Data Plane Programming in Networks

Programación del Plano de Datos en Redes

Belén Brandino

Postgraduate program in Informatics
Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay
September of 2024

Data Plane Programming in Networks

Programación del Plano de Datos en Redes

Belén Brandino

Master's Thesis submitted to the Postgraduate Program in Informatics, Facultad de Ingeniería of the Universidad de la República, as part of the necessary requirements for obtaining the title of Master in Informatics.

Director:
Dr. Prof. Eduardo Grampín

Academic director:
Dr. Prof. Eduardo Grampín

Montevideo – Uruguay
September of 2024

Brandino, Belén

Data Plane Programming in Networks / Belén Brandino. - Montevideo: Universidad de la República, Facultad de Ingeniería, 2024.

xv, 139 p.: il.; 29, 7cm.

Director:

Eduardo Grampín

Academic director:

Eduardo Grampín

Tesis de Maestría – Universidad de la República, Program in Informatics, 2024.

Bibliography: p. 129 – 139.

1. Programación del Plano de Datos, 2. Programabilidad de la red, 3. P4. I. Grampín, Eduardo, . II. Universidad de la República, Postgraduate Program in Informatics. III. Title

MEMBERS OF THE THESIS DEFENSE COURT

D.Sc. Prof. Nombre del 1er Examinador Apellido

Ph.D. Prof. Nombre del 2do Examinador Apellido

D.Sc. Prof. Nombre del 3er Examinador Apellido

Ph.D. Prof. Nombre del 4to Examinador Apellido

Ph.D. Prof. Nombre del 5to Examinador Apellido

Montevideo – Uruguay
September of 2024

RESUMEN

Los requerimientos de las redes de computadoras modernas cambian sin parar. Internet y sus aplicaciones han crecido de manera exponencial, obligando a los operadores de red a innovar. Las redes deben poder adaptarse a nuevas tecnologías, de manera escalable y cuidando los costos asociados. La programabilidad de la red le permite a los usuarios cambiar la funcionalidad de los dispositivos de red, diseñando sus propios algoritmos de procesamiento de paquetes sin depender de fabricantes. A su vez, existe la nueva tendencia de computación "in-network computing", donde se utiliza la programabilidad de la red no solo para la conectividad, sino también para computación. De esta manera, se deja de tratar al dispositivo de red como un simple dispositivo de reenvío, aprovechando sus capacidades y haciendo *offload* de tareas, aportando a la descentralización y toma rápida de decisiones.

En este trabajo se presentan los principales conceptos de la programabilidad de la red, junto con las tecnologías, lenguajes y hardware que lo habilitan. Luego, se presenta una solución en software compleja para demostrar su potencial. En particular, se implementó un *Intrusion Detection System* (IDS) que detecta anomalías de tráfico a nivel de flujo en el dispositivo de red. Se incorpora *Machine Learning* (ML) desplegando un modelo sencillo en el switch, para tomar decisiones rápidas sobre el tráfico en caso de tener confianza suficiente, en caso contrario, se delega la decisión a un experto. Este será un *oracle* con un modelo de ML más poderoso y más datos de entrenamiento. Usando las decisiones del oracle, el dispositivo de red puede ser reentrenado, buscando reducir la dependencia del oracle con el tiempo. Por último, para validar la posible implementación de este problema en hardware, se realizó una pequeña prueba de concepto. Para ambos casos se utilizó el lenguaje de programación de plano de datos más popular: P4 (Programming protocol-independent packet processors). Palabras claves:

Programación del Plano de Datos, Programabilidad de la red, P4.

ABSTRACT

Modern computer networks must continually adapt to evolving requirements driven by the exponential growth of the Internet and its applications. Networks must be able to adapt to new technologies in a scalable manner, while maintaining cost efficiency. Network programmability allows users (typically network operators) to modify the functionality of network devices, defining the packet processing to their specific needs, without relying on vendor-provided solutions. Furthermore, there is a new computing trend known as "in-network computing", which leverages network programmability not only for connectivity but also for computation. In this way, the network devices stop being treated as mere forwarding entities, taking advantage of their capabilities, contributing to task offloading, decentralization and faster decision-making.

This work introduces the key concepts of network programmability, along with the technologies, languages and hardware that make it possible. Then, a complex software solution is introduced to demonstrate the potential and complexity of applications enabled by this concept. In particular, an Intrusion Detection System (IDS) was implemented to detect abnormal traffic at flow level directly on the network device. This approach incorporates Machine Learning (ML) by developing a simple ML model on the switch, to make quick decisions (at line-rate) about traffic, when there is sufficient confidence. Otherwise, it defers to an external oracle that uses a more powerful ML model with additional training data. Based on the oracle's decisions, the network device can go through the retraining process, with hopes of reducing reliance on the oracle over time. Finally, to validate the potential hardware implementation of this problem, a small proof-of-concept was carried out on. Both proof-of-concepts were implemented using the most popular data plane programming language: P4 (Programming protocol-independent packet processors).

Keywords:
Data Plane Programming, Network programability, P4.

List of Figures

Acronyms

Acronyms

ALU Arithmetic Logic Unit 11, 77

API Application Programming Interface 6, 7, 8, 16, 21, 22, 56

ASIC Application-Specific Integrated Circuit 7, 9, 17, 21, 61, 62, 72

BPF Berkeley Packet Filter 29

CLI Command Line Interface 6, 117

CPU Central Processing Unit 29, 31, 33, 38, 39, 43, 49, 56, 59, 62, 63, 64, 65, 68, 69, 71, 72, 74, 75, 76, 77, 78

DPDK Data Plane Development Kit 40

DPI Deep Packet Inspection 76

DPP Data Plane Programming 8

DPU Data Processing Unit 17, 61, 63, 71, 72

DT Decision Tree 93, 94, 99, 106, 108

GPU Graphics Processing Unit 60, 71, 72

GUI Graphical User Interface 70, 117

IDL Interface Definition Language 22

IDS Intrusion Detection System 3, 75, 92, 126

IP Internet Protocol 14, 27, 88

IPS Intrusion Prevention System 75, 76

IPU Infraestructure Processing Unit 61, 63

ISP Internet Service Provider 7

IoT Internet of Things 80

LPM Longest Prefix Match 14

ML Machine Learning 61, 82, 83, 93, 105

NAT Network Address Translation 77

NFV Network Function Virtualization 37, 38, 76, 80, 89

NIC Network Interface Card viii, 1, 25, 31, 32, 33, 34, 38, 55, 56, 57, 59, 61, 62, 63, 64, 68, 69, 71, 77

NPU Network Processing Unit 60

NTMA Network Traffic Monitoring and Analysis 91

OF OpenFlow 8

P4 Programming protocol-independent packet processors vii, 3, 4, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 23, 24, 25, 26, 48, 59, 63, 64, 66, 67, 69, 78, 79, 80, 81, 93, 95, 96, 99, 101, 114, 116, 117, 118, 119, 120, 122, 123, 124, 127

PISA Protocol-Independent Switch Architecture vii, 9, 10, 11, 12, 13

PNA Portable NIC Architecture 21

PSA Portable Switch Architecture 21

QoS Quality of Service 63, 69, 73, 77, 87, 89

RF Random Forest 93, 94, 96, 99, 100, 101, 102, 108, 110

RPC Remote Procedure Call 22, 23

SDN Software Defined Networking 7, 8, 43, 46, 76, 82, 89

SRAM Static Random-Access Memory 11

TCAM Ternary Content-Addressable Memory 11

TPU Tensor Processing Unit 60, 72

VM Virtual Machines 54, 69, 76, 77, 78

protobuf Protocol Buffers 21, 22

vxlan Virtual Extensible LAN 77

Contents

Chapter 1

Introduction

Over the past decade, the Internet and its applications have experienced unprecedented growth, driving significant advancements in both network infrastructure and software development. With the rise of 5G mobile standards, widespread cloud computing, ever-present IoT, and extensive machine learning and big data applications, operators must embrace entirely new approaches to network architecture. This shift will make software-defined networking, edge computing, network function virtualization, and service chaining the standard practices. Upcoming applications will demand that NICs and network devices like switches and routers support a continuously evolving and diverse array of protocols and functions, building on the already extensive range of features available today, such as L2/L3/L4 processing, tunneling and VPN protocols, load balancing, congestion control and Quality of Service, firewalls, and intrusion detection systems [9].

Considering today's applications and their requirements, traditional networks face several challenges. Firstly, it is necessary to have networks that can adapt to the new technologies that emerge every day. Additionally, networks and their protocols need to be changeable in a scalable manner, while also keeping deployment costs and times to a minimum. Typically, when a network device needs to implement a new function, it involves manufacturers, which naturally leads to higher costs and longer timelines. All of this can be addressed using network programmability, allowing users to define the algorithms to be executed directly on the network devices themselves.

Computation history has evolved from traditional parallel and grid computing to cloud computing, which offers service models like IaaS, PaaS, and

SaaS, providing scalability, on-demand resource provisioning, and a pay-as-you-go pricing model. However, with the advent of 5G and beyond, new applications such as mobile video conferencing, connected vehicles, e-healthcare, online gaming, and virtual reality require high data rates and ultra-low latency. Cloud computing struggles to meet these evolving demands due to several challenges. The primary issue is the significant distance between cloud resources and end devices. This connection relies on the Internet, and thus has latency issues. Additionally, the processing power of cloud servers is insufficient for the high data rates and massive traffic volumes generated by emerging applications and IoT devices.

Edge computing was introduced to address cloud computing's limitations by bringing resources closer to end devices, thereby improving latency and processing capacity. However, it may not sustain the growing traffic demands long-term, and the latency still falls short for ultra-low-latency applications. Distributed cloud computing improves upon these paradigms by utilizing the computational and storage capacities of nearby intelligent devices, but it faces challenges like computation and power limitations, device mobility, and security concerns. A more secure, power-efficient, and stable computation infrastructure, such as in-network computing based on programmable data plane technology, could significantly enhance processing capacity at the network edge, complementing existing computational models.

Network elements such as switches and routers facilitate connectivity between end devices and edge infrastructure, as well as between edge and cloud infrastructure. A new computing trend, known as in-network computing, utilizes programmable network elements not just for connectivity but also for computation. Modern programmable switches can handle billions of packets per second with line-rate processing speeds, while maintaining sub-microsecond packet processing delays. By using in-network computing, packets are processed in real-time along their path before reaching edge or cloud servers. This approach provides faster processing closer to end devices compared to traditional edge or cloud computing models.

The in-network computing paradigm promotes the use of network devices, in particular programmable switches, FPGAs, and SmartNICs for computational purposes. Programmable switches can be configured to analyze and modify packet fields as they arrive. FPGAs are semiconductor devices that allow for the programming of logical blocks to perform specific packet process-

ing tasks. Likewise, SmartNICs provide dedicated hardware acceleration and customizable packet processing functions [54].

The problem to be addressed in this work is the need to leverage programmable network devices that adapt to the requirements of today's applications. By viewing network devices as more than just "dumb" devices that merely forward packets, and by empowering them, it is possible to offload multiple network functions, contributing to decentralization and performance improvements, enabling smarter traffic management, enhancing security features, and reducing the load on centralized servers, ultimately leading to a more efficient and resilient network infrastructure. The goal is to showcase the capabilities and benefits of network programmability for both industry and academia, highlighting how it can drive innovation, improve operational efficiency, enable custom solutions, and support advanced research, ultimately transforming the way networks are designed, managed, and optimized.

This thesis, therefore, has two main dimensions: first, an in-depth investigation into network programmability, including the technologies and devices that enable it, and its applications. Additionally, a brief survey of machine learning in networks is conducted. Secondly, a software proof of concept that demonstrates the power of network programmability and how it can contribute to in-network computing, whose results have been published originally in a poster [14] and in an article [13], where more thorough tests were conducted. Additionally, hardware basic implementations were conducted to showcase the real-world applicability of network programmability. In particular, both proof-of-concepts are carried out using the most popular language for data plane programming: P4. The code is available at [17] and all the results are replicable.

In the case of the software proof-of-concept, a complex scenario is demonstrated where network programmability helps create applications tailored to today's needs. Specifically, an Intrusion Detection System (IDS) is designed to detect traffic anomalies at the flow level. In this case, the network device can make decisions about the traffic with a certain degree of confidence, granting it more power than simple packet forwarding and thereby alleviating the load on the control plane. The goal of this solution is to make quick decisions (at line-rate) about traffic when there is sufficient confidence in the model's output. If the device cannot make an appropriate decision, it can delegate this task to an expert (oracle). Furthermore, both the network device and the oracle

use Machine Learning models, allowing them to handle large volumes of data and enabling real-time retraining of the network device to adapt to changing traffic patterns. The oracle, employs a more powerful ML model with access to more training data than the switch that employs a simpler model. Based on the oracle's decisions, the network device can go through the retraining process using the new labeled data, with hopes of reducing dependence on the oracle over time. This is a clear example of how network programmability can contribute to improve current networks by enabling customized applications, reducing loads, integrating Machine Learning models, and allowing for dynamic adjustments during runtime. Finally, to validate the potential hardware implementation of this problem, a small proof-of-concept was carried out on a SmartNIC, where a simplified version of the ML model was deployed on the switch, and its correct functionality was verified.

This document is organized as follows: Chapter 2 presents the state of the art, detailing the concept of network programmability, with a particular focus on the most popular language for this purpose, P4. It includes a comparison of this language with other available options, a survey of different hardware devices for programmability, an overview of various packet processing technologies, a review of various applications of programmable network devices and a review of machine learning in networking. Chapter 3 presents the software proof of concept, along with a variety of tests to measure its effectiveness. Additionally, a small hardware proof of concept was conducted. This proof of concept is supported by a survey of the available hardware devices that led to the selection of the device used. Finally, Chapter 4 presents the conclusions of this work.

Chapter 2

Theoretical foundations

To achieve network programmability, it is essential to understand the enabling technologies. This chapter explores the concept of network programmability in depth, along with the various models that implement it. It also examines the programming languages associated with these models. Alternatives to data plane programming languages, such as programmable packet processing technologies and hardware solutions like SmartNICs, are discussed. Additionally, potential applications for data plane programming and offloading to SmartNICs are introduced, concluding with a review of machine learning in networking, focusing on traffic classification.

2.1 Data Plane Programming

Conventional network devices such as routers or switches integrate both the control plane and the data plane within the same device. The control plane is responsible for establishing packet processing policies, such as determining where to forward a packet or how to change its header, and managing the device's operations. Conversely, the data plane is solely accountable for executing the packet processing policy established by the control plane, namely, it handles forwarding, hence often referred to as the forwarding plane. However, forwarding is not the sole function that the data plane can implement [9].

In the traditional approach, routing algorithms determine the paths that packets follow from their source to their destination, thereby defining the content of the forwarding tables within network devices. As illustrated in the example depicted in Figure 2.1, each network device executes a routing algo-

rithm, thereby possessing both routing and forwarding functionalities.

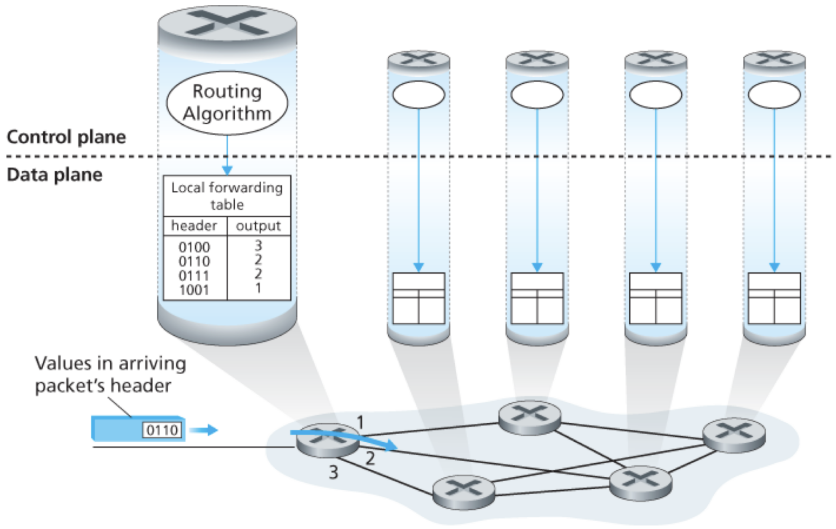


Figure 2.1: Traditional approach: control plane and data plane [57].

Therefore, network devices process packets using algorithms from both the control plane and data plane. Users have some freedom to configure certain aspects of the control plane, such as functionalities or protocols, through Command Line Interface (CLI)s, web interfaces, or management Application Programming Interface (API)s. However, the underlying algorithms can only be modified by the manufacturers. This results in a prolonged development process for adding new features, causing some functionalities to be implemented only when there is high demand, hindering innovation. Moreover, by not allowing the modification of the control plane, all network protocols must be factory-implemented, leading to inefficiencies, e.g. implemented protocols that are not used.

In this context, programmability is defined as the ability of software or hardware to execute an external processing algorithm. This differs from flexible or configurable entities, which only allow changing certain parameters of the internally defined algorithm that remains unchanged. Thus, network programmability refers to the ability to define the processing algorithm executed in a network, particularly in the individual nodes such as switches and routers, among others.

