PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

HENRIQUE BECKER BRUM

LEVERAGING NETWORK-WIDE ORCHESTRATION IN PROGRAMMABLE NETWORKS FOR ENHANCED NIDS PERFORMANCE

Porto Alegre
2024

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

LEVERAGING NETWORK-WIDE ORCHESTRATION IN PROGRAMMABLE NETWORKS FOR ENHANCED NIDS PERFORMANCE

HENRIQUE BECKER BRUM

Master Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Tiago Coelho Ferreto

Porto Alegre
2024

ACKNOWLEDGMENTS

This work was supported by the PDTI Program, funded by Dell Computadores do Brasil Ltda (Law 8.248/91). The author acknowledges the High-Performance Computing Laboratory of the Pontifical Catholic University of Rio Grande do Sul (LAD-IDEIA/PUCRS, Brazil) for providing support and technological resources, which have contributed to the development and the results of this research.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ACRONYMS

ASIC – Application Specific Integrated Circuit

BMV2 – Behavioral Model version 2

CIDR – Classlesss Inter-Domain Routing

DDOS – Distributed Denial-of-Service

DOS – Denial-of-Service

FPGA – Field Programmable Gate Arrays

HIDS – Host-based Intrusion Detection System

IDS – Intrusion Detection System

IP – Internet Protocol

IPS – Intrusion Prevention System

IPV4 – Internet Protocol version 4

IPV6 – Internet Protocol version 6

LCA – Lowest Common Ancestor

MAT – Match-Action Table

NIDPS – Network Intrusion Detection and Prevention System

NIDS – Network Intrusion Detection System

P4 – Programming Protocol-Independent Packet Processors

PISA – Protocol Independent Switch Architecture

PDP – Programmable Data Plane

SDN – Software-Defined Networking

SRAM – Static Random-Access Memory

TCAM – Ternary Content-Addressable Memories

TCP – Transmission Control Protocol

UDP – User Datagram Protocol

VM – Virtual Machine

CONTENTS

POTENCIALIZANDO A ORQUESTRAÇÃO DE TODA A REDE PARA MELHORAR O DESEMPENHO DE NIDS EM REDES PROGRAMÁVEIS

RESUMO

Sistemas de Detecção de Intrusão em Redes (NIDSs) desempenham um papel crucial na proteção de redes contra ameaças cibernéticas, detectando atividades maliciosas e alertando os operadores de rede. No entanto, devido ao crescente volume de tráfego de rede, os NIDSs podem enfrentar problemas de saturação, especialmente na etapa de correspondência de padrões em NIDSs baseados em assinaturas. Para superar esse desafio, diversos estudos exploraram a transferência de regras de assinatura dos NIDSs para dispositivos com programabilidade no plano de dados (PDP), aproveitando sua alta capacidade de processamento de pacotes para pré-filtrar o tráfego antes de chegar aos NIDSs. No entanto, esses trabalhos apresentam duas limitações importantes. Primeiro, a maioria negligencia as restrições de memória dos dispositivos programáveis. Segundo, e mais importante, a grande maioria delega todas as capacidades de pré-filtragem a um único dispositivo. Para abordar essas limitações, este trabalho propõe a orquestração de toda a rede programável para pré-filtrar o tráfego destinado aos NIDSs, melhorando seu desempenho. O objetivo é aliviar o fardo nos NIDSs e aprimorar sua eficiência por meio da transferência estratégica de regras de assinatura para o PDP, direcionando apenas pacotes suspeitos para os NIDSs. Além disso, abordamos as limitações dos trabalhos de ponta empregando dois novos algoritmos de orquestração que levam em conta a memória e a topologia para distribuir estrategicamente as regras para vários dispositivos. A avaliação realizada demonstrou a eficácia desses algoritmos, superando o modelo tradicional de um único dispositivo e garantindo o encaminhamento estável e consistente do tráfego suspeito para o host do NIDS, mesmo em cenários com disponibilidade limitada de memória.

Palavras-Chave: NIDS, Desempenho, Plano de dados programáveis, Orquestração.

LEVERAGING NETWORK-WIDE ORCHESTRATION IN PROGRAMMABLE NETWORKS FOR ENHANCED NIDS PERFORMANCE

ABSTRACT

Network Intrusion Detection Systems (NIDSs) play a crucial role in safeguarding networks against cyber threats by detecting malicious activity and alerting network operators. Due to the escalating volume of network traffic, NIDSs are prone to saturation issues, particularly in the pattern matching stage of signature-based NIDSs. To overcome this, several studies have explored offloading NIDS signature rules to Programmable Data Plane (PDP) devices, leveraging their high packet-processing capacity to pre-filter network traffic for the NIDS. However, these works present two important limitations. First, most of them overlook the memory constraints of programmable devices. Second, and more importantly, the vast majority of them delegates all pre-filtering capabilities to a single device. Neglecting these aspects may prevent the offloading of all required signature rules compromising the effectiveness of the proposed pre-filtering approach. To address these constraints, this work leverages the network-wide orchestration in programmable networks to pre-filter traffic for the NIDS and enhance its performance. Our objective is to alleviate the burden on the NIDS engine and improve its efficiency by offloading NIDS signature rules to the PDP, redirecting only suspicious packets to the NIDS. Furthermore, we address the limitations of state-of-the-art work by employing two novel memory- and topology-aware orchestration algorithms to strategically offload the rules to multiple devices. The evaluation demonstrated the efficacy of the proposed algorithms, as they outperform the traditional single-device model, ensuring the stable and consistent forwarding of suspicious traffic to the NIDS host, even in scenarios with limited memory availability.

Keywords: NIDS, Performance, Programmable Data Plane, Orchestration.

1. INTRODUCTION

With the ever-growing usage of computers and the inherent vulnerabilities of connecting these devices to the Internet, protecting the network is fundamental for guaranteeing the correct functioning of computer systems and the security of sensitive data. According to Cybersecurity Ventures, by 2025, global cybercrimes will cost 10.5$ trillion dollars annually [40]. Despite these staggering costs, cybersecurity does not receive the investments required, especially in small businesses. In Verizon's 2021 Data Breach Investigations Report [41], 46% of all cyber breaches impacted businesses with fewer than 1,000 employees. Still, the average investment in cyber security in small businesses is less than 500 US$ [34] according to Juniper Research. These statistics show that cybercrimes are a massive problem in today's business. However, many companies are still reluctant to invest in cybersecurity, motivating researchers to develop solutions that enhance commercial-grade security software.

Multiple technologies have been developed to protect computer networks from attacks. A popular security tool to address cyberattacks is Network Intrusion Detection Systems (NIDS) [42]. NIDS can detect malicious traffic in the network using either signature-based, anomaly-based, or both techniques. In signature-based NIDS, incoming traffic is matched with a ruleset database to find common attack patterns. Due to the ever-increasing volume of network traffic today, NIDSs have difficulty scaling the signature rules matching process, and intrusions might go undetected [39]. Different studies [28, 17, 42] have examined the performance of three popular open-source NIDS (Snort, Suricata, and Zeek), and concluded that one of the biggest potential performance bottlenecks is the pattern matching stage. Therefore, to guarantee the proper functioning of NIDSs and protect networks against attacks, enhancements to NIDSs must focus on the pattern matching stage.

One way to improve the performance of the pattern matching stage is to offload NIDS capabilities to the network's data plane. This new idea was made possible with the inception of the Programmable Data Plane (PDP). In the PDP paradigm, programmers have complete control over how packets are processed and forwarded, what protocols are allowed, the network statistics to collect, and much more. To make data plane programming easy and flexible, domain-specific languages such as P4 [6] have been created. Several works [31, 20, 25, 37, 2, 1, 26] have explored enhancing the NIDS performance by offloading specific capabilities to the PDP. Among these, offloading NIDS rules to the data plane for pre-filtering network traffic for the NIDS host is a promising approach. By pre-filtering the network traffic destined for the NIDS engine, the number of packets entering the pattern matching stage is reduced, lowering the chance of it becoming a bottleneck and compromising the NIDS.

However, these solutions present important limitations. First, most of these works overlook the memory constraints of the forwarding devices when determining the rules for offloading, resulting in sizable rulesets to offload that may not fit within the available memory. Second, the majority of them only consider scenarios involving a single device for pre-filtering network traffic for the NIDS host. This approach does not take into account that computer networks are usually made up of multiple devices. By focusing on just one device, network resources are underutilized and the security system is vulnerable to the single point of failure (SPOF) problem. Additionally, relying on a single device means that the available memory space to offload NIDS rules is limited. This is not just because programmable switches have a small memory space, but also because the memory must be shared with other network needs, such as packet forwarding rules, which are constantly increasing [19], and other P4 programs. If all NIDS rules cannot be offloaded to the programmable device due to memory limitations, the effectiveness of the solution is compromised. In such cases, suspicious packets may not be forwarded to the NIDS, causing network attacks to go unnoticed and the benefits of using the data plane as a pre-filter lost.

To address the limitations outlined above, this work proposes the network-wide orchestration among PDP devices to pre-filter network traffic for the NIDS host. Our objective is to alleviate the burden on the NIDS engine and enhance its performance by distributing the NIDS signature rules to multiple P4 devices, redirecting only suspicious network traffic to the NIDS. In order to optimize the utilization of network resources, we introduce two network-wide orchestration algorithms that strategically offload rules to switches based on their available memory and placement within the network's topology. In this way, our approach aims to minimize the saturation on the NIDS host and enhance its performance by leveraging the network-wide orchestration of programmable nodes to pre-filter network traffic for the NIDS instance. In summary, our work presents the following contributions:

1. A revised and improved **P4 NIDS rules compiler** derived from the P4-ONIDS [37] rule compiler. This compiler transforms the NIDS rules into P4 table entries, allowing them to be offloaded to the data plane.

2. A **P4 data plane implementation** designed to receive the compiler table entries, monitor suspicious flow, filter incoming traffic, and mirror suspicious packets for the NIDS host.

3. A **network-wide table entries orchestrator** utilizing two novel algorithms to strategically offload compiled table entries to multiple devices while considering the resources of each device and the network topology.

4. An **extensive evaluation**, including the selection of the optimal P4 data plane parameters, and the evaluation of the network-wide offloading algorithms against the baseline method in different topologies and memory availability scenarios.

The evaluation of the proposed solutions demonstrated that, by pre-filtering the network traffic for the NIDS, the volume of packets reaching the NIDS host significantly reduced, alleviating the strain on the pattern matching stage. However, the number of generated alerts using the pre-filtering approach decreased when compared to scenarios without pre-filtering. Regarding the evaluation of the network orchestrator, our network-wide algorithms with memory and topology awareness exhibited superiority over the traditional model, which relies on a single device, especially in scenarios with limited memory availability. The proposed algorithms consistently forwarded the same amount of suspicious traffic to the NIDS host in all memory availability scenarios, resulting in a stable number of alerts. In contrast, the traditional approach failed to maintain a constant number of packets forwarded to the NIDS, leading to a decline in the number of generated alerts. The stability of the proposed approach stems from the strategic offloading of table entries to multiple devices, ensuring that a substantial number of entries are offloaded even in scenarios with restricted memory availability.

The remainder of this document is structured as follows. In Chapter 2, we discuss the theoretical foundation necessary to understand the concepts of NIDS and PDP. Following that, Chapter 3 presents a comprehensive review of related work regarding the offloading of NIDS capabilities to the PDP and network-wide cooperation with PDP devices. Chapter 4 introduces the P4-ONIDS [37] rules compiler work and outlines the enhancements and modifications made to it. Subsequently, the proposed solution is detailed in Chapter 5, covering the general architecture, the functioning of the P4 data plane, and the network-wide orchestrator with its table entries offloading algorithms. Chapter 6 then presents the evaluation of the parameters of the P4 data plane and the evaluation of the network orchestration algorithms. Finally, Chapter 7 concludes this work, highlighting important limitations and proposing future work directions.

2. BACKGROUND

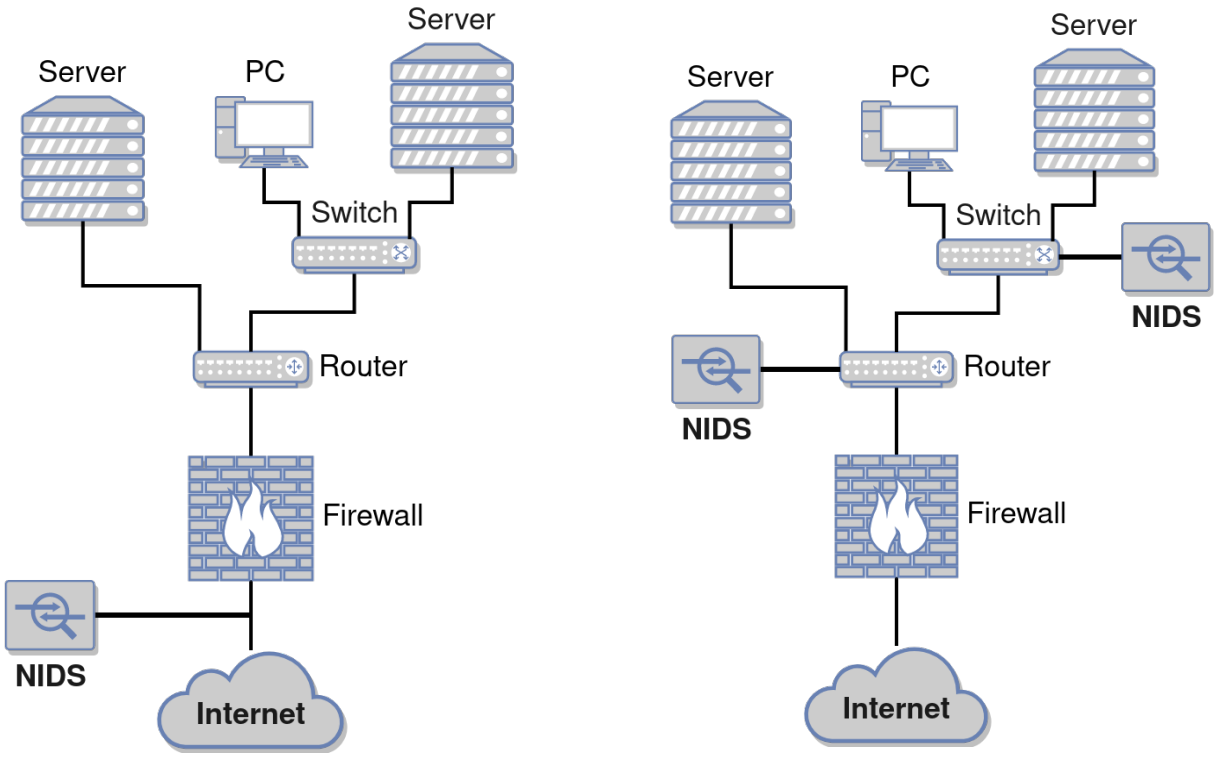
This chapter presents the main topics addressed in this work. First, it presents the concept of Network Intrusion Detection Systems, as well as the main technologies and techniques employed in this area in Section 2.1. Then, it gives a brief overview of the Software-Defined Networking paradigm in Section 2.2. Next, in Section 2.3, this chapter introduces the Programmable Data Plane and explains in detail the P4 language. Finally, it details the Count-Min sketch, a probabilistic and space-efficient data structure in Section 2.4.

2.1 Network Intrusion Detection Systems

Intrusion Detection Systems (IDSs) have become a key technology to protect communication infrastructures [18]. An IDS is a software or hardware system that identifies malicious actions in computer systems to allow the security of the system to be maintained [23]. It is crucial to differentiate an IDS from an Intrusion Prevention System (IPS). An IDS receives the mirrored traffic and alerts network operators if an intrusion is detected. In contrast, an IPS receives the original network traffic and takes preventive actions if an attack is identified.

There are two main types of IDSs based on the class of monitored events: Host-based Intrusion Detection System (HIDS) and *Network Intrusion Detection System* (NIDS). A HIDS monitors and analyzes the internal systems activities (e.g., application activity, system logs) and the host's incoming and outgoing network traffic. Differently, NIDSs are found at specific points on the network to capture and analyze the stream of packets going through a network link [16]. NIDSs can further be categorized regarding the type of deployment. According to Kumar et al. [24], a NIDS can have an early warning, internal, or every host deployment.

In an early warning mode deployment, the NIDS is employed outside the perimeter of the firewall as depicted in Figure 2.1a. The main advantages of such deployment are that the NIDS monitors all traffic entering the network, protecting multiple hosts/devices. Furthermore, since only one NIDS is used, management is easy. The last advantage is the capability of the NIDS to detect attacks targeting the firewall since it monitors the inbound pre-firewall traffic. There are three main disadvantages of this deployment. First, the NIDS can be easily saturated since, typically, only one device with NIDS capabilities is used, and it receives all pre-firewall network traffic. Second, network attacks from one host to another can go unnoticed as the NIDS only monitors the inbound and outbound traffic, not the traffic within the network. Finally, because both the NIDS and the firewall receive the



(a) Early warning deployment

(b) Internal deployment

Figure 2.1: NIDS deployment types

same inbound traffic, conflicting decisions can exist where the firewall classifies certain traffic as malicious while the NIDS categorizes it as benign or vice-versa.

The internal deployment involves placing an NIDS close to the switching nodes or access routers. In this way, the NIDS examines the network traffic passing through any given link. This scenario is shown in Figure 2.1b. One advantage of this deployment is that the NIDS will not have contrasting decisions with the firewall for inbound traffic, as the NIDS receives the traffic already filtered by the firewall. Moreover, saturation becomes harder as there is one NIDS deployed for each link/device. The main disadvantages are maintaining and updating multiple NIDSs, and the firewall is now susceptible to attacks.

The last type of deployment is when every host in the network has its own NIDS, as illustrated in Figure 2.2. Despite being similar to HIDS, the NIDS in each host is decoupled from the operating system and only manages network traffic. This decoupling from the operating system allows a network administrator to manage the multiple NIDSs from a central location. This scenario provides the most protection from all deployments, since every host has a dedicated NIDS to monitor attacks. However, this distributed layout incurs complex management of the NIDS instances, especially in large networks. Furthermore, network-wide attacks will go unnoticed if no communication exchange occurs among NIDS instances.

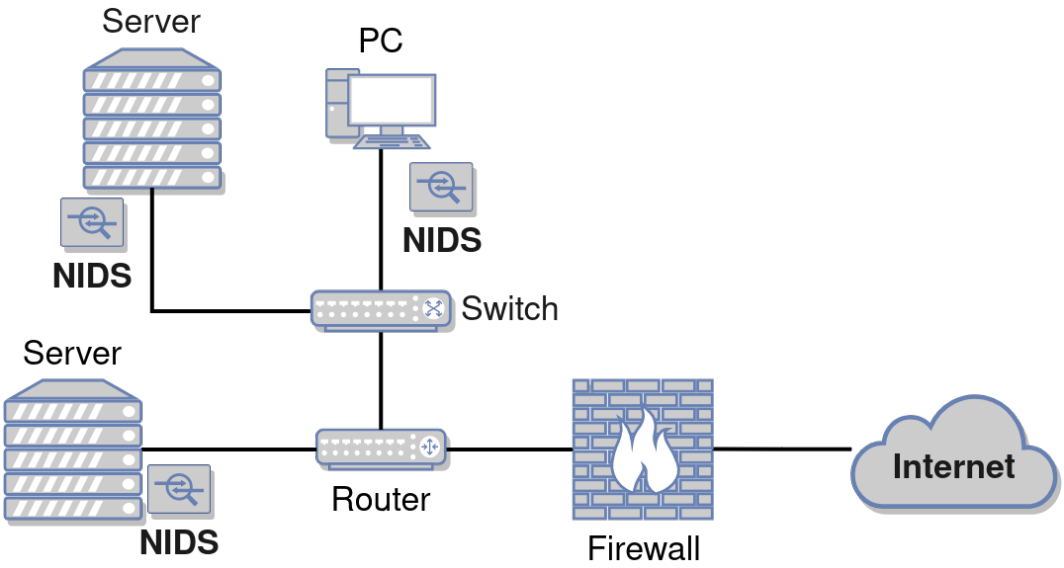


Figure 2.2: Every host NIDS deployment

2.1.1 Signature and Anomaly-Based Detection

The two primary approaches to detecting intrusions in NIDSs are signature-based and anomaly-based. A signature-based NIDS uses pattern matching techniques to find a known attack. In this model, the network traffic is parsed and compared to a database containing rules that characterize the profile of security threats, such as a malware or a DoS attack. If a match is detected, an alert is sent to network administrators so they can perform corrective actions. This technique is explained in more detail in Section 2.1.2, since it is fundamental to our work. Anomaly-based NIDSs appeared as a solution to some limitations of the signature-based model (i.e., detecting an unknown attack).

Anomaly-based NIDSs start by defining a network's normal traffic behavior through artificial intelligence, statistical, or knowledge-based methods, and then comparing this normal traffic to the current traffic. Any significant deviation between normal behavior and observed behavior is regarded as an anomaly, which can be interpreted as an intrusion [23]. The anomaly detection model must continuously update itself to detect intrusions properly. The biggest challenge for anomaly-based systems is defining the normal behavior of a network. If not well-tuned, an anomaly-based NIDS can overflow the network operator with false positives, causing actual attacks to pass unnoticed. Systems that employ both signature- and anomaly-based detection are called hybrid NIDSs.

2.1.2 NIDS Signature Rules

Open-source NIDSs, such as Snort[1], Suricata[2], and Zeek[3], use an extensible rule-based language syntax that allows users to specify signature rules. Besides signatures written for a specific use case, there are also publicly available signatures rulesets of familiar attacks. Despite having a signature language, most public signature rulesets are not made for Zeek, since signatures are not its central defense system. Because of this, in this chapter we will explain only the composition of rules used by Snort and Suricata.

Each signature rule describes the characteristics of a malicious activity and the corresponding action. Snort and Suricata rules have two parts: the rule header and the rule body. The rule header defines the action to take upon any matching traffic, as well as the protocols, network addresses, port numbers, and traffic direction to which the rule should be applied. The rule body defines the informational message associated with a given rule and, most importantly, the payload and non-payload criteria that must be met for a packet to match the rule. A signature rule consists of the following elements:

action proto src_addr src_port direction dst_addr dst_port (body)

The "action" field specifies the operation to perform when a packet matches the rule. Table 2.1 presents some possible actions in Snort. The "proto" field is the networking protocol to match, e.g., IPv4, ICMP, TCP.

Table 2.1: Possible actions for Snort signature rules

Action	**IDS/IPS**	**Description**
alert	both	Generates an alert
block	IPS	Blocks all packets of this flow
drop	IPS	Drops the current packet
log	both	Logs the current packet
pass	IPS	Marks the packet as passed
react	IPS	Responds to a client and terminates a session
reject	IPS	Terminates a session with *TCP Reset* or *ICMP Unreachable*
rewrite	IPS	Enables overwriting a packet’s content

The "src_addr" and "dst_addr" fields indicate the IP addresses for the source and destination, respectively. It is possible to declare an IP address in four ways:

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

- The keyword "any", which means any IP address.
- As an IP address with an optional CIDR block (e.g., "192.168.0.5", "192.168.1.0/24").
- As a variable defined in the Snort configuration file that specifies a network address or a set of network addresses (e.g., "$EXTERNAL_NET", "$HOME_NET").
- A combination of IP addresses, IP address variables, or both, enclosed in square brackets and separated by commas (e.g., "[192.168.1.0/24,10.1.1.0/24]").

The "src_port" and "dst_port" fields specify the source and destination ports, respectively. They can be declared in any of the following ways:

- The keyword "any", meaning any port.
- As a static port (e.g., "80", "445", "21").
- As a variable defined in the Snort configuration that specifies a port or set of ports (e.g., "$HTTP_PORTS").
- As port ranges indicated with the range operator ":" (e.g., "1:1024", "500:").
- A list of either static ports, port variables, port ranges, or all of them, enclosed in square brackets and separated by commas (e.g., "[1:1024,4444,$HTTP_PORTS]").

Note that the "src_addr", "src_port", "dst_addr and "dst_port" fields can have the special "!" negation operator, which will process traffic that does not match the field's value. The "direction" parameter is used to define the direction of the traffic that the rule should apply. It can be unidirectional ("–>") or bidirectional ("<–>"). The unidirectional denotes that the IP addresses on the left side and the port pair represent the source and the right one represents the destination. Bidirectional indicates that the two IP addresses and port pairs can be either source or destination.

Table 2.2: Common Snort options

Option	**Description**
msg	The message to be printed out when a rule matches
sid	The unique signature number assigned to a given rule
classtype	A classification to the rule indicating the type of attack associated with an event
priority	Sets the severity level for appropriate event prioritizing
metadata	Additional and arbitrary information for a rule
content	Used to perform basic pattern matching on strings, hexadecimal, or both
distance	Informs *content* where to start searching for a pattern

Finally, the rule body or rule options defines a series of options. Each option has its own set of option-specific criteria, but the general structure is the same. First, all options are enclosed in parentheses after the rule header. Then, each rule option is declared with its name followed optionally by a ":" character and any option-specific criteria. Lastly, each rule option ends with a ";" character. The options are divided into four groups: General options provide additional context; Payload options are for payload-specific criteria; Non-payload is for non-payload criteria (mainly header related); Post-detection options are triggered after a rule has matched. Table 2.2, presents some common Snort options.

As an example of a Snort and Suricata signature rule, suppose that a network administrator desires to receive an alert for any Domain Name System (DNS) messages containing the string "yourdomain.com" as the Uniform Resource Locator (URL). The signature rule would be the following:

alert dns any any -> any any
(msg:"DNS LOOKUP for yourdomain.com"; dns.query; content:"yourdomain.com"; sid:1;)

2.1.3 Open-Source NIDS

This subsection presents an overview of the architecture and design of the most popular open-source NIDSs: Snort, Suricata, and Zeek (Bro).

