

```
  File
      "/usr/lib/python3/dist-packages/bcc-0.30.0+6a5602ce-py3.10.egg/bcc/
  __init__.py", line 474, in __init__
ctypes.ArgumentError: argument 6: TypeError: wrong type
```

Listing 4.11: BCC incorrect error

According to the BCC repository merge [Support for hardware offload] the correct error when trying to load an offload program, which may appear because the operation is not supported, should be:

```
  File "/usr/local/lib/python2.7/dist-packages/bcc/__init__.py", line
      347, in __init__
    raise Exception("Failed to compile BPF module %s" % (src_file or
        "<text>"))
Exception: Failed to compile BPF module <text>
```

Listing 4.12: BCC correct error

By consulting the BCC source code and analysing the error that occurred. The issue was fixable by casting the interface name to a `const char pointer`, instead of passing this information as a string, as shown in [BCC XDP Examples].

### 4.6.2 eBPF Tail Calls

Before moving on to the implemented solution and its details, it is important to note an element required to move on with development. Until now the entirety of our code could be deployed in a single eBPF program. However, as we reached this stage the program became too complex to be analysed by the Verifier. This required our solution to be split, and use tail calls to link them.

An eBPF tail call, in simple terms, allows an eBPF program to call another function, which is seen as a separate program by the Verifier. This, in turn, makes each targeted function pass independently by the Verifier which resets its allowed complexity [eBPF Documentation]. Our program was split after the flow as been processed and classified by the ML algorithm and before it was analysed against the port scan detection logic.

To support this strategy two new eBPF maps were required. Firstly, the map type `BPF_MAP_TYPE_PROG_ARRAY`, which is indispensable, as it is used to contain the references to the programs. Then an additional array map was added to store the previously processed information and pass it on to the subsequent program to read and continue processing. From our measurements, the process of tailing and passing on the information has minimal impact on performance.

### 4.6.3 Partial Offload

Taking into consideration the limitations documented in section 4.6.1, to implement offloading into our solution, a compromise was established, only offloading what was possible.

The core of our solution works by using the hardware to classify the first packet of a flow. From there, all other packets are classified in the kernel, having this part of the program deployed in driver mode. In normal communications, this doesn't seem to provide enough benefit, as only the first packet gets the hardware processing benefits. However, in the context of port scanning this implementation fits perfectly. Port scan attempts are characterized by a single packet per flow, in most scenarios, therefore all these attempts would be processed in the hardware and not in the host.

In the hardware, the process starts by parsing the packet. Because we can't keep track of flows, again, due to limited map operations, only the first packet of a flow is analysed. However, to save processing, it is ideal for our solution to not process the packet twice, in the hardware and then in the kernel. To solve this issue, a hash map was inserted in the hardware that contains active flows. A flow is active if it possesses more than two packets registered. The kernel is the one that determines if a flow is active, sending an event to the perf output. This event is then received in user space, writing the flow key into the offloaded map, as this is the only way to insert elements. This gives a small time window, where packets are classified in the hardware and kernel, while the user space attempts to insert the new flow. However, from our measurements this operation is extremely fast, taking on average 41.6 ms to complete, with a standard deviation of 0.35 ms.

Because we are using the perf output system to send events to user space, we must not forget the limitations previously documented. To address possible loss of events due to ring buffer overflow, the active flow event, is re-transmitted in the case that it receives a classified packet from the hardware in a flow already marked as active.

After parsing, and determining if the packet belongs to an active flow, in the case that it doesn't, it is classified. To insert ML in the hardware, the compiled RF model was implemented like before, however, the model stored via maps was not accepted by the Verifier.

Because only the map type array and hash are available, to address possible overflow issues when storing active flows, we must incorporate a system that evicts elements from the map when it reaches maximum size. As mentioned the `bpf_delete_elem` is not supported by the public firmware of the NFP. This action is however possible from user space. However, from all BCC functions for deleting elements only one was compatible with offloaded maps. This function deletes all entries in a map. This is not ideal but it is the only means by which we can address the issue.

The process of deleting all entries is time-consuming, causing downtime in the access to that map, figure 4.19 represents these values. While the access to maps is down, the hardware passes a value, indicating that it did not perform any classification, and that process must be done by the kernel. There are no eBPF helper functions in the firmware to get the size of the hash map. For the hardware to know it reached maximum capacity and pass all classifications to the kernel, an array map of a single element was added indicating the active size of the hash map. This array map is updated by the user space program. This map is also used to let the offload program know when a ring buffer overflow occurred, and therefore it can not rely on its active flow table.

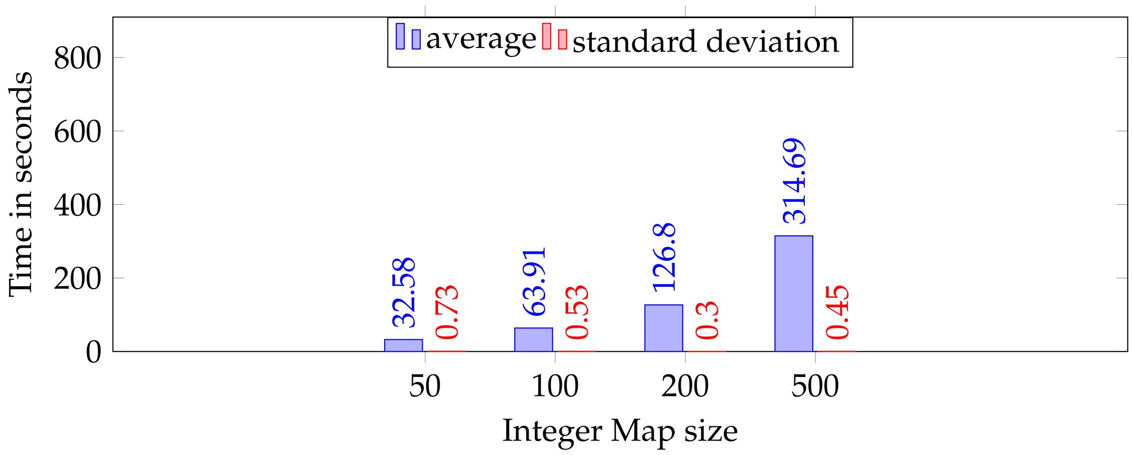


Figure 4.19: Time for resetting offload map entries

From this distribution, a map size that balances the number of active flows it can store and the time it requires to reset its contents must be selected. This depends a lot on the environment where this solution is deployed. After some deliberation, the size of 100 was deemed reasonable. However, for a real scenario, this choice would depend heavily on the environment and the specific objectives.

To send information from the hardware to the kernel, the most intuitive way would be to append information to the *data_meta* parameter. However, as mentioned, this was not possible. The NFP, however, possess an alternative, the helper function `bpf_xdp_adjust_head`. A solution was attempted, incorporating this strategy. We would add a header to the packet containing our classification, and then in the kernel remove this header. Even though functional, this process caused degradation in the IDS processing speeds. To solve this issue a new method to pass information to the kernel was required. Without other direct obvious alternatives to pass information to the kernel, the use of the RX RSS queues was selected to infer information. The RX RSS queue is used to distribute packets across CPUs. The RX RSS queue can be selected with the NFP by changing the variable `rx_queue_index`, refer to section 4.2. Our environment possesses four of these queues, and for our solution to work we require three. In the kernel by examining which queue the packet has assigned decisions are taken. The enumeration that follows details this relation.

- RX RSS Queue 0: if a packet has this assigned queue, it means that this packet is benign or that failed during parsing. If the incoming packet belongs to an active flow is also sent via this queue.
- RX RSS Queue 1: if a packet has this assigned queue, it means that this packet is malicious and is associated with a port scan attempt.
- RX RSS Queue 2: if a packet has this assigned queue, it means that the active flows table is being reset or the ring buffer overflowed and therefore classifications in the hardware cannot take place, as it can't determine if the incoming packet belongs to an existing flow. For this reason, packet classification must be done in the kernel.

This method proved very efficient, as it only requires to change a single variable. Additionally, it is in essence, load-balancing the attack. However, this solution only allows us to know if a port scan was attempted and not the type of scan. To solve this issue we decide to follow a hierarchical model strategy. Put simply, a hierarchical model works by using a model that first distinguishes the main categories, in our case between normal traffic and port scans, then a second model that identifies the sub-categories of the chosen main category. In the context of our solution, we don't require a second model. As previously documented in section 2.3, port scans are extremely similar, being easy to identify by their protocol and used flags. In our solution, the multi-class model is used in the hardware, however, we only can tell the kernel that a port scan was detected. Therefore in the kernel, we implemented a simple decision tree that then identifies what type of scan occurred.

The Figure that follows, 4.20, depicts a high-level representation of how our final solution uses XDP offloading capabilities of the SmartNIC. Note that in this representation, some elements are highlighted differently, namely the eBPF maps storing the ML model and the thread that queues events. This is because these elements are optional. One can choose between the model's deployment and also choose between the alerts generated.