Network programmability entails the ability to specify and change algorithms for both the control plane and data plane. In practice, this means end-users can define these algorithms themselves without involving manufac-

turers. For network infrastructure providers, network programmability means defining data plane algorithms without needing to involve the original designers of the Application-Specific Integrated Circuit (ASIC) for packet processing.

Therefore, network programmability enables manufacturers and end-users to create networks that meet their specific needs more quickly and often at lower costs, without compromising performance or equipment quality. Achieving full programmability of network devices requires both control plane and data plane programmability [56][45].

Currently, telecommunications networks support numerous and heterogeneous use cases in order to support modern technological infrastructures. This widespread use and diversity complicate the design of communication systems, particularly their fundamental components: the network devices. On the one hand, there is a trend towards specialization to optimize network devices for specific tasks; on the other hand, there is a requirement for network devices to be basic and general-purpose to reduce engineering costs. These contrasting demands have driven the necessity (and definition) of programmable network devices, enabling users (typically telecommunication operators) to alter device functionality through a programming interface.

Software Defined Networking (SDN)s, the first and most popular attempt at making the control plane programmable, aim to make the network devices programmable by introducing an API that allows users to bypass built-in control plane algorithms and replace them with their own. This is achieved through the physical (and logical) separation of the control plane and data plane. Typically, these algorithms are developed in software and run on an SDN controller, which has a global view of the network. This remote controller computes and distributes forwarding tables that each network device should use. It is often deployed in a highly reliable and redundant remote datacenter, managed by the Internet Service Provider (ISP) or another provider. In this setup, the data plane functionality remains similar to the traditional approach, with devices primarily handling forwarding tasks, while the remote controller manages routing and other control functions, as depicted in Figure 2.2.

To enable communication between devices and the controller, it was decided to provide a remotely callable API, giving rise to SDNs. Consequently, it became feasible to implement control plane algorithms across the entire network on a centralized controller. In various cases, such as large-scale datacenters, these centralized algorithms were shown to be simpler and more efficient than

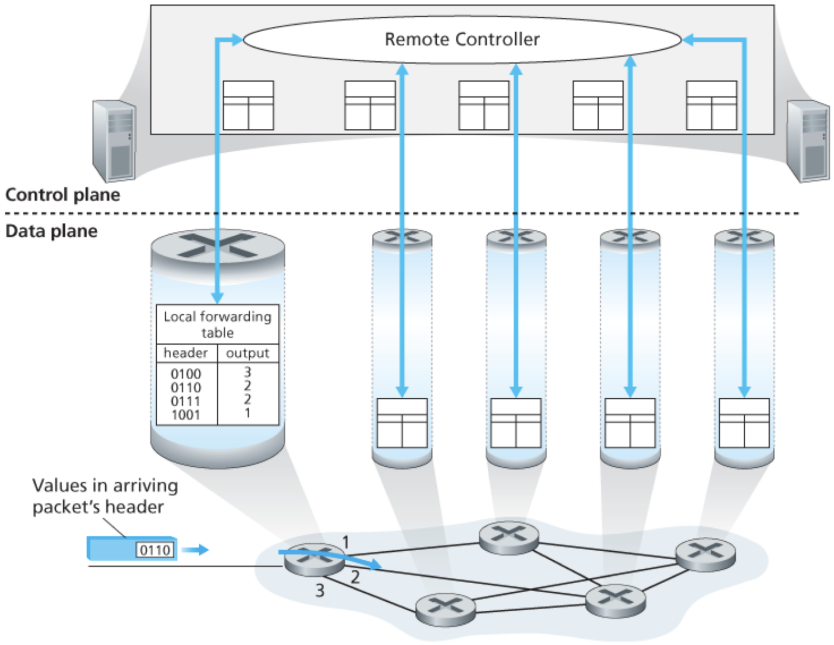


Figure 2.2: SDN: Control plane and data plane [57].

traditional algorithms (e.g., BGP [81]) designed for decentralized control across many autonomous networks. Standardizing this approach led to the development of OpenFlow (OF) [64]. The hope was that once OF standardized the API for controlling data plane functionality, SDN applications could leverage these capabilities to implement network control. However, OF assumed certain specific data plane functionality that was not formally specified and could not be altered. This partly prompted the emergence of data plane programming. Figure 2.3 illustrates the distinction between a traditional network device and one with SDN capabilities.

So, SDNs allow users only to provide their own control plane, which provides the necessary forwarding information to the data plane, while the data plane of different devices remains under manufacturers' control. This limitation is addressed through Data Plane Programming (DPP), enabling end users to define data plane functionality and algorithms themselves. This significantly shifts power to users, allowing them to build custom network equipment without compromising performance, scalability, speed, or power efficiency. For personalized networks, new control planes and SDN applications can be designed, and end users can craft data plane algorithms that fit their exact needs. DPP does not necessarily require APIs or support for external control planes, as OF does. It should be noted that data plane algorithms are responsible

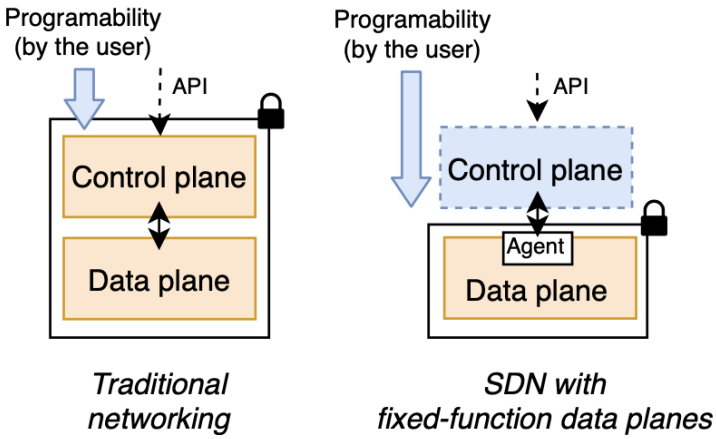


Figure 2.3: Scope of network programmability in traditional network devices versus SDN devices. Traditional network devices allow users to configure certain aspects of the control plane, but the underlying algorithms remain fixed. SDN devices offer a fully programmable control plane, while the data plane remains fixed in functionality [45].

for processing all packets passing through a telecommunications system, thus defining its functionality, performance, and scalability. Attempting to implement data plane functionalities in higher layers, such as the control plane, typically results in performance degradation [45].

Data plane algorithms are typically expressed using standard programming languages, although they often do not map well onto specialized hardware such as high-speed ASICs. This discrepancy has led to the proposal of various data plane programming models as abstractions for hardware. The programming languages are tailored to these data plane models and provide abstract ways to express algorithms. Subsequently, the resulting code is compiled for execution on a specific device that supports the programmed data plane model.

Examples of data plane programming models include Data Flow Graph Abstractions and PISA. In the case of Data Flow Graph Abstractions, packet processing is described using a directed graph. Nodes in the graph represent simple and reusable primitives that can be applied to packets, such as modifying a packet header. Directed edges in the graph depict the internal flow of packets, where decisions about how to handle each packet are made at the nodes. These decisions guide the packet's traversal through the processing graph. Figure 2.4 illustrates an example graph for forwarding IPv4 and IPv6 packets. Programming languages that implement this model include Click [55], Vector Packet Processors [37], and BESS (Berkeley Extensible Soft-

ware Switch) [89].

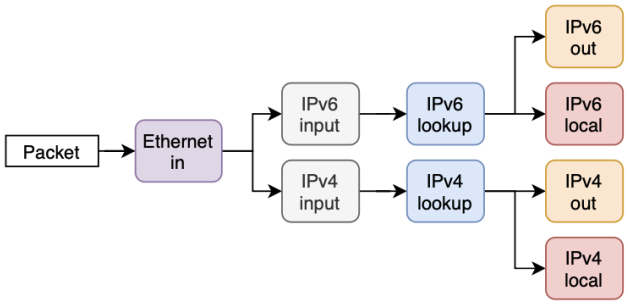


Figure 2.4: An example of a Data Flow Graph Abstractions model graph for IPv4 and IPv6 packet forwarding [45].

In the case of PISA, it is based on the concept of a programmable match-action pipeline, which is well-suited to modern switching hardware. It consists of a programmable parser, a programmable deparser, and between these two, the programmable match-action pipeline itself, which includes multiple stages. Figure 2.5 depicts the PISA model.

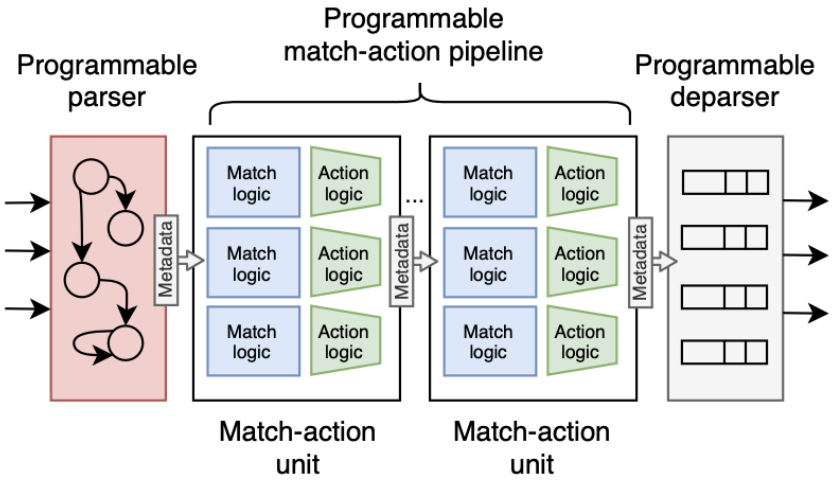


Figure 2.5: Protocol-Independent Switch Architecture [45].

The stages of PISA are as follows:

- Programmable parser: This allows programmers to declare arbitrary headers along with a finite state machine that defines the order of these headers in the packet to be processed. It converts serialized packets into a well-structured form.
- Programmable match-action pipeline: This consists of multiple match-action units, each containing one or more match-action tables. These

tables match an entry with one or more fields of a packet. Each table entry specifies a particular action along with the data to provide, meaning the action executed depends on the packet match. Most of a packet processing algorithm is defined in the form of these tables. Each table includes matching logic coupled with memory (Static Random-Access Memory (SRAM) or Ternary Content-Addressable Memory (TCAM)) to store lookup keys and the corresponding action data. Action logic, such as arithmetic operations or header modifications, is implemented by Arithmetic Logic Unit (ALU)s. Other logic can be implemented using stateful objects, such as counters, meters, or registers stored in SRAM. A control plane manages the matching logic by writing entries to the tables to influence the device's behavior at runtime.

- Programmable deparser: Programmers declare how packets are serialized.

PISA provides an abstract model that is applied in various ways to create concrete architectures, allowing the specification of pipelines with different combinations of programmable blocks. For example, a pipeline without a parser or deparser, one with two parsers and two deparsers, and multiple match-action units in between. It also supports the use of specialized components required for advanced processing, such as checksum calculations or hash functions.

In addition to PISA's programmable components, typical switch architectures also include configurable components with fixed functionality. For example, input/output port blocks that receive or send packets, packet replication engines that implement multicast or packet cloning/duplication, and traffic managers responsible for packet buffering, queuing, and scheduling.

A packet processed by a PISA pipeline consists of the packet's payload and its metadata. PISA processes only the packet metadata (which travels from the parser to the deparser), while the payload travels separately. The metadata can be divided into:

- Packet headers: These are metadata corresponding to network protocol headers, typically extracted in the parser and emitted in the deparser.
- Intrinsic metadata: These are metadata related to fixed-function components. Programmable components can receive information from fixed-function components by reading the intrinsic metadata they produce or

control their behavior by configuring the intrinsic metadata they consume. For example, the ingress port block generates metadata representing the ingress port number, which can be used in match-action units. Conversely, to send a packet, match-action units generate intrinsic metadata representing the egress port number, which is consumed by the traffic manager and/or the egress port block.

- User-defined metadata: Often simply referred to as metadata, this is temporary storage similar to local variables in other programming languages. It allows developers to add information to packets that can be used throughout the processing pipeline.

All metadata is discarded when the corresponding packet exits the processing pipeline, for instance, when the packet is dropped or leaves the switch. Figure 2.6 depicts a typical switch architecture based on PISA. It includes a programmable ingress and egress pipeline and three fixed-function components: an ingress block, an egress block, and a replication engine along with a traffic manager between the ingress and egress pipelines.

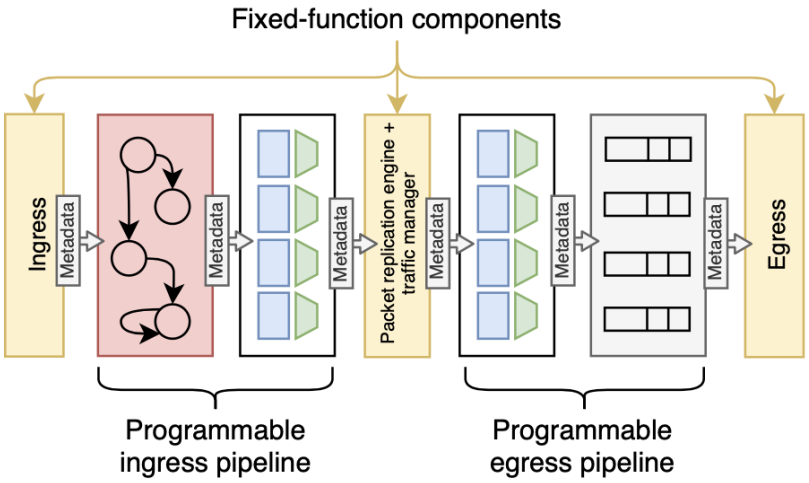


Figure 2.6: Typical switch architecture based on PISA [45].

Including the aforementioned models, there are several approaches for data plane programming, each with its own implementations and programming languages. P4 is currently the most widely adopted abstraction, programming language, and programming concept for the data plane. It is part of the languages used to describe algorithms for PISA. First published as a research paper in 2014 [11], P4 is now developed and standardized by the P4 Lan-

guage Consortium[1]. It is supported by various software and hardware devices and is widely used in academia and industry. Its specification is open and public [45][62][9].

Other data plane programming languages for PISA include FAST [68], OpenState [8], Domino [87], FlowBlaze [79], Protocol-Oblivious Forwarding [88], and NetKAT [3].

2.2 P4

P4 is a language for expressing how packets are processed by the data plane of a programmable forwarding device, whether it be a software or hardware switch, a network interface card, a router, or a network device. These devices are known as targets, meaning a packet processing system capable of executing a P4 program.

P4 allows the user to specify the format (headers) of the packets to be recognized by network devices and the actions to be performed on incoming packets (e.g., forwarding the packet, modifying headers, etc.). P4 also enables the definition of stateful forwarding behaviors based on the use of memory registers that can be accessed when processing a packet.

Many targets implement both the control plane and the data plane, but the P4 program is designed to specify only the data plane functionality of the target. P4 programs also partially define the interface through which the data plane and control plane communicate, but they do not define the control plane functionality. In Figure 2.7, the difference can be seen between a traditional switch with fixed functionality (above) and a programmable switch with P4 (below), where in the traditional switch, it is the manufacturers who decide the data plane functionality. The control plane controls the data plane by managing entries in tables (e.g., forwarding tables), configuring specialized objects, and processing control packets or asynchronous events, such as link state changes. In contrast, in a programmable switch with P4, the data plane functionality is not fixed beforehand; it is defined by the P4 program and thus does not have knowledge of existing network protocols. Additionally, the data plane communicates with the control plane using the same channels as in the traditional switch, except that the set of tables and other data plane objects

[1] https://p4.org

is no longer fixed, as they are defined by the P4 program.

Thus, the P4 program can define the format of the tables, which includes:

- The field (or set of fields) that will be the key of the table, for example, the destination address of the Internet Protocol (IP) header, the Ethernet header protocol, among others.
- The matching algorithm for table lookup, for example, Longest Prefix Match (LPM).
- The possible actions to take on the packets, also defining the logic of these actions and which parameters they will use.

The control plane is responsible for populating this table according to the design specified by the P4 program. For example, if the key of the table defined by the P4 program is the destination IP address, the control plane will select for each IP which action will be taken (from those defined by the P4 program) and provide the parameter values for that action. If the P4 program defined, for example, that there is a parameter representing the port through which the packet should exit, the control plane provides the value for that parameter.

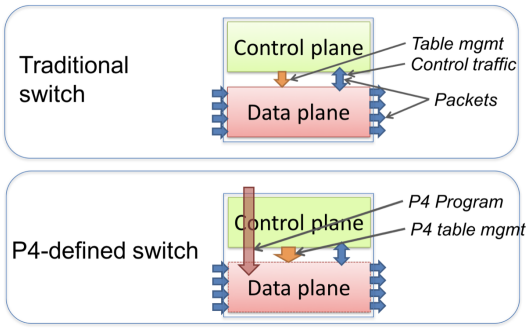


Figure 2.7: Traditional switch vs programmable switch with P4. The traditional switch has fixed data plane functionality, while the control plane manages forwarding tables and handles control packets or events. The programmable switch defines its data plane functionality through a P4 program. The communication between the control and data planes remains similar, but the set of tables and objects is flexible, as they are specified by the P4 program [23].