Snort

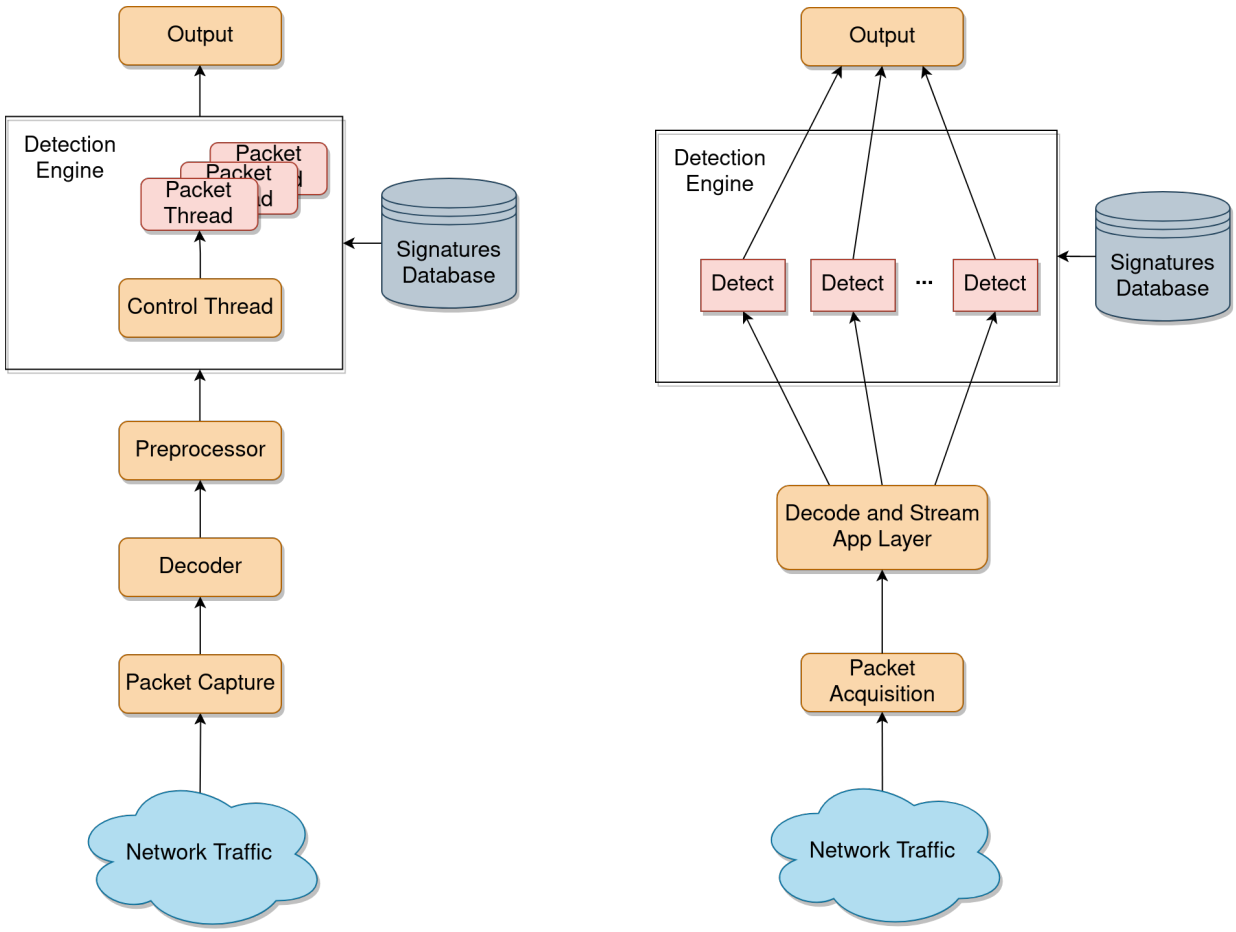
Snort [35] is one of the most widely used signature-based Network Intrusion Detection and Prevention System (NIDPS), that supports both IDS and IPS modes [42]. It was created in 1998 by Marin Roesch and is now maintained by Cisco Systems with the contribution of a substantial and active community of users, special interest groups, and developers. Snort can sniff network packets, log packets to the disk, compare monitored traffic against signature rules, and present attack statistics on the console.

Snort's architecture is depicted in Figure 2.3a. First, the packets are captured with *Libcap* and sent to the *Decoding Module*. Next, the *Pre-processor* normalizes the packets in a format the *Detection Engine* can comprehend. The *Detection Engine* compares the signatures rules with the traffic to detect malicious activity. If a packet matches a rule, the appropriate action is taken. Older Snort versions only supported single-thread detection modules, but Snort 3 now supports multi-thread detection, enhancing the matching capabilities. Finally, the *Output* module saves the logs and alerts.

Suricata

Suricata is a high-performance signature-based NIDPS. It was initially funded by the Department of Homeland Security's Directorate for Science and Technology [44] and is currently maintained by the Open Information Security Foundation (OISF), a 501(c)3 non-profit foundation. Suricata uses the same rule format as Snort and similar detection algorithms. The main advantage of Suricata over Snort is that Suricata was designed to support multiple detection engines via multi-threading. Despite Snort 3 adding multi-thread support to its design, A. Walled et al. [42] conducted an extensive open-source NIDPS evaluation, which showed that Suricata still outperforms Snort 3, even with its new multiple detection modules.

Suricata's architecture is shown in Figure 2.3b. The *Packet Capture* module collects packets from the network, which are then forwarded to the *Decode and Stream Application Layer*, where they are decoded. Next, the *Detection Engine* threads compare the packets with the *Signature Database* to find matches and emit alerts if a match is found. Finally, the *Output* module processes the alerts and events into statistics so that users can better understand the network's behavior.



(a) Snort (b) Suricata

Figure 2.3: Open-source NIDS architectures

Zeek (Bro)

Zeek, formerly Bro, is a network analysis framework for inspecting network traffic against malicious activities. Zeek is not a classic signature-based NIDS; while it supports such standard functionality, Zeek's scripting language facilitates a much broader spectrum of different approaches to finding malicious activity. These include detection of semantic misuse, anomaly detection, and behavioral analysis. In addition to these custom functionalities, Zeek provides some out-of-the-box ones. Zeek's built-in defense mechanisms encompass a wide range of functionalities, including, but not limited to, extracting files from HTTP sessions, detecting malware by interfacing with external registries, reporting vulnerable versions of software seen on the network, identifying popular web applications, detecting SSH brute-forcing, validating SSL certificate chains, etc.

Zeek's architecture is highly scalable in that performance improvements can be easily achieved by dedicating more hardware resources to workers and the manager [42]. Figure 2.4 shows the two main components of Zeek's architecture, the *Event Engine* and the *Script Interpreter*. The *Event Engine* reduces the stream of incoming packets into higher-level events. These events reflect network activity in policy-neutral terms. The *Script Interpreter* executes a set of event handlers written in Zeek's custom scripting language. Zeek's scripts allow users to specify security actions according to the events captured.

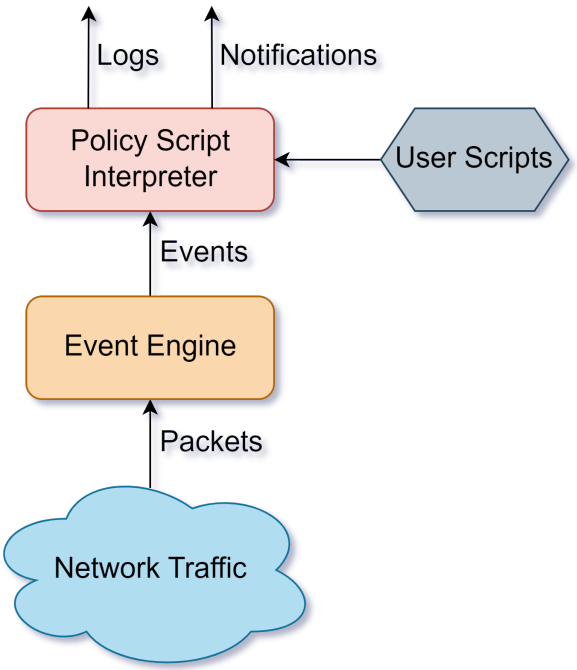


Figure 2.4: Zeek's architecture

2.1.4 NIDS Performance Limitations

NIDSs play an important role in the detection of malicious activities in computer networks. However, due to the increase in link capacity and speeds, NIDSs face the computational challenge of handling higher traffic volumes and performing complex per packet rules [44]. Several studies have tried to find and reduce the factors that affect the performance of NIDS. The authors of [17] evaluated the performance of Snort, Suricata, and Bro (now Zeek) using their default setting and an optimized version on a 10Gb/s network. They demonstrated that the default settings of all open-source NIDSs are unsuitable, as these settings incur a high CPU usage and packet drop rate. Even with the proposed optimizations, the CPU usage was still very high for Snort, although it dropped fewer packets than with the default setting. For Suricata, CPU usage increased slightly with optimizations, but the number of dropped packets was reduced. Bro does not allow the user to modify the pattern matching algorithm, so it was not possible to verify the effectiveness of the optimizations. Overall, Suricata achieved the best results, since it employs multi-threading.

In [18], the authors studied the feasibility of popular open-source NIDS, including Snort and Suricata, on a 100Gb/s network without relying on new packet capture mechanisms or updating existing hardware. Their results have shown that NIDSs can keep accuracy high, but only by considerably increasing CPU usage. In [42], a comprehensive performance evaluation of Snort, Suricata, and Zeek was conducted in both the IDS and IPS modes. Similar to [18], their results demonstrate that CPU usage starts to saturate as throughput increases. Moreover, they showed that packets spend most of the time inside the detection engine and that selecting an efficient pattern-matching algorithm will enhance the performance of the NIDPS.

As elucidated in [42], the most consuming task of a signature-based NIDS is comparing the packets and the signature rules. With the advent of SDN and PDP, researchers have proposed offloading NIDS capabilities from the host to other network nodes to alleviate the work done by the NIDS host. One proposed solution is to use the forwarding devices as a pre-filter mechanism. There are two main ways to achieve this. First, offloading a whitelist of flows to the devices and forwarding only unknown or malicious traffic to the NIDS. This idea is only possible when the network’s benign traffic is small and well-known. The second and more realistic approach is to offload signature rules, which identify suspicious traffic, to programmable devices. Only the traffic that matches these rules is then forwarded to the NIDS host for further examination. Regardless of the approach, the benefits are the same: reduced workload on the NIDS host and increased security scalability.

2.2 Software-Defined Networking

Software-Defined Networking (SDN) delineates a clear separation between the control plane and the data plane while consolidating the control plane so that a single centralized controller can control multiple remote data planes [22]. The SDN paradigm is based on the following ideas: decoupling of the control plane from the data plane; the forwarding decision can be based on other features instead of the destination address; the network control is logically centralized; and network functionalities are programmable through applications running in combination with the controller. Due to the above-mentioned characteristics, SDN is ideal for today's applications' high-bandwidth and dynamic nature. Figure 2.5 presents the SDN reference model according to the Open Networking Foundation (ONF), a nonprofit consortium dedicated to the development, standardization and commercialization of SDN.

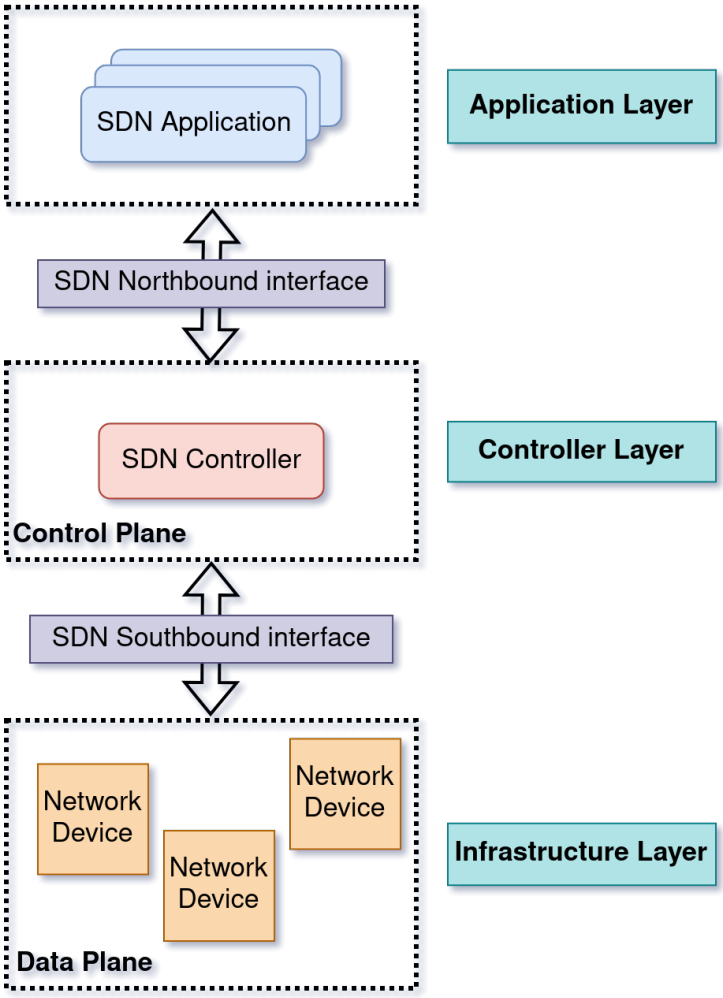


Figure 2.5: SDN reference model.

One of the first proponents of SDN was the *ForCES* [46] framework released by the Internet Engineering Task Force (IETF). The idea of ForCES was to standardize the in-

formation exchange between the control and forwarding planes, allowing them to become physical-separated standard components. Following the ideas proposed by ForCES, *OpenFlow* [30] was introduced in 2008 and since then has become the de facto protocol in SDN for communication between the controller and the forwarding devices [32]. The OpenFlow protocol is rooted in the fact that most modern Ethernet switches and routers contain flow tables (typically built from TCAMs) that run at line rate to implement firewalls, NAT, QoS and to collect statistics. Despite each vendor's flow table being different, there is a set of common functions in them that OpenFlow exploits. With OpenFlow, the controller can populate the flow tables of switches according to the network administrator's needs. At the same time, the controller can query network statistics from the OpenFlow switch. Although OpenFlow started as a solution for academia to run experiments on the university network, it has gained significant traction in the industry over the past few years.

The OpenFlow protocol and the SDN paradigm present certain advantages over the traditional network model: greater control of a network through programming, permitting real-time centralized control of a network based on instantaneous network status and user-defined policies, and offering a convenient platform for experimentation of new techniques [45]. Despite these benefits, the traditional SDN model has some important limitations. Specifically, the data plane is not programmable, as it is limited to a fixed set of protocols and actions made available by the forwarding devices and provided by the controller through an API [15]. Moreover, SDN suffers from scalability and performance issues because SDN switches heavily depend on the control plane to forward the packets, which increases the data control communication overhead [21]. To address these limitations, a new branch of SDN, called the *Programmable Data Plane* (PDP), was introduced.

2.3 Programmable Data Plane

With the advent of the SDN paradigm, the programmability of the control plane became possible. However, in the traditional SDN model, the data plane is not programmable and is highly dependent on the control plane. To address these restrictions, the idea of a programmable forwarding device was first proposed through the *Protocol Independent Switch Architecture* (PISA) [7]. Since the inception of PISA, other architectures have been created based on it to address other needs, such as the *SimpleSumeArchitecture* for FPGA-based targets [15]. In the PISA model, the packets go through a set of stages, each programmable by the user. The PISA architecture is illustrated in Figure 2.6 and contains the following elements: a programmable parser, a programmable match-action pipeline, and a programmable deparser.

1. *The programmable parser* allows programmers to define how protocol headers (custom or standard protocols) should be parsed. The parser can be represented as a

finite state machine with only one starting point, multiple intermediate states, and two final states. The packet is either accepted or rejected at the end of the parsing stage. If the packet is accepted, the extracted header fields are forwarded to the *match-action* pipeline. Conversely, if the packet is rejected, the action depends on the implementation, but it is usually discarded.

2. Following the parsing stage, the packet continues through the *match-action pipeline*, which is further divided into the ingress and egress pipelines. In both pipelines, programmers define one or more *Match-Action Tables*(MATs) to match packets and execute actions based on each packet's data. The lookup keys and corresponding action data for MATs are stored in the programmable device using either Static Random-Access Memory (SRAM) or Ternary Content-Addressable Memory (TCAM). The control plane manages the content of each MAT by writing, modifying, or deleting table entries during runtime. In addition to MATs, this stage allows the definition of separate actions, utilization of stateful objects (e.g., counters, meters, and registers), and the execution of arithmetic and logic operations.

3. The last stage is the programmable *deparser*, where the desired packet headers are reinserted into the outgoing packet.

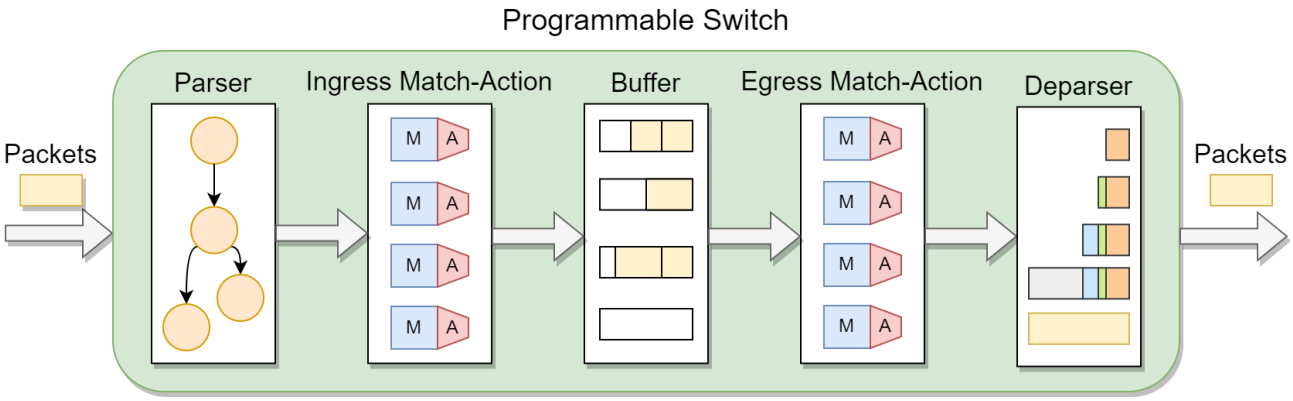


Figure 2.6: Protocol-Independent Switch Architecture (PISA).

In addition to the programmable switches and the PISA model, there is a need for a high-level programming language capable of leveraging the data plane programmability enabled by these technologies. The *Programming Protocol-Independent Packet Processors* language (P4) [6] is one of the most widely used domain-specific programming languages for describing data plane algorithms for PISA and other data plane architectures. Section 2.3.1 details the main components of the P4 language and highlights some of its limitations that are relevant to our work.

2.3.1 P4 Language

The P4 language is an open-source, domain-specific programming language for network devices, specifying how data plane devices (switches, routers, NICs, filters, etc.) process packets. The pillars of the P4 language consist of reconfigurability, protocol independence, and target independence. Reconfigurability is the ability to change the behavior of the forwarding device during runtime. Protocol independence refers to the fact that the network is not limited to any specific networking protocol. Lastly, target independence guarantees that the underlying hardware is hidden from the P4 programmer and it is up to the P4 compiler to turn a target-independent P4 program into a target-dependent binary. The main components of the P4 language are described below:

- **Headers**: Headers describe the sequence and structure of a series of fields [6]. Headers are naturally associated with well-known protocols such as Ethernet, IPv4, TCP, etc., but P4 also allows the declaration of custom headers. Figure 2.1 shows an example of the IPv4 protocol represented as a header in the P4 language.
- **Parsers**: The P4 parser is a finite state machine with a starting state, two final states, and multiple user-defined intermediate states. The primary purpose of the parser is to extract the desired header fields to input them into the match-action stage. For example, an IPv4 header parser would extract the fields displayed in Figure 2.1, and according to the "protocol" field, the next parser could be a TCP or UDP parser.
- **Match-Action Tables (MATs)**: P4 tables are data structures that contain forwarding instructions. Each table entry contains a key and the corresponding set of actions. The matching key consists of one or more header or metadata fields (stateless data) and a match type. The match type field is the algorithm used to match the data plane values with those informed by the control plane. For example, a MAT could be an IPv4 forwarding table, where the destination address is the key while the corresponding action decrements the Time-To-Live (TTL) field and updates the MAC source and destination address.
- **Actions**: Actions are code fragments that describe how packet header fields and metadata are manipulated. For example, an action could decrement the TTL field or update the MAC addresses.
- **Control flow**: Control flow is an imperative program that describes the packet processing on a target, including the data-dependent sequence of match-action unit invocations.
- **Metadata**: Metadata is information about a packet that is not directly related to its headers. There are two types of metadata in a P4 program: the user-defined and

the intrinsic. The user-defined metadata are user-defined data structures associated with each packet. The intrinsic metadata is provided by the architecture and is also associated with each packet, such as the timestamp when the packet arrived at the forwarding device.

Listing 2.1: IPv4 protocol as a P4 header

```
header ipv4_h {
    bit<4>  version;
    bit<4>  ihl;
    bit<8>  tos;
    bit<16> total_len;
    bit<16> identification;
    bit<3>  flags;
    bit<13> flag_offset;
    bit<8>  ttl;
    bit<8>  protocol;
    bit<16> hdr_checksum;
    bit<32> src_addr;
    bit<32> dst_addr;
}
```

2.3.2 P4 Target and Architecture

P4 targets are the programmable nodes on the network where the P4 logic is embedded. P4 supports software-based and hardware-based targets, including NPUs, FPGAs, and ASICs. These targets feature a packet processing pipeline whose structure is target-specific and is described by an architecture model.

As P4 evolved, the P4 architecture concept was introduced to bring more flexibility to the P4 environment. A P4 architecture is a model that identifies the capabilities and logical view of a target's P4 processing pipeline. Each architecture has its functional blocks, fixed functions, and control flow. The P4 programmer only needs to define the behavior of each functional block, since the target manufacturers define the fixed functions. In addition to exposing the functional block of the processing pipeline, a P4 architecture may provide additional functionalities that are not part of the P4 language; these are called *externs*. It is essential to note that P4 programs are not expected to be portable across different architectures, but should be portable across all targets that faithfully implement the corresponding architectural model, provided there are sufficient resources.

2.3.3 P4 Compiler

The P4 compiler translates the P4 programs created for a P4 architecture into target-specific binary code that is executed on the desired P4 target. The compiler works in two stages: the front-end and the back-end. The front-end stage converts the P4 program into an intermediate table dependency graph that represents the dependencies between tables. Then the target-specific back-end maps this graph onto a specific target. Each P4 compiler is designed for a specific P4 target. For example, FPGA-based devices mainly use compilers such as Xilinx P4-SDNet and P4FPGA; BMv2 software switches use compilers such as p4c, PISCES, and P4-to-OVS; while Barefoot Tofino's programmable ASIC uses the Barefoot P4 compiler [13].

2.3.4 P4 Runtime

The P4 runtime is an API that the control plane uses to communicate with the data plane. With the P4 runtime, the controller can reconfigure the behavior of the forwarding devices that already run P4 programs according to new rules or objectives. This means that the control plane can update MATs and *externs* or even load an entirely new P4 program onto the P4 target.

2.3.5 P4 Challenges and Limitations

Recent surveys by AlSabeh et al. [3] and Kfoury et al. [22] have enumerated the latest challenges and limitations for the programmable data plane and the P4 language. Among the many that they have listed, we chose two from each review that are most relevant to our work.

- **Memory Size and Accessibility [3]**: The limited on-chip memory in the switch (e.g., a limited number of SRAM per stage and a limited number of stages), as well as the restrictions on memory access (e.g., a packet can only access few addresses in the memory) have affected several applications. For example, several straightforward functions cannot be performed (e.g., finding the minimum across all elements). Among the affected applications are those related to network security implementations. For instance, it is not feasible to maintain a significant per-flow state, i.e., to track and store every flow in the switch.
- **Processing Capabilities [3]**: Programmable switches process a limited number of primitives, that is, they cannot perform arbitrary operations. Instead, they support a

small set of simple operations. For example, since division is much slower than addition, the switching hardware usually does not support division. Moreover, loops are not supported, and floating-point arithmetic is not a mandatory feature that P4 enforces. In addition, programmable switches support a limited number of operations per packet to operate at a line rate. These limitations affect the number of security applications that are implemented in the network.

- **Control Plane Intervention [22]**: Delegating tasks to the control plane incurs latency and affects the performance of the application. For instance, in congestion control, rerouting-based schemes often use tables to store alternative routes. Since the data plane cannot directly modify table entries, intervention from the control plane is required. The interaction with the control plane in this application hampers the promptness of rerouting. Ideally, the control plane intervention should be minimized when possible. For example, to synchronize the state among switches, in-network cooperation should be considered.

- **Network-Wide Cooperation [22]**: SDN architecture suggests using a centralized controller for network-wide switch management. Through centralization, the state of each programmable switch can be shared with other switches. Consequently, applications can make better decisions as network-wide data is available locally on the switch. The problem with such architecture is the requirement of continuously exchange packets with a software-based system. As an alternative, switches can exchange messages to synchronize their states in a decentralized manner.