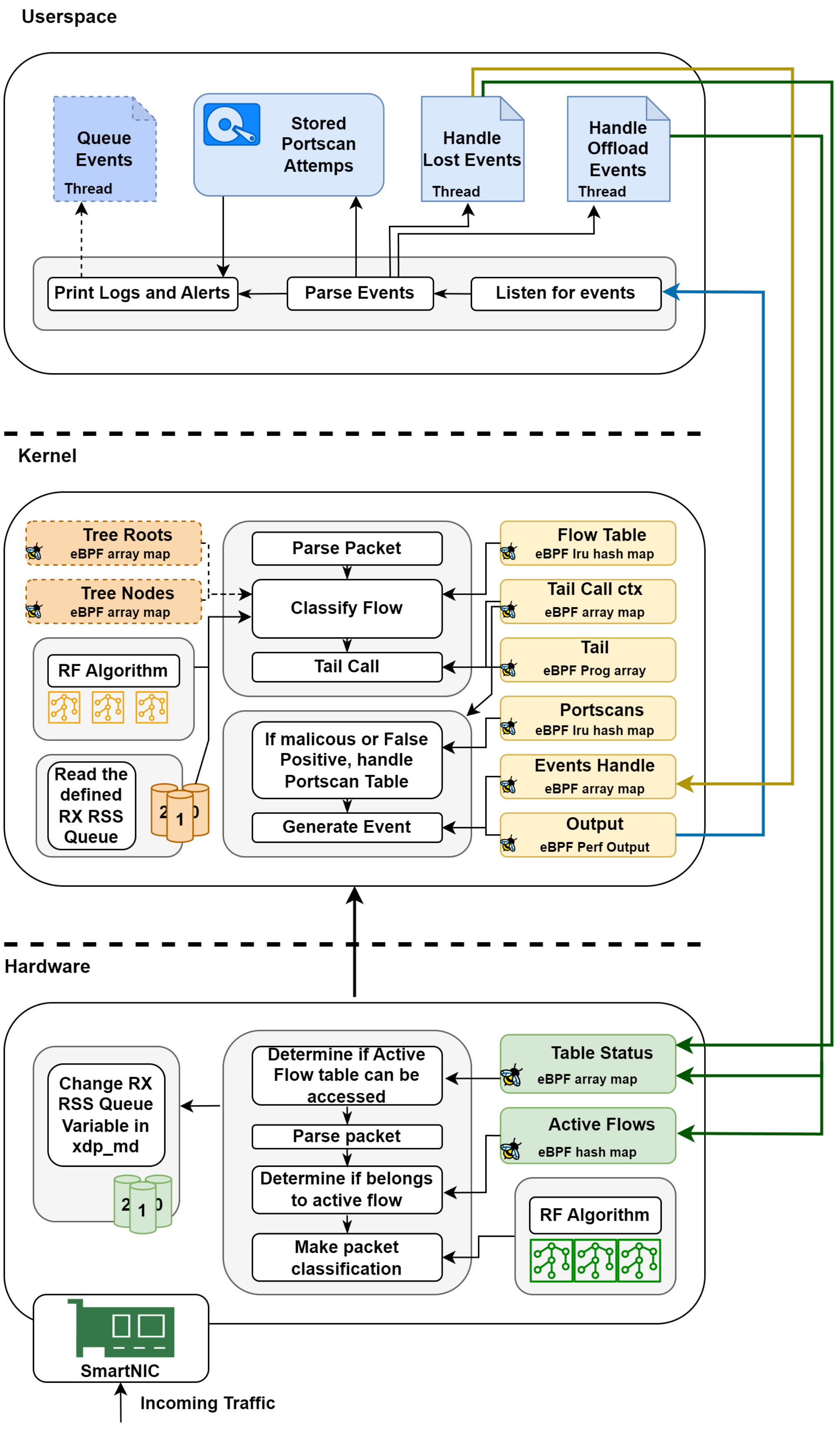


Figure 4.20: Partial offload overview

## 4.7 Chapter Wrap-Up

In this Chapter, the development of the proof of concept was undertaken. We analysed the CIDDS-002, establishing what elements were worthy to be used when training the model. In the end, five features were selected, corresponding to the duration of the flow, protocol, number of packets received, transmitted bytes and flags. From there we analysed the best parameters the model could use, electing 11 trees with a maximum depth of 3 as adequate.

The incorporation of RF within the kernel followed two approaches. The first implementation loaded the model values into eBPF Maps, which could then be retrieved in the kernel to make the flow classification. The second method used *emlearn* to contain the model within a single header file, which could be then called by the eBPF main program.

A single flow classified as malicious is not indicative of a port scan. Logic capable of associating multiple malicious flows and classifying them as a port scan was then developed. Following this, the development of means by which alerts could reach user space, and notify the user, was incorporated into the solution.

The final stage of development focused on the offloading domain. Given the various roadblocks faced, the final solution incorporated a partial offload. The first packet of a flow is classified in the hardware, and the rest of the packets are classified in the kernel. In the context of port scanning where most scan attempts represent a single packet per flow, this solution fits perfectly.

# Chapter 5

# Validation and Evaluation

With the developed work established in the previous chapter, we now consider evaluating the solution in this section. Our study will encompass the evaluation of the two proposed ML approaches and the validation, to assess whether the solution performs the envisioned detection effectively. After this, an assessment of the solution's performance will be made.

Section 5.1 presents the methodology. Section 5.2 examines a comparative analysis between the two developed methods: the ML model loaded via eBPF Maps and the compiled approach. Following that, Section 5.3 validates the solution to determine its effectiveness in achieving its intended purpose. Section 5.4 compares the performance of the solution. Section 5.5 examines the solution response to overflows. Finally, Section 5.6 entails an overview of the chapter.

## 5.1 Evaluation Methodology

Before moving on to the presentation of results, a clarification on how we will attempt to evaluate the solution is necessary. The following enumeration is associated with each Section that follows:

- ML implementation comparison: to examine this element, we must measure the time it takes to classify a flow and the size of the ML model the program can hold. By storing a timestamp right before and after the classification, we can get the classification time. We store these times in an eBPF Map. This is then accessed in user space. In the end, the average and standard deviation are calculated. To determine the size of the model, the number of nodes is the used metric. The number of trees and tree depth is increased incrementally until the model can no longer be deployed.

- Detection capabilities: to determine if detection is appropriate, port scans are conducted against the IDS. If they are identified correctly, this is interpreted as a successful detection. Additionally, a comparison between the two ML Models is also important.

- Performance: To understand the solution's performance, the packets analysed per second were an important metric to consider. When only measuring a continuous flow, this value can be obtained, in user space, by retrieving the flow's packets and duration and dividing them. However, in the presence of multiple flows, more is needed. This is because flows may be removed due to LRU evictions. An eBPF Map is used to store the number of all incoming packets, alongside timestamps of the first and final captured packets. Note that this can cause a small degradation in performance, as in every packet additional calculations are done.
- Overflows: to evaluate how the solution behaves in an abnormal environment, the IDS was put in a scenario where the objective was to overflow the eBPF Maps to see if detection is still effective.

## 5.2 ML via Maps and ML Compiled Performance

During the practical development phase, we explored two approaches for implementing ML in eBPF. The first approach, which intuitively addresses the problem, involves retrieving the roots and nodes of the model. These values are then loaded into maps. Subsequently, in the kernel, we recreate the classification/prediction algorithm and retrieve the stored values from the maps to determine the classification. The second approach involves a library that can unroll the trees into if-else statements. This generated code is contained within a header file that can be called from the eBPF main code. Refer to Section 4.3

Before resuming the examination of how each one performs, it is important to address nuances that have already been encountered. As explained above, the eBPF Verifer imposes limitations on the program size and complexity, refer to section 2.1.1. When models bigger than the ones established were evaluated, it was possible to observe that some models were too large to be hard-coded. However, when using Maps these same models could be used. This issue arises because the hard-coded implementation exceeds the Verifier complexity when attempting to analyze all paths of the if and else statements. Means of circumventing this limit are possible through tailing but have not been implemented as the current solution does not require it. Therefore in its current state, via the Maps solution, we can load a much bigger model if needed. The model size will vary depending on the complexity of the remaining logic in the program. However, in our specific case, the maximum number of nodes in the forest for the hard-coded solution is between 72 and 77 in the kernel, and 112 and 118 offloaded, whereas on Maps it is between 1387 and 1466. Note that in driver mode our model is already reaching the maximum size allowed. Additionally, the reason why offloaded possesses a slightly bigger size is that it doesn't possess all the same logic that is present in the kernel.

We monitored the time that each prediction function needed to come up with the prediction value for that flow. We measured the time of various predictions of various flows (Scans and HTTP connections among others) and then calculated the average and standard deviation. Figure 5.1 shows the results.

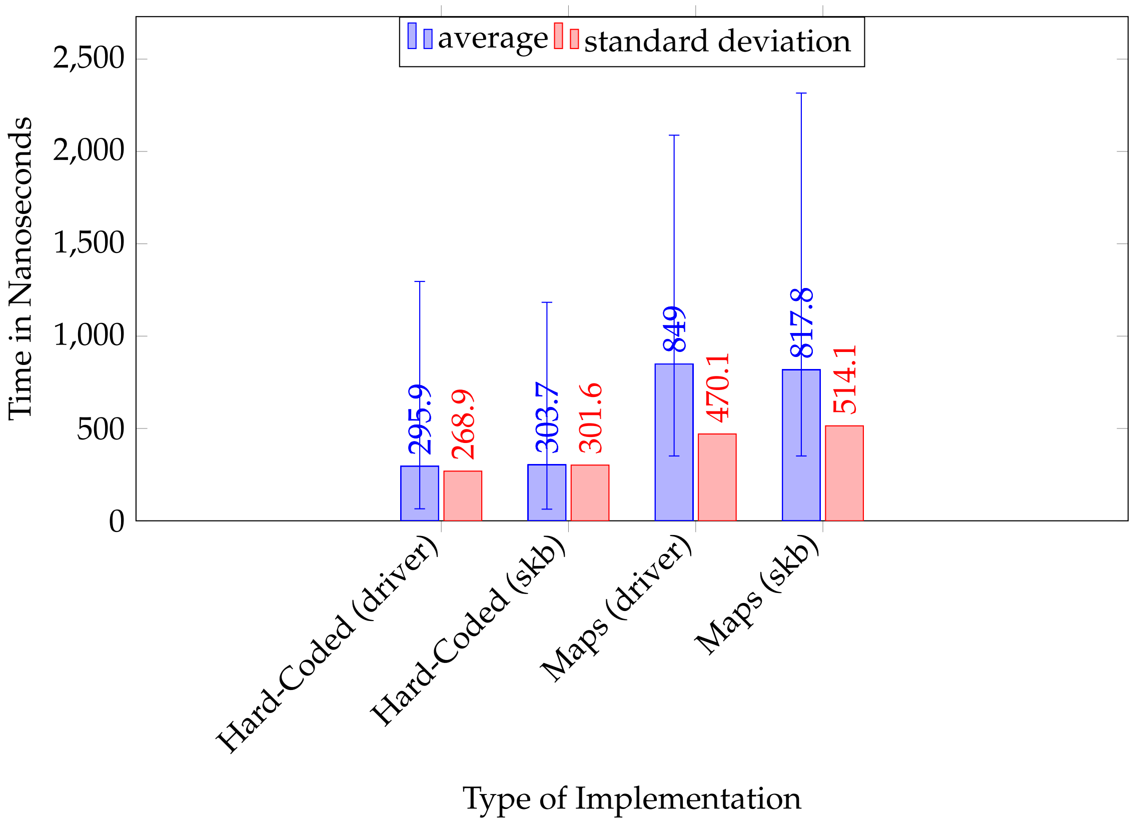


Figure 5.1: Performance Comparisons between ML implementations

There is a significant variance in measurements because the number of decisions to determine a classification is not constant. Some flows get determined much faster. In this test, we attempted to have normal flows and attacks represent nearly the same amount. In Figure 5.2, the times measured can be seen as well as their variance. It is easy to identify where flows begin and end.