A set of programmable components with P4 and the data plane interfaces between them will be called an architecture. The P4 architecture identifies the P4 programmable blocks that can exist within a P4 program (defined for that architecture). It can be considered a contract between the program and

the target, which is why the manufacturer of the target must provide both a compiler and a definition of the architecture for the target. P4 programs are written for architectures and not for particular devices, as several devices can use the same architecture.

The central abstractions provided by the P4 language are:

- Header Types: These describe the format (the set of fields and their sizes) of each header within a packet. For example, the following code defines the Ethernet header named `ethernet_t`, with the fields `dstAddr` of 48 bits, `srcAddr` of 48 bits, and `etherType` of 16 bits, in that order.

```
header ethernet_t {
    bit<48> dstAddr
    bit<48> srcAddr
    bit<16> etherType
}
```

- Parsers: Describe the sequence of allowed headers in received packets, how to identify those sequences, and which headers and fields to extract from the packets. This way, it is possible to access the fields individually.
- Tables: Associate user-defined keys with actions and parameters. These tables can be used to implement forwarding tables, flow tables, etc.
- Actions: Code fragments that describe how to manipulate the fields of packet headers and metadata. They can also have data (parameters) provided by the control plane at runtime.
- Match-Action Units: Construct lookup keys from packet fields or computed metadata, then perform table lookups based on the constructed key, selecting an action (including associated parameters) to execute, and finally executing it.
- Control Flow: Expresses an imperative program that describes packet processing in a target, including the sequence of match-action unit invocations. It ultimately defines the execution sequence of tables, enabling conditional executions.
- Extern Objects: Specific constructions of each P4 architecture that can be manipulated by P4 programs through APIs. Their internal behavior is hardwired, meaning they are not programmable using P4. An example is checksum units. Different architectures have different extern objects.
- User-Defined Metadata: Metadata defined by the user associated with each packet.

- Intrinsic Metadata: Metadata provided by the architecture, associated with each packet, such as the ingress port through which the packet was received.

Figure 2.8 shows a typical tool workflow when programming a target using P4. Manufacturers provide the hardware or software implementation framework, an architecture definition, and a P4 compiler for the target. Programmers supply the P4 program, which must be designed for the specific architecture since the source code is not directly consumed by network devices and therefore needs to be compiled. Compiling the program generates a data plane configuration that implements the forwarding logic described in the input program, and an API to manage the state of data plane objects from the control plane, enabling communication between them [23].

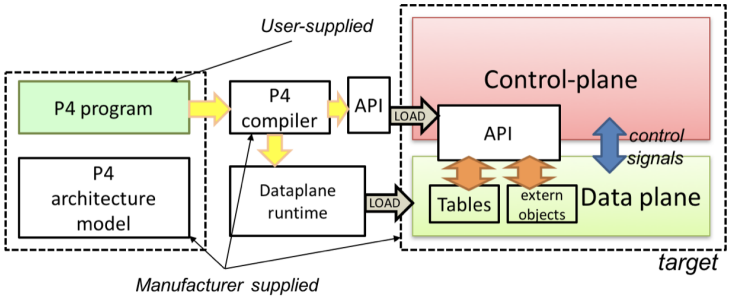


Figure 2.8: Programming a target with P4 [23].

The P4 language has several elements [25]:

- Parsers are the main programmable blocks described in P4. They are implemented as state machines, and their purpose is to extract headers and indicate their order.
- Control blocks are responsible for the main match-action processing. In the control blocks, tables and their actions are defined and applied.
- Expressions: The implementation of parsers and control blocks is provided by the P4 programmer, who uses expressions and data types defined in the language. These expressions can use basic operations and operators, such as standard arithmetic and logical operations.
- Data types:
  - Basic types, such as `bit<n>`, which represents an unsigned bit string of size n, `int<n>` which represents a signed integer of size n, or

`varbit<n>`, which represents a variable-length bit string with a maximum size of n.

- The header type, which is an ordered collection of members that can contain the basic types mentioned. They are byte-aligned, can be valid or invalid, and provide several operations to manipulate the validity bit, such as `isValid()`, `setValid()` and `setInvalid()`.
- The struct type, which consists of an unordered collection of members without alignment restrictions. Generally, P4 programs combine a collection of headers into a struct to use in programmable blocks. It is important to recall that the parser determines the order of the headers to be recognized in the packet.
- Header stack type, which represents an array of headers of the same type.
- Header union type, which is an alternative to the previous type and contains at most one of the different headers. For example, a header union could be declared to consist of IPv4 and IPv6 headers if it is expected that all packets contain only one of these and not both.

Note that with `typedef` an alternative name can be defined for a type, `const` can be used to define constants, and the `#define` primitive can also be used.

P4 is a domain-specific language designed to be implemented on a wide variety of targets, including programmable network interface cards, FPGAs[1], software switches, and hardware ASICs. This is why the language is restricted to constructions that can be efficiently implemented across all these platforms. Some examples of hardware-based targets are Intel Tofino [50], FPGA [48], Netronome Agilio smartNICs [2], Raspberry Pi [59], AMD Pensando Data Processing Unit (DPU)[3] and multiple compilers for these targets (or a variety of, e.g. FPGAs). The Intel Tofino (formerly Barefoot Tofino)[4] is the first Ethernet switch ASIC that allows for user programmability. It is engineered to deliver extremely high throughput of 6.5 Tbit/s (4.88 B packets per second), supporting 65 ports operating at 100Gbit/s. Its next-generation version, the

[1] https://netfpga.org

[2] https://netronome.com/agilio-smartnics/

[3] https://community.amd.com/t5/corporate/amd-pensando-dpu-software/ba-p/630282

[4] https://www.intel.la/content/www/xl/es/products/details/network-io/intelligent-fabric-processors/tofino.html

Intel Tofino 2[5], offers throughput rates of up to 12.8 Tbit/s, with ports capable of running at up to 400Gbit/s. Tofino was originally developed by Barefoot Networks, a startup that Intel acquired in 2019. The Tofino ASIC uses the Tofino Native Architecture (TNA)[6], a custom P4 architecture that significantly enhances the Portable Switch Architecture (PSA). TNA offers support for advanced device capabilities needed to implement complex, high-performance data plane programs. The device is equipped with either 2 or 4 independent packet processing pipelines, each capable of serving 16 100Gbit/s ports. These pipelines can run the same P4 program or operate independently with different programs. Additionally, the pipes can be interconnected, enabling the creation of longer processing pipelines for more demanding applications [46]. Currently, Intel Tofino switches have been discontinued. However, they can still be acquired, and support for them is available.

Software-based ones include Behavioral Model version 2 (BMv2) [24] (the reference software switch for P4), P4OvS [75], P4rt-OVS [74] and PISCES [86]. P4c [26] is the reference compiler for P4, supporting both versions of the P4 language, $P4_{14}$ [30] (discontinued version but still supported on several targets) and $P4_{16}$ [23]. This compiler includes different backends, meaning different targets: simple_switch, DPDK [34], EBPF [22], among others [26].

However, p4c is a modular compiler, making it easy to add different backends. When compiling a $P4_{14}$ or $P4_{16}$ file, two files will be generated:

- A file with the extension `.p4i`, which is the result of executing the preprocessor on the given P4 program.
- A file with the extension `.json`, which is the file format expected by the target.

Additionally, it is possible to add a flag so that the compiler also creates a text file in "P4Info" format, which contains a description of the tables and other objects in the P4 program. This is useful for controlling the data plane.

The performance of programmable devices using P4 assumes a fixed cost for table lookups and interactions with external objects. All P4 programs, including parsers and controls, perform a constant number of operations per byte of a packet received and analyzed. Although parsers can contain loops,

[5] https://www.intel.la/content/www/xl/es/products/details/network-io/intelligent-fabric-processors/tofino-2.html

[6] https://github.com/barefootnetworks/open-tofino

as long as some header is extracted in each cycle, the packet itself limits the parser's total execution. This means the computational complexity of a P4 program is linear in the total size of all headers and does not depend on accumulated state size during data processing (such as the number of flows or the size of processed packets). These guarantees are necessary but not sufficient for fast packet processing across various targets [23, 62]

Compared to modern packet processing systems (e.g., those using microcode on custom hardware), P4 offers several significant benefits [23]:

- Flexibility: P4 allows many forwarding policies to be expressed as programs, unlike traditional switches with fixed forwarding functions.
- Expressiveness: P4 can express sophisticated, hardware-independent packet processing algorithms using only general-purpose operations and table lookups. These programs are portable to other targets implementing the same architecture.
- Resource Mapping and Management: P4 programs describe storage resources abstractly, and compilers map these user-defined fields to available hardware resources, handling low-level details like scheduling and allocation.
- Software Engineering: P4 programs offer benefits such as type verification, information hiding, and software reuse.
- Component Libraries: Libraries provided by manufacturers can wrap specific hardware functions in portable high-level P4 constructs.
- Decoupling Hardware and Software Evolution: Target manufacturers can use abstract architectures to further decouple low-level architectural details from high-level processing.
- Debugging: Manufacturers can provide software models of an architecture to aid in the development and debugging of P4 programs.

2.2.1 Architecture model

As mentioned earlier, the P4 architecture identifies the programmable blocks and their data plane interfaces. The architecture does not need to expose the entire programmable surface of the data plane. For instance, a manufacturer might provide multiple definitions for the same device, each with different capabilities, such as with or without multicast support.

In Figure 2.9, there is a target with two programmable P4 blocks (P4 block #1 and P4 block #2), showing the interfaces between these blocks. Each block is programmed by a separate P4 code fragment. The target interacts with the P4 program through a set of registers or control signals. Input controls provide information to the P4 program, such as the ingress port through which a packet was received, while output controls can be written by the P4 program to alter the target's behavior, such as the egress port to which a packet should be sent. Control registers or signals are represented in P4 as intrinsic metadata. Additionally, P4 programs can store and manipulate data related to each packet using user-defined metadata.

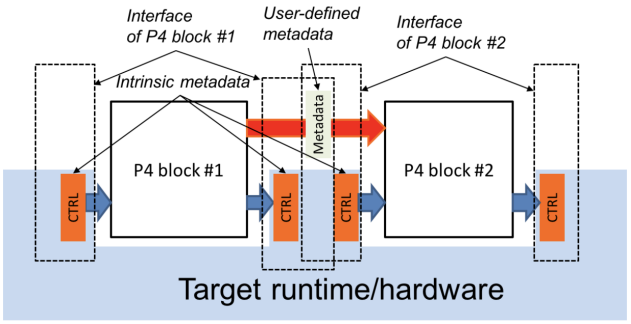


Figure 2.9: Interfaces of a P4 program [23].

The behavior of a P4 program can be entirely described in terms of transformations that map bit vectors to bit vectors. To process a packet, the architecture model interprets the bits written by the P4 program into intrinsic metadata. For instance, to forward a packet to a specific output port, a P4 program might need to write the output port identifier into a dedicated control register. Similarly, to drop a packet, a P4 program might need to set a "drop" bit in another dedicated control register. Note that the details of how the intrinsic metadata is interpreted are architecture-specific.

P4 programs can invoke services implemented by external objects and functions provided by the architecture. Figure 2.10 shows how a P4 program invokes the services of a checksum calculation unit built into a target. The implementation of the checksum unit is not specified in P4, but its interface is. In general, the interface for an external object describes each operation it provides, including its parameters and return types. These external blocks can be viewed as a black box.

In general, P4 programs are not expected to be portable across different

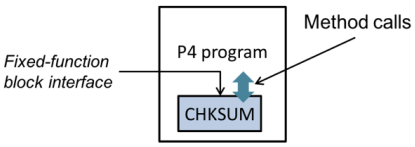


Figure 2.10: A P4 program calling the methods from a fixed function object [23].

architectures. For example, running a P4 program that performs a broadcast of packets by writing to a custom control register will not work correctly on a target that lacks such a control register. However, P4 programs written for a given architecture should be portable across all targets that faithfully implement the model, provided there are sufficient resources [23]. There are several P4 architectures, such as Portable Switch Architecture (PSA)[42], SimpleSumeSwitch[72], Portable NIC Architecture (PNA)[32] and V1Model[31].

2.2.2 P4Runtime

P4Runtime is another component of the P4 ecosystem. It is defined as a standard, open, and hardware-independent API that enables runtime control of P4 data planes. It is open, meaning it can be used to control any ASIC switch, and it is extensible and customizable, allowing different networks to use different protocols and functionalities with the same API [62].

The P4Runtime API allows control of the data plane elements of a device as defined by a P4 program. To achieve this, it generates a controller that can manage P4 devices. This controller can, among other things, manipulate the tables defined by the P4 program (by creating, deleting, modifying, or querying entries), query stateful elements, and change the forwarding pipeline configuration (i.e., change the P4 program running on a switch), all at runtime.

The P4Runtime API defines the messages and the semantics of the interface between the controller and the target it will control. The API is defined using Protocol Buffers (protobuf) [33] files, which are an extensible, language- and platform-independent mechanism for serializing structured data. Developed by Google and open-source, protobuf allows one to define the structure of data once and then use generated source code to easily write and read structured data from and to various data streams and multiple languages. It currently supports generated code in Java, Python, Objective-C, and C++. protobuf was developed as an alternative to XML [27, 33, 41].

First, the structure of the data to be serialized must be defined in a *proto*

file (a text file with the `.proto` extension). This data is structured as messages, where each message is a small logical record of information containing a series of name-value pairs called fields. Once the data structures are specified, the protobuf compiler, called `protoc`, is used to generate data access classes in the preferred programming language. These classes provide simple access to each field, such as *getters/setters*, as well as methods to serialize/deserialize the entire structure from/to raw bytes.

gRPC [1] is an open-source, high-performance framework for Remote Procedure Call (RPC) that can run in any environment, developed by Google. gRPC can use protocol buffers as its Interface Definition Language (IDL) and as its underlying message exchange format. In gRPC, a client application can directly call a method on a server application running on a different machine as if it were local, simplifying the creation of distributed applications and services. As with many RPC systems, gRPC relies on defining a service by specifying the methods that can be called remotely along with their parameters and return types. These gRPC services are defined in `proto` files, with RPC method parameters and return types specified as protobuf messages. The server implements this interface and runs a gRPC server to handle client calls, while the client has a *stub* (simply called a client in some languages) that provides the same methods as the server.

gRPC clients and servers can run and communicate with each other in a variety of environments, from servers to personal computers, and can be written in any of the languages supported by gRPC. For example, one can create a gRPC server in Java with clients in Go, Python, or Ruby [4]. A communication diagram between gRPC clients and servers can be seen in Figure 2.11.

The P4Runtime API is implemented by a program running a gRPC server. This server is associated with an implementation of the P4Runtime service interface (automatically generated). This program is called the "P4Runtime Server." [41]

In Figure 2.12, the reference architecture of P4Runtime is illustrated. The device or *target* to be controlled is shown at the bottom (in dark purple), and a controller is displayed at the top. There can be more than one controller, but P4Runtime only grants write access to a primary controller for each entity, whereas any controller can have read access. Controllers have a gRPC client that remotely calls the methods provided by the gRPC server located

[1] https://grpc.io

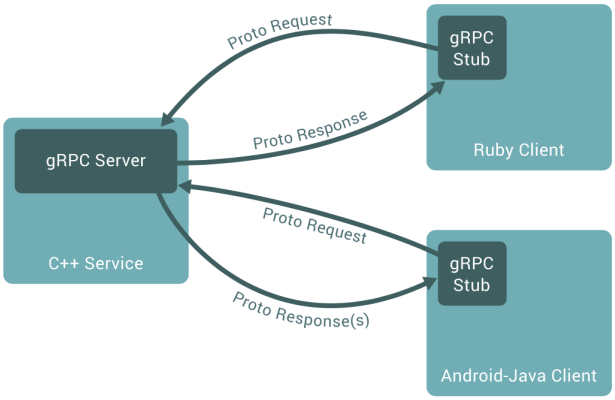


Figure 2.11: Example of a communication diagram between gRPC clients and servers [4].

on the *target*. These methods are used to control P4 elements. For example, to configure the P4 program of a *target*, the controller uses an RPC called `SetForwardingPipelineConfig`.

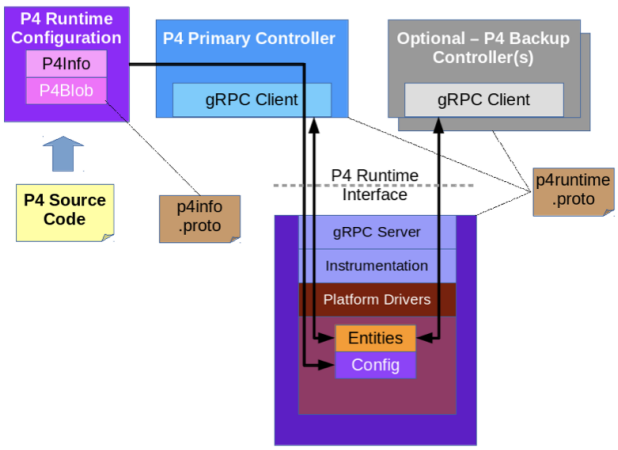


Figure 2.12: P4Runtime reference architecture [41].