When employing the PDP to address the processing limitations of NIDSs, it is crucial to design algorithms and data structures that take into account the PDP constraints, such as the reduced memory size and the restricted set of processing capabilities. An essential function of NIDSs is the monitoring of network flows, which demands significant memory resources as a result of the large number of flows in a network. To efficiently offload this function to programmable devices, researchers [10, 12, 37] have employed sketches. Sketches store summarized information of flows rather than the complete data, significantly reducing the required memory for monitoring flows. Among existing sketch solutions, the Count-Min sketch is widely used to efficiently and accurately store frequencies related to flows.

2.4 Count-Min Sketches

A sketch is a synopsis data structure typically used in algorithms that process data streams, which are also known as data streaming algorithms [43]. The primary objective of a sketch is to provide a summarized representation of a dataset, particularly in

scenarios where storing the entire dataset is prohibitively expensive. This condition often arises when relying solely on memory for data stream processing or when dealing with resource-constrained devices with limited storage capacity. Sketches allow operations to be performed on the data, enabling the retrieval of information about the dataset with a certain probability of error. This error probability can be configured by the programmer by adjusting the amount of resources used. Allocating more resources reduces the probability of errors, while using fewer resources increases it. Examples of sketch data structures include Count-Sketch, Count-Min sketch, and Bloom Filters.

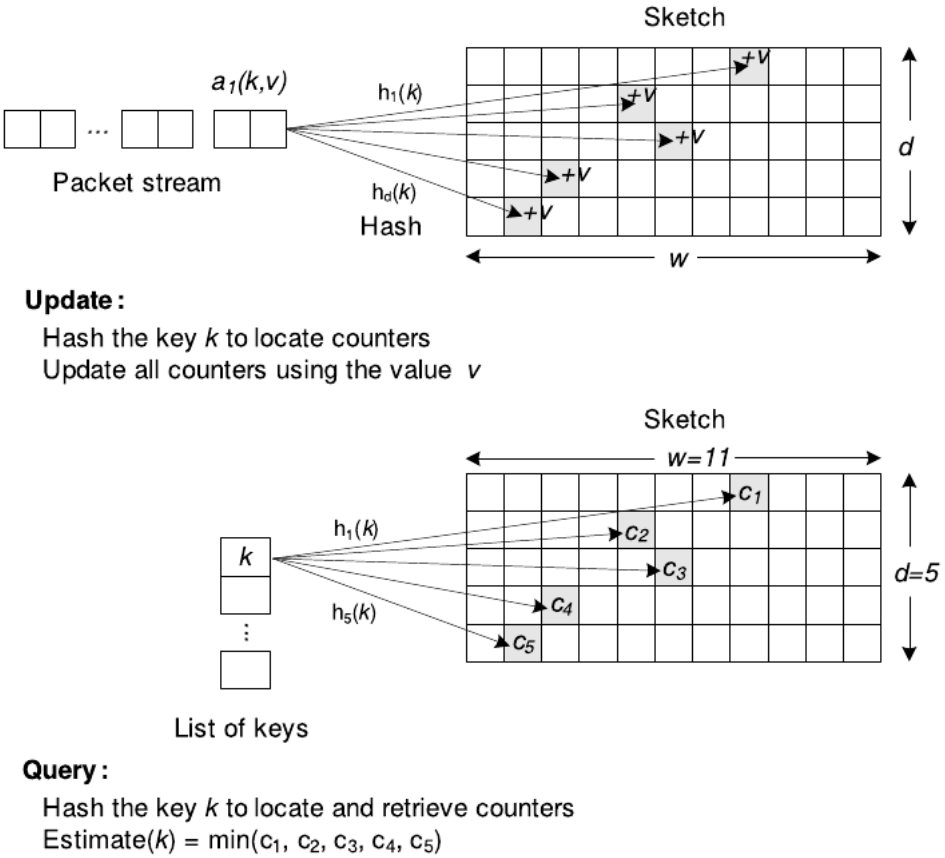


Figure 2.7: Count-Min sketch operations [43]

The *Count-Min* sketch [9] was first proposed in 2003 as an alternative to several other sketch techniques [8]. Its primary objective is to determine the frequency or count of appearances of a specific element. One notable advantage of the Count-Min sketch over the traditional approach of using a hash table is that it uses sublinear space to store information. However, this advantage comes at the cost of potentially overestimating the frequency of an element. The Count-Min sketch is composed of multiple arrays and hash functions, where each hash function is associated with an array, and a hash's output is used to determine the position to store an element's frequency in the associated array. The depth of a Count-Min sketch determines the number of hash functions to use, while the width defines the length of the hash arrays.

There are two main operations in a Count-Min sketch: *update* and *query*. Figure 2.7 details these operations. The *update* operation updates the frequency of appearances for a certain element. For each array, the position to update the frequency of an element is calculated based on the hash function for that array. The *query* procedure works by retrieving the smallest frequency from all the frequencies stored in the arrays. The frequencies to be returned are determined by the positions in the arrays calculated based on their hash functions. With a certain error probability, the query operation returns the real count or a larger count due to the possibility of collisions with other data.

3. RELATED WORK

This chapter explores the related work on offloading NIDS capabilities to the PDP, network-wide security cooperation with the PDP, or scenarios involving both topics. It begins by detailing the search methodology, search queries, and inclusion and exclusion criteria in Section 3.1. Subsequently, in Section 3.2, it showcases works that focus on offloading NIDS capabilities to the PDP, with a particular emphasis on those that offload pre-filtering rules. Next, this chapter presents works that leverage either the data plane alone or a combination of the data plane and the control plane to enhance network-wide security in Section 3.3. Lastly, Section 3.4 discusses related work and highlights the research gap addressed in this work.

3.1 Search Methodology

To find the desired related work, we selected three important research repositories in Computer Science: Association for Computing Machinery (ACM) Digital Library[1], Institute of Electrical and Electronics Engineers (IEEE) Xplore[2], and Scopus[3]. ACM and IEEE were selected for their prominence as primary sources in Computer Science, while Scopus was chosen for its extensive coverage, indexing studies from various sources such as Springer, ScienceDirect, Wiley, and Inderscience, among others. Furthermore, we use Google Scholar[4] to find works that are not normally referenced in the three previously mentioned repositories (e.g., dissertations, thesis, or publications in national events) and to search for citations for some highly related works. The search queries presented next were not used for Google Scholar (only for ACM, IEEE, and Scopus) as it does not provide a comprehensive search tool. Instead, we searched using variations of key ideas, e.g., "offload NIDS P4", "offload NIDS PDP," and "network-wide security P4".

We used two search queries to find relevant work. In both queries, we searched for all fields of the papers (i.e., "AllField" for ACM DL, "Full Text & Metadata" for IEEE Xplore and "ALL" for Scopus). The first query, illustrated in Figure 3.1, focuses on identifying studies that offload NIDS capabilities to the data plane, with particular emphasis on work involving the offloading of signature rules. Since NIDS signature rules typically come from open-source NIDS, we created a query to return PDP works that mentioned Snort, Suricata, or Zeek (Bro). Additionally, we included the term "Intrusion detection system" to encompass works integrating PDP with proprietary NIDS solutions rather than open-source ones.

[1] https://dl.acm.org/
[2] https://ieeexplore.ieee.org/
[3] https://www.scopus.com/
[4] https://scholar.google.com/

To expand the search scope even further, we incorporated the keyword "signature rule". The "?" symbol at the end of some keywords represents one wildcard character.

```
( "P4" OR "Programmable data plane?" OR "Programmable device?" OR "Programmable network?") AND
    ("Snort" OR "Suricata" OR "Bro" OR "Zeek" OR "Instrusion detection system?" OR "Signature rule?")
```

Figure 3.1: Query for works that offload NIDS capabilities to the PDP

To find works on network-wide security cooperation in the PDP, the query in Figure 3.2 was used. Our idea was to add terms that imply a system with multiple components, such as "Framework", "Architecture", or "Orchestrator". Moreover, we added the term "Network-wide" to specifically target works addressing security cooperation across the entire network. To filter out non-NIDS-related works, we added the terms "Intrusion detection system" and "Intrusion prevention system" to the query.

```
( "P4" OR "Programmable data plane?" OR "Programmable device?" OR "Programmable network?") AND
        ("Framework" OR "Architecture" OR "Orchestrator" OR "Network-wide") AND
        ("Intrusion detection system?" OR "Intrusion prevention system?")
```

Figure 3.2: Query for works that employ network-wide security cooperation in the PDP

After retrieving the papers, we filtered out unrelated work. We first removed duplicates, surveys, drafts, and non-computer science works. Then, we investigated the titles and abstracts to remove works that are not security related or have their primary focus on another technology that is not SDN or PDP (e.g., GPU, IoT, Cloud). Afterward, we looked into the manuscript and kept only works that offload NIDSs capabilities (i.e., signature rules and NIDS-related functions) to the PDP or employ some type of network-wide security cooperation (e.g., cooperation among switches, data and control plane cooperation and frameworks). Finally, we compared this final set of works with the ones from Google Scholar search, performed citation searches, and identified the most relevant papers for each search query topic.

3.2 Offloading NIDS Capabilities to the PDP

Since the introduction of P4 in 2014, several studies have explored offloading complex NIDS functions to programmable devices. In P4DDPI [2], P4 is used for the Deep Packet Inspection (DPI) domain name, where the Internet domains are parsed and filtered in the data plane without using the control plane. After extracting the domain name, P4DDPI uses a table containing well-known malicious domain names to filter out malicious

packets. P4DM [1] proposes a new probe-based approach for measuring one-way delays (OWD). In P4DM, hosts generate lightweight probe packets that are then exploited by the P4 programs in the switches to implement the measure. Expanding the scope to include multiple network devices, [4] demonstrates an architecture that uses P4 devices to implement network analysis functions related to IDSs. The proposed model has a complete view of the network and, according to the needs of the system's IDS, offloads to different P4 devices the required functionalities. They implemented three IDS-related functions to be offloaded to programmable devices: DPI, link delay measurement, and asymmetric flow detection.

Instead of deploying complex NIDS functions to the PDP, other works have investigated using the data plane as a pre-filter for network traffic to alleviate the workload on the NIDS host. This approach has advantages over offloading complex NIDS capabilities to the PDP, as the PDP has limited resources and functionalities, but it is specifically designed for packet matching. In [31], the authors propose a two-level network-based IDS for an SDN-based Industrial Control System (ICS). At the first level, a Modbus protocol whitelist is implemented in P4 on network switches. Packets not on the whitelist are forwarded to the second level, where a security engine is used to determine if the packet is malicious and update the off-loaded whitelist accordingly. Although the evaluation of this work produced positive results, offloading a whitelist to the data plane is only feasible in static and easily controlled networks, such as ICS networks. For traditional IT networks, the significant number of normal flows makes offloading whitelists impractical due to the restricted memory of programmable devices.

Unlike [31], the authors of [20] aim to systematically offload Zeek filters of suspicious traffic to programmable switches instead of using a whitelist. By delegating the task of filtering regular traffic to the PDP and forwarding only suspicious packets to the NIDS host, more processing power becomes available for the NIDS host to conduct the thorough analysis of suspicious flows that it needs to perform. Nevertheless, it is worth noting that only a small subset of Zeek traffic filters were offloaded, limiting the solution's security scope. Still trying to reduce the workload on the NIDS host, P4ID [25] proposes to pre-filter network traffic in the data plane by offloading rulesets designed for traditional NIDSs, such as Snort, to the data plane. The architecture of P4ID consists of two components: the *Rule Parser* and the *P4 implementation*. The *Rule Parser* consumes rules made for the popular Snort NIDS and produces table entries that can be installed in the implementation of *P4*. Although improving the number of rules offloaded compared to [20], P4ID does not take into account the limited memory of the device and performs only basic deduplication to optimize the number of rules to offload.

Motivated by the ideas presented by P4ID, P4-ONIDS [37] seeks to optimize the parsing and compilation of NIDS signature rules. This optimization reduces the number of rules offloaded to devices with limited resources while maintaining the accurate for-

warding of suspicious packets to the NIDS host. Additionally, P4-ONIDS uses a Count-Min sketch-based solution to monitor information about suspicious flows and to limit the number of packets redirected to the NIDS. This structure addresses the issue of saturation attacks that could otherwise overwhelm the NIDS with a large number of packets. However, P4-ONIDS has certain limitations. It lacks support for certain types of NIDS rules, such as bidirectional rules, those with negation syntax, and those involving IPV6 addresses. Furthermore, the compilation process is not optimized, leading to long compilation times and excessive memory usage.

More recently, the authors of P4ID [25] updated their solution in [26]. In this improved version, they optimized the data plane memory required for P4ID to track and limit the flows for redirection. Furthermore, they developed a feedback system where Suricata logs are parsed and alerts are translated into 5-tuple table entries within the switch. Through this mechanism, P4ID's new version dynamically identifies suspicious flows and decides whether more packets for this flow should be forwarded to the NIDS or not. However, it is worth noting that P4ID's lack of consideration for the memory limitations of the forwarding devices (regarding the table entries to offload) and the challenge of offloading a large number of rules to them, as demonstrated in [37], remains a limitation of this new version.

One common and crucial limitation observed in most of the works that offload pre-filtering rules to the PDP mentioned above ([20, 25, 26, 37]) is their design to offload to only one forwarding device. This approach introduces multiple drawbacks, including scalability issues, limited NIDS performance enhancements, and reduced switch performance due to the overload of security functionalities on a single switch. The only work that considers more than one forwarding device is [31]. Nevertheless, this work focuses on whitelist instead of signature rules, and it offloads the same filtering rules to all forwarding devices. With this uniform offloading approach, insufficient memory in devices may result in some rules not being offloaded, potentially allowing attacks to go undetected. In contrast, if there is sufficient space for all rules, the network's memory usage might be inefficient, as devices will have multiple duplicate rules that will be unused depending on the network's topology. To address this lack of network-wide security solutions and understand the challenges involved in designing them, we have studied works that employ network-wide security cooperation in the PDP.

3.3 Network-Wide Security Cooperation in the PDP

To thoroughly protect computer networks, it is fundamental that all network devices have security capabilities. However, achieving this goal is difficult, as forwarding and routing devices have limited memory and reduced processing capabilities to protect

themselves against the ever-growing number of attacks. To overcome this issue, instead of delegating the same security functions to all devices, each device performs different security functions, and by cooperating with other devices, individual security limitations are solved. With this architecture, the idea of a completely secure network becomes attainable. In [33], the authors combined programmable network switches, edge devices, and cloud servers to provide scalable and line speed IDS. They achieve this goal by implementing Binarized Neural Networks (BNNs) at the data plane for line-speed packet classification; using the control plane to train the classification model locally; and using federated learning on a cloud server to scale the training process.

Although providing a robust IDS for a distributed scenario, [33] assumes the presence of one edge device for each programmable switch and a cloud server, requirements that may not be feasible in certain environments. Moreover, it is heavily dependent on the control plane and cloud, which could cause congestion in the communication channel when an attack is in progress. In contrast, BUNGEE [14] is a collaborative push-back mechanism in the network for DDoS attack mitigation that runs entirely in the data plane. BUNGEE first detects DDoS attacks through the entropy analysis of the addresses of incoming packets and immediately propagates to upstream devices the detected entropy disturbances. Upstream devices apply filtering rules for suspicious flows and, if necessary, push back these alerts to their upstream counterparts.

Using a similar concept of sharing information among forwarding devices as BUNGEE, the authors of [11] devised a lightweight NIDS that can be embedded in a chain of resource-constrained switches. The idea is to divide a classification model into numerous binary submodels, with each submodel offloaded to a programmable device to detect a specific attack type. The outputs of each device are combined and a final classification of the network traffic is produced on the last device. In [14] and [11], the authors rely basically on the data plane for security decisions. This reliance has positive outcomes as the communication overhead between the data plane and control plane is minimal or nonexistent, thereby decreasing the response time for detecting and responding to attacks. However, depending entirely on the data plane may impact the network's security capabilities, given the restricted functionalities of the data plane compared to the control plane. An ideal solution should balance the communication overhead between the control and the data plane while fully utilizing the resources of the control plane.

Instead of focusing on providing a new algorithmic or theoretical contribution to the network security field, Poseidon [27] proposes a practical and system-level defense system for DDoS attacks using PDP. With Poseidon, users can specify defense policies for different DDoS attacks, and Poseidon takes care of partitioning the policy-needed functions across switches and servers for effective defense. Additionally, Poseidon uses a runtime management mechanism to reconfigure the defense system for dynamic protection without interrupting legitimate flows. Poseidon's biggest limitation is that it does not

consider the resource constraints of programmable devices when deploying the policies. This limitation is evident in [29] when it was found that Poseidon recorded 65K legitimate sessions for an SYN proxy table, making it non-scalable.

Jaqen [29] is a switch-native approach to volumetric DDoS defense that can run detection and mitigation functions entirely inline on switches without relying on additional data plane hardware. Jaqen's API allows users to configure sketches, query relevant metrics, compute detection decisions, and define the mitigation actions for the data plane to perform. In addition, Jaqen incorporates a network-wide resource manager that optimally deploys detection and mitigation modules on the network. Despite being resource-efficient, Jaqen's mitigation response is too slow, a limitation relevant to our work. Summarizing, we can conclude from the limitations observed in Poseidon and Jaqen that security solutions working with the PDP should consider the restrictions of the network devices in order to become scalable and that time is an important factor.

3.4 Discussion

In the previous sections, we analyzed several works that offload NIDS capabilities to the PDP, employ network-wide security cooperation in programmable networks, or both. In the following discussion, we will outline some key characteristics of the works that offload pre-filtering rules to the PDP (Section 3.2). This discussion is summarized in Table 3.1.

Table 3.1: Related work comparison table

Work	Pre-filtering Rules Type	Ruleset Optimization	Resource Aware Offloading	Runtime NIDS Feedback	Deployment Type
[31]	Whitelist	None	No	Yes	Network-wide rules replication
[20]	Zeek filters	None	No	No	One device
P4ID [25, 26]	Snort/Suricata rulesets	Basic deduplication	No	Yes	One device
P4-ONIDS [37]	Snort/Suricata rulesets	Deduplication and aggregation	Yes	No	One device
This Work	**Snort/Suricata rulesets**	**Deduplication and aggregation**	**Yes**	**No**	**Memory- and topology-aware network-wide**

As presented in Table 3.1, open-source NIDS rulesets [25, 37, 26] are favored over Zeek filters [20] and whitelists [31] due to their ability to provide protection against a wide range of well-known attacks. When dealing with NIDS rulesets, it is crucial to implement rule optimization strategies to prevent overloading the network devices' memory with an excessive number of rules. Alongside the removal of duplicates, rule aggregation is a strategy that can help reduce the necessary number of rules.

In addition to creating an optimized ruleset for offloading, understanding the resources of the networking devices and offloading accordingly is essential to avoid excessive memory usage and potential operational impacts. However, most works lack this resource-aware offloading, except for [37]. Following the offloading of rules to the PDP, a feedback mechanism, as employed by [31, 26], is fundamental for the continuous monitoring of attacks and the corresponding update of the offloaded rules. Despite the diverse characteristics of the related work, their deployment approaches are not comprehensive, as most of them consider only one device [20, 25, 37, 26]. Even when multiple devices are considered [31], the proposed approach is to distribute the same set of rules to all devices. These strategies can lead to the incomplete offloading of rules and the inefficient utilization of the network's available memory, compromising the solutions' objectives.

Based on this discussion and the observed characteristics and limitations of the related work, we propose to address the lack of a comprehensive rules offloading method by developing a network-wide orchestrator capable of offloading NIDS rules to multiple devices, considering the memory of each device and the network topology. By addressing this gap in existing research, our work aims to provide a more comprehensive approach capable of improving NIDS performance in more scenarios, ultimately enhancing network security. To achieve this goal, we first improve the existing P4-ONIDS [37] rules compiler in Chapter 4. Subsequently, in Chapter 5, we detail our proposal, including the P4 data plane to pre-filter network traffic for the NIDS, as well as the memory- and topology-aware orchestrator that distributes the table entries based on two novel offloading algorithms.

4. IMPROVEMENTS TO THE P4-ONIDS RULES COMPILER

This chapter outlines the improvements made to the P4-ONIDS rules compiler. It begins by providing a detailed overview of the P4-ONIDS rules compiler in Section 4.1. This chapter then details the improvements made to the P4-ONIDS rules compiler and the results obtained from these modifications in Section 4.2.

4.1 The P4-ONIDS Rules Compiler

The P4-ONIDS architecture is divided into two components: the *P4 NIDS Rules Compiler* and the *P4 Switch Data Plane*. The P4 NIDS Rules Compiler is responsible for converting the NIDS rules into P4 table entries, while the P4 Switch Data Plane is responsible for detecting, filtering, and redirecting suspicious traffic to the NIDS host. This section introduces the P4 NIDS Rules Compiler, detailing its input and the compilation steps required to convert NIDS rules to P4 table entries. Figure 4.1 illustrates the architecture of the P4 NIDS Rules Compiler.

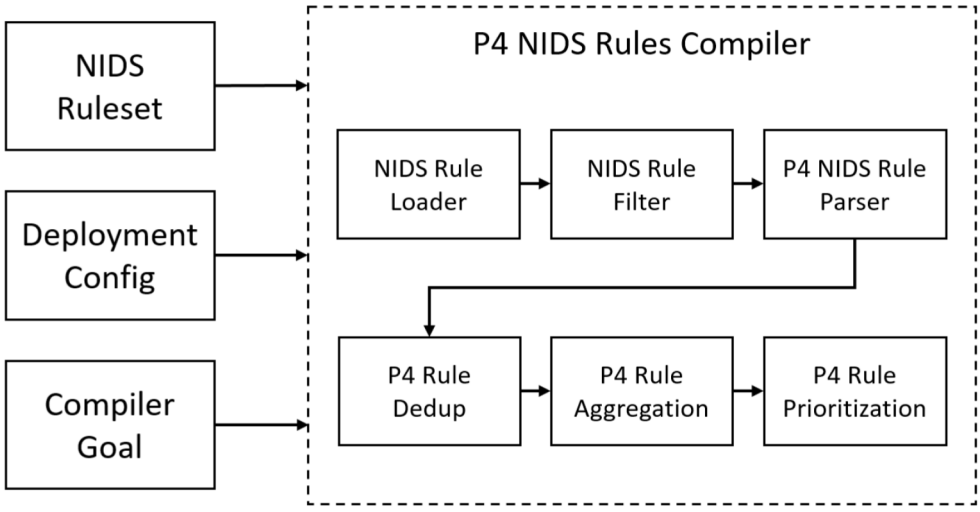


Figure 4.1: P4 NIDS Rules Compiler architecture [38]

4.1.1 Inputs

The P4 NIDS Rules Compiler requires three inputs: (i) a NIDS Ruleset, (ii) a Deployment Configuration, and (iii) a Compiler Goal. The NIDS Ruleset is composed of a list of signature rules written using Snort and Suricata's syntax. These rules define suspicious network behavior and the appropriate action to take in case a packet complies with

the rule (see Section 2.1.2). Listing 4.1 shows a sample of rulesets containing two TCP signature rules targeting browser attacks.

Listing 4.1: Snort signature rules

```
alert tcp $EXTERNAL_NET 80 -> $HOME_NET any ( msg:"BROWSER-OTHER Netscape 4.7
    client overflow"; flow:to_client,established; content:"3|C9 B1 10|?|E9
    06|Q<|FA|G3|C0|P|F7 D0|P"; metadata:ruleset community; reference:bugtraq,822;
    reference:cve,1999-1189; reference:cve,2000-1187; classtype:attempted-user;
    sid:283; rev:14; )

alert tcp $HOME_NET any -> $EXTERNAL_NET 80 ( msg:"BROWSER-OTHER Netscape 4.7
    unsucessful overflow"; flow:to_server,established; content:"3|C9 B1 10|?|E9
    06|Q<|FA|G3|C0|P|F7 D0|P"; metadata:ruleset community; reference:bugtraq,822;
    reference:cve,1999-1189; reference:cve,2000-1187;
    classtype:unsuccessful-user; sid:311; rev:15; )
```

Rules can have configuration variables, such as $HOME_NET, enabling the reuse of a ruleset in different deployments. The *Deployment Configuration* outlines the addresses and ports related to the configuration variables and the priority for each type of rule. The configuration of each deployment directly impacts the number of processed rules due to the different possibilities in defining the addresses and ports (see Section 2.1.2). Therefore, using the same NIDS ruleset with distinct deployment configurations can lead to significant differences in the final number of rules. Lastly, the *Compiler Goal* defines specific goals for the compiler, such as the maximum number of rules to offload, the P4 target architecture, and the prioritization scheme (e.g., random, severity, etc.).

4.1.2 Compilation Steps

The P4-ONIDS compiler performs six main steps: (1) file parsing and rule adjusting, (2) rule filtering, (3) transforming the NIDS rules into P4 table entries, (4) removing duplicate P4 table entries, (5) aggregating table entries, and (6) prioritizing certain table entries. The compiler begins by parsing the three inputs: the NIDS Ruleset, the Deployment Configuration and the Compiler Goal. For the *NIDS Ruleset* input, the compiler reads the Snort/Suricata file(s), parses each rule, and saves them into a data structure. Subsequently, the compiler adjusts the rules by replacing the configuration variables in them by the actual addresses and ports defined in the *Deployment Configuration*. Once the rules are adjusted, invalid rules are removed. Invalid rules include those containing IPv6 headers, bidirectional flow, negation syntax, and rules composed only of "any" matches.