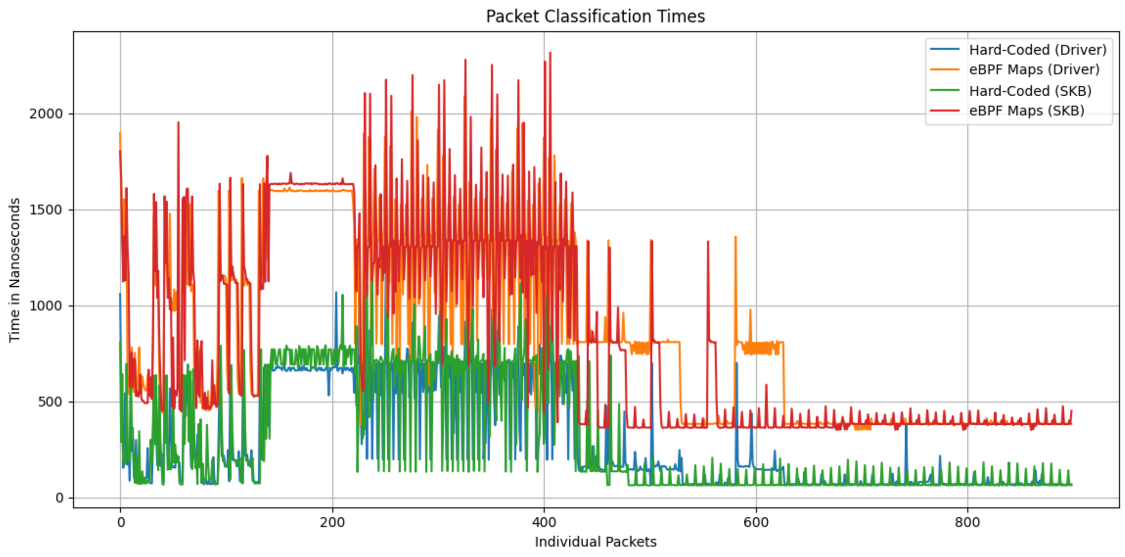


Figure 5.2: Packet Classifications Times

Has we can see, the prediction made using the hard-coded model is much faster the the one using Maps. However, we cannot conclude that this is the best solution due to the factors explained before, concerning the model size. When choosing the one that is the most suitable for the job at hand one must consider the speed and the size of the model. If the size is small enough to be hard-coded, then this is the optimal solution, otherwise, it must employ one using Maps.

Unfortunately, we could not get a measurement of the time it takes for the offloaded mode to classify a packet. The helper function needed to obtain the time is not available by the NFP firmware. Note that, if it existed it would only allow us to compare to the other hard-coded measurements as the solution via maps was not implemented in the offloaded domain.

## 5.3 eBPF-IDS Detection Analysis

Another element we wish to evaluate is whether the solution can detect port scans. The dataset used to train the model possessed the following port scans: SYN, ACK, FIN, and UDP. During development, our initial model was binary and then progressed to multi-class. In this transition, we observed various behaviours which we believe are important to document. To perform these tests, the Nmap tool was used [Nmap Reference Guide]. Then the scans in question were used against the machine running the eBPF-IDS solution. The scans were used against a multitude of ports, both opened and closed. Table 5.1 depicts the observed results in the binary model and the final solution.

| **Scan Type** | **Detected (binary)** | **Detected (multiclass)** |
|---|---|---|
| SYN | Yes / No | Yes |
| FIN | Yes | Yes |
| ACK | Yes | Yes |
| UDP | Yes | Yes |

Table 5.1: Main Port Scans Detected by the solution

From the analysis conducted, both FIN and UDP were always identified and the flow was classified as a port scan. However, some nuances were observed with SYN, on the binary model, and with ACK.

The SYN scan was detected when the scan performed was against a closed port. However, when facing an open port, the SYN scan sends a second packet, the RST packet. This characteristic led to the following behaviour: when receiving the SYN packet the eBPF-IDS solution would classify the received flow as a port scan, however, when the flow was updated with the RST packet this flow would be declassified back to normal traffic. A possible reason for this behaviour has to do with the distribution of the dataset concerning SYN scans to open ports. Figure 5.3 is meant to illustrate these distributions.

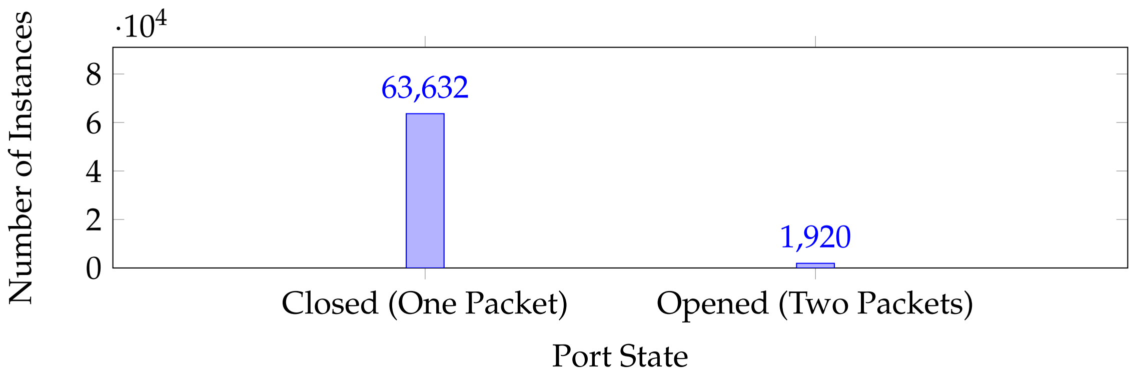


Figure 5.3: SYN Scan Distribution concerning Number of Packets

It is clear that SYN scan to open ports, which will utilize more than one packet, is much lower. Also, we are grouping all scan types into a single-port scan classification. Taking this into account, we observe that all other scan types only utilize one packet, making this discrepancy even greater. Note that these results consider the pre-processing done in Chapter 4. However, when updating to the multi-class model this behaviour was no longer present. The model was able to classify SYN scans to open ports correctly.

Another observed issue relates to the ACK scan. When performing an ACK scan, this could be detected by both solutions. However, it was observed that some normal flows could sometimes be classified as a Port Scan in the first packet and then, immediately after the flow updated with a new packet, it would be declassified, reverting to normal traffic. When observing the first packet of this flow, it becomes clear why. When placing both this first packet of a flow where this would happen and an ACK scan side by side, they would have the same characteristics. With this, further examination was conducted which revealed that when training a model, this would fall to one of two sides. Flows with the previous characteristics would be classified either as an ACK Scan or in rare situations classified as normal traffic. We believe that the first situation is acceptable because even though a flow can be wrongfully classified it will immediately change once updated keeping actual ACK Scans identified, unlike the other situation where they are never identified.

Given that our solution uses ML, it is also important to analyse if it can detect scans outside the ones it was trained on, refer to Section 2.3. When testing this process we also observed changes when updating the model. Table 5.2 depicts the observed results.

From these results, it becomes apparent that some of the detection capabilities were lost when updating the model when it comes to other scans, different from the ones the model was trained on. When identified, these scans had classifications that represented the scan they were most similar to. For example, the Window scan was classified as ACK, this is because it sends a packet equal to one of an ACK, however, what changes is the interpretation of the response, refer to Section 2.3.2. It is important to note that when using the offloaded mode, these other scans could not fall within the decision tree logic and therefore classified only as a port scan, being unable to determine further details.

| Scan Type | Detected (binary) | Detected (multiclass) |
|---|---|---|
| Xmas | Yes | No |
| NULL | Yes | Yes |
| Maimon | Yes | No |
| Window | Yes | Yes |
| TCP Connect | No | Yes |

Table 5.2: Other Port Scans Detected by the Solution

We also analysed details that surround evasion techniques like changing the size of the probes being sent. It was observed that the binary model failed when this was done. However, the multi-class model managed to still correctly identify these scans.

With these measurements, one can conclude that using the binary or multi-class model can have advantages and disadvantages. If one desires to implement a solution to detect a broad spectrum of a category, the binary model, which aggregates all subcategories into a single one, seems to be the most appropriate. However, if one's intention is to detect specific targets within a category, following a multi-class approach is the best option.

## 5.4 eBPF-IDS Performance Analysis

The performance of our solution is also an important metric to consider. With this, one can have a greater perspective on its suitability for a real-world scenario. It would also allow us to place our solution concerning other authors' implementations.

### 5.4.1 Data Processed per Second

To analyse the packets per second our solution can process, the iPerf3 [iPerf3] tool was used. However, to have some point of reference, some other metrics are necessary. Firstly a measurement with no IDS. Secondly, like other authors presented [ANAND et al., 2023; Bachl et al., 2021; Carvalho et al., 2023; Pradhan and Mannepalli, 2021], measurement of a user space solution. For this, we developed a similar implementation of what was done in the kernel but in user space. This solution was developed in Python using Scapy [Scapy]. This solution does not produce the best performance, however, it allows us to have some perspective. Additionally, we also measured Snort's performance. Even though Snort belongs to a different category of IDSs and it is not fair to compare signature-based solutions to anomaly-based ones, we believe it is still important to have it as a reference.

Since the offloaded solution only analyses the initial packet, its performance will closely resemble that of the driver mode solution. This offloaded approach is specifically tailored for detecting port scans. To thoroughly assess its performance, it's best to deploy it in such a scenario. The iPerf3 tool does not allow to generate of traffic where every packet belongs to a new network flow. To simulate this environment, we simply tell the offloaded program to always classify a packet and tell the kernel program to never classify it. This effectively puts the solution in a scenario where all packets are classified in the hardware. Note that, because the hardware does not have context of the previous packets, its classification would not be correct, this test would only allow us to understand the speed benefit of classifications.