The controller can access the P4 entities declared in the `P4Info` file's metadata. The structure of this file is defined by the `p4info.proto` file. The controller can also modify the forwarding pipeline configuration, which involves installing and running the compiled P4 program output and installing the associated P4Info metadata. Additionally, the controller can query the *target* to obtain information about the device configuration and P4Info. In summary, P4Runtime not only allows for loading different P4 programs onto devices but also enables querying device configuration information (such as control plane tables) and other objects (like counters), as well as modifying control plane

rules, all at runtime [41].

2.2.3 Available Resources

The resources available for P4, catering to both beginners and experts, are extensive. These include documentation, tutorials, sample code, videos, and tools. The P4 community is active and consistently contributes to the ecosystem.

- The official P4 website is available at [1]. This site provides the latest updates, relevant information, working communities, communication channels, events, publications, and more.
- The original P4 paper is available at [11].
- A P4 forum is available at [2]. This forum covers a wide range of topics, from advanced discussions to beginner-friendly ones. It is highly active, featuring responses from experts, P4 members, and enthusiasts.
- The P4 language repository is available at [3]. It contains multiple repositories, among the following:
  - The P4 programming language specifications repository [4]. The $P4_{16}$ specification is available at [23] and the $P4_{14}$ specification is available at [30].
  - The official P4 tutorial [5], that includes a virtual machine with all necessary tools already installed and a set of different use cases. The tutorial slides are available at [6]. These are very comprehensive and highly useful for anyone who is starting to explore the world of P4.
  - The behavioral model, the reference P4 software switch [7].
  - The P4Runtime repository [8]. Is specification is available at [9]. It also includes a shell[10] for P4Runtime, easing its use.

[1] https://p4.org
[2] https://forum.p4.org
[3] https://github.com/p4lang
[4] https://github.com/p4lang/p4-spec
[5] https://github.com/p4lang/tutorials
[6] https://docs.google.com/presentation/d/1zliBqsS8IOD4nQUboRRmF_19poeLLDLadD5zLzrTkVc/edit?pli=1#slide=id.g37fca2850e_6_141
[7] https://github.com/p4lang/behavioral-model
[8] https://github.com/p4lang/p4runtime
[9] blob:https://p4.org/191510e9-32f6-4f15-a5ed-1e8ba0451225
[10] https://github.com/p4lang/p4runtime-shell

  - The reference compiler for P4 (p4c) repository [11].

- A repository[1] containing extensive information about P4, including scripts to install all the necessary tools for developing and testing P4 programs.
- A Mininet extension that allows the user to create and test virtual networks that can include P4 switches is available at[2].
- The specification for the Portable Switch Architecture (PSA) is available at [3], the specification for the P4 Portable NIC Architecture (PNA) is available at [4] and the V1Model source code is available at [5].
- The P4 Language Cheet Sheet is available at [6].
- The P4 Language Consortium Youtube Channel is available at [7].
- Documents for multiple pasts events (including hackatons, workshops, between others) are available at [8].
- P4 ONF's blog is available at [9].

2.2.4 Applications

The applications of P4 are countless. P4 has been used in both hardware and software versions to implement hundreds of solutions. An example of several solutions using P4 can be found in section 3.1.1, which presents the related work for the proof-of-concept software. In this section both software and hardware solutions are presented, all implemented in P4. To illustrate, here are a few interesting examples of P4 applications:

- NDP protocol: Re-architecting datacenter networks and stacks for low latency and high performance [44]. This is a clear example for using P4 (and network programmability in general). The authors designed a new protocol for the data center and implemented a switch in P4 that can process packets in the format of the new protocol.

[11] https://github.com/p4lang/p4c
[1] https://github.com/jafingerhut/p4-guide
[2] https://nsg-ethz.github.io/p4-utils/introduction.html
[3] https://p4.org/p4-spec/docs/PSA.html
[4] https://p4.org/p4-spec/docs/PNA.html
[5] https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4
[6] https://drive.google.com/file/d/1Z8woKyElFAOP6bMd8tRa_Q4SA1cd_Uva/view
[7] https://www.youtube.com/@p4languageconsortium267/videos
[8] https://p4.org/events/
[9] https://opennetworking.org/tag/p4/

- SRv6 uSID implementation on P4[10]. The SRv6 "micro segment" solution, known as uSID, is an enhancement to the SRv6 Network Programming model. It enables the representation of SRv6 segments in a more compact and efficient manner, utilizing only two bytes for uSID compared to the standard 16-byte IPv6 address used for regular SRv6 segments. The authors developed the SRv6 uSID solution using the P4 language.
- P4 LB: Load Balancing Using P4 in Software-Defined Networks [52]. This work implements a load balancer using P4, analyzing packet headers and utilizing stateful objects to record data flow information, introducing four packet scheduling schemes: connection hash, random, round-robin, and weighted round-robin.

2.3 Other DPP Languages

Despite P4 being the most well-known and widely used language for data plane programming, this alone is not a sufficient reason to use it. Therefore, it was decided to evaluate the other available languages, assessing each one in terms of its architecture, potential applications, available resources, and more.

2.3.1 Click

The Click Router [55] is introduced as a versatile, modular software architecture designed for constructing routers. These routers are composed of packet processing modules known as elements. The core interface of an element mainly includes functions for initialization and packet handling, but it can be expanded to support additional functionalities. To configure a router, the user links various elements together in a directed graph, where the connections, or edges, represent the different paths for packet forwarding. Users can also enhance a configuration by creating new elements or by recombining existing ones in various ways. Click was developed as an extension to the Linux kernel on a general-purpose PC [55].

[10] https://netgroup.github.io/p4-srv6-usid/

2.3.1.1 Architecture

A Click element, as previously described, is a packet processing module designed for conceptually simple tasks, like decreasing an IP packet's time to live field, rather than more complex operations such as IP routing. A router configuration is represented as a directed graph, where the elements serve as vertices, and the edges between them indicate possible paths for packet transfer. Every action performed by the software of a Click router, including device handling, routing table lookups, queueing, and packet counting, is encapsulated within an element. The user controls the behavior of a Click router by selecting which elements to use and how they are interconnected.

Each element in Click is a C++ object that may maintain its own private state, with connections (or edges) represented as pointers to these objects. Passing a packet through a connection is implemented as a single virtual function call.

The key properties of an element include:

- Element class: Each element belongs to a specific class that defines the code executed when processing a packet, as well as its initialization procedure and data structure.
- Ports: Elements can have multiple input and output ports, with connections defined between an output port of one element and an input port of another. Ports can also have specific roles, for example, the second output port is generally used to send erroneous packets.
- Configuration string: An optional string provided during router initialization that can be used to set the element's state or modify its behavior.
- Method interfaces: While every element supports the basic packet-transfer interface, they can also create and export additional interfaces for runtime communication, which may include both methods and data, such as a queue exporting its length.

In summary, a Click element is a C++ object with defined ports and customizable behavior through configuration strings and method interfaces, enabling modular packet processing within a router configuration.

Figure 2.13 shows a diagram of a sample element "Tee(2)", including the properties described before. "Tee" is the element class, which copies every packet received through its input port (triangular port in the figure) and sends

it to each output port (rectangular ports). The configuration string is represented by the string between the parenthesis. In this case, the string "2" is interpreted by the class Tee as a request for two outputs. Last, method interfaces are not shown explicitly, as they are implied by the class. Figure 2.14 shows a router configuration (directed graph), that counts incoming packets and then drops them all. This configuration is compound by three elements, an element that receives the packet, passing it through an element that serves as a counter, and finally passing it through an element that drops it.

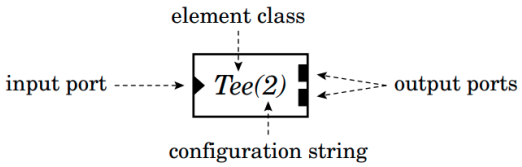


Figure 2.13: A sample element "Tee(2)" [55].

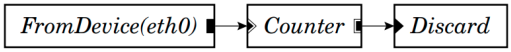


Figure 2.14: A click router configuration diagram, that counts and then drops all packets [55].

Click's configuration language is straightforward, comprising two primary constructs: declarations, which generate elements, and connections, which specify the linking of these elements. The example below illustrates the creation of three elements and their connections.

```
// Declare three elements...
src :: FromDevice(eth0);
ctr :: Counter;
sink :: Discard;
//... and connect them together
src -> ctr;
ctr -> sink;

//Alternate definition using syntactic sugar
FromDevice(eht0) -> Counter -> Discard;
```

Users can also create what are known as compound elements, which allow them to define custom element classes. A compound element is a fragment of a router configuration that includes various connected elements, which can then be used in the overall configuration just like a standard Click element class.

There are two drivers available for running Click router configurations: a Linux in-kernel driver and a user-level driver that interfaces with the network using Berkeley Packet Filter (BPF) or similar packet socket mechanisms. The user-level driver is particularly useful for profiling and debugging, whereas the in-kernel driver is more suited for production use.

To install a Click router configuration, the user provides the configuration file to the kernel driver via the Linux proc[1] file system. Click operates as a kernel thread within the Linux kernel, running the router driver, which continuously processes tasks in a loop. There are two ways to modify a configuration in Click without losing data:

- Handlers: They serve as access points for user interaction, and each element can have multiple handlers. This method is ideal for making changes that are specific to a single element. For example, a Click routing table might offer "add_route" and "del_route" handlers.
- Hot swapping: This method is suitable for more extensive configuration changes. The user can create a new configuration file and install it using the hot-swapping option.

Additionally, element class definitions can be dynamically added or removed from the Click kernel driver.

However, there are some limitations. Click's elements may not be suitable for all scenarios, as the system is primarily organized around packet flow. For instance, complex protocols such as BGP do not naturally break into parts (elements) among which packets flow. Click also lacks the ability to schedule Central Processing Unit (CPU) resources per individual flow and does not support variables, which can lead to information duplication [55].

There is an enhanced version of the Click Modular Router called FastClick [6], which is backwards compatible with standard Click elements and is designed as a high-speed userspace packet processor. FastClick introduces features like batching, advanced multi-processing, and improved support for Netmap [82] and DPDK [34].

2.3.1.2 Available resources

The available resources for Click include various tools and documentation to assist users in building and configuring Click routers. These include:

[1] https://docs.kernel.org/filesystems/proc.html

- The Click paper "The Click Modular Router" is available at [55], while the previous version of Click has its own paper available at [67]. The main difference includes more in-depth implementation details and performance analysis.
- The Click modular router toolkit is available at [1], showing how to compile and run a Click router.
- The FastClick code is available at [2], in conjunction with tutorials on Click in general and how to achieve high throughput and low latency.

2.3.1.3 Applications

Regarding applications of the Click router, the following could be found, among others:

- 5G-EmPOWER[3]. It proposes a mobile network operating system designed for heterogeneous wireless/mobile networks. The Agent is built using the Click modular router and implements a reference 5G-EmPOWER WiFi Access Point.
- Hummingbird: a validation solution over fully dynamic paths [47]. Hummingbird as well as a non-sampling Baseline is implemented using the Click router.
- Hierarchical Multiresource Fair Queueing for Packet Processing [94]. The authors present two new multi-resource fair queueing algorithms to support hierarchical scheduling, collapsed Hierarchical Dominant Resource Fair Queueing (collapsed H-DRFQ) and dove-tailing H-DRFQ. The proposed algorithms are implemented on the Click router.

2.3.1.4 Comparison with P4

Click was first introduced in 1999, making it an older technology that has certainly influenced subsequent data plane programming languages. The most recent commit to the Click modular router toolkit repository was in July 2022, and the presence of many unresolved issues suggests that it may no longer be actively maintained.

[1] https://github.com/kohler/click
[2] https://github.com/tbarbette/fastclick
[3] https://github.com/5g-empower/empower-lvap-agent

Click supports several platforms, including Linux, macOS, BSD, and partially Windows [35]. This compatibility allows Click to run on OpenWRT [4], a specialized Linux distribution for wireless routers. However, unlike P4, Click does not have the capability to map programs to various target hardware switches [11].

Click is considered relatively easy to learn, thanks to its modular design, which simplifies the composition of different elements. However, this modularity can also have a negative impact on performance. As previously mentioned, Click is well-suited for programs that align with packet flows, which excludes certain protocols that do not match well.

Click is a fairly expressive language, ideal for defining how a CPU's kernel processes packets. However, it cannot map to the parse-match-action pipelines on dedicated hardware. Moreover, Click is not designed for controller-switch architectures, which means programmers cannot specify match-action tables that can be dynamically filled with correctly-typed rules. Finally, Click makes it difficult to determine dependencies that would limit concurrent execution [11].

2.3.2 BESS

BESS (Berkeley Extensible Software Switch) [43] (formerly known as SoftNIC) introduces a new architecture aimed at enhancing NIC features, which often fall short of application requirements and lack flexibility. BESS allows users to develop features in software with minimal performance overhead. It operates as a hybrid software-hardware architecture by adding a software shim layer between the NIC hardware and the network stack, thereby extending NIC capabilities with software. This approach enables high-performance packet processing in software while still leveraging hardware features. BESS is designed to offer high performance, modularity, and backward compatibility with existing software. While BESS is based on Click [55], it aims to simplify and extend the design choices of Click. According to the authors, it offers similar hardware-level performance with the flexibility of software, allowing for advanced NIC functionalities [43, 58].

[4] https://openwrt.org

2.3.2.1 Architecture

Figure 2.15 illustrates the BESS architecture, where packet processing is depicted as a dataflow multigraph made up of modules. Each module represents a NIC feature and carries out specific operations on the packets passing through it. Ports fulfill the role of sources and destinations, with packets entering at one port, flowing through the various modules in the graph, and exiting through another port. The dataflow graph in BESS supports two types of ports:

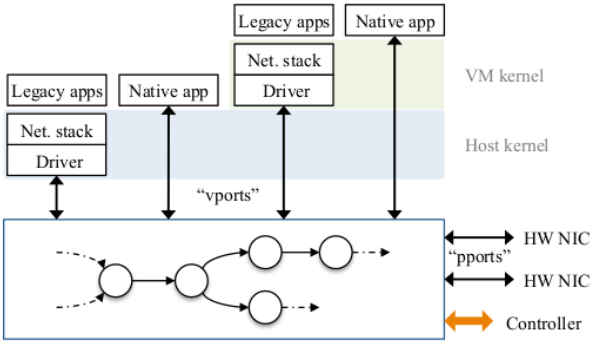


Figure 2.15: BESS architecture [43].

- Virtual ports (vports): These ports serve as interfaces between BESS and upper-layer applications. Vports connect BESS to a peer, which can either be the BESS device driver (used for supporting legacy applications that rely on the kernel's TCP/IP stack) or a BESS-aware application that bypasses the kernel. These peers can reside either on the host machine or within a virtual machine.
- Pyshical ports (pports): These ports interface BESS with the NIC hardware. Pports provide a set of hardware-implemented primitives, such as checksum calculations.

Vports emulate an ideal NIC port, supporting all features required by their connected peer. The modules within the dataflow pipeline are responsible for implementing these features, using software, hardware, or a combination of both. BESS thus creates a hardware abstraction layer, shielding peers from the limitations of the underlying hardware. An explicit control channel allows an external controller to provide forwarding policies, while BESS focuses exclusively on the data plane. This control channel supports three types of operations:

- Updating the data path (e.g., adding or removing modules or ports).
- Configuring resource allocations (e.g., limiting CPU or bandwidth usage for applications).
- Managing individual modules (e.g., updating flow tables or collecting statistics).

Each module can produce certain metadata based on its function, which is carried along the pipeline. Packets carry this metadata as a list of key-value pairs. Modules specify the metadata fields they require as input and the fields they produce as output. This specification is useful for error detection if a required field is not provided by an upstream module, and for discarding unused metadata during configuration.

Figure 2.16 illustrates an example pipeline where packets are processed through a switching service, TCP segmentation offload (TSO) if necessary, and checksum offloading. In this scenario, the NIC lacks TSO capabilities and only supports checksum offloading for TCP/IPv4 packets. From the peer's perspective, the vport appears as a fully functional NIC, providing all the mentioned features. This example also demonstrates how BESS modules can implement NIC features by offloading computations to hardware when needed.

The packet processing in the example shown in Figure 2.16 is carried out as follows, with each rectangle in the figure representing a different module:

- vport_inc: This module pulls a packet descriptor, creates a BESS packet buffer containing the packet data, and adds metadata fields for the input port ID (`inc_port`) and checksum offloading description (`csum_sw`).
- parser: This module inspects the packet's headers from layers two to four and records the results in the metadata field `hdr_info`.
- switch: Using the metadata `inc_port` and `hdr_info`, this module performs L2 switching. It then uses the destination address to determine which output edge to send the packet to, in this case, the TSO module.
- TSO: This module checks whether the packet is a TCP packet larger than the Maximum Transmission Unit (MTU) of the `output_port`. If so, it performs TSO, segmenting the packet into multiple MTU-sized packets, copying all necessary metadata into each packet, and updating `csum_sw`.
- checksum: Based on `csum_sw` and `hdr_info`, this module determines whether the checksum needs recalculation and if it should be done in software. If the NIC associated with `out_port` can handle the checksum,

the module sets `csum_sw` to "on"; otherwise, it computes the checksum in software.