The next step involves converting the adjusted and filtered NIDS rules into P4 table entries. Each P4 table entry is composed of the action and the following 6-tuple match: the protocol type, source IP address, source port range, destination IP address,

destination port range, and TCP flags if available. Before converting a NIDS rule to P4 table entries, the compiler groups the ports of a rule into ranges, with the aim of minimizing the number of ungrouped ports. With the ports combined into ranges, the compiler iterates through every combination of source IP address, source port, destination IP address, and destination port of a rule to create a unique P4 table entry. These procedures can generate a single entry into the P4 table or multiple entries, depending on the number of addresses and ports involved. Listing 4.2 presents the Snort rules from Listing 4.1 in the table entries format, considering the configuration variables *$HOME_NET* as "[10.0.10.0/24, 10.0.20.5]" and *$EXTERNAL_NET* as "any." The protocol is represented in hexadecimal format. The IP addresses have the host address separated from the network mask using "&&&", and the "any" address is translated to "0.0.0.0&&&0.0.0.0". The ports are converted into ranges by default, even if there is only one port, to align with the P4 data plane restrictions. Lastly, the TCP flags are represented in binary form.

Listing 4.2: P4 table entries from NIDS rules

```
// Alerts for the first Snort rule
alert 0x6 0.0.0.0&&&0.0.0.0 80->80 10.0.10.0&&&255.255.255.0 0->65535 00000000
alert 0x6 0.0.0.0&&&0.0.0.0 80->80 10.0.20.5&&&255.255.255.255 0->65535 00000000

// Alerts for the second Snort rule
alert 0x6 10.0.10.0&&&255.255.255.0 0->65535 0.0.0.0&&&0.0.0.0 80->80 00000000
alert 0x6 10.0.20.5&&&255.255.255.255 0->65535 0.0.0.0&&&0.0.0.0 80->80 00000000
```

Due to the aforementioned procedure, the P4 table entries of one rule can be duplicates of another rule. Consequently, the compiler needs to remove duplicates and merge them into one. When merging duplicate rules, the final rule retains the severity level and identifier of all merged rules. Following the deduplication stage, the number of rules is further reduced by aggregating rules within the same IP address range or port range. For example, consider that table entries A and B have the same source IP, source port, destination IP, and protocol, but the destination port of rule A is "1024", and the destination port of rule B is the range "[0:2000]". In the aggregation stage, rule A can be grouped into rule B since packets that would match rule A are covered by rule B. Similarly to the deduplication stage, the aggregation process retains the severity level and the identifier of the original rules. The last step is to select the final ruleset based on the *Compiler Goal*. For instance, the final goal could be to return 100 table entries, ordered by severity.

4.2 Improvements Performed

Although the P4-ONIDS compiler successfully achieved its goal of reducing the total number of P4 table entries to offload compared to the P4ID compiler [25], it has some important limitations. This work addresses two of its main limitations. The first limitation addressed is the lack of support for certain types of NIDS rules, such as those with IPv6 headers and negation syntax. In this work, we expand the set of accepted rules to include those not accepted by P4-ONIDS. The second one is that the compilation process of the P4-ONIDS compiler takes too long and uses a significant amount of memory, rendering it impractical. As a result, we have tackled this issue by modifying parts of the P4-ONIDS compiler that were causing prolonged compilation time and excessive memory usage. This section outlines these improvements along with the results obtained. It begins by presenting the inputs used to evaluate the improvements implemented. Subsequently, this section describes the modifications made to the original P4-ONIDS compiler and presents the results obtained from these improvements. Finally, the modified P4-ONIDS compiler architecture is depicted.

4.2.1 Experiments Input

The evaluation of the original P4-ONIDS compiler and the improved version requires two inputs: a NIDS ruleset and a Deployment Configuration. Table 4.1 provides details on the rulesets tested, including the number of valid rules according to the Snort 3 engine and the date of download, as these rulesets are updated daily or weekly. The *Snort 3 Community*[1] ruleset is the basic ruleset provided by Snort, which does not require an account to access. It consists of 3944 rules and receives daily updates. The *Snort 2 Emerging Threats*[2] is a publicly maintained ruleset consisting of 31405 rules with daily updates. Lastly, the *Snort 3 Registered*[3] is an extensive ruleset maintained by Snort that requires an account for download. It has 44705 valid rules and is updated weekly.

Table 4.1: NIDS rulesets information

NIDS ruleset	Amount of rules	Download date
Snort 3 Community	3944	April 5, 2023
Snort 2 Emerging Threats	31405	April 27, 2023
Snort 3 Registered	44705	May 19, 2023

[1] https://snort.org/downloads
[2] https://rules.emergingthreats.net/open/snort-2.9.0/
[3] https://snort.org/downloads

For the deployment configuration, Listing 4.3 displays the IP (ipvar) and port (portvar) configuration variables used by the NIDS rulesets. The network addresses are based on the CICIDS2017 dataset [36], since this dataset is later used in Chapter 6 as input for the main experiments of this work. The ports and file structure are derived from Snort's default configuration file.

Listing 4.3: Deployment configuration

```
# List of IP network addresses
ipvar HOME_NET [192.168.10.0/24,172.16.0.0/16,205.174.165.68,205.174.165.66]
ipvar EXTERNAL_NET any
ipvar DNS_SERVERS [192.168.10.3/32]
ipvar SMTP_SERVERS $HOME_NET
ipvar HTTP_SERVERS $HOME_NET
ipvar SQL_SERVERS $HOME_NET
ipvar TELNET_SERVERS $HOME_NET
ipvar SSH_SERVERS $HOME_NET
ipvar FTP_SERVERS $HOME_NET
ipvar SIP_SERVERS $HOME_NET

# List of ports
portvar HTTP_PORTS [36,80,81,82,83,84,85,86,87,88,89,90,311,383,443,555,591,593,
   631,801,808,818,901,972,1158,1220,1414,1533,1581,1719,1720,1741,1801,1830,
   1942,2231,2301,2375,2381,2578,2809,2869,2980,3000,3029,3037,3057,3128,3443,
   3702,4000,4343,4592,4848,5000,5054,5060,5061,5117,5222,5250,5416,5443,5450,
   5555,5600,5814,5894,5984,6080,6173,6988,7000,7001,7005,7070,7071,7144,7145,
   7180,7181,7510,7770,7777,7778,7779,8000,8001,8008,8014,8015,8020,8028,8040,
   8080,8081,8082,8085,8088,8090,8095,8118,8123,8180,8181,8182,8222,8243,8280,
   8300,8333,8344,8393,8400,8443,8484,8500,8509,8694,8787,8800,8852,8880,8888,
   8899,8983,9000,9002,9060,9080,9090,9091,9111,9200,9201,9290,9443,9447,9700,
   9710,9788,9830,9850,9999,10000,10080,10100,10250,10255,10297,10443,11371,
   12601,13014,14592,15489,16000,17000,18081,19980,29991,30007,30018,33300,
   34412,34443,34444,36099,40007,41080,44449,49152,49153,50000,50002,50452,
   51423,53331,55252,55555,56712]
portvar SHELLCODE_PORTS !80
portvar MAIL_PORTS [110,143]
portvar ORACLE_PORTS 1024:
portvar SSH_PORTS 22
portvar FTP_PORTS [21,2100,3535]
portvar SIP_PORTS [5060,5061,5600]
portvar FILE_DATA_PORTS [$HTTP_PORTS,110,143]
portvar GTP_PORTS [2123,2152,3386]
```

4.2.2 Improvements and Results

The first improvement's objective is to expand the types of NIDS rules that the compiler can parse. The new rules supported include those with IPv6 headers, that are

bidirectional, or contain negated ports. Adding support for rules with IPv6 headers involved programming the necessary parsing capabilities. For bidirectional rules, the approach required splitting the rule into two new rules with opposite source and destination fields. And for rules with ports using the negation syntax, the negated ports were replaced with the corresponding nonnegated ports within the TCP port range (0 to 65535). Despite adding support for these rules, rules containing negated IP addresses are not yet supported.

The results of the first improvement are evident when comparing the "Parsing, Adjusting, and Filtering Rules" column of Table 4.2 between P4-ONIDS and this work. Table 4.2 compares the number of rules and table entries between P4-ONIDS [37] and this work at every stage of the P4-ONIDS rules compiler and the new stage introduced in this work (i.e., "Rule Dedup.") for the three rulesets of Table 4.1.

Table 4.2: Number of rules and P4 table entries during each stage

Work	**Ruleset**	**Parsing, Adjusting, and Filtering Rules**	**Rule Dedup.**	**P4 Table Entries**	**P4 Table Entries Dedup.**	**P4 Table Entries Aggregation**
P4-ONIDS [37]	Snort 3 Community	2937	-	1199677	4328	854
	Snort 2 Emerging Threats	24462	-	9188484	105219	750
	Snort 3 Registered	41115	-	15732312	13251	1699
This Work	Snort 3 Community	3955	**318**	8980	5052	1841
	Snort 2 Emerging Threats	31408	**554**	145356	111830	839
	Snort 3 Registered	44223	**1721**	53247	14202	14202

The second improvement focuses on reducing the compilation time and memory usage. After examining the source code, it was found that the third stage, which involves converting NIDS rules into P4 table entries, is the slowest and most time-consuming. This is due to the fact that the number of table entries generated from a rule corresponds to every combination of its source IP addresses, source ports, destination IP addresses, and destination ports. The necessity to iterate over all combinations of these fields is because the P4 table entries do not allow lists, unlike the NIDS rules. Due to this constraint, the algorithm itself cannot be optimized, as lists of IPs and ports in a NIDS rule need to be unwound to generate valid P4 table entries.

Taking into account this limitation, the chosen approach was to reduce the number of NIDS rules and the number of IPs and ports per rule before starting the rule-to-table entry conversion stage. With this goal in mind, we noticed that multiple NIDS rules had the same fields used to uniquely identify a table entry. These fields are: protocol, source IP addresses, source ports, destination IP addresses, destination ports, and TCP flags, if available. Knowing this information, duplicate NIDS rules are removed based on these fields on the new "Rule Dedup." stage before converting them into table entries. Additionally, for further optimization, ports are grouped into ranges. For instance, by converting the list of ports "[10,11,12,13]' into the port range "[10:13]", the number of elements to iterate goes from four to just one. This reduction is noticeable by comparing the number of table entries in the "P4 Table Entries" stage between P4-ONIDS and this work in Table 4.2.

The last stage of the P4-ONIDS compiler, the aggregation stage, poses another significant bottleneck. This is because for each table entry, the P4-ONIDS compiler iterates over all the other table entries to determine which ones can be grouped. This time-consuming process results in an $O(N^2)$ algorithm, where N is the number of table entries entering the aggregation stage. The key insight to improve this algorithm was obtained by analyzing the two rightmost columns of Table 4.2 in the "P4-ONIDS" row. The notable reduction in table entries between the pre-aggregation stage compared to the post-aggregation stage indicates the presence of numerous groups. With this knowledge, we modified the aggregation algorithm so that the nested (inner) loop iterates over a hashmap containing the groups' key (i.e., the 6-tuple match) rather than iterating over the table entries list. Although the worst-case scenario of this new algorithm still is $O(N^2)$ (i.e., no groups), we exploit the fact that there are multiple groups in the evaluated datasets, resulting in a smaller inner loop to iterate.

The results of the performance improvements obtained after modifying the P4-ONIDS compiler to improve the compilation time and memory usage are shown in Table 4.3. As presented in the table, our modifications significantly decreased the compilation time and memory usage, with a reduction of over 80% for all metrics in the three evaluated datasets. One of the main reasons for this is the decrease of input rules to the rule-to-table entry conversion stage ("P4 Table Entries" column). This is evident by comparing the "'Parsing, Adjusting, and Filtering Rules" column for the "P4-ONIDS" row against the "Rule Dedup." column for the "This Work" row in Table 4.2.

The benefits of the modifications to the aggregation stage are evident when comparing the results of the "Snort 2 Emerging Threats" rows with the "Snort 3 Registered" rows in Table 4.3. Although the Snort 2 Emerging Threats ruleset has 13k fewer rules than the Snort 3 Registered ruleset (see Table 4.1), compiling it takes nine times longer than compiling the Snort 3 Registered ruleset with P4-ONIDS. The main cause of this problem is the considerable number of table entries remaining after deduplication of P4 table entries

Table 4.3: Performance improvements results

Work	Ruleset	Compilation time	Memory usage
P4-ONIDS [37]	Snort 3 Community	101.53s	1875.7 MiB
	Snort 2 Emerging Threats	11219.18s	14073.7 MiB
	Snort 3 Registered	1252.36s	23648.0 MiB
This Work	Snort 3 Community	19.60s **(80.6% reduction)**	124.1 MiB **(93.38% reduction)**
	Snort 2 Emerging Threats	70.07s **(99.37 % reduction)**	1205.3 MiB **(91.43% reduction)**
	Snort 3 Registered	69.45s **(94.45% reduction)**	1021.8 MiB **(95.67% reduction)**

for the Snort 2 Emerging Threats ruleset, as illustrated in the column "P4 Table Entries Deduplication" in Table 4.2. Our modifications to the aggregation stage addressed this issue, and with them, the Snort 2 Emerging Threats compilation time is now similar to the Snort 3 Registered one. More importantly, this work's compilation time for the Snort 2 Emerging Threats ruleset is 158 times faster than the compilation time with P4-ONIDS.

4.2.3 Modified P4-ONIDS Compiler Architecture

The modified P4-ONIDS compiler architecture is illustrated in Figure 4.2. The inputs of the original P4-ONIDS compiler are preserved, but the structure of the compilation steps and the content of some steps has been modified. The only structural change was the addition of the "NIDS Rule Deduplication" stage. In the new version, the "NIDS Rules Loader" not only loads IPv4, ICMP, TCP, and UDP rules but also parsers IPv6 rules. As for the "NIDS Rule Filter," the only types of rules removed are those with negated IPs. It also removes rules with UDP or TCP without flags that contain "any" for both the source and destination ports and at least one "any" IP address, as they are excessively inclusive. Lastly, the "P4 Rule Aggregation" stage still performs the same function, but the algorithm has been modified to improve the compiler's performance.

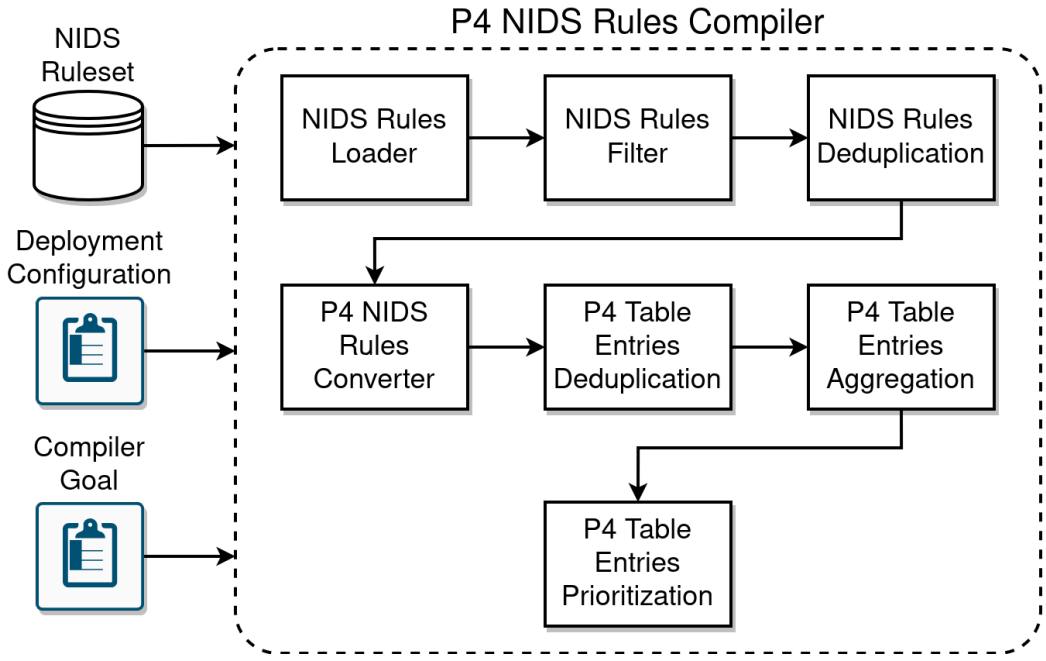


Figure 4.2: P4-ONIDS modified compiler architecture

5. NETWORK-WIDE ORCHESTRATION IN PROGRAMMABLE NETWORKS

Network Intrusion Detection Systems play a crucial role in securing computer networks, providing a comprehensive defense system that identifies multiple network attacks and suspicious behaviors. One method employed by NIDSs to detects attacks is by matching network traffic with signature rules created from known patterns of malicious activity. One drawback of this approach is that it is a time-consuming and resource-intensive procedure, as all packets undergo the pattern matching process. Consequently, this stage can become a bottleneck [42], leading to detection delays and potentially allowing attacks to go unnoticed, particularly in high-throughput networks with limited resources. Therefore, to minimize the likelihood of the NIDS not functioning as expected, improvements must address the saturation problem in the pattern matching stage.

With the advent of the Programmable Data Plane, researchers have explored ways to overcome the saturation problem in the NIDS pattern matching process by offloading certain functionalities to the PDP. A prevalent approach employed by these works is to offload NIDS signature rules to the PDP in order to pre-filter the network traffic for the NIDS host. The idea is to reduce the overall traffic that goes to the NIDS host, sending only suspicious packets to it, thus addressing the saturation problem of the packet matching stage without damaging the attack detection performance of the NIDS.

However, most of these approaches overlook the memory constraints of the forwarding devices when offloading the rules, resulting in sizable rulesets to offload that may not fit within the available memory. Furthermore, the majority of these works only consider one programmable switch to receive all rules, leading to scalability issues, underutilization of the network's total available memory, and decreased switch performance due to the overload of security functionalities in one device. Even in works that consider multiple network devices for rule offloading, the chosen method is to offload the same rules to all devices, causing the inefficient usage of the total available memory. This becomes particularly critical when considering that the limited memory size of programmable switches must be shared with other tables, such as packet forwarding and routing tables, which are constantly increasing [19], and other concurrent P4 programs. The combination of limited available memory, the need to share it with other networking functionalities, and inefficient memory usage by the proposed solutions can result in incomplete offloading of the NIDS ruleset, ultimately compromising the effectiveness of the solution.

Knowing the scalability constraints of NIDS, the memory restrictions of PDP, and the limitations of the state-of-the-art approaches, this work proposes leveraging the network-wide orchestration of programmable devices to enhance the NIDS performance. More specifically, NIDS rules are compiled into P4 table entries and strategically offloaded to

multiple PDP devices to pre-filter network traffic. These offloaded table entries ensure that only suspicious packets are forwarded to the NIDS host for further analysis, therefore reducing the overall volume of packets reaching the NIDS host.

Our solution, as in previous works, enhances the NIDS performance by overcoming the pattern matching stage saturation problem through the reduction of unnecessary packets reaching the NIDS host. However, unlike previous works and aiming to solve their limitations, our new approach mitigates the risk of not offloading all compiled rules by strategically distributing them to multiple devices while considering the devices' memory limitations and the network topology. The proposed solution consists of the following three components:

1. A *P4 NIDS rules compiler* to condense NIDS signature rules from open-source NIDS rulesets to a reduced set of P4 table entries without compromising the network traffic coverage of the original ruleset. This component was detailed in Chapter 4.
2. A *P4 data plane implementation* was devised to receive the compiler table entries, monitor suspicious flow, filter incoming traffic, and mirror suspicious packets for the NIDS instance.
3. A *network-wide table entries orchestrator* employing two novel algorithms to strategically offload the generated table entries to multiple devices while considering the resources of each device and the network topology.

This chapter outlines the proposed solution, starting with Section 5.1, where the architecture of the solution and the interaction among its components are presented. Afterward, in Section 5.2, it details the implementation of the P4 data plane, with its logic and the data structures used. Finally, in Section 5.3, the network-wide orchestrator and the table entries offloading algorithms are introduced. Combined, the orchestrator and algorithms are designed for network-wide memory- and topology-aware offloading of table entries to the PDP.

5.1 Architecture

The proposed architecture is illustrated in Figure 5.1. The control plane is responsible for converting the NIDS rules into entries in the P4 table through *P4 NIDS Rules Compiler*. This compiler, an evolution of the P4-ONIDS [37] compiler, is detailed in Chapter 4, where modifications and improvements made to it are also discussed. Following the conversion of rules into table entries, the control plane strategically offloads them to the data plane devices based on the network information. In the data plane, the forwarding

devices forward packets to the end hosts and filter the network traffic for the NIDS, sending only suspicious packets to it. Since our work focuses on NIDSs, suspicious packets are not dropped; instead, they follow their original paths, and a copy is sent to the NIDS host. Besides these two planes, there is also the NIDS engine, which analyzes the forwarded suspicious traffic and determines if they are actually network attacks, and the end hosts of the network.

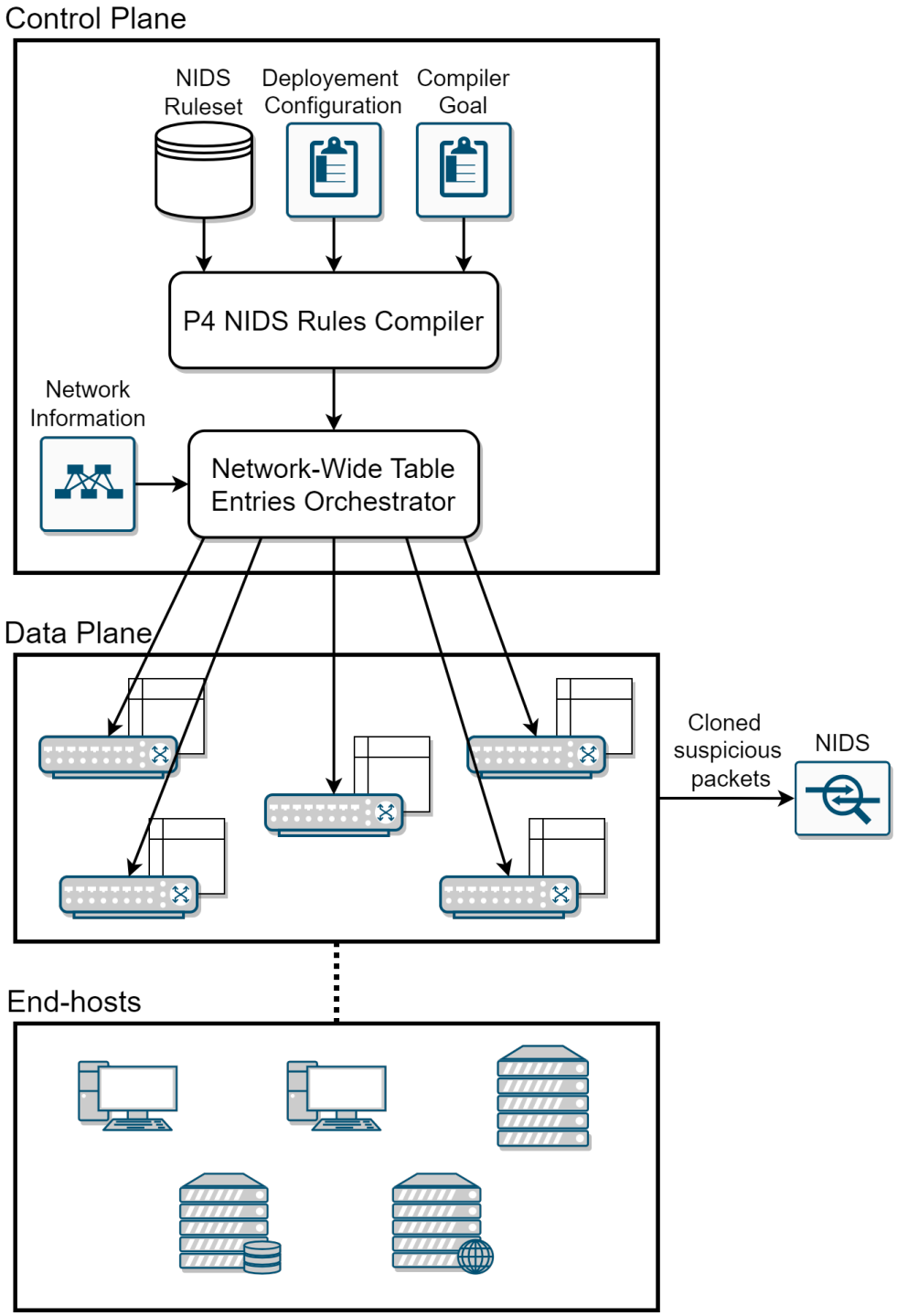


Figure 5.1: Proposed architecture

5.2 P4 Data Plane

The *P4 data plane* provides switching capabilities to forward network packets to their destination, as well as filtering functionalities for the NIDS engine. By monitoring suspicious flows based on the table entries created by the compiler and forwarding only suspicious packets to the NIDS host, the P4 data plane effectively reduces the volume of packets sent to the NIDS, addressing the saturation problem of the pattern matching stage. The P4 data plane of this work is built upon the one developed in P4-ONIDS [37], with some modifications. It employs a similar MAT to forward suspicious traffic to the NIDS and the same Count-Min data structure to track suspicious flows. However, certain adjustments have been made to the original P4-ONIDS data plane with respect to these two components.

5.2.1 Match Action Tables

Listing 5.1: P4 table that filters IPv4 traffic for the NIDS engine

```
1  table ipv4_nids {
2      actions = {
3          pass; clone_to_ids; NoAction;
4      }
5      key = {
6         meta.protocol:     exact;
7         hdr.ip.v4.srcAddr: ternary;
8         meta.srcPort:      range;
9         hdr.ip.v4.dst\_addr: ternary;
10        meta.dstPort:      range;
11        meta.flags:        exact;
12     }
13     size = 10240;
14     default_action = pass();
15     counters = ipv4_ids_table_hit_counter;
16 }
```

The *MATs* used by the data plane are exemplified in Listing 5.1. This listing presents the IPv4 NIDS table that compares incoming IPv4 packets with the table entries created by the P4 NIDS Rules Compiler to determine suspicious packets. Within this table are defined the actions to execute, the packet fields to match and their matching type (i.e, exact, ternary or range), the table size, the default action in the case of no match, and the counters to keep track of the table entries matched. When a packet's fields match a table entry, the packet is considered suspicious and flagged for cloning. The cloned packet is then sent to the NIDS host for further analysis, where the NIDS engine determines whether

the packet constitutes an attack or not. If a packet does not match any entries, this table does not take action on the packet.