Table 5.3 depicts the measured values of the solution in its different XDP modes and ML model implementations. Bellow it, is a sub table, to have the other measurements as a reference.

| **XDP Mode** | **ML mode** | **Avg Pkts/sec** | **Std Dev Pkts/sec** | **Avg Gbits/sec** | **Std Dev Gbits/sec** |
|---|---|---|---|---|---|
| Generic | Maps | 266 405 | 2 750 | 3.05 | 0.02 |
| Generic | Compiled | 290 483 | 2 751 | 3.33 | 0.02 |
| Driver | Maps | 490 827 | 3 945 | 5.58 | 0.04 |
| Driver | Compiled | 596 942 | 7 660 | 6.77 | 0.09 |
| Offload | Maps | 503 680 | 4 927 | 5.74 | 0.05 |
| Offload | Compiled | 605 861 | 2 230 | 6.87 | 0.02 |
| Offload[a] | - | 618 723 | 964 | 7.00 | 0.01 |

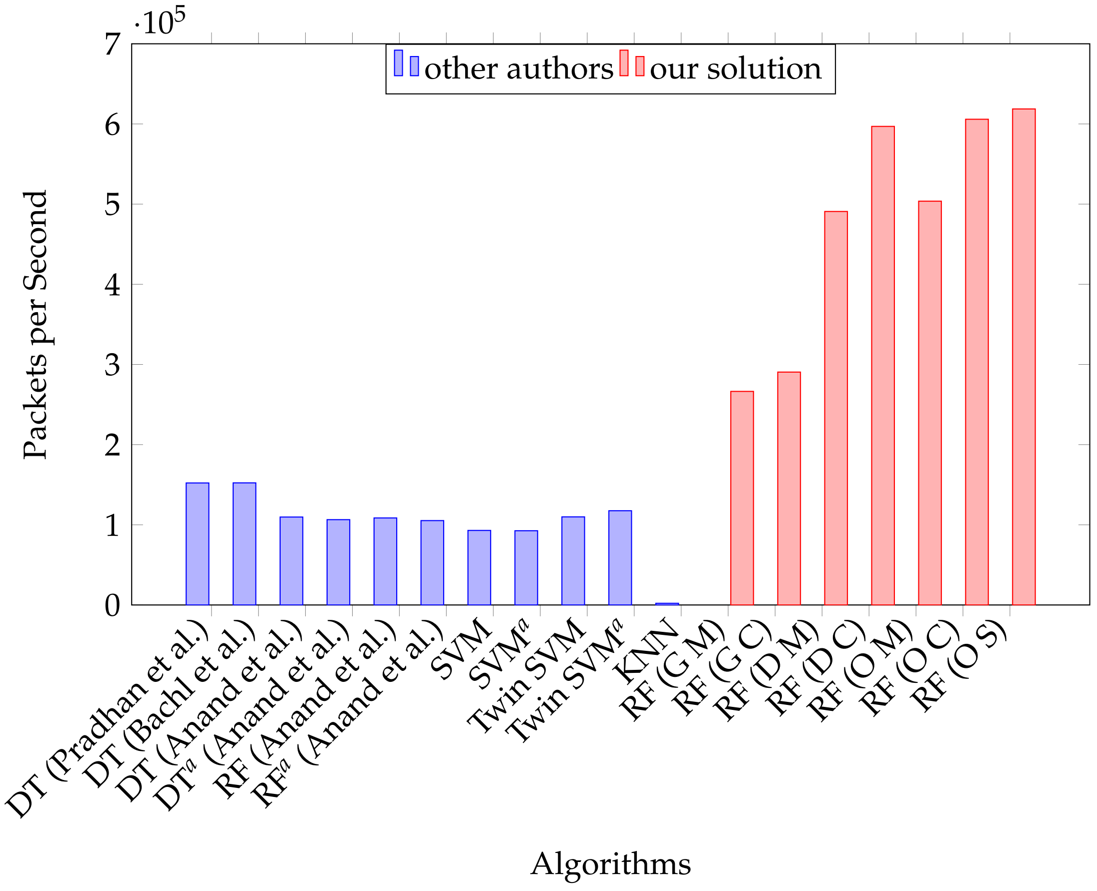
| **Solution** | **Avg Pkts/sec** | **Std Dev Pkts/sec** | **Avg Gbits/sec** | **Std Dev Gbits/sec** |
|---|---|---|---|---|
| User Space IDS | 1 094 | 12 | 9.20 | 0.01 |
| Snort | 51 840 | 2 765 | 9.06 | 0.11 |
| Default (no IDS) | - | - | 9.41 | 0.01 |

[a] Simulated Environment

Table 5.3: eBPF-IDS Performance Evaluation

From these measurements, the difference between the ML deployment becomes apparent. We already knew the compiled version of the model was faster. However, now, we can see its benefit in the performance of the overall solution. When it comes to the XDP modes, the difference in speed is also noticeable. Primarily when moving from generic to driver mode. As expected the driver and offloaded mode, in a normal scenario, presented similar performance. This is obvious, as only the first packet took advantage of the offloading. However, in the simulated scenario, the offload solution was 3.5% faster than in the driver, which may look small but is over a 20,000 packet per second difference. Compared to their user space counterparts, the eBPF solutions greatly exceed their performance. Note that the Gbits per second are superior because the user-space solutions do not sit in line of traffic.

Figure 5.4 places our solution against the other authors that implemented an eBPF IDS. Making these comparisons does allow us to have a perspective on how successful our solution is. However, consider that there are multiple variables at play. Firstly, different algorithms are employed. Secondly, these authors developed solutions for multi-class detection, requiring, therefore, more features to be analysed, and a bigger model to produce correct results. Finally, the hardware aspect is also relevant, for example, the availability of XDP modes will hinder performance.



[a] DoS/DDoS of CICIDS2017 Dataset
G - Generic, D - Driver, O - Offload, C - Compiled, M - Maps, S - Simulated

Figure 5.4: Packets per Second of eBPF IDS solutions[1]

[1]It is important to reinforce that this comparison in terms of performance analysis is not fair, due to variables such as hardware, ML model size and solution's target. Other ML solutions did not use a SmartNIC and used different models. All this makes a difference. It is only to provide a view of how it performs to existing solutions.

### 5.4.2 Load Balancing the Port Scan Attack

The previous test, simply showed the performance when processing a continuous flow of traffic. However, given the nature of our solution, there is a test that can show the true performance benefit of our offloaded approach. Taking a closer look at our proposed detection system, one can infer that it is load-balancing the attack. This means, that the malicious flows and normal ones are routed through different paths and processed by different CPUs, effectively improving performance.

To test this situation, both iPerf3 and hping3 are used. Because in a hping3 flood, each packet resembles the one of a port scan, it effectively simulates a scenario where multiple port scan attempts are generated at a high rate.

It is important to note that to obtain these metrics, the code had to be changed, as explained in Section 5.1. This addition is only to measure the values. However, this modification leads to a small performance decrease. Because of that, the most appropriate evaluation metric is the performance increase from one solution deployment to the other.

Another point to take into account is that with the hping3 flood, the perf output map will eventually overflow. The iPerf3 traffic will keep being sent by RX RSS queue 0. However, the rest of the packets will shift from queue 1 to queue 2 because the hardware cannot know for certain if the packet does belong to an active flow as it could have been lost in the overflow. Nonetheless, all new traffic is still being load-balanced. Some performance is lost as classifications are now done in the kernel.

Table 5.4 depicts the measured values. Consider that the iPerf3 Gbits per second is decreased as the hping3 flood is running simultaneously.

| **XDP Mode** | **ML mode** | **Avg Pkts/sec** | **Std Dev Pkts/sec** | **Avg Gbits/sec** | **Std Dev Gbits/sec** |
|---|---|---|---|---|---|
| Generic | Maps | 240 852 | 1 923 | 1.06 | 0.01 |
| Generic | Compiled | 254 108 | 2 412 | 1.23 | 0.04 |
| Driver | Maps | 430 425 | 14 806 | 3.43 | 0.08 |
| Driver | Compiled | 472 686 | 17 130 | 4.03 | 0.01 |
| Offload | Maps | 538 437 | 2 883 | 5.01 | 0.03 |
| Offload | Compiled | 616 196 | 3 002 | 5.86 | 0.25 |

Table 5.4: eBPF-IDS Attack Load Balance Performance Evaluation

In this test, the difference between XDP driver mode and XDP offload is now noticeable. Placing our solution in its designed environment is the only way to measure its true effectiveness. The offload solution performance increase, from driver mode, with ML in maps and compiled, is 20.06% and 23.29% respectively. This is above 100,000 packets per second increase.

Within this domain, it would also be interesting to examine the data in the receive buffers. The higher the values, the greater is degradation in performance, as the data is put on the buffer but not called to the application fast enough.

Figure 5.5 depicts the receive queue values when only using the iPerf3 tool. Figure 5.6 shows this same trend, however, when using iPerf3 and hping3. Table 5.5 presents the average and standard deviation of these values.

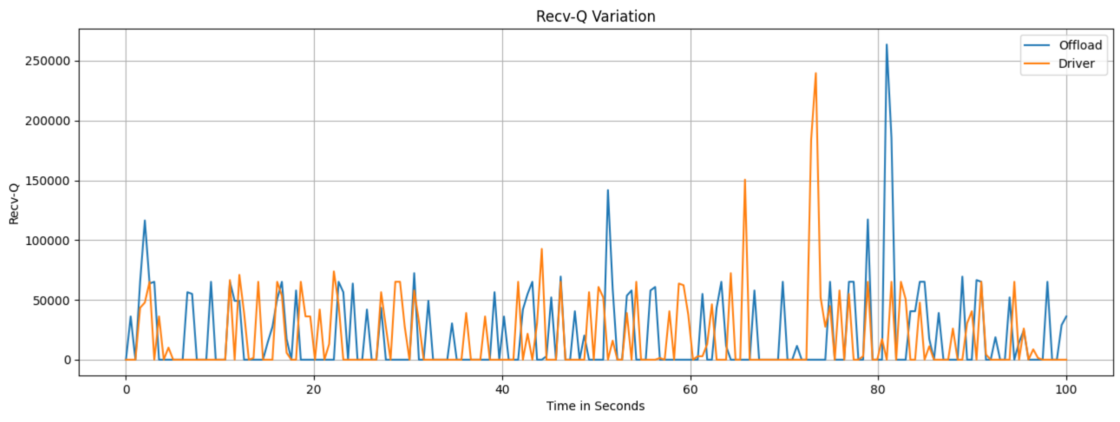


Figure 5.5: Receive Buffer with iPerf3

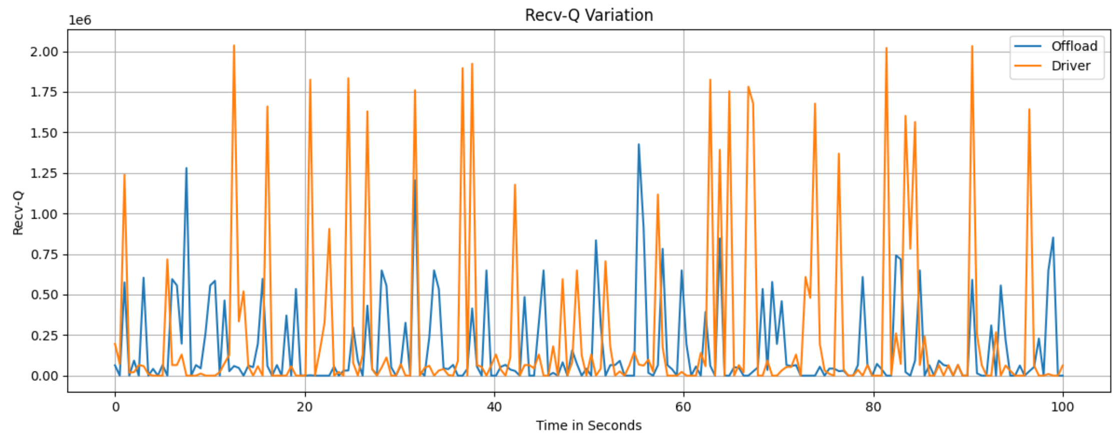


Figure 5.6: Receive Buffer with iPerf3 and hping3

With only the iPerf3, both solutions operate similarly. However, when in the presence of the attack, even though possessing a bigger load of traffic, the offloaded solution still possesses, in the receive queues, less data than driver mode. This indicates that the partial offload approach moves data from the queues to the application much faster.