- pport_out: This module sends the packet to the NIC, with flags indicating whether the hardware should compute the checksum.

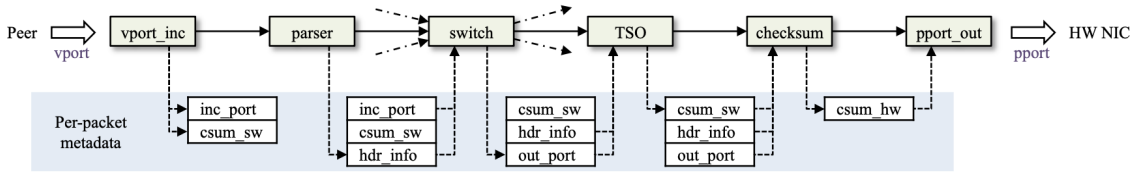


Figure 2.16: BESS example pipeline showing the metadata passing through the pipeline [43].

2.3.2.2 Overview

The BESS prototype is implemented in 14,000 lines of C code and runs on unmodified Linux and QEMU/KVM, with the expectation that supporting other operating systems or virtualization platforms would be straightforward. BESS operates as a user-mode program on the host, using one or more dedicated cores to reduce the costs of context switching.

BESS pipeline components are:

- pports: Built on the Intel Data Plane Development Kit (DPDK) [34] for high-performance packet I/O. Pports allow direct access to NIC hardware without kernel intervention. Two module instances are associated with each pport: `pport_out` converts packet metadata into hardware-specific offloading primitives, while `pport_in` accepts incoming packets and converts hardware-offload outcomes into metadata for other modules.
- vports: Vports consist of RX and TX queues with two one-way ring buffers for transmitting packet buffers and receiving completion notifications. Ring buffers are allocated in shared memory between BESS and the peer. The authors developed a device driver for traditional TCP/IP applications as a Linux kernel module, which can be used by both hosts and guests. This driver is designed with the expectation that it can be easily ported to other operating systems. The kernel network stack and applications remain unaffected, as the driver presents a vport as a

standard Ethernet device. For applications that bypass the kernel and use their own specialized or streamlined network stacks, the authors provide a user-level library, enabling direct access to vport queues with the option of zero-copy support. However, when vports interface with the kernel device driver, packet data must be copied, as implementing zero-copy support within the kernel would require significant modifications. BESS uses inter-core interrupts to notify peers when packets are transmitted, although this notification is unnecessary when a peer sends a packet, as BESS employs polling. Peers can choose to disable interrupts temporarily or permanently.

The main components of BESS are illustrated in figure 2.17, which include:

- bessd: The core of the software switch, the "BESS daemon", that transfers packets between ports and modules.
- ports: Entry and exit points for packets in bessd, which can be connected to network interfaces, virtual machines, containerized applications, or user-space processes.
- modules: The building blocks of the packet processing.
- bessctl: The controller for bessd, allowing administrators to manage port-module connections, monitor traffic flow, and execute administrative commands through a CLI.

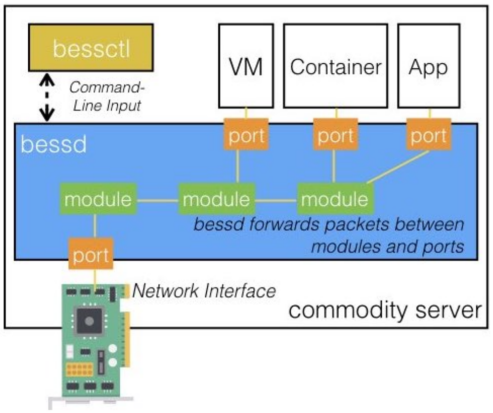


Figure 2.17: BESS main components [58].

BESS operates entirely in userspace and directly binds to network interfaces using DPDK, bypassing the kernel. This kernel bypass is a key factor in BESS's high performance, as packets can be processed directly in userspace, avoiding

the overhead of the kernel networking stack [58]. Figure 2.18, similar to Figure 2.17, illustrates this setup, highlighting the separation between kernel and userspace.

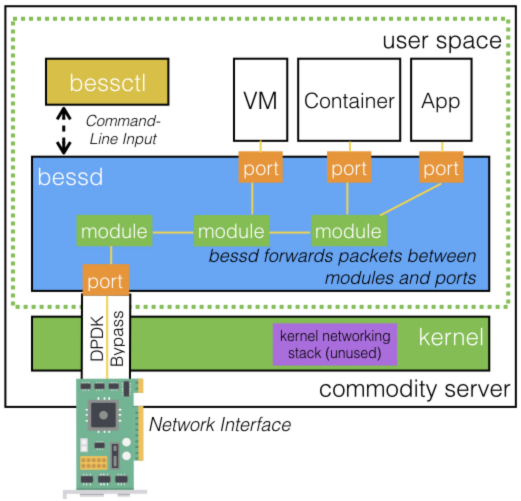


Figure 2.18: BESS main components with kernel/user space division [58].

BESS scripts are essentially Python programs with some additional features. Below is an example of a BESS configuration script for an Access Control List (ACL):

```
import scapy.all as scapy
import socket

def aton(ip):
     return socket.inet_aton(ip)

# Craft a packet with the specified IP addresses
def gen_packet(proto, src_ip, dst_ip):
    eth = scapy.Ether(src='02:1e:67:9f:4d:ae', dst='06:16:3e:1b:72:32')
    ip = scapy.IP(src=src_ip, dst=dst_ip)
    udp = proto(sport=10001, dport=10002)
    payload = 'helloworld'
    pkt = eth/ip/udp/payload
    return str(pkt)

packets = [gen_packet(scapy.UDP, '172.16.100.1', '10.0.0.1'),
           gen_packet(scapy.UDP, '172.12.55.99', '12.34.56.78'),
           gen_packet(scapy.UDP, '172.12.55.99', '10.0.0.1'),
           gen_packet(scapy.UDP, '172.16.100.1', '12.34.56.78'),
           gen_packet(scapy.TCP, '172.12.55.99', '12.34.56.78'),
           gen_packet(scapy.UDP, '192.168.1.123', '12.34.56.78'),
          ]

fw::ACL(rules=['src_ip': '172.12.0.0/16', 'drop': False])
```

```
Source() -> Rewrite(templates=packets) -> fw -> Sink()
```

Modules are created by declaring module objects and assigning them a name using `::` (as in `fw::ACL(...` in the example above). The modules are connected with arrows (`->`), representing the unidirectional flow of packets. The final line of the code demonstrates a chain of four modules. The `Source()` module generates all packets (initially empty), which are then passed to the `Rewrite` module. This module takes a parameter (a packet "template") and fills each incoming packet with a copy of the specified template. Once the packets are filled, they are sent to the `fw` module, which also takes parameters. By default, `fw` drops all packets, but with a rule that allows packets (drop:False) where the src_ip matches 172.12.0.0/16, packets from this prefix are permitted through. These packets then reach the `Sink()` module, which simply deletes any packet that enters it [1].

2.3.2.3 Available resources

The available resources for BESS include tools and documentation on installation, usage and configuration.

- The original BESS technical report "SoftNIC: A Software NIC to Augment Hardware" is available at [43]. This report presents the idea behind BESS and evaluation of performance.
- The BESS website is available at [1]. This website provides a brief overview of BESS and centralizes all resources (code, papers, etc.).
- The BESS code is available at [2]. This repository includes fairly detailed documentation on how to install, use, and configure BESS.

2.3.2.4 Applications

Regarding applications of BESS, not many could be found. Both cases are based on Network Function Virtualization (NFV), as BESS has been specifically optimized for NFV use cases [78]:

- E2 [77]. E2 is a framework for NFV applications, a scalable and application-agnostic scheduling framework for packet processing.

[1] https://github.com/NetSys/bess/wiki/Writing-a-BESS-Configuration-Script
[1] https://span.cs.berkeley.edu/bess.html
[2] https://github.com/NetSys/bess

- ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining [95]. It proposes a packet processing architecture that, whenever feasible, dynamically distributes packets to NFVs in parallel and intelligently merges their outputs to ensure the preservation of correct sequential processing semantics.

2.3.2.5 Comparison with P4

BESS is built on the foundation of Click, but the authors assert that they do not compromise performance for programmability. Instead, they aim to simplify and extend Click's design choices. In BESS, modules represent broader actions, such as switching, whereas Click modules are designed for more granular tasks. The authors claim that this approach enhances performance by reducing the need for transitions between modules. BESS also assumes that each module internally manages its own queues as needed, which simplifies scheduling and improves scalability.

BESS is positioned as an efficient framework for advancing and supporting new networked systems, particularly those that aim to remove network policy enforcement from the data plane. Traditional network stacks have often relied on hardware techniques to implement NIC functionalities due to the challenges they present. While making NIC hardware more programmable, such as with FPGA, is an option, it still faces limitations in terms of programmability and resource constraints. An alternative to BESS could involve placing a general-purpose CPU on the NIC card as a bump-in-the-wire. However, the authors argue that BESS's approach, which reuses existing server resources, is superior in terms of resource efficiency, elasticity, and maintaining a system-wide perspective [43].

As of today, the last commit on the BESS repository was in March 2022 (more than two years ago), with many open issues, suggesting that it is no longer actively maintained. Additionally, BESS is more suitable for execution in traditional servers and NFVs.

2.3.3 VPP

Vector packet processing is a widely used approach in high-performance packet processing applications like VPP and DPDK. On the other hand, scalar-based processing is typically employed by network stacks that do not have stringent

performance requirements.

In a scalar packet processing stack, packets are handled one at a time: an interrupt handling function takes a single packet from a network interface and processes it through a series of functions. While this model is straightforward, it can be inefficient in certain aspects.

Conversely, a vector packet processing stack handles multiple packets simultaneously by using a "vector of packets". The interrupt handling function passes this entire vector through the functions, allowing for optimizations such as pipelining and prefetching. These techniques help reduce read latency on table data and parallelize the packet processing workload [38].

2.3.3.1 Overview

The VPP platform is an adaptable framework that offers production-ready switch and router functionality right out of the box. It is the open-source version of Cisco's Vector Packet Processing (VPP) technology, a high-performance packet processing stack that can run on standard CPUs. The VPP platform is a part of the Fast Data Project (FD.io)[1].

VPP is built around the concept of a "packet processing graph", which makes the design modular and easily extendable. The nodes in this graph represent basic actions to be performed on packets, designed to be small, modular, and loosely coupled. This structure facilitates the addition of new nodes or changes to the connections between existing ones. New nodes can be introduced, or the graph's order can be modified, through the use of plugins. Plugins can also be developed as standalone components and installed by simply adding them to the plugins folder, allowing them to function as independent components. These plugins are shared libraries that VPP loads at runtime.

At runtime, the VPP platform creates a vector of packets by collecting all available packets from RX rings. The packet processing graph is then applied, node by node, to the entire vector of packets. Figure 2.19 illustrates an example of a packet processing graph with the packet vector as its input.

The VPP platform is versatile and its authors affirm it can be used to develop any packet processing application. The engine operates entirely in user space, so plugins can be created without modifying Linux kernel code. These plugins are written in C.

[1] https://fd.io

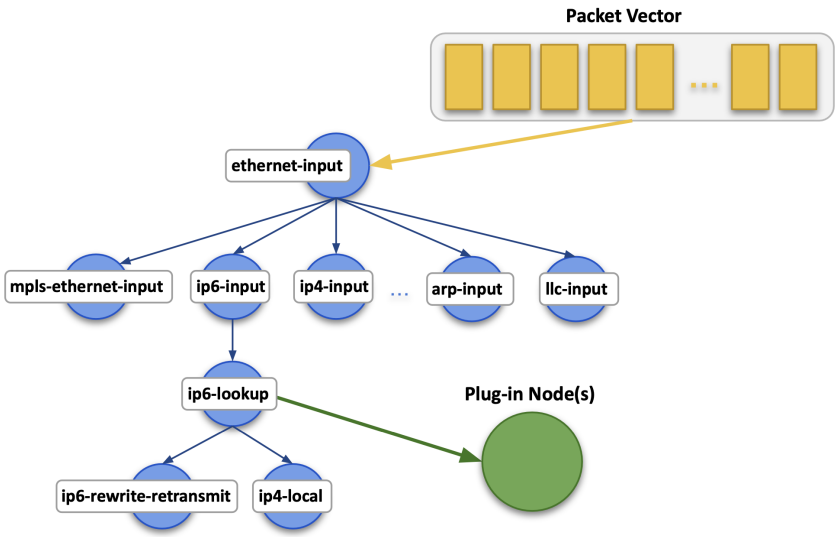


Figure 2.19: An example packet processing graph [38].

The VPP platform supports x86/64 and ARM-AArch64 architectures and can be installed on Debian and Ubuntu [37, 38].

VPP is continually enhanced through the extensive use of plugins. A notable example of this is the Data Plane Development Kit (DPDK), which provides essential drivers and functionalities for VPP [38].

VPP processes an entire vector of packets through one graph node before proceeding to the next node. This process, illustrated in Figure 2.20, optimizes performance. The first packet in the vector typically primes the instruction cache, enabling the subsequent packets to be processed rapidly. The fixed processing costs for the vector are distributed across the length of the vector, resulting in very high and consistent performance. If VPP experiences a slight delay, the next vector contains more packets, which allows the fixed costs to be spread over more packets, reducing the average processing cost per packet and enabling the system to catch up. As a result, both throughput and latency remain relatively stable. If multiple cores are available, the graph scheduler can assign (vector, graph node) pairs to different cores [36].

2.3.3.2 Available resources

The available resources for VPP include tools and extensive documentation, between various use cases presented. These include:

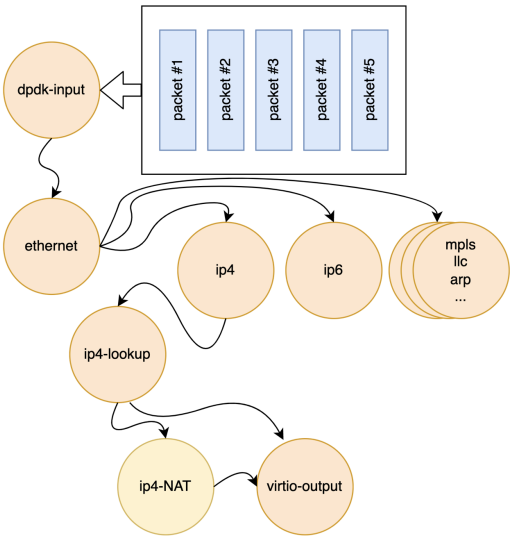


Figure 2.20: An example packet processing graph showing how VPP processes an entire vector of packets through one graph node before proceeding to the next node [36].

- VPP documentation is available at [1], where a general overview of VPP is presented. Then, in [2], an exhaustive documentation is provided, including a description of the language, multiple use cases, installation and execution guides for VPP along with troubleshooting, development documentation, and more.
- VPP's code is available at [1].
- The FD.io website is available at [2]. The Fast Data Project (FD.io) is an open-source project aimed at providing fast and secure networking data plane through Vector Packet Processing (VPP).
- The FD.io Youtube channel is available at [3]. This channel includes numerous videos, including tutorials, presentations, use cases, and more.

2.3.3.3 Applications

According to the FD.io website, there are multiple FD.io users. Commercial solutions based on FD.io are offered by companies like Netgate, Cisco, PANTHEON.tech, among others. These users include:

[1] https://wiki.fd.io/view/VPP/What_is_VPP%3F
[2] https://s3-docs.fd.io/vpp/23.10/index.html
[1] https://github.com/FDio/vpp
[2] https://fd.io
[3] https://www.youtube.com/channel/UCIJ2OP6_i1npoHM39kxvwyg/videos

- Cisco with the ASR 9000 Router[4] that provides advanced programmability and with the Carrier Grade Services Engine (CGSE). It provides IPv6 translation and DDoS mitigation solution[5].
- Alibaba Cloud does Network Service Optimization with the VPP Platform[1]. VPP is used as the foundation to develop a product similar to a VPN network gateway. The product is applied in scenarios such as hybrid cloud, retail chains, and the interconnection between a headquarters and its branches.
- TNSR [2] from Netgate is a high-performance software router based on FD.io's Vector Packet Processing (VPP). By using VPP, DPDK and other open-source technologies, they promise a high-performance router for a low cost.
- Inocybe promises an Open Networking Platform that enables customers to deploy integrated, cutting-edge networks[3]. It is based on FD.io, between other technologies.
- PANTHEON.tech with the StoneWork Enterprise solution[4]. It proposes a performant and modular networking solution, integrating a VPP dataplane, including multiple networking functionalities.
- Samsung's 5G UPF (User Plane Function) [40] stack is built on top of VPP. Samsung utilizes DPDK and VPP technologies, between others for higher performance processing.