As mentioned previously, this work expands on the P4-ONIDS data-plane MATs. The main enhancement is the support for IPv6 packets, achieved through the creation of an IPv6 packet parser and a new IPv6 NIDS table. This IPv6 table is similar to the table shown in Listing 5.1, with the only differences being its name, "ipv6_nids", the table entries counter name, and the replacement of the string "v4" for "v6" in the "hdr.ip.v4.srcAddr" and "hdr.ip.v4.dst_addr" key fields.

5.2.2 Tracking Suspicious Flows with a Count-Min Sketch

A NIDS operates by analyzing packets and flows to determine if they categorize suspicious behavior or even a network attack. The MAT introduced in Section 5.2.1 clones packets and forwards them to the NIDS, but lacks any flow monitoring functionalities. To keep track of the flows during their lifetime, a *Count-Min sketch* (see Section 2.4) is used. This probabilistic data structure functions as a frequency table for flows. Hash functions map a packet's 5-tuple (source and destination IP, source and destination port, and the protocol field) to positions or cells in the arrays, allowing the Count-Min sketch to track the packet count of a flow. In addition to flow tracking, the P4-ONIDS Count-Min uses a safeguard against saturation attacks such as DDoS attacks by sending only the first N packets of a suspicious flow to the NIDS. The Count-Min P4 implementation utilized in this work is virtually the same as the one by P4-ONIDS. It uses P4 registers to build the hash arrays and P4's native provided *hash* function to determine the positions in the arrays. The Count-Min *query* procedure is also the same, but the *update* procedure has been changed due to a different implementation of the Count-Min *aging mechanism*.

To maintain the Count-Min size within reasonable bounds without compromising its accuracy, it is crucial to reset the entries of idle flows from the Count-Min. However, the resetting of idle entries from Count-Min is not one of its standard operations. Therefore, P4-ONIDS adopted the timing-based aging method presented in [5] to handle idle flows. In the original paper outlining this method, the cells of the bloom filters are marked to indicate whether operations have been performed on them during a predefined time window or *phase*. Bloom filters are probabilistic data structures similar to the Count-Min sketches. At the start of a new phase, all flags are set to zero. By the end of a phase, non-flagged cells are reset, effectively cleaning idle flow entries, while flagged cells remain unchanged. Although the original paper employed this method for Bloom Filters, it can be easily adapted to Count-Min sketches.

Although still inspired by this approach, the Count-Min aging implementation of this work differs from the P4-ONIDS one due to the impracticality of the P4-ONIDS imple-

mentation. This impracticality arises from the fact that, in order to overcome the absence of loops in P4, P4-ONIDS proposed to unfold the loop that resets all Count-Min cells. However, the original loop iterates over all cells of the data structure, meaning that the number of lines of the unfolded loop will be at least the number of cells. For instance, suppose a small Count-Min sketch with depth equal to four and width equal to 2048; the unfolded loop would require at least 8192 lines of code. Due to this limitation, we changed the approach for resetting Count-Min cells.

The new approach involves aging only the cells related to the current suspicious packet in the switch that have not been updated in the previous phase. In this way, the cells are only reset when operations are performed on them, rather than resetting all cells every time a phase transition occurs, therefore, removing the necessity of unfolding the loop. The first step of this approach is the transition of a phase, and this logic is displayed in Listing 5.2. Similarly to P4-ONIDS, the phase transition in this work occurs every T seconds, also known as the aging threshold. It occurs each time an IP packet enters the network and before the cell aging process. After checking if the time has elapsed (line 6), the phase tracker is incremented (lines 8-10), and the timestamp of the last increment of the phase tracker is updated (line 12).

Listing 5.2: Updating the global phase tracker

```
1  apply {
2      if (hdr.ip.v4.isValid() || hdr.ip.v6.isValid()) {
3          bit<48> last_timestamp = 0;
4          last_phase_transition_time.read(last_time, 0);
5          bit<48> time_diff = standard_metadata.ingress_global_timestamp -
       last_time;
6          if (time_diff > 1000000 * T) {
7              bit<16> global_window_id = 0;
8              global_window_tracker.read(global_window_id, 0);
9              global_window_id = global_window_id + 1;
10             global_window_tracker.write(0, global_window_id);
11
12             last_phase_transition_time.write(0, standard_metadata.
       ingress_global_timestamp);
13         }
14         ...
15     }
16 }
```

With the phase transition defined, all that remains is to age the cells of the current packet on the switch. Listing 5.3 details this logic. The *update_count_min* procedure (line 1) occurs every time a packet matches a table entry and is deemed suspicious. First, the positions of the hash array or cells are determined based on the chosen hash functions (lines 2-7). Then, the main aging logic occurs (lines 10-21). It begins by reading the frequency of each related Count-Min cell and its respective phase. With this information, a cell is aged if the difference between its phase and the global phase is greater than or equal to two, indicating that the cell has not been updated in the previous phase (cate-

gorizing an idle flow). Each cell is aged separately since there is the possibility of hash collisions leading to shared cells, and resetting a shared cell means that another flow's frequency could be underestimated, contradicting the Count-Min definition. Following this process, the cells are incremented to account for the new packets while ensuring that they do not exceed the *N* threshold. The last step is to update the phase tracker for these cells to reflect the current global phase (lines 24-25).

Listing 5.3: Aging the Count-Min cells

```
1  action update_count_min(bit<16> protocol, bit<128> src_ip, bit<128> dst_ip, bit<16> src_port, bit<16> dst_port) {
2      bit<32> hash1;
3      bit<32> hash2;
4      ...
5
6      hash(hash1, HashAlgorithm.crc16, 32w0, {src_ip, dst_ip, src_port, dst_port, protocol}, COUNTMIN_WIDTH);
7      hash(hash2, HashAlgorithm.csum16, 32w0, {src_ip, dst_ip, src_port, dst_port, protocol}, COUNTMIN_WIDTH);
8      ....
9
10     bit<16> gl_phase = 0;
11     global_phase_tracker.read(gl_phase, 0);
12     bit<10> count;
13     bit<16> aux_phase;
14
15     cm_array1.read(count, hash1);
16     phase_id_tracker1.read(aux_phase, hash1);
17     cm_array1.write(hash1,  (gl_phase - aux_phase) >= 2? 1 :(count < N ? count + 1 : N+1));
18
19     cm_array2.read(count, hash2);
20     phase_id_tracker2.read(aux_phase, hash2);
21     cm_array2.write(hash2,  (gl_phase - aux_phase) >= 2? 1 :(count < N ? count + 1 : N+1));
22     ...
23
24     phase_id_tracker1.write(hash1, gl_phase);
25     phase_id_tracker2.write(hash2, gl_phase);
26     ...
27 }
```

5.2.3 Data Plane Workflow

Figure 5.2 summarizes the logic presented in this section by illustrating the data plane flow diagram for an incoming packet. When a packet enters the data plane, the first step is to determine the egress port of the packet is based on the forwarding entries provided by the control plane. Subsequently, the Count-Min aging mechanism updates

the global phase tracker. Then, the data plane checks if the packet is suspicious by comparing it with the table entries of the NIDS table. If the packet does not match any entry, indicating it is not suspicious, the packet is forwarded to the egress port determined at the beginning. If the packet matches an entry, meaning that it is suspicious, the Count-Min cells are checked to see if they need to be reset. If positive, the cells to age are zeroed. Following the aging process, the data plane verifies if the reported frequency is lower than the maximum packets to be sent per flow (the *N* threshold). If it is, the Count-Min cells are incremented, the packet is cloned to the NIDS host, and the original packet is forwarded to the determined egress port. If the reported frequency is greater than or equal to *N*, no action is taken, besides forwarding the packet to its egress port.

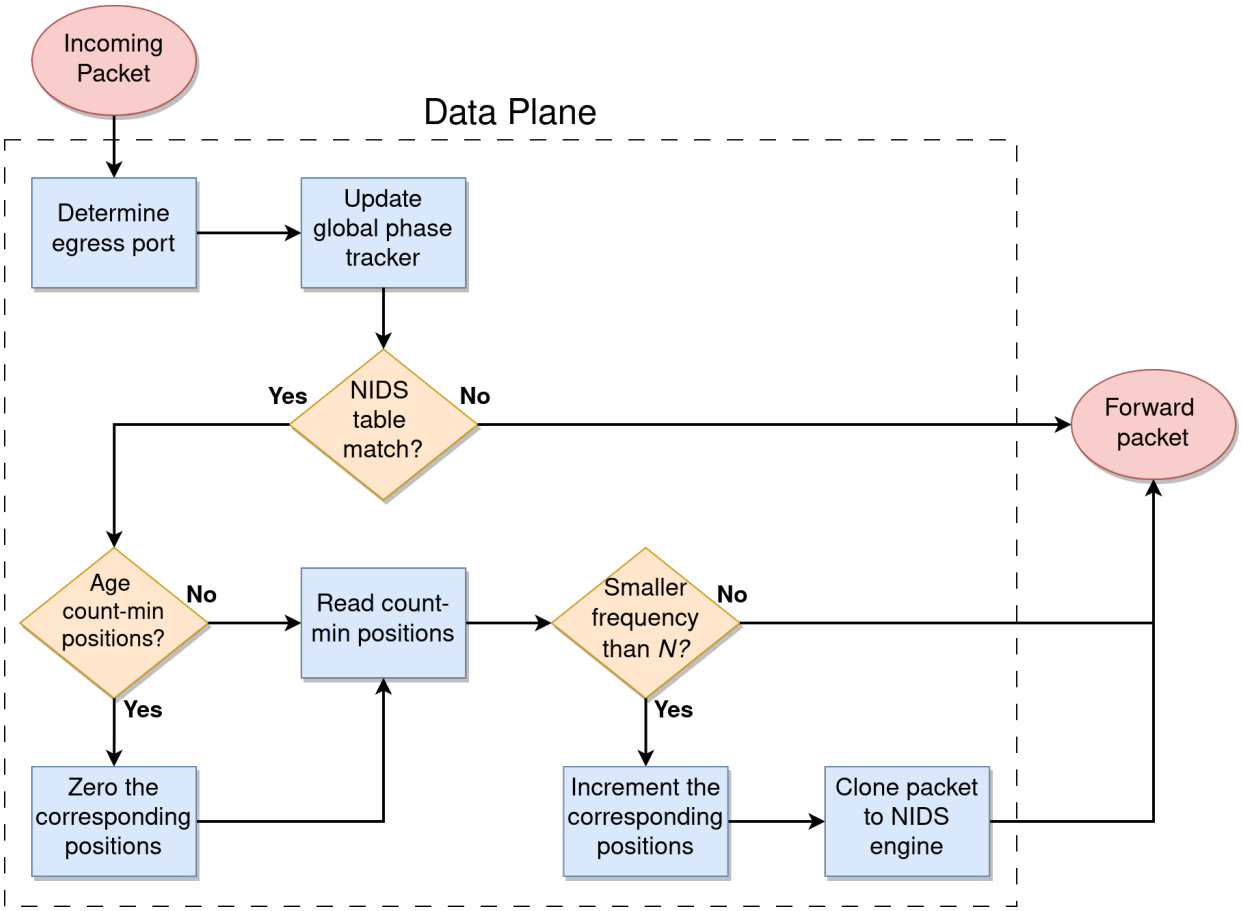


Figure 5.2: Data plane workflow

5.3 Network-Wide Table Entries Orchestrator

With the rules converted to table entries, and the data plane programmed to receive them and filter traffic for the NIDS host, the last step is to strategically distribute those table entries to multiple PDP devices by taking into account the memory of each device and the network topology. The *Network-Wide Table Entries Orchestrator* is respon-

sible for this task. Before presenting the algorithms used by the orchestrator to distribute the table entries, it is essential to understand the assumptions for these algorithms. First, malicious network traffic originates only from external devices, not internal ones, and flows through a **single** forwarding node referred to as the ***source switch***. Consequently, the network is modeled after a directed acyclic graph (DAG), featuring a single source and multiple sinks, where the network end-hosts are connected to only one forwarding device. Furthermore, the table entries have severity levels, and their destination addresses can take on three distinct values: "all", indicating all end hosts in the network; a subnetwork (e.g., "172.16.30.0/24") encompassing multiple end-hosts; or a specific IP address (e.g., "192.168.30.40/32") targeting only one end host. Lastly, all the switches in the network are P4 switches, and they have the same amount of available space for the compiled table entries.

The only input for the orchestrator is the network information, as depicted in Figure 5.1. The network information includes the network nodes (networking devices and end hosts) and their links, the end hosts' IP addresses and the available space for table entries in each switch. Taking advantage of this information, the orchestrator offloads the table entries to the data plane using the traditional approach, referred to as *Simple*, or the two novel proposed algorithms, named *First-fit* and *Best-fit*. These methods are described in the following:

- **Simple:** This approach, employed by the majority of the state-of-the-art works (except [31]) and considered as our baseline, offloads all table entries to a single switch randomly or ordered by their severity. In this work, the switch that receives all rules is the *source switch*. The drawbacks of this method are extensively discussed in Chapter 3.

- **First-Fit:** Modeled after the First-Fit algorithm for the bin-packing problem, this algorithm initially attempts to offload all table entries to the *source switch*. If unsuccessful, it starts distributing entries to downstream switches, beginning with those closest (in hops) to the *source switch*. The process continues until all table entries have been offloaded or there are no more switches available for offloading. Section 5.3.2 details this algorithm.

- **Best-Fit:** Inspired by the best-fit approach of the bin-packing problem, this algorithm aims to place each entry in its *best* position. If it cannot offload an entry to its *best* position, the algorithm searches for other switches where it can place the table entry without altering the desired result of offloading to the *best* position. If there is sufficient space to offload the missing entry to other device(s), it proceeds with the offloading process; otherwise, it continues searching until all switches are exhausted. This algorithm is described in Section 5.3.3

The approach presented by [31] is not evaluated in this work, since the results would be very similar to the Simple approach, but with a higher amount of offloaded table entries. This similarity arises because the set of table entries offloaded to the data plane would be the same, albeit replicated across all network devices. However, this replication would not offer improvements since the input traffic comes through only the *source switch*, and all suspicious packets would already be matched in it before reaching the other devices.

The remainder of this chapter is organized into three sections. First, in Section 5.3.1, the initial procedure performed by both algorithms proposed in this work is detailed. This procedure divides the table entries into subsets according to their destination address. Then, the First-Fit algorithm is detailed in Section 5.3.2. Finally, Section 5.3.3 outlines the Best-Fit approach step by step.

5.3.1 Grouping the Table Entries into Subsets

The two network-wide offloading algorithms introduced in this work employ the same grouping of table entries into subsets based on the destination address of the entries. This logic occurs before offloading the table entries to the PDP devices and is detailed in Algorithm 5.1. There are three types of subsets, the "generic" subset, subsets identified after a specific switch ID (e.g., "S1" and "S2"), and subsets identified by more than one switch ID. The "generic" subset encompasses table entries destined for all end hosts in the network (lines 4-5). For subsets named after a specific switch's ID, the pertinent table entries are those where the associated end hosts are connected to only one switch, which names the subset (lines 6-8 and lines 11-12). In subsets containing more than one switch ID, the related table entries are those where the associated end hosts are connected to more than one switch (lines 9-21). To name this multi-switch subset, the first step is to define the group of switches connected to the end hosts, called the *related switches* (line 10). Subsequently, the Lowest Common Ancestor(LCA) switch among the *related switches* is defined (line 14). The *source switch* can also be a *related switch*. With the *LCA switch* defined, the name of the subset is created by adding the *LCA switch* ID, followed by the "+" and the list of *related switches* separated by the "-" sign (lines 15-18). The order of the *related switches* is determined by the topological order of the network starting at the *LCA switch*.

To better understand how Algorithm 5.1 works, Figure 5.3 illustrates an example scenario. The figure presents a network topology with four switches and five end hosts, illustrating network links and highlighting the *source switch* in red. It further provides an example scenario by specifying the number of table entries to offload for each destination and the available space in each switch for these table entries. In this example, the 15

```
1: function create_subsets(network_info, table_entries)
2: subsets ← {}
3: for each table entry t in table_entries do
4:   if t.dst_addr is "all" then
5:     subset["generic"].append(t)
6:   else if t.dst_addr.CIDR is "/32" then
7:     sw_id ← get_switch(network_info, t.dst_addr)
8:     subset[sw_id].append(t)
9:   else if t.dst_addr.CIDR is not "/32" then
10:    related_switches ← get_related_switches(network_info, t.dst_addr)
11:    if len(related_switches) = 1 then
12:      subset[related_switches[0]].append(t)
13:    else
14:      LCA_switch ← LCA(network_info, related_switches)
15:      subset_name ← LCA_switch + " + "
16:      for each switch s in related_switches do
17:        subset_name ← +" − " + s
18:      end for
19:      subset[subset_name].append(t)
20:    end if
21:  end if
22: end for
23: return subsets
```

Algorithm 5.1: Group the table entries into subsets based on their destination address

table entries intended for all end hosts form the "generic" subset. The 10 entries for the "192.168.100.0/24" subnetwork constitute the "S1+S3-S4" subset, named after the S1 *LCA switch* and the S3 and S4 *related switches*. The 15 table entries destined to the "172.16.10.0/24" subnetwork are grouped into the "S2" subset, while the 10 table entries for the "10.0.10.100/32" end-host are assigned to the "S3" subset.

5.3.2 First-Fit Algorithm

The main idea of the First-Fit algorithm is to offload all table entries to the *source switch*, and if this is not possible, to the closest switches to the *source switch* in terms of hop count. Described in Algorithm 5.2, the algorithm operates as follows. Once the subsets are created (line 2), it orders the switches for offloading using the topological sort (line 4). The algorithm then iterates over the ordered switch list, aiming to offload all table entries associated with each switch that have not been offloaded to other switches (lines 5-12). Subsets related to a switch are those that contain table entries destined for the end hosts connected to that switch (line 6). Each time a switch initiates the offloading

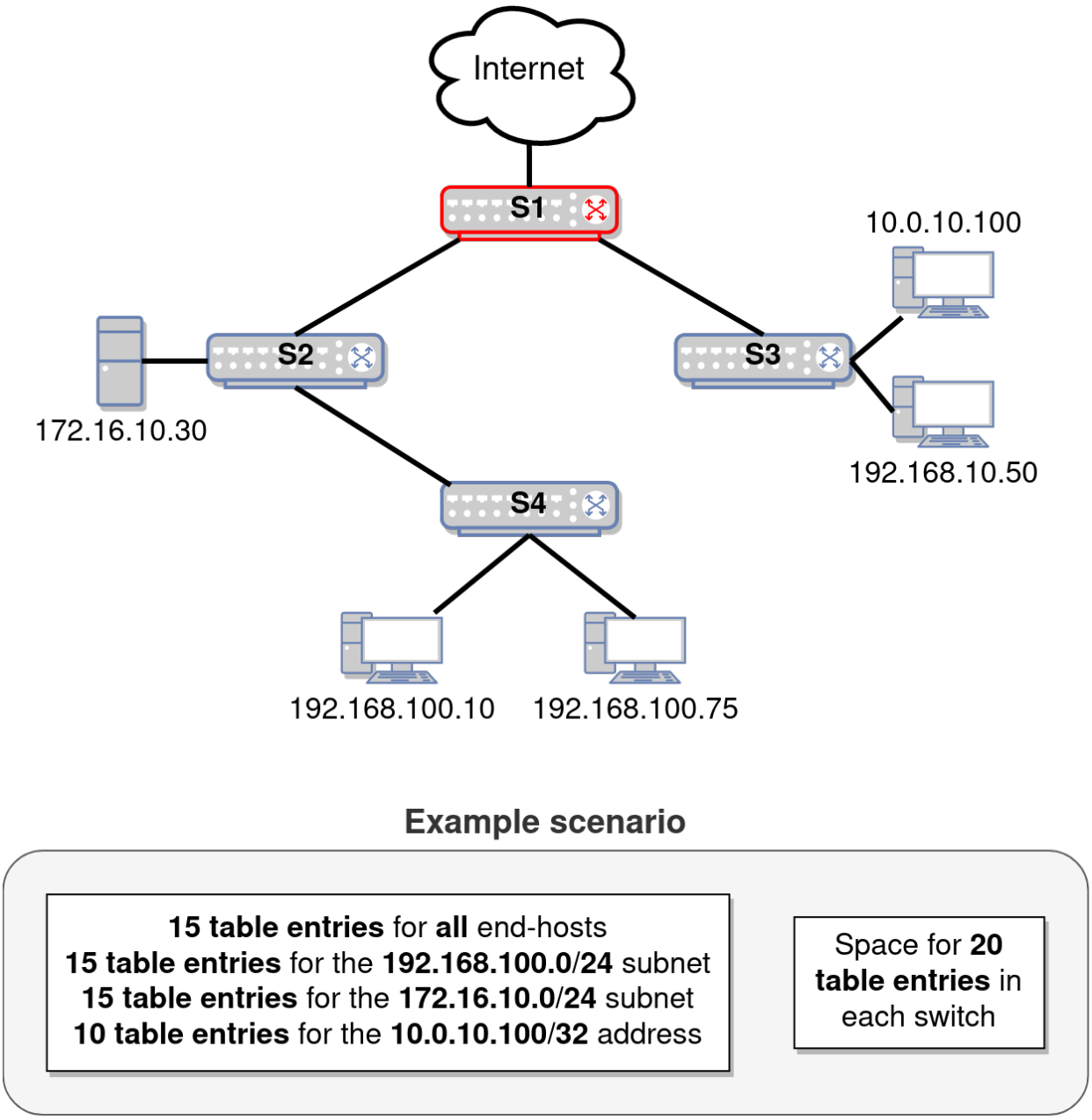


Figure 5.3: Example scenario

process, it examines the previously offloaded table entries to identify the entries it still needs to receive (line 6). In addition to determining the subsets to be offloaded for a switch, the algorithm also orders them (line 6). The prioritization for the subsets of a switch is as follows: the "generic" subset takes precedence, then the multi-switch subsets where the switch is a *related switch*, followed by the specific subset named after the switch, and lastly, subsets destined for "descendant" nodes. If all the table entries designated to a switch are already offloaded, the algorithm follows to the next switch (lines 7-9); otherwise, the entries are offloaded (line 10), and the information about the offloaded subsets is updated (line 11). After iterating over all switches, the algorithm finishes, having either offloaded all table entries or not.

```
1: function first_fit (network_info, table_entries)
2: subsets ← create_subsets (network_info, table_entries)
3: offloaded_subsets_runtime_info ← {}
4: ordered_switches ← topological_sort_switches (network_info)
5: for each switch sw in ordered_switches do
6:    entries_to_offload ← get_subsets_for_switch (sw, subsets, offloaded_subsets_runtime_info)
7:    if len (entries_to_offload) = 0 then
8:       continue
9:    end if
10:   missing_entries ← offload (sw, entries_to_offload)
11:   update_runtime_subsets_info (offloaded_subsets_runtime_info, missing_entries)
12: end for
```

Algorithm 5.2: First-Fit algorithm

Taking into account the example scenario in Figure 5.3 and the subsets created using Algorithm 5.1 (see Section 5.3.1), the First-Fit algorithm offloads as follows. First, it offloads the 15 table entries of the "generic" subset and five entries from the "S1+S3-S4" subset ("192.168.100.0/24" subnetwork) to S1. Moving to S2, as not all entries were offloaded, the algorithm offloads the 15 entries of the "S2" subset since it prioritizes subsets related to the S2 switch over "descendant" node subsets, i.e., the "S1+S3-S4" subset. For the remaining space of S2, it offloads five entries of the "S1+S3-S4" subset to S2. Proceeding to S3, the algorithm offloads the remaining 10 table entries of "S1+S3-S4" and the 10 table entries of the "S3" subset. Finally, S4 receives the remaining 5 table entries of the "S1+S3-S4" subset.

Overall, there were 15 duplicate table entries offloaded, all from the "S1+S3-S4" subset. This duplication occurred because the "S1+S3-S4" subset could not be offloaded at S1 alone, necessitating the replication of table entries to ensure the protection of end-hosts in both paths. The Firs-Fit algorithm successfully offloaded the table entries required by all end hosts of the example scenario, a capability that the Simple method could not offer, as it would have only offloaded 20 table entries out of the total of 55 entries.

5.3.3 Best-Fit Algorithm

The core of the Best-Fit algorithm is to offload the table entries directly to their *best* positions. Algorithm 5.3 details the Best-Fit logic, with its operation described next. Once the subsets are created (line 2), the algorithm orders them based on their type (line 3). First in line are the subsets named after specific switch IDs (e.g., "S1" and "S2"), arranged according to the topological sort of the switches. Next in the sequence are the

multi-switch subsets, ordered according to their related switches, again using the network topological sort. The "generic" subset takes the last position in this ordering.