| Tool | XDP Mode | Average Data in Recv-Q | Standard Deviation of Data in Recv-Q |
|---|---|---|---|
| iPerf3 | Offload | 19 978 | 35 153 |
| iPerf3 | Driver | 18 015 | 32 644 |
| iPerf3 + hping3 | Offload | 164 782 | 270 215 |
| iPerf3 + hping3 | Driver | 259 859 | 535 613 |

Table 5.5: eBPF-IDS Receive Queue Data

### 5.4.3 CPU Usage Analysis

Another metric worth analysing is the CPU usage between the solution in driver mode and offloaded mode. This was measured using the `top` command. Given our partial offload, the rest of the solution runs in the driver. It would be interesting to see if, in the simulated scenario, there is decreased CPU usage. To measure this, both solutions were placed under the same load by iPerf3, 6 Gbits per second.

Figure 5.7 depicts the CPU usage trend in user space and kernel of the solutions using the compiled model. Table 5.6 shows the average of these measurements.

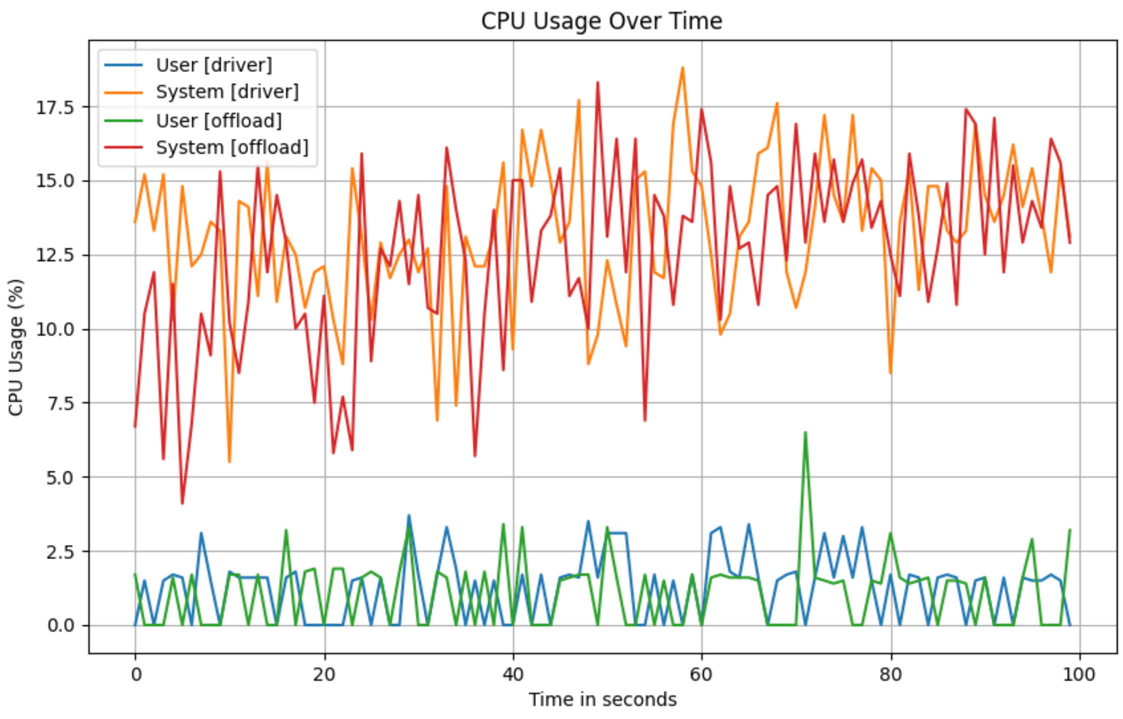


Figure 5.7: eBPF-IDS CPU usage over time comparison

| **Space** | **XDP mode** | **Average CPU usage (%)** | **Standard Deviation** |
|---|---|---|---|
| User | Driver | 1.28 | 1.062 |
| System | Driver | 13.221 | 2.45 |
| User | Offload | 1.088 | 1.15 |
| System | Offload | 12.553 | 3.023 |

Table 5.6: eBPF-IDS CPU usage

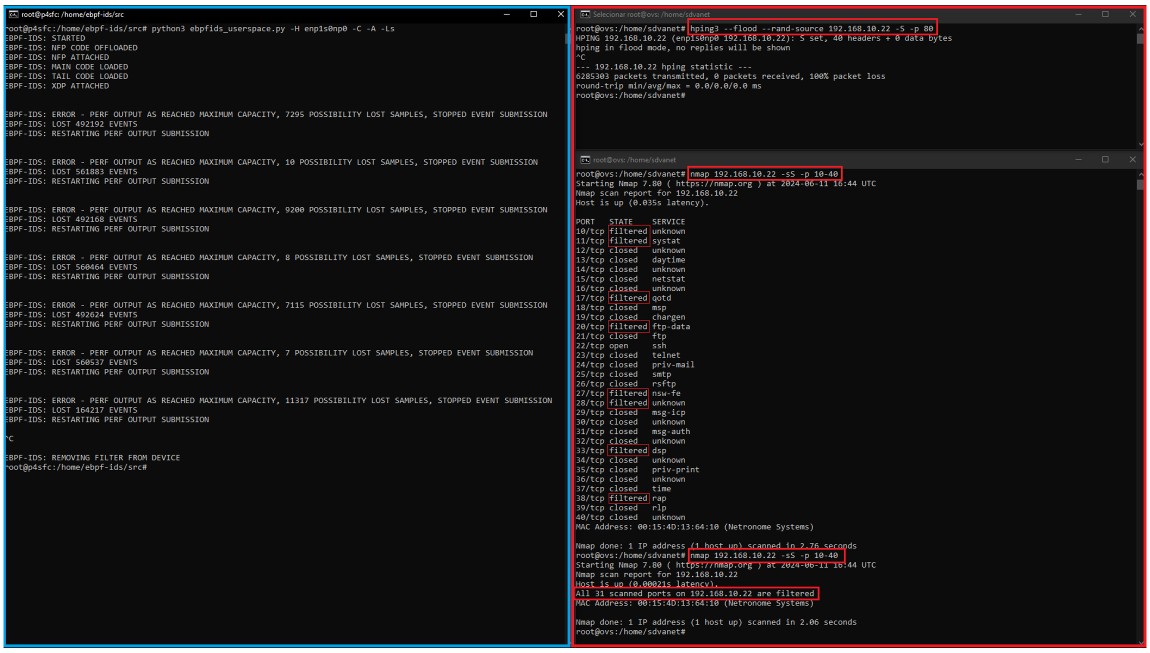
We can see that the trends overlap, and the average difference is very similar, saving less than 1% in CPU usage from the driver to the offloaded solution. The part of the program that is offloaded is small, therefore it seems that our solution does not have much benefit when it comes to CPU usage.

## 5.5 eBPF-IDS Resilience Tests and Known Problems

The previously conducted tests were made in scenarios where the solution was designed to work properly. However, in a real-world environment, the solution may face adversity. During development, careful attention was given to possible overflow events. It would be interesting to examine how performance and detection keep up in these situations.

Using hping3, a flood scenario was tested. Each packet sent equals the one of a port scan attempt therefore triggering perf output events. As expected, an overflow occurs which triggers the process of waiting for the ring buffer to stabilise. During this process, our solution works as expected. No events are generated but the IDS keeps detection active. When using an active response mode, scan attempts are still dropped. This is backed up, by analysing the output of a conducted scan, where it is stated that all ports are filtered. Figure 5.8 depicts this scenario. We can clearly see that the ring buffer is overflowed by the errors given, however, looking at the attacker terminal scans are blocked. In the first scan after enough probes, the ports' status becomes filtered. This is more evident as the second scan gives only filtered results even to ports previously identified as open.