2.3.3.4 Comparison with P4

In the Alibaba application[1], one of the aforementioned applications, the authors assert that VPP is mainly suitable for proof-of-concept (POC) projects and has several drawbacks. They argue that to use VPP for commercial purposes, a deep understanding of the platform is necessary, and that it is far from being a fully mature product.

Currently, the VPP repository is active, with commits made as recently

[4] https://www.cisco.com/c/en/us/products/routers/asr-9000-series-aggregation-services-rout
index.html
[5] https://www.cisco.com/c/en/us/products/collateral/routers/
carrier-routing-system/data_sheet_c78-614893.html
[1] https://www.alibabacloud.com/blog/network-service-optimization-with-the-vpp-platform_
593985
[2] https://www.netgate.com/tnsr#get-to-know
[3] https://www.cengn.ca/project/case-studies/inocybe/
[4] https://pantheon.tech/products/stonework/

as days ago. VPP runs on commodity CPUs rather than traditional switches. While VPP is similar to Click and BESS, it leverages a packet processing approach to enhance performance.

2.3.4 OpenState

The concept behind OpenState was to create a stateful data plane programming abstraction, as opposed to the stateless OpenFlow match/action table, with the goal of achieving high performance while aligning with vendors' preferences for closed platforms. The authors suggest using extended finite state machines as an enhancement to the OpenFlow match/action model. They argue that many stateful tasks involving only local states within individual links or switches are unnecessarily centralized for easier management and programmability, primarily because these tasks cannot be implemented on local OpenFlow devices without the explicit involvement of the controller for any state updates. The authors propose that the "dumbness" of the data plane is more a result of the limited capabilities of the OpenFlow data plane API than a deliberate design choice or a fundamental principle of SDN. The proposed solution is an abstraction that can formally describe the desired stateful processing of flows within the device itself, without requiring the device to be open-source or reveal its internal workings [8].

2.3.4.1 Abstraction

Figure 2.21 illustrates a state machine for a well-known use case called "port knocking", a technique used to open a port on a firewall. To model the desired behavior, the state machine is linked to each host. Starting from a default state, each correct port knock triggers a transition to intermediate states until the final open state is reached. Any knock on an unexpected port will cause a transition back to the default state. Once in the OPEN state, packets directed to port 22 will be forwarded, while all other packets will be dropped without resetting the state to default.

Every transition is caused by an event, which consists in a packet matching some criteria (in this case a given port number), and every state transition is associated to a forwarding action (in this case, forward or drop). We have two new aspects (from the OpenFlow match/action rule):

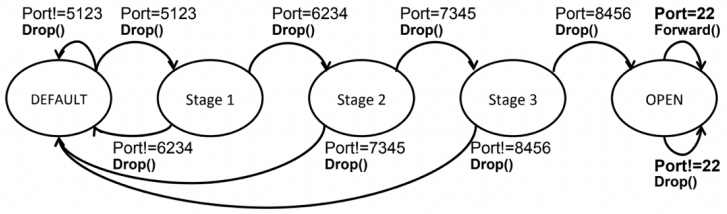


Figure 2.21: An example State Machine: port knocking [8].

- XFSM abstraction: The match that specifies an event not only depends on the packet header, but also depends on the current state. This event, not only causes an action, but also a transition to the next state. All this can be modeled by a Mealy Machine. The abstraction is made concrete (while maintaining platform independence) by restricting the set of possible actions to those available in current OpenFlow devices, and by restricting the set of events to OpenFlow matches on header fields and metadata easily implementable in hardware platforms. The states and transitions are left to the programmers' freedom.
- State Management: The authors recommend separating the matches that define events from the ones that define flows, meant as entities which are attributed a state. There are two tables: the State Table, that is queried with the packet header fields, in order to get the current state of the flow. Then, the XFSM Table is queried, using the current state and the headers, to find the action and next state that correspond. This next state is updated in the State Table.

Figure 2.22 shows how the proposed approach supports the port knocking example. The proposed abstraction misses a way to allow the update of states for a given flow by events occurring on different flows. This models a subset of important stateful operations, such as MAC learning, where the forwarding is done using the destination MAC address, but the table is updated using the source MAC address. Giving the programmer the ability to use an eventually different header field in the two accesses to the State Table (lookup and update) solves this problem. These are called lookup-scope and update-scope [8].

OpenState is an extension of OpenFlow, meaning that the implementation and installation of tables can utilize standard OpenFlow match tables and flow messages without requiring code changes. However, some additional modifications are necessary. This process is depicted in Figure 2.23, illustrating what

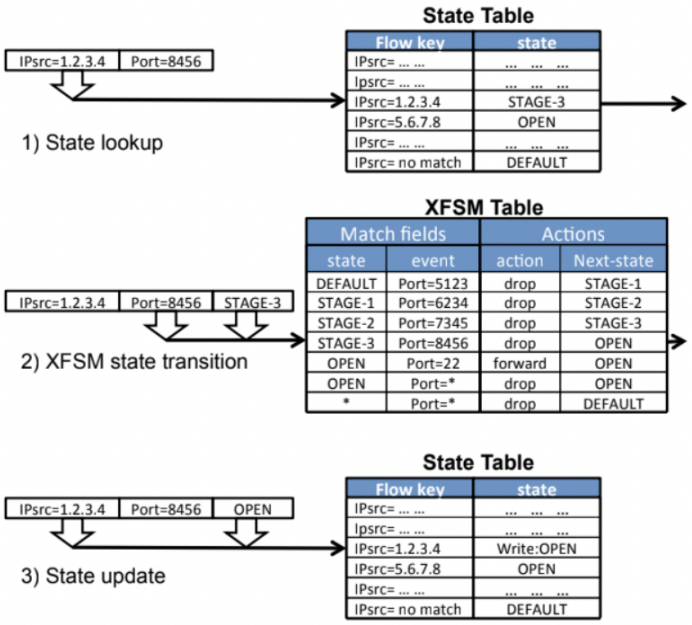


Figure 2.22: State and XFSM Tables for port knocking example [8].

is is known as a stateful stage - a logical block that comprises a State Table and an XFSM table, that implements the OpenState abstraction. The state machines are directly executed inside the network device, offloading controllers.

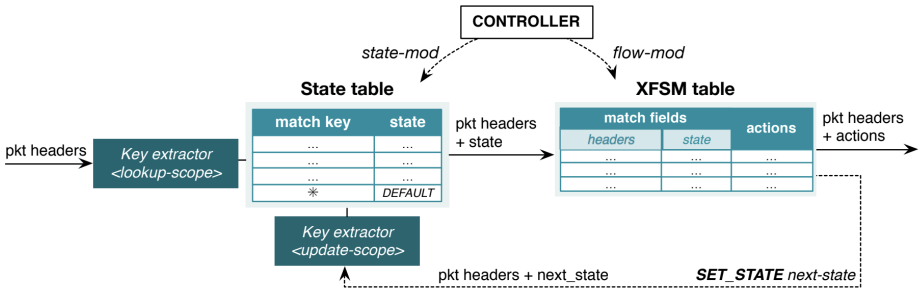


Figure 2.23: Architecture of the stateful stage. The XFSM table is represented by a standard Openflow table, while a SET STATE action is used to trigger updates on the state table [8].

2.3.4.2 Available resources

The available resources for OpenState include some reduced tools and documentation. These are:

- The original paper is available at [8].

- The specification is available at [1]. The provided documentation is full of to-do comments and questions.
- The code is available at [1].
- Other repositories from OpenState team are available at [2]. These include some use cases and other implementations (e.g. a controller).

2.3.4.3 Applications

Regarding applications of the OpenState, the following could be found, among others:

- SPIDER: Fault Resilient SDN Pipeline with Recovery Delay Guarantees [20]. This application comes from the OpenState authors. It proposes a fault resilient SDN pipeline design with programmable failure detection and recovery. This solution is implemented and experimentally validated using OpenState.
- Detour planning for fast and reliable failure recovery in SDN with OpenState [19]. The solution introduces a protection scheme against link or node failure and aims for zero packet loss after failure detection, regardless of controller availability. The mechanism is built on OpenState, that enables programmable, stateful forwarding rule adaptation, minimizing dependence on remote controllers.
- StateSec: Stateful Monitoring for DDoS Protection in Software Defined Networks [10]. The solution proposes an approach to protect end-hosts based on in-switch processing capabilities to detect and mitigate DDoS attacks.

2.3.4.4 Comparison with P4

As of today, last commit in the repository is from 2015, and the repository has only 30 commits, suggesting that is the original code presented for the paper and hasn't been maintained or updated. Actions taken on a packet are limited to the instructions defined in OpenFlow, thus fixed.

[1] https://github.com/OpenState-SDN/openstate-spec
[1] https://github.com/OpenState-SDN/ofsoftswitch13
[2] https://github.com/OpenState-SDN

2.3.5 Other languages

Other languages include FAST[68], Domino[87], FlowBlaze[79], POF[88] and NetKAT[3], all these used for programming PISA devices. These were also analyzed, but compared to P4, all fall short. Some remarks about these languages include:

- FAST: Abstractions seems compact, but it does not seem easy to write. There is no publicly available code and actions taken on a packet are limited to the instructions defined in OpenFlow. Also, the data plane algorithms are compiled in the controller rather than in the device itself.
- Domino: Domino targets line-rate switches, thus it is more constrained than other languages. The concept behind Domino seems overly complicated. There are problems for which Domino isn't a good fit, including algorithms that perform various changes within a packet, performing a variety of computations per packet, or algorithms that perform complex computations for some packets, but not all. As of today, last commit in the compiler's repository is from 2019.
- FlowBlaze: It is based on Extended Finite State Machines (EFSM). FlowBlaze is built on top of the NetFPGA open platform and both hardware and software sources are publicly available. As of today, last commit in the repository is from 2020, and the repository has only 27 commits, suggesting that is the original code presented for the paper and hasn't been maintained or updated.
- Protocol Oblivious Forwarding (POF): The repository has only four commits, from 2016 and the source code is not accessible anymore. POF support is discontinued, confirmed by the authors.
- NetKAT: Is promoted as having reasoning based on a strong semantic foundation, Kleene algebra with tests. NetKAT has a clear advantage that is the ability to prove and answer questions about the desired behavior of the network and forwarding policies. Besides this, it seems to be tedious to write programs, and restrictive to the available operations and headers. In search of the programmability of the network, NetKAT seems to be limited, and useful for cases where is of utmost necessity to guarantee the correct functionality of the policies. As of today, last commit on the repository was on July 2020, suggesting that it is not currently being maintained.

2.3.6 Remarks

P4 benefits from a large and active community, with several working groups dedicated to various aspects (Language Design, API, Architecture, Applications, and Education). Its key repositories, such as the compiler, tutorials, p4runtime, and behavioral model, are well-maintained with frequent updates and responses to open issues. The P4 ecosystem is extensively documented, with numerous tutorials and videos, offering far more comprehensive resources compared to other languages, which often lack sufficient documentation beyond the original paper. It is also one of the few, if not the only, that is actively maintained. P4 is highly expressive, enabling the implementation of a wide range of network protocols, and its code is easy and intuitive to write. Additionally, it has a powerful controller, P4Runtime, which allows for dynamic changes to the P4 program during runtime. Many experts regard P4 as the *de facto* language for data plane programming.

2.4 Programmable packet processing technologies

Several technologies are specifically designed for programmable network packet processing, and can be used for accelerating the data plane. These technologies aim to process packets with high efficiency, optimizing resource utilization. Software-based data plane acceleration solutions enhance packet processing capabilities, either in user space or within the kernel. In addition, there are kernel-level technologies for packet processing and filtering, which are optimized for handling packets at high speed. Among these are Berkeley Packet Filter (BPF) [63], eXpress Data Path (XDP)[1], and Data Plane Development Kit (DPDK)[2].

2.4.1 BPF

BPF (Berkeley Packet Filter) is a highly flexible and efficient virtual machine-like construct in the Linux kernel allowing to execute bytecode at various hook points in a safe manner. BPF has the original version, called “classic”

[1] https://www.iovisor.org/technology/xdp
[2] https://www.dpdk.org

BPF (cBPF), which is mostly obsolete. The other version is called extended BPF (eBPF). Nowadays, the Linux kernel runs eBPF only. BPF is a general purpose RISC instruction set. Although the name suggests that BPF's purpose is packet filtering, the instruction set is generic and flexible enough to allow for many more use cases [22].

2.4.1.1 cBPF

Initially, the code for an application was passed from the user space to the kernel, where it would be checked in order to guarantee safety and avoid kernel failures. After passing this check, the program would be attached to a socket and it would be executed for every packet that arrived. BPF emerged as a way to filter packets the earliest possible, avoiding the need to copy the packets from the kernel space to the user space in order to filter them through network monitoring tools from the user space. In this way, packet filtering performance was significantly improved, in comparison to existing solutions. BPF allows to add a filter to any socket from a program in the user space. BPF also defines a packet-based memory model, two registers: accumulator (A) and index register (X), an implied program counter, and a temporary auxiliary memory [90, 91].

BPF is a kernel architecture for packet capturing, which led to an important improvement of the performance in comparison to existing packet capturers of the time. Consequently, BPF is the packet filter language used by tools such as tcpdump and its successors [22]. This performance increase derives from two principal improvements in the architecture:

- BPF uses a "*filter machine*" based on registers that can be efficiently implemented on today's register based RISC CPUs. CSPF (CMU/Standford Packet Filter) [65] uses a filter machine based on memory stack, that doesn't work well with modern bottle-necked CPUs.
- BPF uses a simple, non-shared buffer model, that is possible thanks to today's larger address spaces. This model is very efficient for the "usual cases" of packet capture.

BPF has to main components: the network tap and the packet filter. The network tap gathers copies of the packets form network device drivers and delivers them to listening applications. The filter decides if a packet should be accepted and if it is the case, how much of the packet should be copied to the listening application. Figure 2.24 shows the BPF interface with the rest

of the system. Normally, when a packet arrives at a network interface, the link level device driver sends it to the protocol stack. Instead, when BPF is listening on this interface, the driver first calls BPF, which gives the packet to every filter of each participating process. These filters (defined by the user) decide if the packet should be accepted or not, and how many bytes of every packet should be saved. For every filter that accepts the packet, BPF copies the solicited amount of data to the buffer associated with that filter. Then the device driver regains control. If the packet wasn't destined to the local host, the driver returns from the interrupt, otherwise normal protocol processing proceeds. It is not possible to make a read system call for every packet, since a process could want to watch every packet in the network and time between packets could be of a few microseconds. Because of this, BPF should gather data from various packets, and return them as an unit when the monitoring application makes a read. In order to maintain packet boundaries, BPF encapsulates the information of every captured packet with a header that includes a timestamp, length and offset.

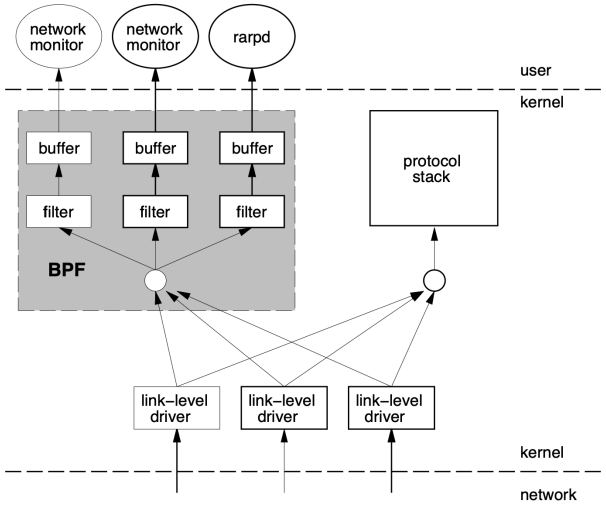


Figure 2.24: BPF Overview [63].

Packet filter When monitoring a network, normally only a small set of network traffic is wanted. There is an important performance improvement by filtering not wanted packets in the interrupt. To minimize memory traffic, the packet should be filtered "in place" rather than copying it to some other kernel buffer before filtering. So, when a packet is not accepted only those bytes needed for filtering are referenced by the host.

There are other designs, like SunOS's STREAMS NIT (Network Interface Tap) [49] which copies the packets before filtering resulting in performance degradation. This design is modular, being the packet filter module on top of the packet filter module. Authors claim that while a STREAMS-like design might appear to be elegant, there is very little design advantage in factoring the packet filter into a separate streams module, but great performance advantage in integrating the packet filter and the tap into a single unit.

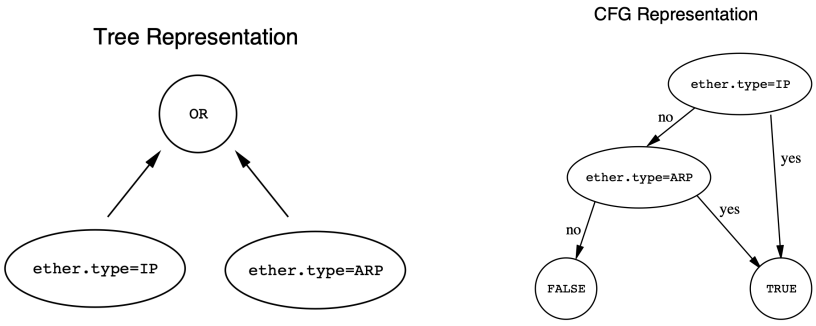
The Filter Model Assuming a reasonable design of the buffering model, this will be the dominant cost of the accepted packets, while the packet filter computation will be the dominant cost for the dropped packets. Most applications of packet capture drop more packets than they accept, so the good performance of the packet filter is critical to overall good performance. A packet filter is a boolean function evaluated on a packet. If the value is *true*, the kernel copies the packet to the application, if it is *false*, the packet is ignored. Traditionally, there has been two approaches to the filter abstraction:

- A boolean expression tree: in this model every node represents a boolean operation, while the leaves represent test predicates on packet fields. The edges represent operator-operand relationships.
- A directed acyclic control flow graph (CFG), which is the model used by BPF. In this model every node represents a packet field predicate while the edges represent control transfers.