With the subsets ordered, the algorithm proceeds with the offloading phase (lines 4-15). For subsets associated with specific switch IDs, it begins by offloading them to the switch that names the subset, their *best switch* (lines 7-14). If the algorithm fails to offload all entries to the *best switch*, it extends the offload to the switches in the path(s) between the *best switch* and the *source switch* starting with the switches closest to the *best switch* (lines 7-14). This ensures that packets destined for a switch still pass through all the entries related to that switch. Moving on to the multi-switch subsets, the offloading starts at their *LCA switch* (the multi-switch subsets *best switch*) as offloading to the *LCA switch* eliminates duplicate entries (lines 7-14). If not all table entries are offloaded to the *LCA switch*, the algorithm continues down the paths to the related switches (lines 5-10), offloading duplicate table entries when required (lines 7-14). Finally, the "generic" subset is offloaded, initially to the *source switch* (lines 7-14). In cases where offloading to the *source switch* is not feasible, the distribution of the "generic" subset extends to its descendant nodes, continuing until all table entries are successfully offloaded or there is no more available space (lines 7-14).

```
 1: function best_fit (network_info, table_entries)
 2: subsets ← create_subsets (network_info, table_entries)
 3: ordered_subsets ← order_subsets (network_info, subsets)
 4: for each subset sub in subsets do
 5:    related_switches ← get_subset_path (network_info, sub)
 6:    entries_to_offload ← sub
 7:    for each switch sw in related_switches do
 8:       missing_entries ← offload (sw, entries_to_offload)
 9:       if len(missing_entries) = 0 then
10:          break
11:       end if
12:       update_switch_info (network_info)
13:       entries_to_offload ← missing_entries
14:    end for
15: end for
```

Algorithm 5.3: Best-Fit algorithm

Based on the example scenario depicted in Figure 5.3 and the subsets created using Algorithm 5.1 in Section 5.3.1, the Best-Fit algorithm performs the following offload. Initially, it offloads all 15 table entries of the "S2" subset to S2. Subsequently, it offloads the 10 table entries from the "S3" subset to S3. Moving to the multi-switch subsets, it offloads the 15 table entries from the "S1+S3-S4" subset and five from the "generic" subset to S1. Finally, it offloads the remaining table entries of the "generic" subset: five table entries to S2, 10 table entries to S3, and five table entries to S4.

Overall, 10 duplicate table entries were offloaded, all from the "generic" subset. Although the Best-Fit algorithm successfully offloaded all table entries, five entries from the "generic" subset were not offloaded along the path to the S2 end host from the *source switch*. This omission leaves S2's end host susceptible to network attacks against which those table entries could have protected. By prioritizing subsets specific to individual switches, the Best-Fit approach may result in some multi-switch or "generic" table entries not being offloaded, contrasting with the First-Fit algorithm, which might not offload entries from subsets identified by individual switches.

6. EVALUATION

In this chapter, the offloading algorithms for network-wide table entries introduced in Section 5.3 are evaluated. Initially, the experimental setup, including the computing resources and input workload, is presented. Subsequently, this chapter details the evaluation of the data plane parameters to determine the optimal configuration for the data plane algorithms. Finally, the table entry offloading algorithms are evaluated in different scenarios.

6.1 Experimental Setup

The P4 NIDS Rules Compiler and the Network-Wide Table Entries Orchestrator are implemented in *Python 3.10*, while the P4 Data Plane is written in *P4-16* for the *Behavioral Model version 2* (BMv2) software switch. The P4 environment is created through the *P4 tutorials*[1] virtual machine, which includes all the necessary tools to emulate a network with Mininet[2] and run P4 programs. For this VM we allocated 22 cores, 146GB of memory, and 32GB of disk space. This VM was hosted inside another machine, an Ubuntu 22.04.3 VM with 26 cores, 200GB of memory, and 300GB of disk space. The NIDS software engine used in this evaluation to receive and analyze the packets forwarded from the data plane is *Snort 3*. The input ruleset for the compiler and the Snort 3 instance is the *Snort 3 Registered* ruleset, chosen for its extensive coverage of attack types. The deployment configuration for the Snort 3 engine in this evaluation is the one discussed in Chapter 4 and illustrated in Listing 4.3.

For the experiment's workload, we selected the *CICIDS2017* dataset[3] [36]. This dataset contains benign and malicious network traffic stored in PCAP (Packet Capture) files to resemble real-world data. It consists of five PCAPs, each corresponding to a day of the week and featuring realistic traffic. Monday simulates a normal day and contains only benign traffic, while Tuesday, Wednesday, Thursday, and Friday contain a mixture of benign and malicious traffic. The attacks implemented include Brute Force FTP, Brute Force SSH, DoS, Heartbleed, Web Attack, Infiltration, Botnet, and DDoS. Table 6.1 shows the number of packets per PCAP and the alerts generated by Snort 3 when using these PCAPs as input with the three rulesets of Table 4.1. Despite Monday containing only benign traffic, it generated a significant number of alerts, especially for the Snort 3 Community and Snort 3 Registered rulesets. This occurs because these rulesets contain many rules indicating suspicious traffic, which may not necessarily signify attacks, generating alerts

[1] https://github.com/p4lang/tutorials
[2] http://mininet.org/
[3] https://www.unb.ca/cic/datasets/ids-2017.html

even for benign traffic. Throughout the remainder of this chapter, this table serves as the *baseline data* which all subsequent experiments are compared.

Table 6.1: CICIDS2017 dataset and Snort rulesets

PCAP	**Amount of packets**	**Quantity of alerts**		
		Snort 3 Community	Snort 2 Emerging Threats	Snort 3 Registered
Monday	11709971	40384	7144	65903
Tuesday	11551954	51563	24652	83940
Wednesday	13788878	214634	20801	415093
Thursday	9322025	38204	40180	74275
Friday	9997874	37720	97914	67570

6.2 Data Plane Parameters Experiments

Before evaluating the proposed network-wide offloading algorithms, it is crucial to determine the optimal parameters for the data structures and algorithms of the P4 data plane described in Section 5.2. This evaluation centers on three parameters: the phase transition threshold for the Count-Min aging method (T), the number of packets to clone from a suspicious flow to the NIDS engine (N), and the width of the hash arrays of the Count-Min sketch (W). To determine the best configuration, this section explores various combinations of these three parameters and compares the results. Section 6.3 focuses on the T and W parameters, while Section 6.4 analyzes the N and W parameters.

In these experiments, two metrics are employed to establish the best configuration: the number of alerts generated by the NIDS engine compared to the baseline alerts and the number of cloned packets sent to the NIDS host compared to the baseline. The first metric is associated with the capacity of the NIDS engine to detect attacks when the network traffic is pre-filtered by the PDP. A similar match to the baseline alerts is mandatory. The second relates to the pattern matching saturation problem caused by the excessive number of packets this stage must process. Lower values are preferred, but only if the number of alerts does not decrease. The baseline data mentioned in these metrics are detailed in Table 6.1.

The topology used to evaluate the data plane parameters is shown in Figure 6.1. It includes a source host, the P4 switch with the data plane outlined in Section 5.2, the Snort 3 NIDS engine, and an end host. The source host sends all network traffic to the end host. The network traffic input consists of the five PCAPs from the CICIDS2017 dataset, transmitted at a rate of 1000 packets per second (pps) through the *tcpreplay*[4] tool. This

[4]https://tcpreplay.appneta.com/

rate was selected due to the performance limitations of the P4 BMv2 switch. In this topology, the original packets traverse the switch towards the end host, while suspicious packets are cloned and forwarded to the NIDS engine. The NIDS engine processes these packets to identify suspicious behavior, generating an alert if positive. This setup represents a worst-case scenario and provides an upper bound for the parameters since the switch must process all the network traffic.

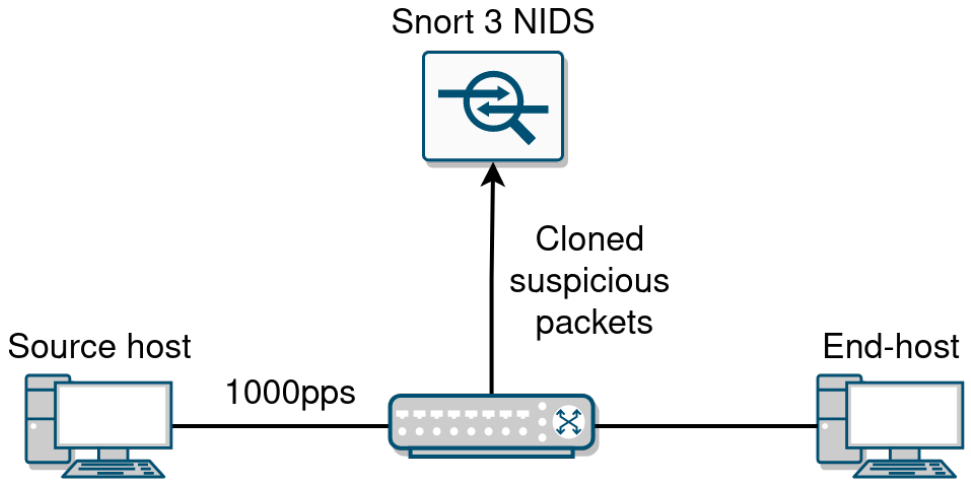


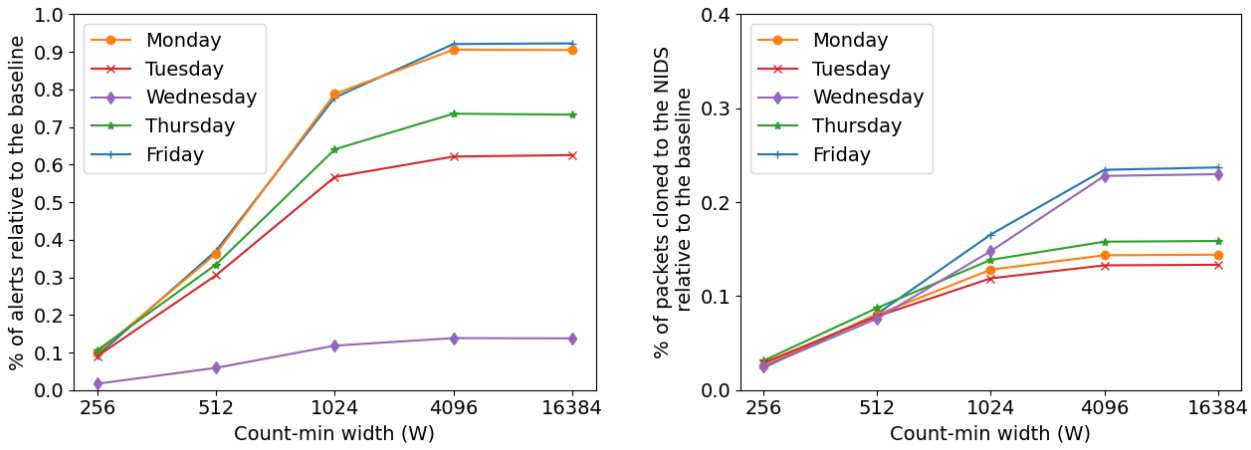
Figure 6.1: Topology used for evaluating the data plane parameters

6.3 Analyzing the *T* and *W* Parameters

We begin the experiments by investigating the phase transition threshold (T) and the Count-Min width parameter (W). Initially, we set the phase transition threshold to 10 seconds, then 25 seconds, and finally 50 seconds. Regarding the Count-Min width, we test with 256, 512, 1024, 4096 and 16384 with each of the T values. It is important to note that the width is associated with the length of the hash arrays, not the number of hash arrays, which remained fixed at 4 throughout the experiments. The four hash functions used were: *crc16*, *csum16*, *crc16_custom* and *crc32*. Future research should explore the variations of the number of hash arrays and the hash functions to assess its impact on performance. The N parameter is fixed at 10 packets for this initial set of experiments. Figure 6.2 displays the graphs for the number of alerts and cloned packets per PCAP compared to the baseline with T=10 while varying the W parameter. Figure 6.3 presents the same metrics for T=25, while Figure 6.4 shows it for T=50.

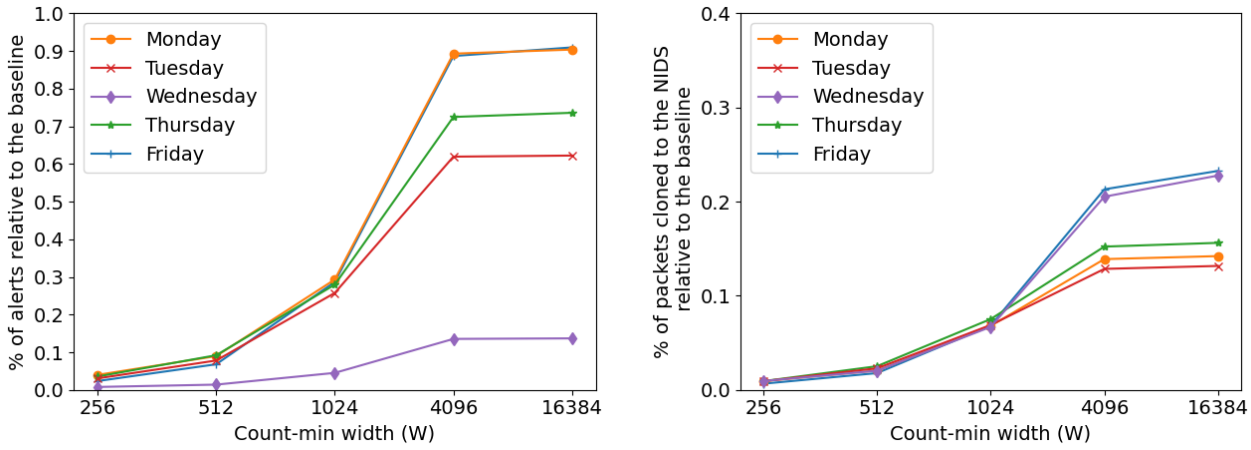
When analyzing these graphs, it is visible that larger W values lead to an increase in both the number of alerts and packets sent to the NIDS host, regardless of the T parameter. This is attributed to larger Count-Min widths causing fewer hash collisions. However, improvements due to increasing the Count-Min width are observed only up to W=4096, as experiments with W=16384 show similar results. Regarding the results for each PCAP,

the "Wednesday" PCAP had only a small fraction of the alerts compared to the baseline data for all T and W values. The "Tuesday" and "Thursday" PCAPs showed poor results with small Count-Min widths, but as it increased, the number of alerts increased, reaching over 60% correspondence with the baseline alerts. The PCAPs for "Friday" and "Monday" initially had poor alert results, but as the width increased, the alerts reached a 90% match with the baseline data. For all PCAPs and experiments, the number of packets sent to the NIDS host accounted for less than 25% of the baseline data. Lastly, comparing the results among the different T values reveals that the smaller thresholds generated more alerts. This is attributed to the more frequent aging of the Count-Min entries and, consequently, the increase in packets cloned to the NIDS. However, when using larger W values, the impact of the T threshold is not as pronounced.



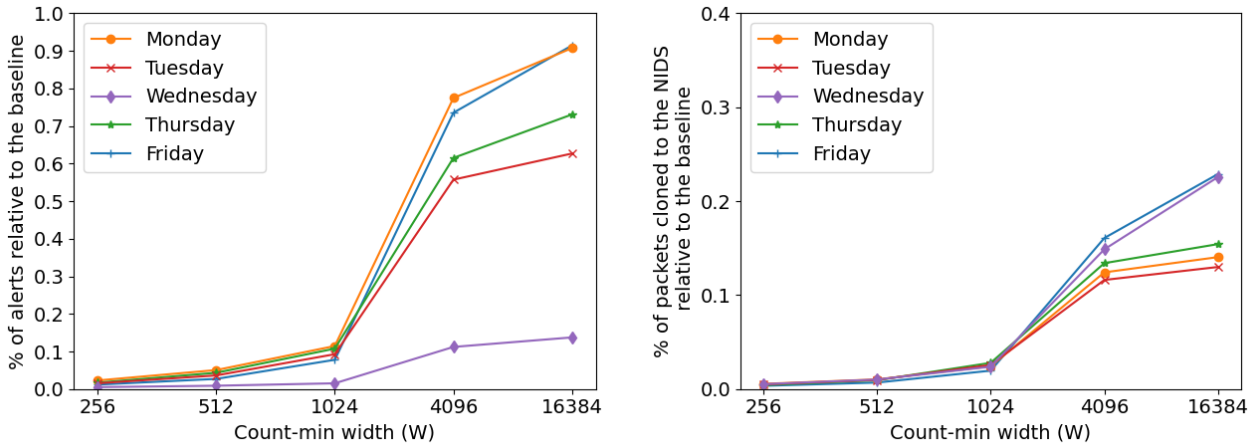
(a) Alerts (b) Packets cloned

Figure 6.2: Percentage of alerts and packets cloned for N=10 and T=10



(a) Alerts (b) Packets cloned

Figure 6.3: Percentage of alerts and packets cloned for N=10 and T=25

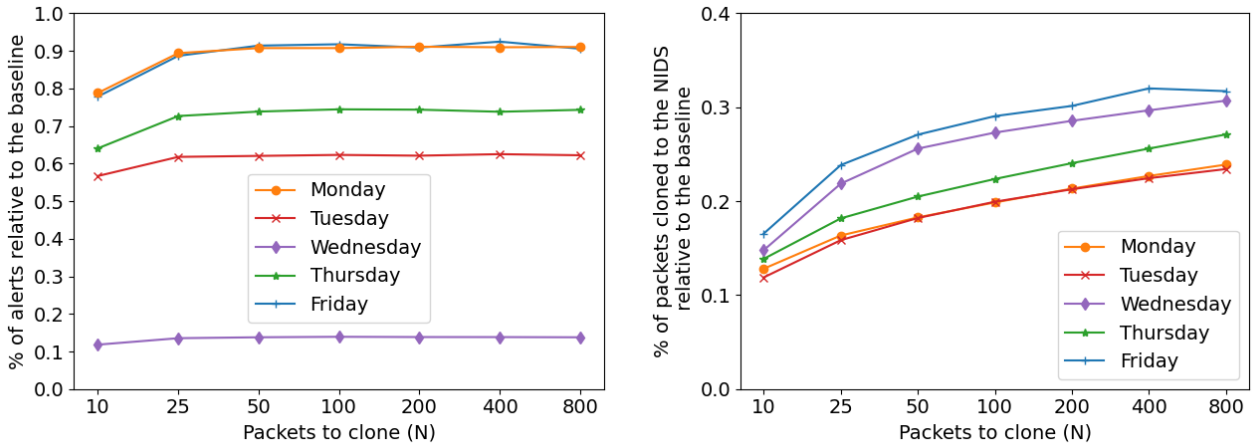


(a) Alerts (b) Packets cloned

Figure 6.4: Percentage of alerts and packets cloned for N=10 and T=50

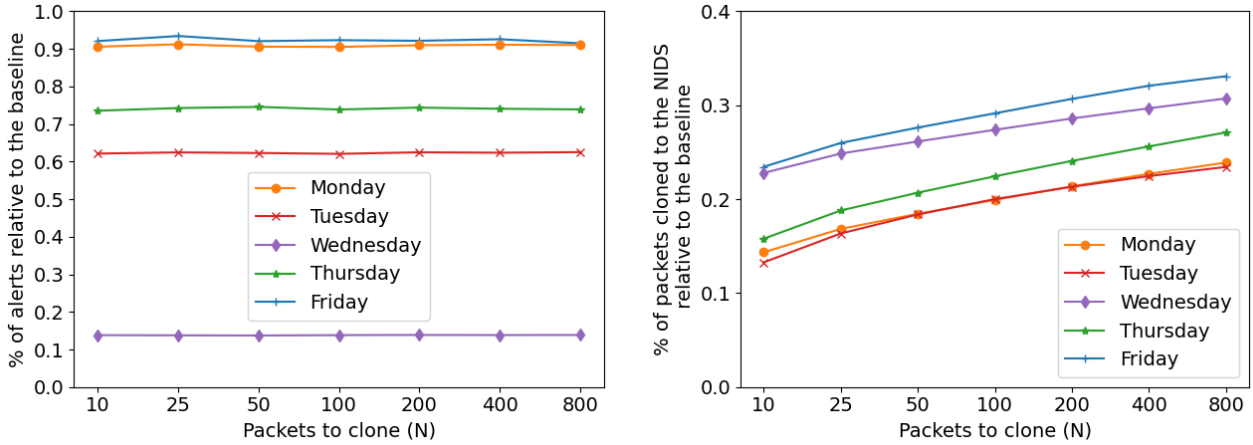
6.4 Analyzing *N* and *W* Parameters

After experimenting with T and W parameters, the number of packets required to send a suspicious flow to the NIDS engine (threshold N) was evaluated. To do this, T was fixed at 10 seconds, since this value produced the highest number of alerts generated for all PCAPs, and the W parameter varied between 1024, 4096 and 16384, since these values resulted in the best outcomes with T=10. The N parameter was tested with 10, 25, 50, 100, 200, 400 and 800 packets for each one of the W values. Figure 6.5 shows the number of alerts and cloned packets per PCAP compared to the baseline data with W=1024 while varying the N parameter. Figure 6.6 presents the same metrics for W=4096, while Figure 6.7 shows it for W=16384.



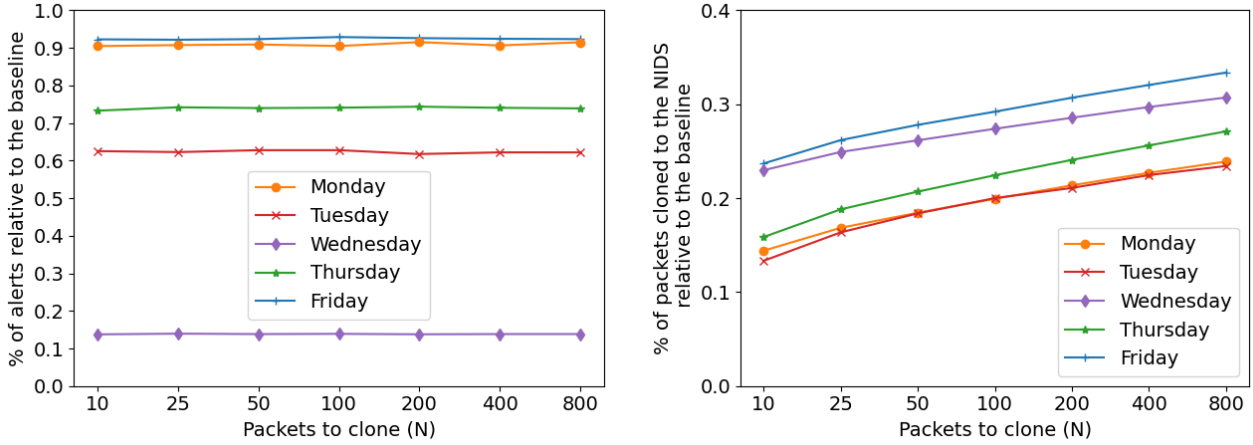
(a) Alerts (b) Packets cloned

Figure 6.5: Percentage of alerts and packets cloned for T=10 and W=1024



(a) Alerts (b) Packets cloned

Figure 6.6: Percentage of alerts and packets cloned for T=10 and W=4096



(a) Alerts (b) Packets cloned

Figure 6.7: Percentage of alerts and packets cloned for T=10 and W=16384

As observed in these graphs, the number of alerts compared to baseline data remained practically the same for experiments and PCAPs. Only in the results for W=1024 (Figure 6.7b), it is possible to see a growth in alerts when increasing N from 10 to 25 packets. Despite the alerts remaining the same, the number of packets sent to the NIDS host increased as the N parameter grew for all PCAPs and W sizes. This demonstrates that despite the fact that more packets reached the NIDS host, the additional packets were not suspicious or, at least, did not trigger further alerts. Based on these results, we selected the following configuration for the network-wide offloading algorithms evaluation: ***T*=10** for the phase transition threshold of the Count-Min aging method, as it demonstrated the best results for all W sizes; ***W*=16384** for Count-Min hash arrays width, as it yielded the highest number of alerts among all tested T values; and ***N*=200** for the number of packets to send from a suspicious flow to the NIDS engine, as an average value, considering that all N values tested showed good results.

With the data plane parameters defined, the memory footprint of the P4 data plane can be calculated. The Count-Min sketch and its aging method use registers to store information. A register is a vector containing multiple positions or entries of a certain type of data. The Count-Min sketch entries store 10-bit values, whereas the aging method arrays' entries used to track the phase of each Count-Min position store 16-bit values. These sizes can be adjusted to better fit the desired behavior, but for our experiments, they are sufficient. Regarding the memory footprint of the P4 data plane, we have the following: the Count-Min sketch uses four hash arrays each with 16384 entries of 10 bits, resulting in 82KB of memory, while the aging method uses 16-bit entries for the same number of entries, resulting in 131KB of memory. Combined, the total memory usage for the P4 data plane to track suspicious flows is 213KB, which is quite small.

6.5 Evaluating the Network-Wide Offloading Algorithms

With the data plane parameters selected, the next set of experiments evaluates the algorithms outlined in Section 5.3, which the network-wide orchestrator employs to offload the compiled table entries. The three algorithms discussed are as follows: Simple, First-Fit, and Best-Fit. They are assessed in two network topologies, linear and tree, with varying memory availability in the switches. The memory availability scenarios are described in Table 6.2, and are based on the number of table entries generated by the compiler for the Snort 3 Registered ruleset. The amount of space for the table entries is the same in all the switches in the network. Three main metrics guide this evaluation: the number of alerts generated by the NIDS engine compared to the baseline alerts, the number of cloned packets sent to the NIDS host compared to the baseline, and the number of table entries offloaded to the data plane. In the two topologies considered in this evaluation, there is only one source host (simulating the external network) responsible for sending all packets from the CICIDS2017 dataset at a rate of 1000 pps via the *tcpreplay* tool.