It is important to note, however, an issue. Due to this high traffic, the flow table reaches maximum size and flows start being evicted. This can come at the cost of losing the context of important stored flows. In this situation, a normal flow may keep getting reinserted and therefore trigger a port scan alert. Additionally, this reinserted flow could also cause the offloaded table to exceed the size limit therefore resetting it. Finally, during the overflow of the ring buffer, the offload program cannot trust the active flow table, because there might be active flows that are not being received. So all traffic that is not in the active flow table is sent on RX RSS queue two. This may hinder performance as previously the load of the attack and normal traffic was split among queues. These issues are only important in the offload domain. In driver or generic mode this type of problem, related to the offload map, is not present.



```
root@p4sfc:/home/ebpf-ids/src# python3 ebpfids_userspace.py -H enp1s0np0 -C -A -Ls
EBPF-IDS: STARTED
EBPF-IDS: NFP CODE OFFLOADED
EBPF-IDS: NFP ATTACHED
EBPF-IDS: MAIN CODE LOADED
EBPF-IDS: TAIL CODE LOADED
EBPF-IDS: XDP ATTACHED


EBPF-IDS: ERROR - PERF OUTPUT AS REACHED MAXIMUM CAPACITY, 7295 POSSIBILITY LOST SAMPLES, STOPPED EVENT SUBMISSION
EBPF-IDS: LOST 492192 EVENTS
EBPF-IDS: RESTARTING PERF OUTPUT SUBMISSION


EBPF-IDS: ERROR - PERF OUTPUT AS REACHED MAXIMUM CAPACITY, 10 POSSIBILITY LOST SAMPLES, STOPPED EVENT SUBMISSION
EBPF-IDS: LOST 561883 EVENTS
EBPF-IDS: RESTARTING PERF OUTPUT SUBMISSION


EBPF-IDS: ERROR - PERF OUTPUT AS REACHED MAXIMUM CAPACITY, 9200 POSSIBILITY LOST SAMPLES, STOPPED EVENT SUBMISSION
EBPF-IDS: LOST 492168 EVENTS
EBPF-IDS: RESTARTING PERF OUTPUT SUBMISSION


EBPF-IDS: ERROR - PERF OUTPUT AS REACHED MAXIMUM CAPACITY, 8 POSSIBILITY LOST SAMPLES, STOPPED EVENT SUBMISSION
EBPF-IDS: LOST 560464 EVENTS
EBPF-IDS: RESTARTING PERF OUTPUT SUBMISSION


EBPF-IDS: ERROR - PERF OUTPUT AS REACHED MAXIMUM CAPACITY, 7115 POSSIBILITY LOST SAMPLES, STOPPED EVENT SUBMISSION
EBPF-IDS: LOST 492624 EVENTS
EBPF-IDS: RESTARTING PERF OUTPUT SUBMISSION


EBPF-IDS: ERROR - PERF OUTPUT AS REACHED MAXIMUM CAPACITY, 7 POSSIBILITY LOST SAMPLES, STOPPED EVENT SUBMISSION
EBPF-IDS: LOST 560537 EVENTS
EBPF-IDS: RESTARTING PERF OUTPUT SUBMISSION


EBPF-IDS: ERROR - PERF OUTPUT AS REACHED MAXIMUM CAPACITY, 11317 POSSIBILITY LOST SAMPLES, STOPPED EVENT SUBMISSION
EBPF-IDS: LOST 164217 EVENTS
EBPF-IDS: RESTARTING PERF OUTPUT SUBMISSION

^C

EBPF-IDS: REMOVING FILTER FROM DEVICE
root@p4sfc:/home/ebpf-ids/src#
```

```
root@ovs:/home/sdvanet# hping3 --flood --rand-source 192.168.10.22 -S -p 80
HPING 192.168.10.22 (enp1s0np0 192.168.10.22): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
^C
--- 192.168.10.22 hping statistic ---
6285303 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@ovs:/home/sdvanet#
```

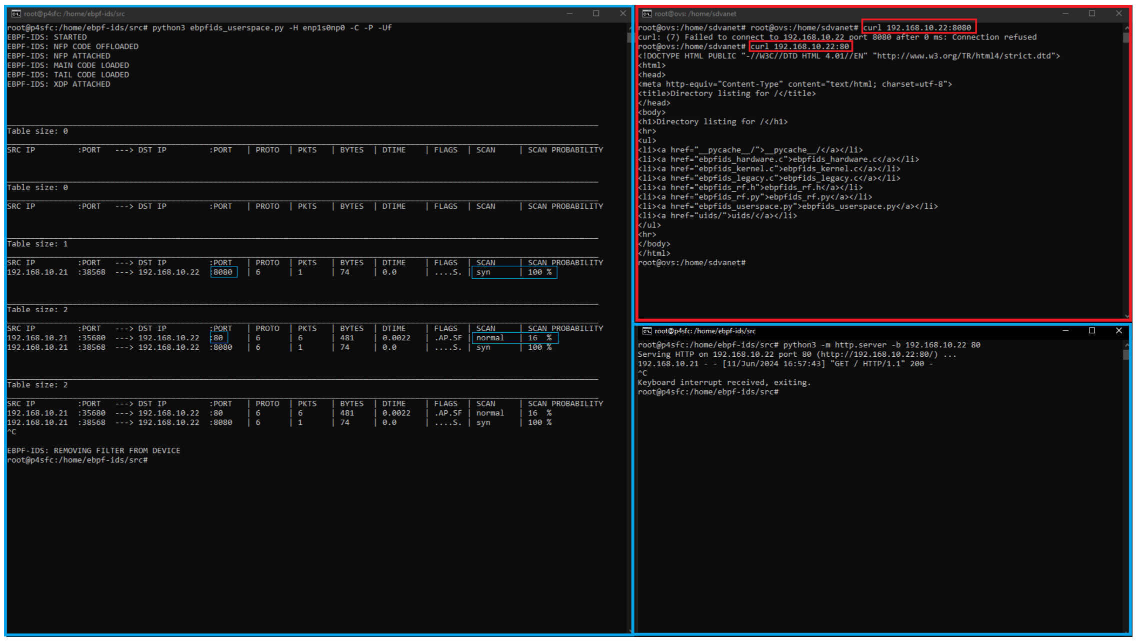
```
root@ovs:/home/sdvanet# nmap 192.168.10.22 -sS -p 10-40
Starting Nmap 7.80 ( https://nmap.org ) at 2024-06-11 16:44 UTC
Nmap scan report for 192.168.10.22
Host is up (0.035s latency).

PORT   STATE    SERVICE
10/tcp filtered unknown
11/tcp filtered systat
12/tcp closed   unknown
13/tcp closed   daytime
14/tcp closed   unknown
15/tcp closed   netstat
16/tcp closed   unknown
17/tcp filtered qotd
18/tcp closed   msp
19/tcp closed   chargen
20/tcp filtered ftp-data
21/tcp closed   ftp
22/tcp open     ssh
23/tcp closed   telnet
24/tcp closed   priv-mail
25/tcp closed   smtp
26/tcp closed   rsftp
27/tcp filtered nsw-fe
28/tcp filtered unknown
29/tcp closed   msg-icp
30/tcp closed   unknown
31/tcp closed   msg-auth
32/tcp closed   unknown
33/tcp filtered dsp
34/tcp closed   unknown
35/tcp closed   priv-print
36/tcp closed   unknown
37/tcp closed   time
38/tcp filtered rap
39/tcp closed   rlp
40/tcp closed   unknown
MAC Address: 00:15:4D:13:64:10 (Netronome Systems)

Nmap done: 1 IP address (1 host up) scanned in 2.76 seconds
root@ovs:/home/sdvanet# nmap 192.168.10.22 -sS -p 10-40
Starting Nmap 7.80 ( https://nmap.org ) at 2024-06-11 16:44 UTC
Nmap scan report for 192.168.10.22
Host is up (0.00021s latency).
All 31 scanned ports on 192.168.10.22 are filtered
MAC Address: 00:15:4D:13:64:10 (Netronome Systems)

Nmap done: 1 IP address (1 host up) scanned in 2.06 seconds
root@ovs:/home/sdvanet#
```

Figure 5.8: scan Detection with Ring Buffer Overflowed

The offloaded map must also be considered. As mentioned the time it takes to reset the full map is expensive. For that reason during that time, the kernel makes all classifications. While the map is being reset, the iPerf3 performance test is unaffected. Detection also works properly. However, if a flood is present because only one queue is being used, this may cause severe degradation due to the many flows entering the queue. There might be ways to tackle this problem, but if we start trying to include solutions for every potential abnormal situation, our development process will become overly complex and extend beyond the scope of a proof of concept. In a real-world scenario, however, we would need to consider these issues.

As the time to reset the offload map is long, we also decided to examine if the constant insertion of flows would affect performance. To test this, an HTTP page was opened and a script that curled that page in a loop was run. This script would trigger the insertion of a new flow on the offloaded map in every curl. iPerf3 performance test was unaffected.

Interestingly, each curl would result in a flow with a 16% probability of being a scan, this can be seen in Figure 5.9. This is because the first packet was similar to one of a port scan, however, the five remaining packets would declassify the flow to normal traffic. This percentage is not enough to determine that a false positive occurred. The constant behaviour of curling a web page does resemble a port scan attempt, as it keeps checking a port for activity. Our solution does alert for a port scan attempt after enough curls are performed. Additionally, in Figure 5.9, a curl is done to a different destination, which gets classified as an SYN scan, as it resembles one. No more packets are exchanged resulting in it remaining identified.



```
root@p4sfc:/home/ebpf-ids/src# python3 ebpfids_userspace.py -H enp1s0np0 -C -P -Uf
EBPF-IDS: STARTED
EBPF-IDS: NFP CODE OFFLOADED
EBPF-IDS: NFP ATTACHED
EBPF-IDS: MAIN CODE LOADED
EBPF-IDS: TAIL CODE LOADED
EBPF-IDS: XDP ATTACHED


__________________________________________________________________________________________________
Table size: 0
__________________________________________________________________________________________________
SRC IP        :PORT   ---> DST IP         :PORT   | PROTO  | PKTS   | BYTES  | DTIME    | FLAGS  | SCAN     | SCAN PROBABILITY


__________________________________________________________________________________________________
Table size: 0
__________________________________________________________________________________________________
SRC IP        :PORT   ---> DST IP         :PORT   | PROTO  | PKTS   | BYTES  | DTIME    | FLAGS  | SCAN     | SCAN PROBABILITY


__________________________________________________________________________________________________
Table size: 1
__________________________________________________________________________________________________
SRC IP        :PORT   ---> DST IP         :PORT   | PROTO  | PKTS   | BYTES  | DTIME    | FLAGS  | SCAN     | SCAN PROBABILITY
192.168.10.21 :38568  ---> 192.168.10.22  :8080   | 6      | 1      | 74     | 0.0      | ....S. | syn      | 100 %


__________________________________________________________________________________________________
Table size: 2
__________________________________________________________________________________________________
SRC IP        :PORT   ---> DST IP         :PORT   | PROTO  | PKTS   | BYTES  | DTIME    | FLAGS  | SCAN     | SCAN PROBABILITY
192.168.10.21 :35680  ---> 192.168.10.22  :80     | 6      | 6      | 481    | 0.0022   | .AP.SF | normal   | 16  %
192.168.10.21 :38568  ---> 192.168.10.22  :8080   | 6      | 1      | 74     | 0.0      | ....S. | syn      | 100 %


__________________________________________________________________________________________________
Table size: 2
__________________________________________________________________________________________________
SRC IP        :PORT   ---> DST IP         :PORT   | PROTO  | PKTS   | BYTES  | DTIME    | FLAGS  | SCAN     | SCAN PROBABILITY
192.168.10.21 :35680  ---> 192.168.10.22  :80     | 6      | 6      | 481    | 0.0022   | .AP.SF | normal   | 16  %
192.168.10.21 :38568  ---> 192.168.10.22  :8080   | 6      | 1      | 74     | 0.0      | ....S. | syn      | 100 %
^C

EBPF-IDS: REMOVING FILTER FROM DEVICE
root@p4sfc:/home/ebpf-ids/src#
```

```
root@ovs:/home/sdvanet# root@ovs:/home/sdvanet# curl 192.168.10.22:8080
curl: (7) Failed to connect to 192.168.10.22 port 8080 after 0 ms: Connection refused
root@ovs:/home/sdvanet# curl 192.168.10.22:80
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
<li><a href="__pycache__/">__pycache__/</a></li>
<li><a href="ebpfids_hardware.c">ebpfids_hardware.c</a></li>
<li><a href="ebpfids_kernel.c">ebpfids_kernel.c</a></li>
<li><a href="ebpfids_legacy.c">ebpfids_legacy.c</a></li>
<li><a href="ebpfids_rf.h">ebpfids_rf.h</a></li>
<li><a href="ebpfids_rf.py">ebpfids_rf.py</a></li>
<li><a href="ebpfids_userspace.py">ebpfids_userspace.py</a></li>
<li><a href="uids/">uids/</a></li>
</ul>
<hr>
</body>
</html>
root@ovs:/home/sdvanet#
```

```
root@p4sfc:/home/ebpf-ids/src# python3 -m http.server -b 192.168.10.22 80
Serving HTTP on 192.168.10.22 port 80 (http://192.168.10.22:80/) ...
192.168.10.21 - - [11/Jun/2024 16:57:43] "GET / HTTP/1.1" 200 -
^C
Keyboard interrupt received, exiting.
root@p4sfc:/home/ebpf-ids/src#
```