In figure 2.25 we can see both models with a filter that recognizes an ARP or IP packet over Ehternet. These models are computationally equivalent, meaning any filter that can be expressed in one model can be expressed in the other. However, regarding implementation, they are very different: the tree model maps naturally to code for a stack machine, while CFG model maps naturally to a register machine.

BPF uses the CFG filter model since it has a significant performance advantage over the expression tree model. While the tree model may need to redundantly parse a packet many times, the CFG models "remembers" a parsed packet state in the graph. In order to reach a particular node, you know what paths you must have traversed, so a graph can be organized in a way that once a subexpression is evaluated, the value is used only at nodes that follow the original computation, so it does not need to be recomputed.

The BPF Pseudo-Machine The BPF machine abstraction consists of



(a) Tree model.

(b) CFG model.

Figure 2.25: Filter function representations [63].

an accumulator, an index register (x), a scratch memory store, and an implicit program counter. The operations on these elements can be categorized into the following groups:

- Load instructions: they copy a value into the accumulator or index register.
- Store instructions: they copy either the accumulator or index register into the scratch memory store.
- ALU instructions: they perform arithmetic or logic operations on the accumulator using the index register or a constant as an operand.
- Branch instruction: they alter the flow of control, based on comparison tests.
- Return instructions: they terminate the filter and indicate what portion of the packet to save.
- Miscellaneous instructions: comprise everything else.

The fixed length instruction format is defined as `opcode: 16 | jt:8 | jf:8 | k:32` where:

- The opcode field: indicates the instruction type and addressing modes.
- The jt and jf fields: are used by the conditional jump instructions and are the offests from the next instruction to the true and false targets.
- The k field: generic field used for various purposes.

Figure 2.26 shows the entire BPF instruction set, using the "assembler syntax" as a means of illustrating. The actual encodings are defined with C macros.

The following example code shows how packet filters can be expressed using the BPF instruction set. This filter accepts all IP packets.

opcodes	*addr modes*				
ldb	[k]			[x+k]	
ldh	[k]			[x+k]	
ld	#k	#len	M[k]	[k]	[x+k]
ldx	#k	#len	M[k]	4*([k]&0xf)	
st	M[k]				
stx	M[k]				
jmp	L				
jeq	#k, Lt, Lf				
jgt	#k, Lt, Lf				
jge	#k, Lt, Lf				
jset	#k, Lt, Lf				
add	#k			x	
sub	#k			x	
mul	#k			x	
div	#k			x	
and	#k			x	
or	#k			x	
lsh	#k			x	
rsh	#k			x	
ret	#k			a	
tax					
txa					

Figure 2.26: BPF instruction set [63].

```
        ldh     [12]
        jeq     #ETHERTYPE_IP, L1, L2
L1:     ret     #TRUE
L2:     ret     #0
```

The first instruction loads the Ethernet type field, then it compares it to the IP type. If the comparison fails, zero is returned and the packet is rejected. If it is successful, TRUE is returned and the packet is accepted (TRUE is some non-zero value that represents the number of bytes to save) [63].

2.4.1.2 eBPF

Although BPF is very useful for packet filtering, other areas can benefit from its ability to program the kernel. Many enhancements were made to BPF to make it a universal in-kernel virtual machine. This new version is called eBPF. Several aspects of the architecture were changed, with the addition of some important functionalities such as the ability to persist data between executions, and to share information between these and the user space, and the option of being able to call functions that run inside the kernel. Programs can also be changed, modified or reload at execution time. Function calls were also added to the architecture, in which parameters are passed through registers, just like in native hardware. This makes it possible to map an eBPF function

call to a hardware instruction, resulting in almost no overhead. That's to why cBPF instructions are internally translated to eBPF instructions, improving performance. In sum, eBPF provides an instruction set and an execution environment within the Linux kernel and can be used to modify packet processing, as well as allowing programming of network devices. In order to achieve this, a programmer develops an application in C restricted language and compiles it to eBPF [91].

The program shown in Figure 2.27 illustrates the structure of a simple eBPF program that drops all received packets, as soon as it arrives at the network interface.

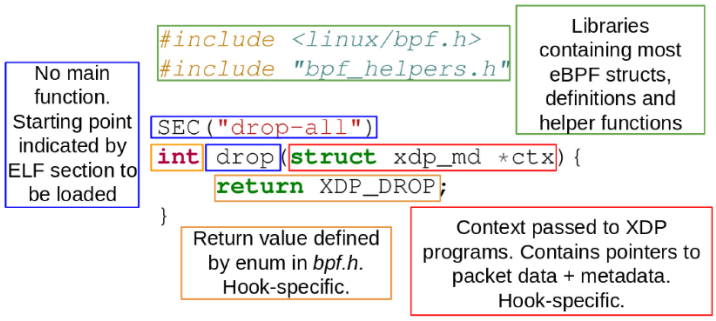


Figure 2.27: eBPF program that drops all packets [91].

One application of eBPF is The Case for Pluginized Routing Protocols [92]. Traditional routing protocols like BGP and OSPF/IS-IS have not been replaced by newer technologies such as SDN. While standardization ensures interoperability between routers from different vendors, it also slows down innovation since the process of standardizing protocols can take decades. The solution proposed to allow network operators to create custom extensions for the traditional protocols, was integrating a modified eBPF Virtual Machines (VM) into FRRouting[1]. This VM enables the execution of operator-supplied bytecode at various points within the FRRouting code, allowing for the implementation of extensions for both OSPF and BGP [80, 92].

2.4.1.3 uBPF

Last, there is the the user-space BPF (uBPF) Virtual Machine, which re-implements the virtual machine based on the eBPF kernel. While BPF was originally designed to allow safe execution of code in the kernel, the uBPF

[1] https://frrouting.org

project makes it possible to run BPF programs in user space. Therefore, the uBPF virtual machine can be easily integrated with applications that bypass the kernel (for example, DPDK and XDP) [76].

2.4.2 XDP

In computer networks, hooks are used to intercept packets before or during a system call. The Linux kernel exposes various hooks to which one can attach eBPF programs, enabling data gathering and personalized event handling. Two of these hooks are the XDP (eXpress Data Path) and the Traffic Control (TC), which together enable packet processing near the NIC, enabling the creation and development of multiple network applications.

Packets processed by the operating system are processed through various layers in the kernel; the Socket Layer, the TCP (Transmission Control Protocol) Stack, Netfilter, TC, XDP and the Network Interface, as shown in Figure 2.28. XDP is the lowest layer from Linux kernel network stack. It is only found

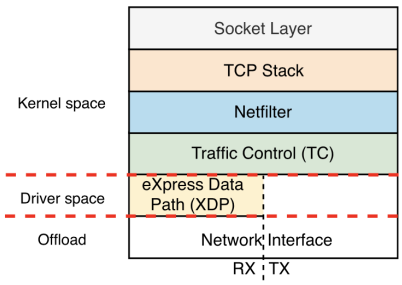


Figure 2.28: Linuk kernel network stack [91].

in the Receive (RX) path of a device's network driver and enables packet processing at the very beginning of the network stack, even before the operating system allocates memory. It exposes a hook to which eBPF programs can be attached. Programs can make arbitrary changes to arriving packets at these hooks and make quick decisions about them, avoiding the overhead imposed by the processing at the kernel. This makes XDP the hook with best performance in terms of speed for applications. After processing a packet, an XDP program returns an action, that represents the decision regarding what should be done to the packet after program exit. These actions include: dropping the packet (silently or raising an exception), allowing further processing by the kernel stack, transmit from the interface it came from and transmit from

another interface (another NIC, another CPU for further processing, or an AF_XDP socket for userspace processing) [91].

If the example shown in Figure 2.27 was saved in a `dropworld.c` file, that code can be compiled into an ELF object file using the `clang` compiler:

```
$ clang -target bpf -O2 -c dropworld.c -o dropworld.o
```

The `ip` tool can load the object file into the kernel, using the following command:

```
# ip -force link set dev [DEV] xdp obj dropworld.o sec .text
```

Since there is no section tag in the example code, the generated bytecode is located in the default section (.text) of the ELF object file. The `-force` parameter indicates that the program should be loaded even if there's another program loaded on that interface, therefore getting replaced. The [DEV] parameter should be the corresponding interface [91].

2.4.2.1 Traffic Control Hook

XDP can only handle ingress traffic, so in order to process egress traffic, the closest layer to the NIC that has access to the whole Ethernet frame is the Traffic Control (TC) layer. On Linux, this layer is in charge of carrying out traffic control policies. In it, the network administrator can add filters to modify or drop packets as well as configure various queuing disciplines (*qdisc*) for the various packet queues that are present in the system. The TC has a special queuing discipline called `clsact`. It exposes a hook that enables eBPF programs to specify queue processing activities. The configured eBPF software receives pointers to the packet to be processed as part of its input context. This structure is a UAPI (Userspace API) for specific fields that the program is permitted to access from the kernel's socket buffer internal data structure. The kernel has already processed the packet at the TC level to extract protocol metadata, which accounts for the greater context information supplied to the eBPF program compared to the XDP case. The input packet may be modified while the program is running, and the return value tells TC what should be done about it. One can load a program on the TC hook using the `tc` tools, available in the `iproute2` package. The following command shows how to create the `clsact` *qdisc* and load and eBPF program to process packets on interface `eth0`, where `<direction>` indicates which direction the

program should follow (ingress or egress), `<ebpf-obj>` and `<section>` are the names of the file containing the compiled eBPF code and the section to load the program, respectively [91].

```
# tc qdisc add dev eth0 clsact
# tc filter add dev eth0 <direction> bpf da obj <ebpf-obj> sec <section>
```

Similar uses for both hooks (XDP and TC) include DDoS mitigation, tunneling, and managing link layer information. However, XDP can achieve better throughput values than programs on TC since it executes before any socket buffer allocation occurs. However, by being the lowest layer on TX, TC can take advantage of additional parsed data and run eBPF programs for both incoming and outgoing traffic [91].

2.4.3 DPDK

DPDK is an optimized data plane open source software solution created by Intel for its multi-core processors. The goal is to give programmers a straightforward and comprehensive framework for quick packet processing. According to their requirements, users can utilize this method to build prototypes or add their own protocol to the stack [7]. In sum, DPDK is a user space library that is in charge of giving functions that enables to intercept packets before they pass through the kernel and process them in accordance with the programmer demands, hence accelerating processing. All resources must be allocated before calling data plane programs in the DPDK run-to-completion model for packet processing. These data plane applications are running as execution units in logical processing cores, or lcores. Polling is used to access all devices in order to avoid using interrupts due to the performance overhead they cause [34]. In figure 2.29 traditional packet processing can be seen on the left, whereas packet processing with DPDK can be seen on the right. In the latter, it can be seen that all interactions with the NIC are done through special drivers and libraries [93]. DPDK can be used to implement the host stack needed to be able to comprehend a new user-defined protocol.

The framework generates a collection of libraries tailored to specific environments by creating an Environment Abstraction Layer (EAL). This layer can be customized for different modes of Intel architecture (32-bit or 64-bit), Linux user space compilers, or particular platforms. These environments are built using make files and configuration files. Once the EAL library is gener-

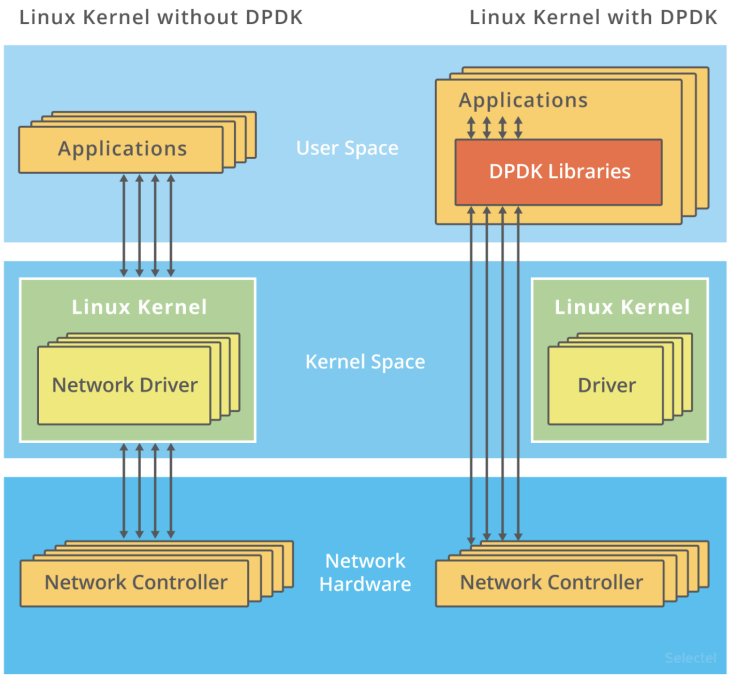


Figure 2.29: Traditional packet processing vs packet processing with DPDK [93].

ated, users can link it with their own applications. In addition to the EAL, other libraries like Hash, Longest Prefix Match (LPM), and rings are provided. The EAL manages access to low-level resources like hardware and memory. It offers a standardized interface that abstracts the details of the underlying environment from applications and libraries. The initialization routine is in charge of determining how to allocate these resources, including memory, PCI devices, timers, consoles, and other components [34].

2.4.4 Remarks

These technologies, while not solutions for network devices themselves, can be utilized in certain use cases for efficient custom packet processing. They can be deployed on servers, hosts, or network devices that support Linux.

In particular, BPF relies on the Linux kernel, which can be either an advantage or a disadvantage since it doesn't require specialized hardware. However, it is limited in the logic it can implement compared to P4. In the case of XDP, it also shares the dependency on the Linux kernel and device limitations. DPDK allows bypassing the kernel stack and is compatible with various

devices such as CPUs, NICs, SmartNICs, among others[1]. However, it is significantly more complex than P4 when it comes to writing programs, but it is possible to access the whole packet with DPDK (including payload).

The technology to be used largely depends on the use case. When ultra-customized processing on network devices is required, P4 can be very useful. On the other hand, if packet processing is done on a traditional server, using technologies like DPDK or BPF may be beneficial due to their fast packet processing capabilities. It can be said that these technologies serve as alternatives to P4.

2.5 SmartNICs

In 1965, Gordon Moore predicted that the number of transistors on a chip would double annually, later revising this to every two years [66]. In 1974, Robert Dennard observed that power density remained constant as transistor size decreased, allowing for enhanced performance of integrated circuits by packing more transistors per chip. However, by 2003, Dennard scaling reached its limits, reducing the rate of processor performance improvement. This led to the development of multi-core processors, which improved performance but have also faced limitations due to Amdahl's Law [2], which highlights constraints of parallelism in computing, as applications also have tasks that must be executed sequentially.

In today's world, most data reaches computing locations as network packets. The traditional link between networks and hosts is the NIC. Historically, NICs were basic hardware devices that received packets from the network and stored them in the host's memory, where they would wait for the general-purpose processor to handle them. While this approach worked well for many years, it now faces several challenges in modern environments. With the conclusion of Moore's Law and Dennard scaling, adding more processing power to handle increasing traffic is no longer feasible. A substantial portion of processor tasks now involve infrastructure-related functions (rather than user application-related) like TCP/IP operations, encryption, and compression, which consume processing cycles that could otherwise be dedicated to user applications. Traditional software approaches to managing these packet-

[1] https://core.dpdk.org/supported/

related tasks are inefficient in terms of throughput, latency, and energy use. While hardware advancements previously compensated for these inefficiencies, today's solutions can no longer depend on future processor performance improvements. Moreover, the rapid increase in network traffic, alongside significant enhancements in the physical layer and bandwidth capacity, has outpaced processors' ability to keep up, widening the gap between processing power and network speed.

With the end of Dennard scaling and no further increases in the energy budget, many believe that the best way to improve energy efficiency, performance, and cost is by using domain-specific processors instead of power-hungry general-purpose processors. SmartNICs are seen as a groundbreaking technology designed to tackle these challenges by integrating various domain-specific processors that are specialized for specific infrastructure tasks, such as compression/decompression, programmable pipelines, and encryption/decryption. These SmartNICs also include general-purpose processors to manage the system, assist the domain-specific processors, and allow users to run control-plane applications. In this context, domain-specific processors are often referred to as accelerators or engines. The development of domain-specific processors has already proven successful in areas like graphics with Graphics Processing Unit (GPU)s in the 2000s, machine learning with Tensor Processing Unit (TPU)s in the mid-2010s, networking with Network Processing Unit (NPU)s following the PISA model in the late 2010s, and genomic analysis in 2018.