Table 6.2: Memory availability scenarios

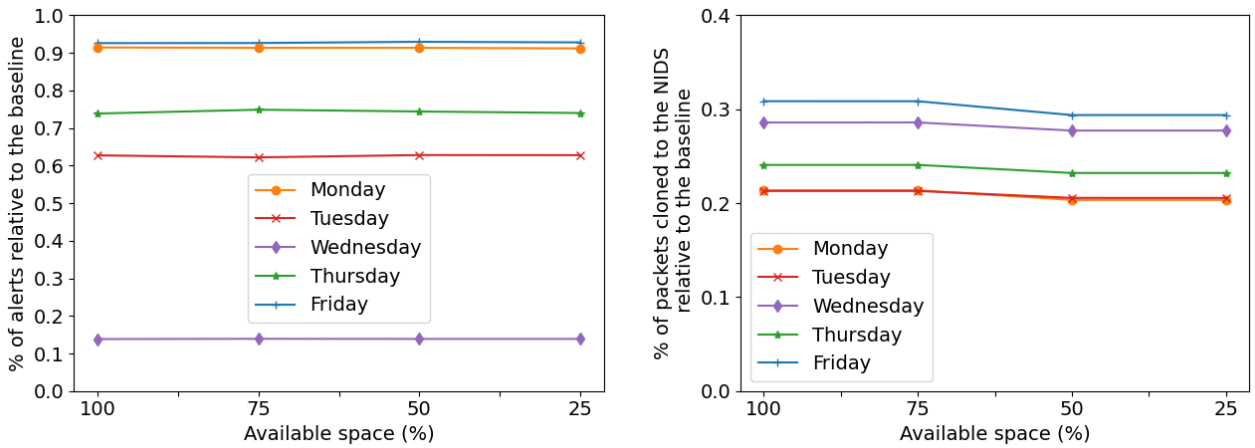
Memory availability scenario	**Table entries available per switch**
100%	1465
75%	1099
50%	733
25%	367

For the Simple algorithm, which is our baseline algorithm and is used in most state-of-the-art work, the results are independent of the topology, since it only offloads

to the *source switch* (and one of the presuppositions is the existence of only one *source switch*), where the outside traffic enters the network. The results of the Simple algorithm are detailed in Section 6.5.1. For the First-Fit and Best-Fit algorithms, their evaluations on different topologies and the corresponding results are detailed in Section 6.5.2 for the linear topology and Section 6.5.3 for the tree topology. The host IP addresses in the topologies are based on the attackers' and victims' IPs from the CICIDS2017 dataset. Forwarding rules are installed on the switches to correctly direct traffic to the end hosts (victims) from the *source switch*.

6.5.1 Simple Algorithm Results

Figure 6.8 presents the generated alerts and the cloned packets in the NIDS engine for the Simple algorithm when prioritizing the offloading of the table entries with higher severity. The results are, at first, unexpected because the alerts remained stable in all scenarios, although only 25% of all table entries were offloaded. A closer analysis reveals that only a few table entries match most of the traffic considered suspicious (i.e., the traffic cloned to the NIDS engine), and these entries are always offloaded in the scenarios outlined in Table 6.2. These entries are always offloaded due to their high severity, making them a priority in the offloading process. In addition to a few table entries that match most of the suspicious traffic, the alerts generated by Snort predominantly originate from a small set of rules.



(a) Alerts (b) Packets cloned

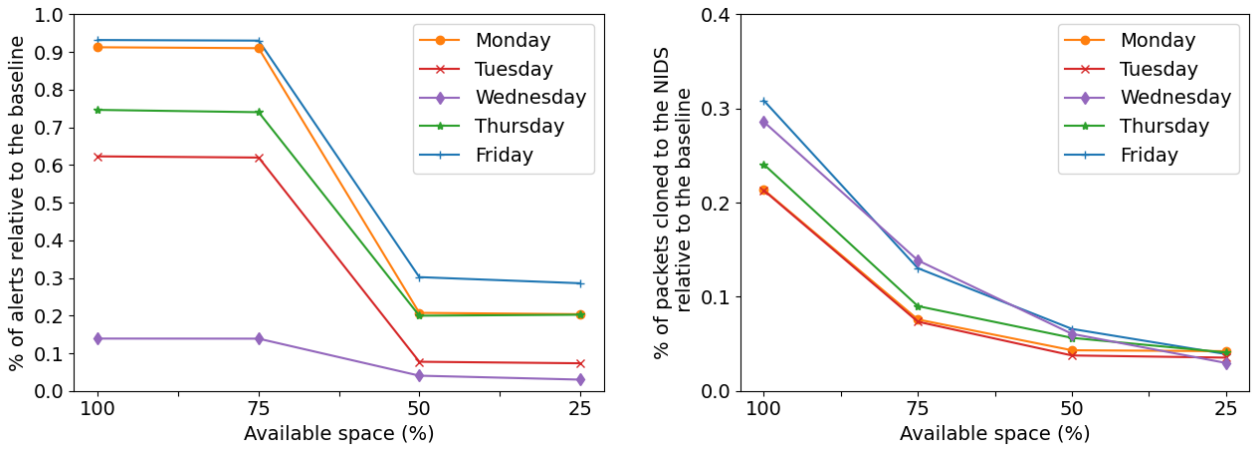
Figure 6.8: Percentage of alerts and packets cloned for the Simple algorithm with table entries ordered by severity

To exemplify this condition, consider the high-severity table entry presented in Listing 6.1, designed to match TCP SYN packets coming from "any" location and reaching the end hosts in subnet "192.168.10.0/24" at "any" port.

Listing 6.1: Table entry matching most of the Friday PCAP traffic

```
alert 0x6 0.0.0.0&&&0.0.0.0 0->65535 192.168.10.0&&&255.255.255.0 0->65535
    00000010
```

Listing 6.1 entry matched 3,921,168 packets for the Friday PCAP, approximately 40% of the total packets for this PCAP. This entry of high severity was always offloaded, contributing to the stable number of cloned packets in the NIDS observed in Figure 6.8b. Although no other table entry matched as many packets, a few other high-severity entries matched several hundred thousand packets. Together, these entries constituted practically all traffic forwarded to the NIDS engine, despite being a small group. This monopolizing behavior is also seen in the alerts generated by the Snort 3 engine. For the Friday PCAP, the Snort rule with Signature ID (SID) 254, generated 36,325 alerts, almost half of all alerts from the baseline data.



(a) Alerts

(b) Packets cloned

Figure 6.9: Percentage of alerts and packets cloned for the Simple algorithm with randomly ordered table entries

Due to the concentration of alerts and packets to clone in a few high-severity rules and table entries across all PCAPs, a new approach was necessary to demonstrate the importance of memory and topology-aware algorithms that strategically offload table entries to multiple PDP devices. To this end, the table entries are offloaded without considering their severity, and the offloading order is random. The results of this new approach are shown in Figure 6.9, and in them, the decrease in performance is evident when there is limited space available to offload the table entries. In the scenario with 75% memory availability, the number of packets sent to the NIDS sharply declined, while the number of alerts remained constant. This implies that certain table entries that matched several packets, but not suspicious ones, were not offloaded to the PDP. When the available space was further reduced to 50% of the size of the table entries, the number of alerts dropped abruptly, indicating that the table entries that corresponded to a significant volume of

suspicious traffic were not removed. For the remainder of this evaluation and to maintain consistency, the table entries to be offloaded are the randomly ordered table entries used in this new approach.

6.5.2 Linear Topology Experiments

The linear topology used in the experiments is illustrated in Figure 6.10. It consists of one *source switch* (S1) (with a red border) where the network traffic enters the network, and four forwarding switches (S2, S3, S4 and S5) connected to the end hosts. The *source switch* is also connected to one host (a firewall).

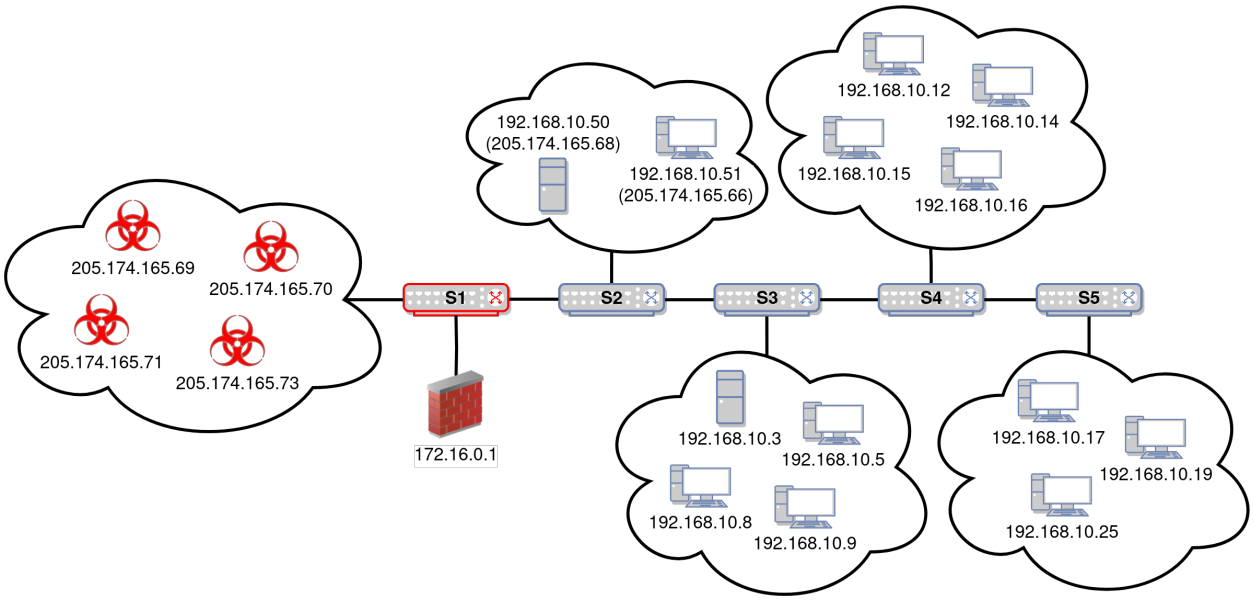


Figure 6.10: Linear topology diagram

Before offloading the table entries to the data plane for the First-Fit and Best-Fit algorithms, there is the initial step of dividing the entries into subsets based on their destination address. Considering the linear topology of Figure 6.10 and the table entries compiled from the Snort 3 Registered ruleset, the subsets are the following: the "generic" subset containing 77 table entries with the destination set to all hosts in the network; the "S2+S2-S3-S4-S5" multi-switch subset comprising 347 entries that contain the "192.32.10.0/24" network as the destination (to understand the name of this subset go to Section 5.3); the "S1" subset consisting of 347 entries with the "172.16.0.0/16" network as the destination; and the "S2" subset with 694 entries utilizing the "205.174.165.68" or "205.174.165.66" IP as the destination address.

With the subsets defined, the table entries are offloaded. Figure 6.11 illustrates the number of unique and duplicate table entries offloaded to the PDP by each algorithm, while Table 6.3 specifies the switches to which the entries were offloaded. The number of

table entries offloaded by the Simple algorithm is directly linked to the available memory of the *source switch* (S1). In contrast, the other two algorithms, First-Fit and Best-Fit, are not limited by this constraint, since they consider multiple networking devices. This advantage is evident in Figure 6.11, as the only scenario in which they were unable to offload all the table entries is the 25% memory availability scenario. Even in this case, they managed to offload at least half of all table entries.

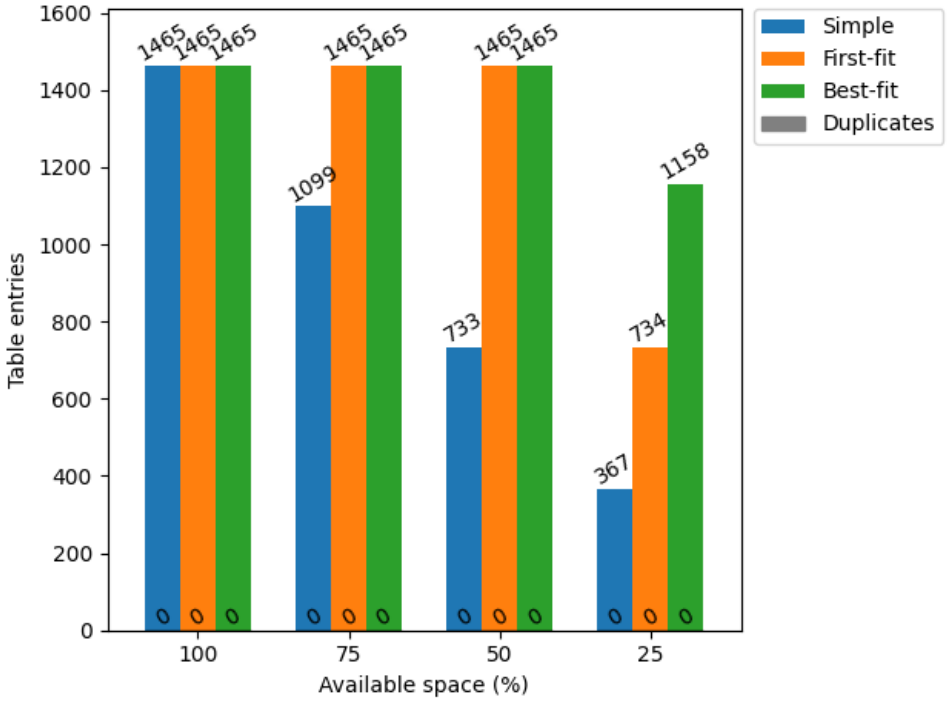


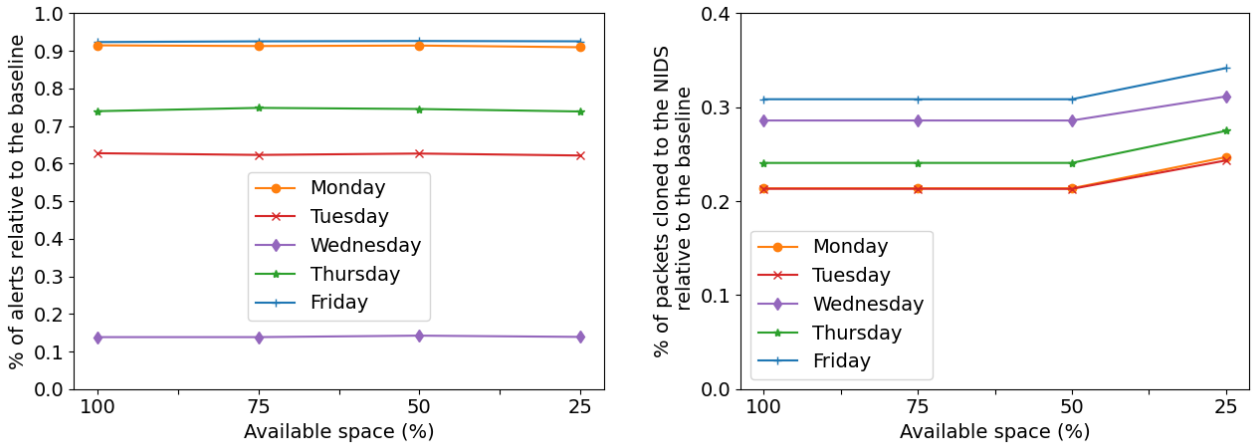
Figure 6.11: Table entries offloaded to the data plane by each algorithm with the linear topology

Table 6.3: Table entries per switch for the First-Fit and Best-Fit algorithms in the linear topology

Memory availability scenario	**First-Fit**					**Best-Fit**				
	S1	**S2**	**S3**	**S4**	**S5**	**S1**	**S2**	**S3**	**S4**	**S5**
100%	1465	0	0	0	0	771	694	0	0	0
75%	1099	366	0	0	0	771	694	0	0	0
50%	733	732	0	0	0	733	732	0	0	0
25%	367	367	0	0	0	367	367	367	57	0

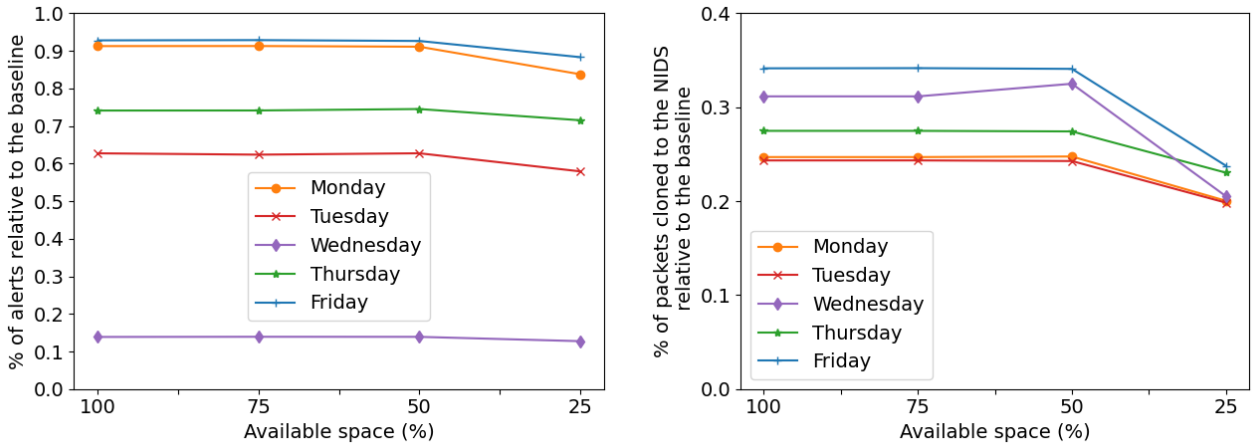
Figure 6.12 illustrates the alerts generated and the cloned packets sent to the NIDS engine for the *First-Fit* algorithm. The number of alerts remains constant for all PCAPs, even in the 25% scenario, where only half of all table entries are offloaded. This stability is attributed to the grouping into subsets of the table entries and the prioritization of the "generic" and "S2+S2-S3-S4-S5" subsets, as they contain the table entries responsible for cloning most of the packets to the NIDS engine. An interesting observation arises

in the 25% scenario of Figure 6.12b: the number of packets sent to the NIDS increased, but the alerts did not. This occurs because, in the 25% scenario, the First-Fit algorithm offloads all "generic" table entries and some "S1" entries to S1, and all "S2+S2-S3-S4-S5" (i.e., the table entries with "192.32.10.0/24" as destination) and some "S2" entries to S2. With this distribution, some packets first match the "generic" table entries and, later, the "S2+S2-S3-S4-S5" table entries, causing the same packet to be cloned twice to the NIDS host. This does not happen when the "generic" and "S2+S2-S3-S4-S5" table entries are offloaded to the same switch (50%, 75% and 100% scenarios), since P4 tables permit just one match. To avoid overcounting alerts in this situation, packets sent twice to the NIDS engine that generate the same alert are not counted.



(a) Alerts (b) Packets cloned

Figure 6.12: Percentage of alerts and packets cloned for the First-Fit algorithm with the linear topology



(a) Alerts (b) Packets cloned

Figure 6.13: Percentage of alerts and packets cloned for the Best-Fit algorithm with the linear topology

Figure 6.13 shows the results for the *Best-Fit* algorithm. Unlike the First-Fit results, both the number of alerts and cloned packets decreased in the 25% scenario, although the Best-Fit algorithm offloaded more table entries in this scenario (see Figure 6.11). This occurs because the Best-Fit algorithm prioritizes the subsets destined for a specific switch, namely the "S1" and "S2" subsets, which do not contain the most relevant entries for the input PCAPs. Due to this prioritization, the Best-Fit algorithm offloads the "S2+S2-S3-S4-S5" and "generic" subsets only at S3 and S4, respectively, leaving the hosts in S1, S2, and S3 exposed to malicious packets and the NIDS oblivious to them. Another observation is a slight increase in packets forwarded to the NIDS host in the 50% scenario for the Wednesday PCAP. The reason for this is similar to the First-Fit increase in the 25% scenario: the table entries in different switches match the same packets, being sent twice to the NIDS.

6.5.3 Tree Topology Experiments

The tree topology used in this evaluation is shown in Figure 6.14. It includes one *source switch* (S1) (with a red border) in which the network traffic enters the network, three switches (S3, S4, S5) connected to end hosts, and one switch without hosts (S2). The *source switch* is also connected to one host (a firewall).

The subsets of table entries created for the First-Fit and Best-Fit algorithms, based on the tree topology illustrated in Figure 6.14 and the table entries compiled from the Snort 3 Registered ruleset, are as follows: the "generic" subset containing 77 table entries with the destination set to all hosts in the network; the "S1+S3-S4-S5" multi-switch subset comprising 347 entries that contain the "192.32.10.0/24" network as the destination (to understand the name of this subset go to Section 5.3); the "S1" subset consisting of 347 entries with the "172.16.0.0/16" network as the destination; and the "S3" subset with 694 entries utilizing the "205.174.165.68" or "205.174.165.66" IP as the destination address.

After the subsets are defined, the table entries are offloaded. Figure 6.15 displays the number of unique and duplicate table entries offloaded to the data plane by each algorithm, and Table 6.4 details the switches to which the entries were offloaded. The number of table entries offloaded by the Simple algorithm is the same as in the linear topology. For the First-Fit and Best-Fit algorithms, they offloaded all entries and protected all hosts down to the 50% scenario. For the 25% scenario, not all entries were offloaded, and a considerable number of offloaded table entries are duplicates. The presence of duplicate table entries in the tree topology for both proposed algorithms is attributed to the existence of two paths: one going from S1 (the *source switch*) to S2 and the other going to S3.

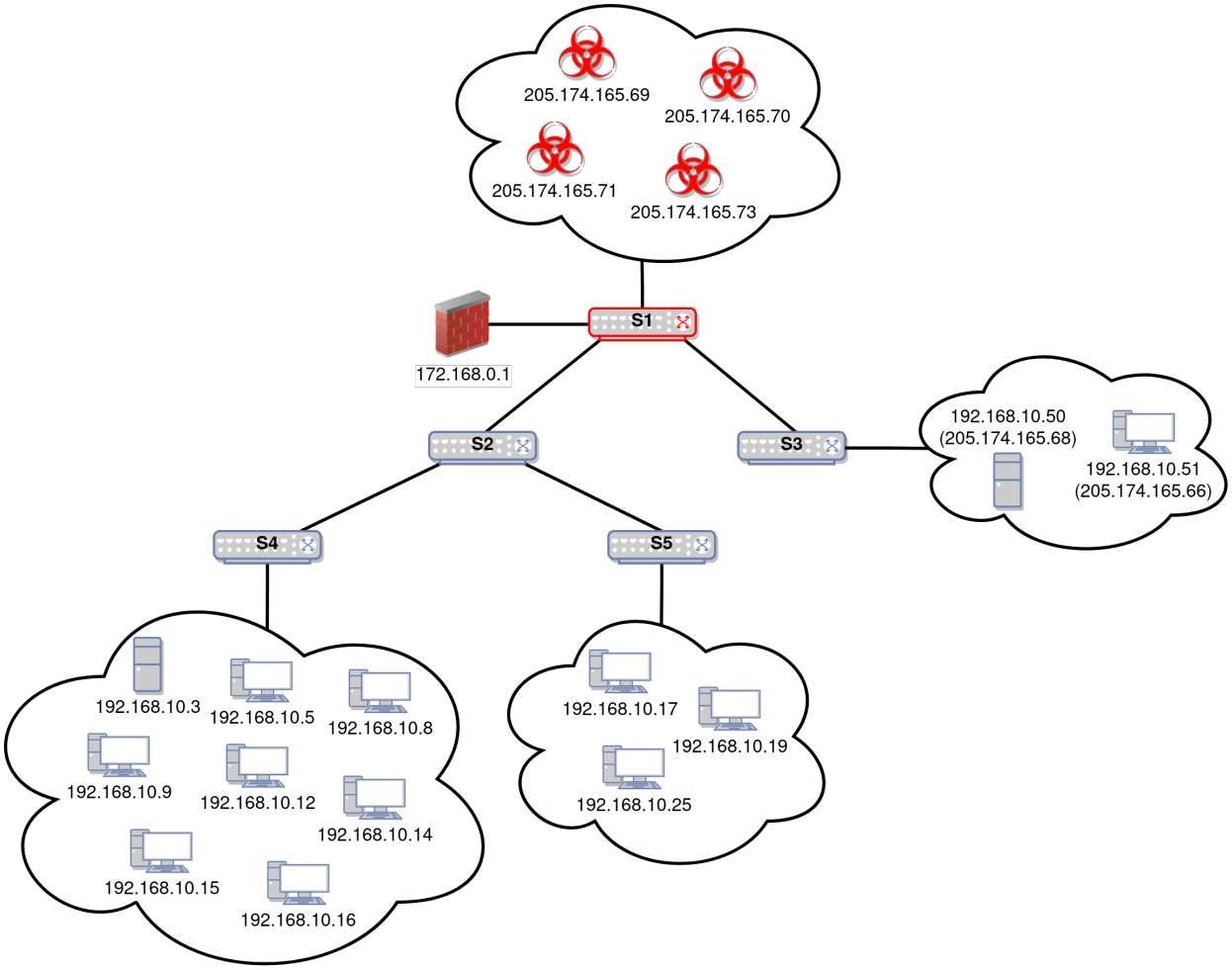


Figure 6.14: Tree topology diagram

Table 6.4: Table entries per switch for the First-Fit and Best-Fit algorithms in the tree topology

Memory availability scenario	**First-Fit**					**Best-Fit**				
	S1	**S2**	**S3**	**S4**	**S5**	**S1**	**S2**	**S3**	**S4**	**S5**
100%	1465	0	0	0	0	771	0	694	0	0
75%	1099	0	366	0	0	771	0	694	0	0
50%	733	38	732	0	0	733	38	732	0	0
25%	367	367	367	0	0	367	367	367	57	57

The percentage of generated alerts and cloned packets sent to the NIDS engine for the *First-Fit* algorithm with tree topology is shown in Figure 6.16. The results of the First-Fit algorithm in tree topology are similar to those in linear topology. The alerts remain stable throughout the four memory availability scenarios, whereas the number of cloned packets increases with the 25% scenario due to table entries in different switches sending the same packets. Regarding the duplicate table entries for the 50% scenario, the 38 duplicate table entries are from the "S1+S3-S4-S5" subset offloaded both to S2 and S3.