Figure 5.9: Curl with IDS active

## 5.6 Chapter Wrap-Up

This Chapter focused on examining the developed solution. It addressed the two approaches proposed to implement RF, its detection capabilities and performance.

From the two methods, it was concluded that the hard-coded solution operates faster than the one via Maps. However, the hard-coded solution comes at the cost of only being capable of loading smaller models.

The current solution can identify port scans. Its detection goes beyond the port scans it was trained on. As we moved from a binary to a multi-class model changes were observed in the detection. Selecting the best option depends on the final goals envisioned.

The performance of the solution was measured, showing the benefit of using the compiled model implementation and the difference in packets per second between XDP modes. Where driver and offload scored the best.

Finally, some tests that placed the developed proof of concept in abnormal scenarios was also done. This would allow to know the existing limitations more clearly.

# Chapter 6

# Conclusion

This Chapter serves as a reflective stage to evaluate the progress made throughout the project and to provide a conclusion that encapsulates the key findings. Additionally, Section 6.1, details what future work and exploration could be done.

From our analysis, it was possible to conclude that eBPF can be a great auxiliary to an IDS. This is due to where the eBPF can be placed, allowing not only for improved performance but increased visibility, both important characteristics that can improve an IDS.

From our research, it becomes apparent that this type of solution is a new area of study that requires more research. This investigation showed that there are various limitations, however, the authors either addressed, circumvented or deemed them acceptable. All solutions compared to their user space counterpart concluded that the eBPF solution was greater.

The practical development revealed that the integration of ML in eBPF is not only possible but is indeed a good approach. We were able to implement a solution capable of differentiating between network flows, classifying them as normal or as a port scan. When it came to the offload domain various limitations were encountered. These led to the development of a partial offload solution. We believe, however, that we still fulfilled that objective, as our final proposal, still operates in the hardware and employs eBPF and ML in that domain, detecting port scans. However, for the solution to be functional, more was required, placing part of the program in the kernel, and using some user space memory for alert generation.

The final IDS demonstrated its efficacy in successfully detecting the attacks it was trained for. Furthermore, its performance exhibited not only effectiveness but also promising potential real-world applications.

## 6.1 Future Work

We believe that the final solution meets all the objectives established and expectations, given that it is meant to be only a proof of concept. However, there is always room for improvement. One path that could be taken to improve the current solution is on the detection front. Improving the current dataset with more scan types, or searching for new datasets could be options to improve the amount of scan types the solution can detect.

For a real-world scenario, detecting more than only port scans would be ideal. Our solution is very focused on the domain of port scanning. Pivoting to a multi-class attack vector detection at this stage would be very difficult. The features required to detect scans are different from the ones required to detect other attacks. This transition would require all the research done, on the most effective dataset and ML model, to be conducted once again. However, from the gained insights, some strategies for this type of implementation can be thought out. The compiled version of the ML model is much faster but it only allows for smaller models. A good strategy would be to follow a hierarchical model, like the one implemented with the XDP offload solution, in the case that the model needed to properly detect multiple attack vectors, is too large. Training only the model to distinguish between the main attack classes, gives the possibility to make it smaller. Then build sub-models that can make the distinction within that specific class. This would also allow, to have these sub-models stored in eBPF maps, if needed, effectively splitting the model and taking advantage of the compiled model capabilities as much as possible.

When it comes to XDP offloading, our solution merged successfully. Allowing for ML in the hardware to make sense. However, other attack vectors may require features for detection not supported by the current hardware. Therefore, ML may not be a good strategy to follow, as the current state of the art is designed for more simple filter-like solutions.

# References

Dogukan Aksu and M Ali Aydin. Detecting port scan attempts with comparative analysis of deep learning and support vector machine algorithms. In *2018 International congress on big data, deep learning and fighting cyber terrorism (IBIGDELFT)*, pages 77–80. IEEE, 2018.

Akram QM Algaolahi, Abdullah A Hasan, Amer Sallam, Abdullah M Sharaf, Aseel A Abdu, and Anas A Alqadi. Port-scanning attack detection using supervised machine learning classifiers. In *2021 1st International Conference on Emerging Smart Technologies and Applications (eSmarTA)*, pages 1–5. IEEE, 2021.

NEMALIKANTI ANAND, MA SAIFULLA, and Pavan Kumar Aakula. High-performance intrusion detection systemusing ebpf with machine learning algorithms. 2023.

Stefan Axelsson. Intrusion detection systems: A survey and taxonomy. 2000.

Longe Olumide Babatope, Lawal Babatunde, and Ibitola Ayobami. Strategic sensor placement for intrusion detection in network-based ids. *International Journal of Intelligent Systems and Applications*, 6(2):61, 2014.

Maximilian Bachl, Joachim Fabini, and Tanja Zseby. A flow-based ids using machine learning in ebpf. *arXiv preprint arXiv:2102.09980*, 2021.

BCC pull request 5051. https://github.com/iovisor/bcc/pull/5051.

BCC Reference Guide. https://github.com/iovisor/bcc/.

BCC XDP Examples. https://github.com/iovisor/bcc/tree/master/examples.

Monowar H. Bhuyan, D.K. Bhattacharyya, and J.K. Kalita. Surveying port scans and their detection methodologies. *The Computer Journal*, 54(10):1565–1581, 2011. doi: 10.1093/comjnl/bxr035.

BPF & XDP Reference Guide. `https://docs.cilium.io/en/stable/bpf`.

BPF Hardware Offload Deep Dive Webinar. https://open-nfp.org/the-classroom/bpf-hardware-offload-deep-dive-webinar/.

Diego Couto de Carvalho et al. Detecção de intrusões em dispositivos de rede com o filtro de pacote berkeley. 2023.

Mehiar Dabbagh, Ali J. Ghandour, Kassem Fawaz, Wassim El Hajj, and Hazem Hajj. Slow port scanning detection. In *2011 7th International Conference on Information Assurance and Security (IAS)*, pages 228–233, 2011. doi: 10.1109/ISIAS.2011.6122824.

Gustavo de Carvalho Bertoli, Lourenço A Pereira, Cesar Marcondes, and Osamu Saotome. Evaluation of netfilter and ebpf/xdp to filter tcp flag-based probing attacks. In *Proceedings of XXII symposium on operational applications in defense area (SIGE)*, pages 35–39, 2020.

eBPF Documentation. `https://ebpf.io/what-is-ebpf`.

eBPF-IDS. https://github.com/joaolopix/ebpf-ids.

eBPF Offload Getting Started Guide (Netronome CX SmartNIC). https://help.netronome.com/support/solutions/articles/36000050009-agilio-ebpf-2-0-6-extended-berkeley-packet-filter.

Peter Manev Eric Leblond. Introduction to ebpf and xdp support in suricata. 2019.

Ever Deeper with BPF – An Update on Hardware Offload Support. https://www.netronome.com/blog/ever-deeper-bpf-update-hardware-offload-support/.

Carrie Gates. Coordinated scan detection.

Nutan Farah Haq, Abdur Rahman Onik, Md. Avishek Khan Hridoy, Musharrat Rafni, Faisal Muhammad Shah, and Dewan Md. Farid. Application of machine learning approaches in intrusion detection system: A survey. *International Journal of Advanced Research in Artificial Intelligence*, 4(3), 2015. doi: 10.14569/IJARAI.2015.040302. URL `http://dx.doi.org/10.14569/IJARAI.2015.040302`.

iPerf3. https://iperf.fr/.

Mahdi Nsaif Jasim, Ali Munther Abdul Rahman, and Muthanna Jabbar Abdulredhi. Machine learning classification-based portscan attacks detection using decision table. 2023. doi: 10.11591/ijeecs.v29.i3.pp1466-1472.

Magnus Karlsson and Jesper Brouer. Xdp (express data path) as a building block for other foss projects. In *FOSDEM 2019*, Brussels, 2019.

Jakub Kicinski. Bpf hardware offload deep dive, 09 2018. URL `https://www.netronome.com/technology/ebpf/educational-webinars/`.

Jakub Kicinski and Nicolaas Viljoen. ebpf hardware offload to smartnics: cls bpf and xdp. a.

Jakub Kicinski and Nicolaas Viljoen. Xdp hardware offload: Current work, debugging and edge cases. b.

Cynthia Bailey Lee, Christian Roedel, and Elena Silenok. Detection and characterization of port scan attacks. 2003. URL `https://api.semanticscholar.org/CorpusID:15075241`.

Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013.

Linux's uapi/linux/bpf.h, a. `https://elixir.bootlin.com/linux/v5.15.86/source/include/uapi/linux/bpf.h#L878`.