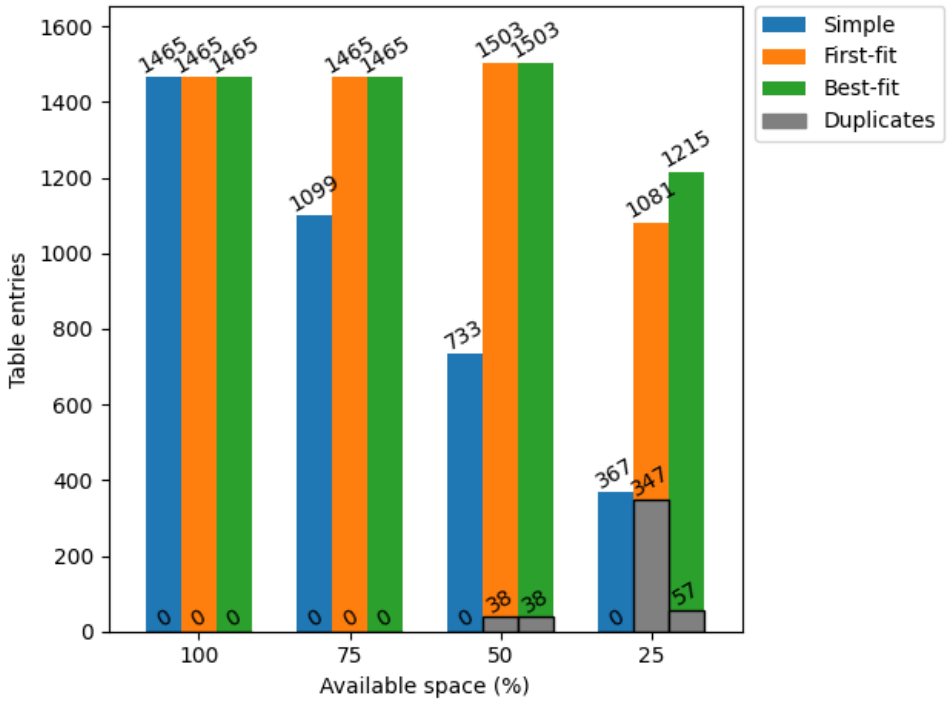
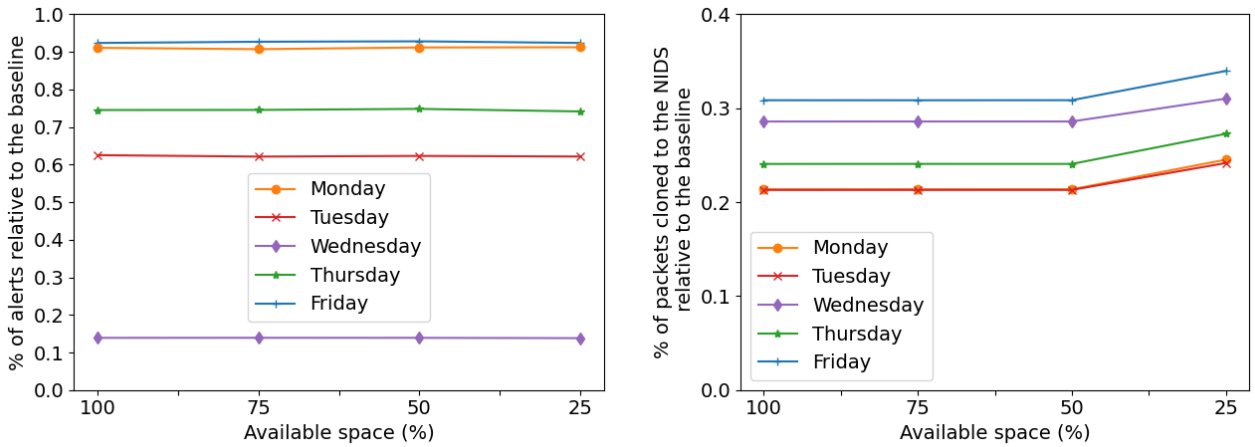


Figure 6.15: Table entries offloaded to the data plane by each algorithm with the tree topology

This duplication occurs because these switches are on separate paths, where both have end hosts that need the table entries of this subset that were not offloaded to S1. In the 25% scenario, the same problem occurs, but in this case, all 347 table entries of the "S1+S3-S4-S5" subset must be offloaded to S2 and S3. Once again, the First-Fit algorithm demonstrated stable and good results, given its prioritization of the "generic" and "S1+S3-S4-S5" subsets, which contain the table entries that clone most packets to the NIDS for the input dataset.

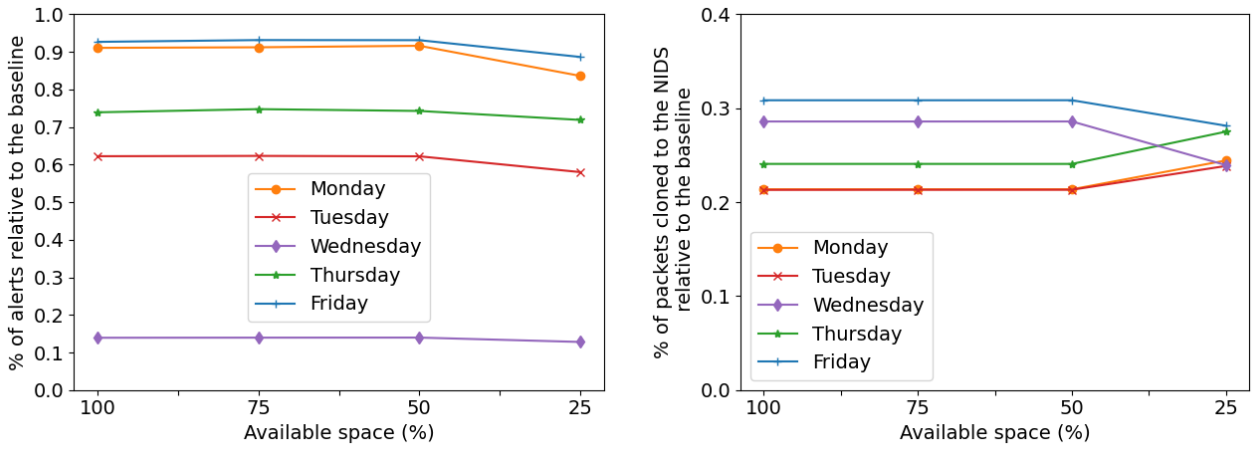


(a) Alerts

(b) Packets cloned

Figure 6.16: Percentage of alerts and packets cloned for the First-Fit algorithm with the tree topology

Figure 6.17 presents the results for the *Best-Fit* algorithm. The number of alerts remained stable in all scenarios, with a slight decrease in the scenario of 25% memory availability. Instability in the volume of packets forwarded to the NIDS occurred only in the 25% scenario. In this scenario, some PCAPs had a slight increase in the number of cloned packets, while others had a minor decrease. The increase was caused by entries in different switches matching the same packet, while the decrease was due to not all table entries being offloaded. The improved performance of the Best-Fit algorithm in the tree topology compared to the linear topology is attributed to the complete offloading of the "generic" and "S1+S3-S4-S5" subsets (the subset with the "192.168.10.0/24" network as the destination) for more end hosts. Regarding duplicate entries, the duplicates in the 50% scenario are for the "generic" subset entries in S2 and S3 that could not be offloaded to S1. Whereas for the 25% scenario, the duplicate 57 entries are from the "generic" subset and they were offloaded to S4 and S5 since the "generic" subset could not be offloaded to S1 and S2, and both S4 and S5 contain end hosts encompassed by the "generic" subset.



(a) Alerts

(b) Packets cloned

Figure 6.17: Percentage of alerts and packets cloned for the Best-Fit algorithm with the tree topology

6.6 Discussion

This chapter detailed the evaluation of the network-wide table entries offloading algorithms presented in Section 5.3. To start, an assessment of the parameters of the data plane was conducted to determine the optimal configuration. The evaluation showed that the *W* parameter had a more significant impact compared to the other parameters, while the *N* variable did not prove as influential as anticipated. More importantly, it became apparent that the number of alerts compared to the baseline data stagnated for all input PCAPs, never achieving an exact match. The Wednesday PCAP, in particular, had an

extremely low correlation with the baseline alerts. For future work, it is crucial to replicate the evaluation with a higher packet rate, as the main attacks of CICIDS2017 are DoS and certain Snort rules only generate alerts with a specific packet rate. The chosen packet rate of 1000pps, defined due to BMv2 limitations, hardly configures a DoS behavior. Furthermore, it is imperative to revisit the P4 data plane to identify potential shortcomings that contribute to this limitation. Addressing these aspects will contribute to a more comprehensive understanding and enhancement of the proposed solution.

In the experimental analysis of the algorithms, it becomes evident that the choice of table entries to offload and the characteristics of the input traffic significantly influence the results. As described in Section 6.5.1, a small number of table entries were responsible for forwarding most of the traffic to the NIDS engine, and a small set of Snort rules generated the majority of alerts. Regarding the final results, the network-wide algorithms demonstrated superior performance compared to the Simple algorithm, as they consistently offloaded a larger number of table entries to the PDP. In particular, the First-Fit algorithm exhibited better overall performance, despite offloading fewer table entries compared to the Best-Fit algorithm. This can be attributed to the First-Fit algorithm’s prioritization of the "generic" subset and the subset of table entries with the "192.168.10.0/24" network as the destination. These subsets contained the table entries that matched a substantial portion of the packets, leading to more packets being forwarded to the NIDS engine and, consequently, generating more alerts.

7. CONCLUSION

This work explored the possibility of leveraging the network-wide orchestration of PDP devices to pre-filter the network traffic for the NIDS host with the goal of enhancing its performance. Although NIDSs play a crucial role in protecting networks against attacks, the ever-increasing volume of network traffic poses a substantial challenge to them, particularly in high-throughput networks. The bottleneck typically occurs in the pattern matching stage, where incoming packets are meticulously compared against thousands of signature rules indicative of malicious or suspicious traffic patterns. State-of-the-art approaches propose leveraging the PDP paradigm to offload these signature rules directly to the forwarding devices, effectively prefiltering the network traffic for the NIDS and forwarding only suspicious packets to it. However, existing proposals present some limitations. Most of them do not address the memory limitations of PDP devices correctly. More importantly, they overlook the potential orchestration among multiple networking devices, opting instead to offload all rules to a single device or replicating the rules in all devices. These limitations shown by the proposals can impact their effectiveness, as these solutions rely on the assumption that all signature rules are successfully offloaded to the PDP. Failure to offload these signature rules could compromise the solution's efficacy, as suspicious packets might not reach the NIDS engine, allowing attacks to go unnoticed.

To overcome these shortcomings while still addressing the NIDS saturation problem in a similar manner to state-of-the-art works, we leveraged the network-wide orchestration of programmable devices to prefilter network traffic for the NIDS. We started by improving the P4 NIDS rules compiler of P4-ONIDS [37] by broadening the types of rules it accepts. We also improved its compilation time and reduced its memory usage, as detailed in Chapter 4. Then, in Chapter 5, we presented the P4 data plane responsible for prefiltering network packets and forwarding suspicious packets to the NIDS engine based on the compiler's table entries. Lastly, we designed a network-wide table entries orchestrator to strategically distribute the compiled entries to multiple PDP devices while considering the device's memory and network topology. This orchestrator can offload these entries through three approaches: Simple, similar to those used in state-of-the-art works, where all entries are offloaded to the device(s) of the network entry point (named *source switch* in this work); First-Fit, where the table entries are initially offloaded to the *source switch*, and if not possible, then to downstream devices in the network until all entries are offloaded or the available space is exhausted; and Best-Fit, which attempts to offload a table entry to its *best* position, and if not possible to other devices in the path(s) from the *source switch* to the *best* device.

For our experiments, detailed in Chapter 6, we started by selecting the optimal parameters for the P4 data plane algorithms through experimentation. Once defined, we proceeded to evaluate the network-wide orchestrator algorithms. The evaluation en-

compassed two classical network topologies, linear and tree, across four distinct memory availability scenarios. The results underscored the superior adaptability of algorithms that consider multiple devices. The two network-wide algorithms successfully offloaded all table entries in three of the four scenarios and at least half of the entries in the 25% memory availability scenario. In particular, the First-Fit approach exhibited slightly better results than the Best-Fit approach, given its prioritization of table entries destined for all end hosts of the network and those relevant to the CICIDS2017 victims network. In conclusion, the proposed network-wide table entries offloading algorithms demonstrated their advantage over the traditional model by effectively offloading more table entries in resource-constrained scenarios, thus ensuring the effectiveness of the proposed prefiltering mechanism in a broader range of situations.

7.1 Limitations and Future Work

Research developed in this work highlighted the advantages of algorithms that employ network-wide orchestration of PDP devices compared to traditional models that focus on a single device. However, our work exhibits some shortcomings, especially the poor correlation between the number of alerts generated in the experiments compared to the baseline data. Therefore, in this Section, we detail the main limitations of our research, discuss the probable causes, and propose future work to address them.

The experiments were conducted within a Mininet emulated environment using tools with limited performance. While this environment served as a valuable initial testing ground, its inherent constrains limit the accurate representation of real-world networks, thereby restricting our solution’s applicability. As a consequence, future work should evaluate this research in a real PDP testbed equipped with multiple P4 switches and a real NIDS engine host. Besides the restricted testing environment, the substantial difference in the number of alerts generated in all experiments compared to the baseline data is another significant limitation of this work. The potential causes for this shortcoming are described below.

1. The primary reason for this disparity is that certain rules in the NIDS engine (Snort 3) are designed to detect DoS and DDoS attacks, and they only trigger when the packet rate surpasses specific thresholds. However, in our experiments, the packet rate is constrained to 1000 pps (due to BMv2 limitations), hardly characterizing a DoS or DDoS attack. This limitation becomes evident when examining the low number of alerts for the Wednesday PCAP, which predominantly features DoS attacks. To address this issue, it is essential to replicate the experiments in a real P4 testbed with the real packet rate of the input dataset.

2. Another contributing factor to the observed discrepancy is that not all packets that generated alerts in the baseline data are cloned into the NIDS in our experiments. This could be caused by offloaded table entries not properly cloning these packets or potential limitations in the proposed P4 data plane. However, concerning the later assumption, the experiments showed that increasing the number of packets to clone per flow did not improve the number of alerts, indicating that the cause of this limitation probably lies in the first assumption. Future work must review the set of table entries offloaded and the P4 data plane to address this restriction.

In addition to addressing the detailed limitations outlined above, future research must evaluate the network-wide orchestrator with more complex network topologies to enhance the robustness of the proposed algorithms. These new topologies should include networks with multiple sources of unknown or malicious network traffic and networks that contain a mix of traditional networking devices and PDP devices. Lastly, the ability to offload table entries and remove unused table entries from the data plane during runtime could further address the memory limitations of the PDP and improve the NIDS performance.

REFERENCES

[1] Al Sadi, A.; Berardi, D.; Callegati, F.; Melis, A.; Prandini, M. “P4dm: Measure the link delay with p4”, *Sensors*, vol. 22–12, 2022, pp. 4411.

[2] AlSabeh, A.; Kfoury, E.; Crichigno, J.; Bou-Harb, E. “P4ddpi: Securing p4-programmable data plane networks via dns deep packet inspection”. In: Proceedings of the 2022 Network and Distributed System Security (NDSS) Symposium, 2022, pp. 1–7.

[3] AlSabeh, A.; Khoury, J.; Kfoury, E.; Crichigno, J.; Bou-Harb, E. “A survey on security applications of p4 programmable switches and a stride-based vulnerability assessment”, *Computer Networks*, vol. 207, 2022, pp. 108800.

[4] Alsadi, A.; Berardi, D.; Callegati, F.; Melis, A.; Prandini, M. “A security monitoring architecture based on data plane programmability”. In: 2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit), 2021, pp. 389–394.

[5] Bonomi, F.; Mitzenmacher, M.; Panigrah, R.; Singh, S.; Varghese, G. “Beyond bloom filters: From approximate membership checks to approximate state machines”, *ACM SIGCOMM computer communication review*, vol. 36–4, 2006, pp. 315–326.

[6] Bosshart, P.; Daly, D.; Gibb, G.; Izzard, M.; McKeown, N.; Rexford, J.; Schlesinger, C.; Talayco, D.; Vahdat, A.; Varghese, G.; et al.. “P4: Programming protocol-independent packet processors”, *ACM SIGCOMM Computer Communication Review*, vol. 44–3, 2014, pp. 87–95.

[7] Bosshart, P.; Gibb, G.; Kim, H.-S.; Varghese, G.; McKeown, N.; Izzard, M.; Mujica, F.; Horowitz, M. “Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn”, *ACM SIGCOMM Computer Communication Review*, vol. 43–4, 2013, pp. 99–110.

[8] Cormode, G. “Count-min sketch.” Source: https://users.cs.utah.edu/~pandey/courses/cs6968/spring23/papers/cms.pdf, 2009.

[9] Cormode, G.; Muthukrishnan, S. “An improved data stream summary: the count-min sketch and its applications”, *Journal of Algorithms*, vol. 55–1, 2005, pp. 58–75.

[10] da Silveira Ilha, A.; Lapolli, Â. C.; Marques, J. A.; Gaspary, L. P. “Euclid: A fully in-network, p4-based approach for real-time ddos attack detection and mitigation”, *IEEE Transactions on Network and Service Management*, vol. 18–3, 2020, pp. 3121–3139.

[11] Dao, T.-N.; Nguyen, H.-N.; et al.. “A network intrusion detection architecture based on class parallelism on distributed switches”. In: 2022 13th International Conference on Information and Communication Technology Convergence (ICTC), 2022, pp. 1284–1289.

[12] Ding, D.; Savi, M.; Antichi, G.; Siracusa, D. “An incrementally-deployable p4-enabled architecture for network-wide heavy-hitter detection”, *IEEE Transactions on Network and Service Management*, vol. 17–1, 2020, pp. 75–88.

[13] Gao, Y.; Wang, Z. “A review of p4 programmable data planes for network security”, *Mobile Information Systems*, vol. 2021, 2021, pp. 1–24.

[14] González, L. A. Q.; Castanheira, L.; Marques, J. A.; Schaeffer-Filho, A.; Gaspary, L. P. “Bungee: An adaptive pushback mechanism for ddos detection and mitigation in p4 data planes”. In: 2021 IFIP/IEEE International Symposium on Integrated Network Management (IM), 2021, pp. 393–401.

[15] Hauser, F.; Häberle, M.; Merling, D.; Lindner, S.; Gurevich, V.; Zeiger, F.; Frank, R.; Menth, M. “A survey on data plane programming with p4: Fundamentals, advances, and applied research”, *Journal of Network and Computer Applications*, vol. 212, 2023, pp. 103561.

[16] Hodo, E.; Bellekens, X.; Hamilton, A.; Tachtatzis, C.; Atkinson, R. “Shallow and deep networks intrusion detection system: A taxonomy and survey”, *arXiv preprint arXiv:1701.02145*, 2017.

[17] Hu, Q.; Asghar, M. R.; Brownlee, N. “Evaluating network intrusion detection systems for high-speed networks”. In: 2017 27th International Telecommunication Networks and Applications Conference (ITNAC), 2017, pp. 1–6.

[18] Hu, Q.; Yu, S.-Y.; Asghar, M. R. “Analysing performance issues of open-source intrusion detection systems in high-speed networks”, *Journal of Information Security and Applications*, vol. 51, 2020, pp. 102426.

[19] Huston, G. “Bgp in 2019 – the bgp table”. Source: https://blog.apnic.net/2020/01/14/bgp-in-2019-the-bgp-table/, Jan 2020.

[20] Ilha, A. D. S. “Accelerating real-time intrusion detection by offloading capture filters to p4 programmable switches”, *Universidade Federal do Rio Grande do Sul*, 2019.

[21] Kaur, S.; Kumar, K.; Aggarwal, N. “A review on p4-programmable data planes: Architecture, research efforts, and future directions”, *Computer Communications*, vol. 170, 2021, pp. 109–129.

[22] Kfoury, E. F.; Crichigno, J.; Bou-Harb, E. “An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends”, *IEEE Access*, vol. 9, 2021, pp. 87094–87155.

[23] Khraisat, A.; Gondal, I.; Vamplew, P.; Kamruzzaman, J. “Survey of intrusion detection systems: techniques, datasets and challenges”, *Cybersecurity*, vol. 2–1, 2019, pp. 1–22.

[24] Kumar, S. “Survey of current network intrusion detection techniques”, *Washington Univ. in St. Louis*, 2007, pp. 1–18.

[25] Lewis, B.; Broadbent, M.; Race, N. “P4id: P4 enhanced intrusion detection”. In: 2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), 2019, pp. 1–4.

[26] Lewis, B.; Broadbent, M.; Rotsos, C.; Race, N. “4midable: Flexible network offloading for security vnfs”, *Journal of Network and Systems Management*, vol. 31–3, 2023, pp. 52.

[27] Li, G.; Zhang, M.; Wang, S.; Liu, C.; Xu, M.; Chen, A.; Hu, H.; Gu, G.; Li, Q.; Wu, J. “Enabling performant, flexible and cost-efficient ddos defense with programmable switches”, *IEEE/ACM Transactions on Networking*, vol. 29–4, 2021, pp. 1509–1526.

[28] Lin, P.-C.; Lee, J.-H. “Re-examining the performance bottleneck in a nids with detailed profiling”, *Journal of Network and Computer Applications*, vol. 36–2, 2013, pp. 768–780.

[29] Liu, Z.; Namkung, H.; Nikolaidis, G.; Lee, J.; Kim, C.; Jin, X.; Braverman, V.; Yu, M.; Sekar, V. “Jaqen: A {High-Performance}{Switch-Native} approach for detecting and mitigating volumetric {DDoS} attacks with programmable switches”. In: 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 3829–3846.

[30] McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S.; Turner, J. “Openflow: enabling innovation in campus networks”, *ACM SIGCOMM computer communication review*, vol. 38–2, 2008, pp. 69–74.

[31] Ndonda, G. K.; Sadre, R. “A two-level intrusion detection system for industrial control system networks using p4”. In: 5th International Symposium for ICS & SCADA Cyber Security Research 2018 5, 2018, pp. 31–40.

[32] Neu, C. V. “Detecting encrypted attacks in software-defined networking”, Ph.D. Thesis, Pontifícia Universidade Católica do Rio Grande do Sul, 2019.

[33] Qin, Q.; Poularakis, K.; Leung, K. K.; Tassiulas, L. “Line-speed and scalable intrusion detection at the network edge via federated learning”. In: 2020 IFIP Networking Conference (Networking), 2020, pp. 352–360.

[34] Research, J. “Cybersecurity breaches to result in over 146 billion records being stolen by 2023”. Source: https://www.juniperresearch.com/press/cybersecurity-breaches-to-result-in-over-146-bn, Aug 2018.

[35] Roesch, M.; et al.. “Snort: Lightweight intrusion detection for networks.” In: Lisa, 1999, pp. 229–238.

[36] Sharafaldin, I.; Lashkari, A. H.; Ghorbani, A. A. “Toward generating a new intrusion detection dataset and intrusion traffic characterization.” In: ICISSp, 2018, pp. 108–116.

[37] Tavares, K.; Ferreto, T. C. “P4-onids: A p4-based nids optimized for constrained programmable data planes in sdn”. In: Anais do XXXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, 2021, pp. 434–447.

[38] Tavares, K. C. P. “P4-onids: P4-based pre-filtering nids solution optimized for constrained programmable data plane in sdn”, Master’s Thesis, Pontifícia Universidade Católica do Rio Grande do Sul, 2021.

[39] Varghese, J. E.; Muniyal, B. “An efficient ids framework for ddos attacks in sdn environment”, *IEEE Access*, vol. 9, 2021, pp. 69680–69699.

[40] Ventures, C. “Cybercrime to cost the world $10.5 trillion annually by 2025”. Source: https://cybersecurityventures.com/cybercrime-damage-costs-10-trillion-by-2025/, Nov 2020.

[41] Verizon. “Cybercrime thrives during pandemic: Verizon 2021 data breach investigations report”. Source: https://www.verizon.com/about/news/verizon-2021-data-breach-investigations-report, May 2021.

[42] Waleed, A.; Jamali, A. F.; Masood, A. “Which open-source ids? snort, suricata or zeek”, *Computer Networks*, vol. 213, 2022, pp. 109116.

[43] Wellem, T.; Lai, Y.-K.; Huang, C.-Y.; Chung, W.-Y. “A flexible sketch-based network traffic monitoring infrastructure”, *IEEE Access*, vol. 7, 2019, pp. 92476–92498.

[44] Wong, K.; Dillabaugh, C.; Seddigh, N.; Nandy, B. “Enhancing suricata intrusion detection system for cyber security in scada networks”. In: 2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE), 2017, pp. 1–5.

[45] Xia, W.; Wen, Y.; Foh, C. H.; Niyato, D.; Xie, H. “A survey on software-defined networking”, *IEEE Communications Surveys & Tutorials*, vol. 17–1, 2014, pp. 27–51.

[46] Yang, L.; Dantu, R.; Anderson, T.; Gopal, R. “Forwarding and control element separation (forces) framework”. Source: https://www.rfc-editor.org/rfc/rfc3746.html, April 2004.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br