Linux's uapi/linux/bpf.h, b. `https://elixir.bootlin.com/linux/latest/source/include/uapi/linux/bpf.h#L6294`.

Batta Mahesh. Machine learning algorithms-a review. *International Journal of Science and Research (IJSR).[Internet]*, 9(1):381–386, 2020.

Peter Mell. Understanding intrusion detection systems. *IS Management Handbook*, pages 389–398, 2003.

Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8, 2018. doi: 10.1109/HPSR.2018.8850758.

Yawar Rasool Mir and Navneet Kaur Sandhu. Port scan detection using ai. 2019.

Steve Morgan. Cybercrime to cost the world $10.5 trillion annually by 2025. *Cybercrime Magazine*, 2020.

NCS Workshop. https://www.cisuc.uc.pt/en/ncs-workshop.

Netronome. `https://netronome.com`.

NFVSDN. https://nfvsdn2024.ieee-nfvsdn.org/.

Nmap Reference Guide. https://nmap.org/book/man.html.

Nmap Reference Guide - Host Discovery. https://nmap.org/book/man-host-discovery.html.

Jon Nordby, Mark Cooke, and Adam Horvath. emlearn: Machine Learning inference engine for Microcontrollers and Embedded Devices, March 2019. URL `https://doi.org/10.5281/zenodo.2589394`.

Giuseppe Ognibene. *Toward Efficient DDoS Detection with eBPF*. PhD thesis, Politecnico di Torino, 2021.

OneR Algorithm. https://www.saedsayad.com/oner.htm.

Racyus DG Pacífico, Marcos AM Vieira, Lucas FS Duarte, and José AM Nacif. Function as a service offloaded to a smartnic. In *2022 IEEE Latin-American Conference on Communications (LATINCOM)*, pages 1–6. IEEE, 2022.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Jason M Pittman. Machine learning and port scans: A systematic review. *arXiv preprint arXiv:2301.13581*, 2023.

P Pradhan and P Mannepalli. Machine leaning for flow based intrusion detection using extended berkley packet filter. *Int. J. Eng. Res. Curr. Trends*, 3:5–7, 2021.

README.sfportscan. `https://www.snort.org/faq/readme-sfportscan`.

Liz Rice. *Learning eBPF*. O'Reilly Media, Inc., 2023.

Markus Ring, Sarah Wunderlich, Dominik Gruedl, and Djourn Landes. Creation of flow-based datasets for intrusion detection. *Information Warfare*, 16, 2017, to appear.

Markus Ring, Sarah Wunderlich, Deniz Scheuring, Dieter Landes, and Andreas Hotho. A survey of network-based intrusion detection data sets. *Computers & Security*, 86:147–167, 2019.

RTCM. https://rtcm.inesctec.pt/eventos/35o-seminario/.

Amin Sadiq, Hassan Jamil Syed, Asad Ahmed Ansari, Ashraf Osman Ibrahim, Manar Alohaly, and Muna Elsadig. Detection of denial of service attack in cloud based kubernetes using ebpf. *Applied Sciences*, 13(8):4700, 2023.

T Saranya, S Sridevi, C Deisy, Tran Duc Chung, and MKA Ahamed Khan. Performance analysis of machine learning algorithms in intrusion detection system: A review. *Procedia Computer Science*, 171:1251–1260, 2020.

Scapy. https://scapy.net/.

Snort 3 Inspector Reference. `https://www.cisco.com/c/en/us/td/docs/security/secure-firewall/snort3-inspectors/snort-3-inspector-reference/port-scan-inspector.html`.

Support for hardware offload. https://github.com/iovisor/bcc/pull/2502.

Cássio Giordani Tatsch. Sistema de detecção e prevenção de ataques port scan em redes openflow sdn. 2017.

Shie-Yuan Wang and Jen-Chieh Chang. Design and implementation of an intrusion detection system by using extended bpf in the linux kernel. *Journal of Network and Computer Applications*, 198:103283, 2022. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2021.103283. URL `https://www.sciencedirect.com/science/article/pii/S1084804521002769`.

HD Wieren. Signature-based ddos attack mitigation: Automated generating rules for extended berkeley packet filter and express data path. Master's thesis, University of Twente, 2019.

# Appendices

# Appendix A

# eBPF Map Updates via Private Agilio SmartNIC Firmware

This Appendix will focus on the use of the private firmware provided by Netronome. It is intended to provide a more in-depth insight into the reason behind not following its use for the final solution.

The following Sections, A.1 and A.2, will entail the limitations of the firmware provided by the private documentation as well as the tests conducted with this tool, respectively.

## A.1 eBPF Offload Map Updates Limitation

Documentation was also provided, along with the private firmware. In it, it is justified the reason behind, the helper function required to make Map updates via the hardware, be private. The processing of network packets in the SmartNIC is handled differently than in the Kernel. Packets are processed in parallel.

This environment can lead to race conditions as map updates may start before the previous information is stored. To handle this, synchronisation is needed. This comes in the form of locks. Access to map values can only be done one at a time resulting in map actions having to wait, which will in consequence lead to degradation in performance.

## A.2 eBPF-IDS Offload Challenges and Drawbacks

To try and understand the possible impact of this feature on our proof of concept, a test was conducted. This test would immediately allow us to determine if the use of the feature was worth following. This test aims to compare a solution's performance deployed in the hardware and in the kernel.

The deployed solution could not be the already proposed proof of concept in its entirety, as some things would need to be changed. However, to make the assessment needed, we only need to examine the map update section of our solution. The deployed eBPF program does the simple task of keeping track of flows' characteristics. No classifications regarding intrusions are done. Note that Map writes, which are used in our solution had to be changed to Map updates, this also damages performance in XDP Driver and Generic. iPerf3 was used to generate traffic.

| **XDP Mode** | **Avg Pkts/sec** | **Std Dev Pkts/sec** | **Avg Gbits/sec** | **Std Dev Gbits/sec** |
|---|---|---|---|---|
| Generic | 324950 | 2375 | 3.76 | 0.03 |
| Driver | 577949 | 3732 | 6.70 | 0.05 |
| Offload | 2231 | 17 | 1.26 | 0 |

Table A.1: Performance Evaluation of the Private Firmware

From these results, it becomes clear the drastic difference between the map updated in the kernel compared to the hardware. Following this strategy would completely change the initial paradigm of improved performance through the use of eBPF. Additionally, there are other drawbacks to consider:

- Map type: there is no LRU variant for the offload domain. The Map used in this test is only the Hash type. Developing means of handling overflows would be necessary, through flow evictions.
- BPF time helper function: the `bpf_time_get_ns` is an eBPF helper function used to calculate the duration of a flow. This function is, however, not supported by the firmware. As documented in Section 4.1.2, duration is in fact not one of the most important features. Not using it for detection could be possible. However, one of our port scan thresholds is related to time. The absence of this function would make it impossible to calculate such a metric. In conclusion, not using time could hinder threat detection and decrease false positive detection.
- Atomic writes: one may argue that the function `__sync_fetch_and_add` could be used to increment the values of the characteristics of a flow. However, a new flow would still need to be added via the Map update function. Additionally, the implementation only allows incrementing of integers, given that many values are not of such type some method of storing and incrementing these values properly is needed. However, even if developed, the implemented atomic operations do not allow reads in eBPF programs, meaning that we can only write to these values and not read them to make a classification.

In conclusion, developing the solution using this strategy did not seem appropriate given all these results. Additionally, if further research is conducted on the developed proof of concept, the accessibility to the firmware could be a roadblock.

# Appendix B

# Agilio eBPF Firmware SmartNIC installation

To allow this document to be self-contained, except for the source code, documenting the process of installing the SmartNIC Firmware was considered relevant. The installation followed the documentation provided by the vendor [eBPF Offload Getting Started Guide , Netronome CX SmartNIC].

Note that the documentation provides a guide for different environments. The following is only for our own setup, Ubuntu. For this, kernel 4.17 or above is the recommended

## B.1 Installation

The firmware can be downloaded from the Netronome's Support page. Additionally, via the following command, it can also be obtained:

```
wget https://help.netronome.com/helpdesk/attachments/36019898763
```

When using the command, the file may not possess the correct name and extension. Please change it to `agilio-bpf-firmware-2.0.6.124-1.deb` to follow the next steps better. Note that, the firmware version number may be updated in the future, for that reason make the appropriate changes to not misidentify the version that is being used. The next steps are:

- Install the Firmware

```
dpkg -i agilio-bpf-firmware-2.0.6.124-1.deb
```

- Update NFP driver symbolic links

```
cd /lib/firmware/netronome
ln -s agilio-bpf/* .
```

- Remove and Reload the driver

```
modprobe -r nfp
modprobe nfp
```

By this stage, the Firmware should be loaded. The documentation also provides some commands to check if everything is working correctly. With the command `dmesg`, verify the logs for the presence of BPF. With the command `ip link` ensure the status of the interface is UP. Finally `ethtool -i $ETHNAME` should output the firmware version. The output of these commands can be seen on page 9 of [eBPF Offload Getting Started Guide , Netronome CX SmartNIC].

Additionally, iproute2 utilities and Clang are required. In the case that these are not present, they should be installed.

## B.2 Verification

The documentation demonstrates some programs that can be loaded to verify the NIC's functionality. However, given that our proof of concept uses BCC, the examples provided by this library can be used [BCC XDP Examples].

Note the small alteration needed for these programs to be loaded correctly, refer to Section 4.6.1. The device parameter should be passed as a `char pointer` and not as a string. Take as an example the file `xdp_drop_count.py`, changing line 44. Our proposed correction has been accepted [BCC pull request 5051], however, this error may be present in other sources.

```
27 offload_device = None
28 if len(sys.argv) == 2:
29     device = sys.argv[1]
30 elif len(sys.argv) == 3:
31     device = sys.argv[2]
32 
33 maptype = "percpu_array"
34 if len(sys.argv) == 3:
35     if "-S" in sys.argv:
36         # XDP_FLAGS_SKB_MODE
37         flags |= BPF.XDP_FLAGS_SKB_MODE
38     if "-D" in sys.argv:
39         # XDP_FLAGS_DRV_MODE
40         flags |= BPF.XDP_FLAGS_DRV_MODE
41     if "-H" in sys.argv:
42         # XDP_FLAGS_HW_MODE
43         maptype = "array"
44         offload_device = ctypes.c_char_p(device.encode('utf-8'))
45         flags |= BPF.XDP_FLAGS_HW_MODE
```

Listing B.1: BCC XDP example change