The growing adoption of SmartNICs is evident across the global IT landscape. Hyperscalers like Google, Amazon, and Microsoft are developing their own SmartNICs to handle infrastructure tasks and enhance performance and revenue. Companies such as Intel, NVIDIA, and AMD are focusing on creating SmartNICs for a wide market, offering Systems on a Chip (SoCs) with programmable, domain-specific processors for security, networking, storage, and telemetry. Projects like VMware's Monterey are redefining cloud architectures by incorporating SmartNICs to manage storage, network, and security services, significantly improving performance and freeing up processor cycles for user applications. Research and education networks (RENs), like the Energy Sciences Network (ESnet), are upgrading their infrastructure with SmartNICs to support data-intensive science. Software vendors are also leveraging SmartNICs; for instance, VMware's ESXi, vCenter, and NSX components for virtualizing High Performance Computing (HPC) environments can now be

efficiently offloaded onto SmartNICs. Palo Alto Networks introduced the "Intelligent Traffic Offload" service, offloading firewall functions to SmartNICs, and Juniper Networks has enabled its virtual router/firewall to do the same. Telecommunication operators are increasingly shifting core services to SmartNICs, and workloads in serverless and edge computing, including Machine Learning (ML) training and inference, can be accelerated using SmartNICs. Testbeds such as FABRIC [5] and the GÉANT project[1], used globally for fundamental research, rely on SmartNICs and other programmable devices to allow experimenters to program data path behavior and process network traffic at line rate in innovative ways [53].

In order to achieve the high parallelism required to achieve high-speed network packet processing, current SmartNICs rely on multiple hardware architectures including (i) ASIC (e.g., Netronome NFP); (ii) System-on-Chip (e.g., Nvidia BlueField); and (iii) FPGA (e.g., Xilinx Alveo).

SmartNICs are often referred to as DPUs or Infraestructure Processing Unit (IPU)s. Although there have been efforts, primarily marketing-driven, to differentiate these terms technically, a widely accepted distinction is not established [61]. FPGA stands for Field Programmable Gate Array. FPGAs are semiconductor integrated circuits where much of the device's functionality can be modified; this can be done by the design engineer, during the Printed Circuit Board (PCB) assembly process, or even after a product has been deployed. Changes are made by altering which electrical inputs and outputs are connected, which logic gates and flip-flops are implemented, and how these gates are interconnected. An FPGA consists of a grid of configurable logic, known as Adaptive Logic Modules (ALMs), and specialized blocks such as Digital Signal Processing (DSP) blocks and Random Access Memory (RAM) blocks. These programmable blocks are combined through configurable routing interconnections to implement complete digital circuits[1].

2.5.1 Evolution of SmartNICs

There are three main generations of NICs: traditional NICs, offload NICs, and SmartNICs. Figure 2.30 illustrates a simplified diagram of the three NICs.

[1] https://geant.org

[1] https://www.intel.la/content/www/xl/es/support/programmable/support-resources/fpga-training/getting-started.html

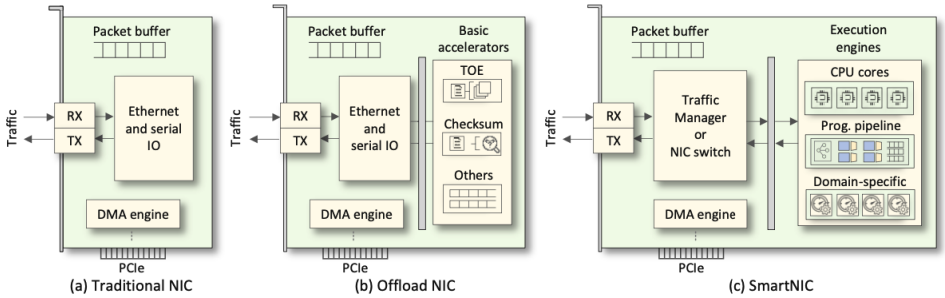


Figure 2.30: Key functional components of (a) traditional NICs, (b) offload NICs, and (c) SmartNICs.

2.5.1.1 Traditional NIC

Traditional NICs (Figure 2.30 (a)) are devices that handle basic services at the physical and data-link layers. These services include tasks like serializing/deserializing frames, managing link access, and providing error detection. These functions are typically carried out by a fixed-function component on a specialized chip within the NIC. On the transmission side, this component takes a datagram from the host, encapsulates it in a link-layer frame, and transmits the frame through the communication link according to the link-access protocol. On the receiving side, the component receives the frame and forwards it to the host via a Peripheral Component Interconnect Express (PCIe) card [53].

2.5.1.2 Offload NIC

Offload NICs (Figure 2.30 (b)) integrate hardware, such as ASICs and/or FPGAs, to handle basic "infrastructure" functions (tasks that facilitate data movement to the host and do not involve application data) that were traditionally managed by the host CPU. The aim is to free up the host CPU's processing power for user applications rather than infrastructure tasks. Examples of these functions include:

- Basic packet processing, such as parsing and reassembling IP datagrams, computing IP checksums, and encapsulating and de-encapsulating TCP segments.
- Managing TCP connections on the NIC, including connection establishment, checksum and sequence number calculations, TCP Offload Engine (TOE) functions, sliding window calculations for segment acknowledg-

ment, and congestion control.

- Additional functions that modify TCP/IP header fields to perform basic filtering and traffic classification.

Offload NICs enable users to perform pre-programmed functions on the NIC but do not allow for the creation and execution of custom applications directly on the NIC. Even with complete transport layer offload, application protocols must still be implemented on the host processor [53].

2.5.1.3 SmartNIC

The definition of a SmartNIC is not universally agreed upon. One can consider a SmartNIC being a programmable NIC. Historically, NICs that handled tasks beyond basic packet processing were termed SmartNICs. The term SmartNIC can be used to refer to the latest generation of NICs, which are also known as SoC SmartNICs, IPUs, DPU, and Auxiliary Processing Units (xPUs).

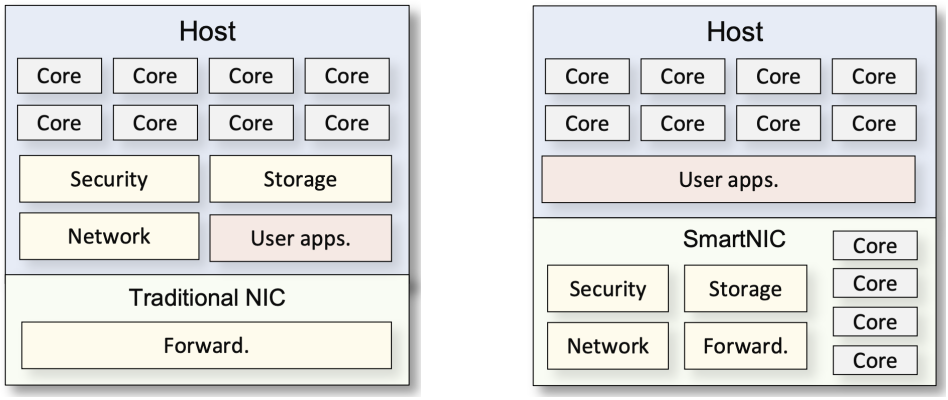
SmartNICs offer varying degrees of programmability. Some vendors enable the complete rewriting of the hardware description, as seen with FPGA-based SmartNICs, while others allow the offloading of only specific networking tasks to computing units, typical of SoC-based SmartNICs. To accommodate this level of programmability, SmartNICs often depend on diverse hardware platforms and programming languages, such as P4, Micro-C, and VHDL/Verilog. However, programming, debugging, and operating SmartNICs continue to be challenging tasks [61].

Figure 2.30 presents a simplified diagram of a SmartNIC. The SmartNIC includes a Traffic Manager (TM) or a NIC switch that handles Quality of Service (QoS) and directs traffic to the NIC execution engines. These engines consist of a combination of processors specialized in custom packet processing and other domain-specific tasks. Some SmartNICs, like NVIDIA's BlueField-2 DPU[1], use a multi-core CPU for custom packet processing, while others, like AMD's Pensando DSC[2], utilize embedded flow engines with a P4 programmable ASIC pipeline. Additionally, some SmartNICs, such as AMD's Xilinx SN1000[3], employ FPGAs for custom packet processing. The domain-specific processors are designed for high-performance and energy-efficient processing of specific tasks,

[1] https://network.nvidia.com/files/doc-2020/pb-bluefield-2-dpu.pdf

[2] https://www.amd.com/system/files/documents/pensando-dsc-200-product-brief.pdf

[3] https://docs.amd.com/v/u/en-US/ds989-sn1000



(a) Traditional NIC.

(b) SmartNIC.

Figure 2.31: In a deployment using a traditional NIC (a), the host CPU cores handle both infrastructure functions and user applications. However, with SmartNICs (b), the host CPU cores are dedicated entirely to running user applications, while the SmartNIC's CPU cores work alongside other accelerators to manage the infrastructure functions [53].

such as cryptography. The execution engines typically feature a memory hierarchy that includes an L1 cache, scratchpad, L2 cache, and DRAM.

SmartNICs also have general-purpose CPU cores for executing control plane functions, enabling them to operate independently with their own operating system, separate from the host system. The programmable components of a SmartNIC allow it to handle infrastructure functions without relying on the host CPU. As shown in 2.31, in a deployment with a traditional NIC (a), the host CPU cores manage both infrastructure functions (such as network, security, and storage) and user applications. In contrast, with SmartNICs (b), the host CPU cores focus solely on running user applications, while the SmartNIC's CPU cores, along with other domain-specific accelerators, manage the infrastructure functions.

Custom Packet Processing: SmartNICs allow developers to create custom packet processing routines on their execution engines, which can be implemented using CPU cores, FPGAs, or programmable ASIC pipelines. Regardless of the hardware architecture employed, the packet processing engines of SmartNICs typically include a programmable parser, a programmable match-action pipeline, and a programmable deparser—components that closely resemble those in the PISA architecture. Although different vendors have their own approaches to programming these pipelines, there is a shared industry goal to make them P4-programmable. P4, initially designed as a domain-

specific language for programmable data plane switches, has become popular for programming packet data paths due to its simplicity and flexibility.

Domain-specific Packet Processing: Infrastructure tasks can generally be divided into network functions, security functions, and storage functions. Given the repetitive nature of these tasks, it is efficient to hardcode them into hardware. For example, hardware-based crypto processors, which have been in use for some time, are domain-specific processors integrated into SmartNICs. Other examples of domain-specific processors include those used for regular expression (RegEx) tasks in Deep Packet Inspection (DPI), Non-Volatile Memory Host Controller over Fabrics (NVMe-oF) for remote storage, data compression, data deduplication, and Remote Direct Memory Access (RDMA) [53].

Control Plane and Management: SmartNICs include CPU cores dedicated to running control plane functions and managing the SmartNIC. These CPU cores can also be utilized to implement functions that cannot be accommodated within the ASIC or FPGA execution engines. Incorporating CPU cores within a SmartNIC offers several advantages [53]:

- SmartNIC CPU cores can handle specific infrastructure functions, such as key distribution for TLS sessions, reducing the load on the host CPU and allowing it to focus on user applications.
- Running infrastructure functions on SmartNIC CPU cores is more efficient since they are isolated from the compute-intensive workloads of user applications.
- Security is enhanced by completely isolating infrastructure functions from the host system.
- While ASIC/FPGA engines are limited in handling complex operations due to the need for high-speed packet processing, the SmartNIC's CPU cores can manage these tasks, though this may increase latency.

2.5.1.4 SmartNIC benefits

SmartNICs provide a variety of features and advantages that address modern network challenges:

- Infrastructure offloads: Data center infrastructure tasks currently consume up to 30% of processing capacity, a phenomenon known as the Data Center Tax. Offloading these tasks to SmartNICs frees up this

30%, making it available for user applications, which can significantly boost revenue for cloud providers. This is a key reason why hyperscalers (large companies that provide cloud computing and data management services at a massive scale[1]) are among the first to adopt this technology.

- Application acceleration: With hardware-based accelerators, SmartNICs offer better performance per watt than host-based applications, resulting in lower latency and increased overall efficiency.
- Agility and reprogrammability: Developing new silicon is a lengthy, costly process that requires extensive testing. By the time this cycle is complete, rapid technological advancements may render the hardware outdated. SmartNICs address this issue by providing programmable components, enabling adaptability and timely updates to meet evolving technological demands.
- Security isolation: SmartNICs improve security by isolating infrastructure function execution from the server's execution environment.

2.5.2 SmartNICs development tools and frameworks

Figure 2.32 illustrates a taxonomy that classifies the development tools and frameworks used for programming SmartNICs, based on the specific component within the SmartNIC being targeted.

2.5.2.1 Programmable pipeline

Packet processing logic is typically implemented using ASICs or FPGAs, and the development of offloaded applications depends on the hardware architecture and the vendor's Software Development Kits (SDKs) [53].

- P4 Language: Initially designed to program the data plane of PISA-based switches, the P4 language has proven versatile for programming data planes in various packet processing devices. Although different vendors use varied programming models, there is a unified goal to make their pipelines programmable in P4.
- FPGA Programming: FPGAs consist of configurable logic blocks and programmable interconnects, enabling users to tailor the chip's functionality to their application needs. FPGA-based SmartNICs use similar

[1] https://www.redhat.com/en/topics/cloud-computing/what-is-a-hyperscaler

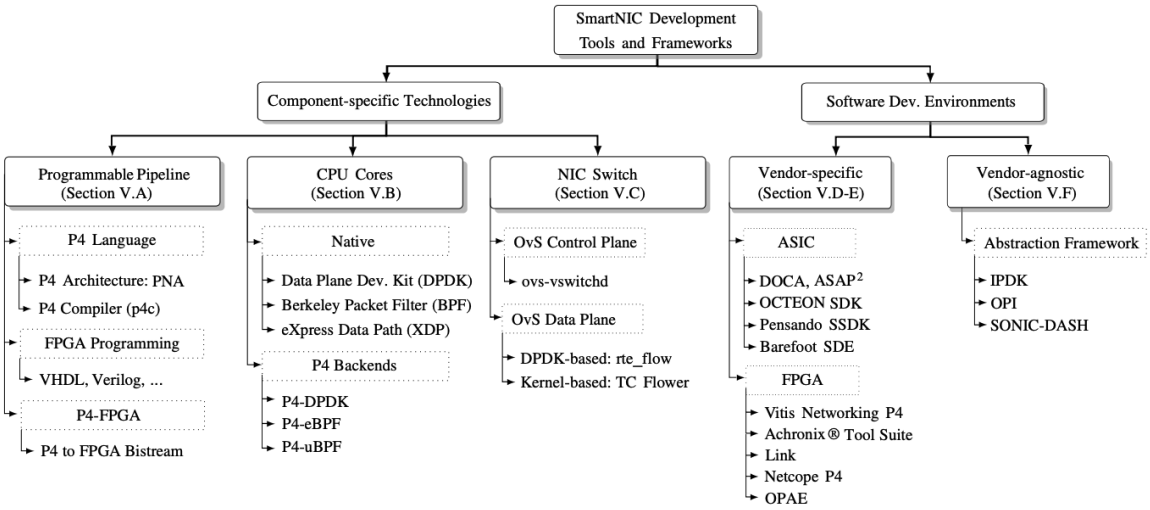


Figure 2.32: Taxonomy of SmartNIC development tools and frameworks, categorized by component-specific technologies and software development environments [53].

programming workflows to traditional FPGAs. This means that development tools, methodologies, and languages used for traditional FPGA programming, such as Integrated Development Environments (IDEs) and compilers that convert Hardware Description Languages (HDLs) such as VHDL and Verilog into FPGA configuration files, are also applicable to SmartNICs.

- P4-FPGA: Programming FPGAs with VHDL or Verilog can be complex and time-consuming, especially for beginners. To simplify this, frameworks have been developed to translate P4 code into FPGA bitstreams. P4, being a high-level and user-friendly language for programming datapaths, provides a faster and more efficient alternative for FPGA programming. However, challenges remain in creating a compiler that can effectively translate P4 code into VHDL or Verilog. Issues include the use of low-level libraries in FPGA programming that are not portable across devices and the difficulty of generating an efficient implementation from diverse P4 programs and varying architectural trade-offs. The community is actively working on developing P4 FPGA compilers to address these challenges.

2.5.2.2 CPU Cores

User applications run on CPU cores, whether on the SmartNIC's cores or the host's cores. The process for an application to handle a packet from the NIC are: when a packet arrives, the NIC generates an interrupt to notify the operating system of the packet's memory location. The OS then moves the packet to the network stack, which triggers system calls from the OS kernel to forward the packet to the appropriate user-level application. These procedures introduce overheads that significantly reduce bandwidth throughput. Modern NICs ports linerate now exceed 200Gbps, and as NIC speeds increase, the time available for processing each packet decreases. For example, at 200Gbps, the interval between consecutive 1500-byte packets is just 60 nanoseconds (ns), making the standard network stack insufficient to handle such high traffic rates effectively [53].

- Data Plane Development Kit (DPDK) [34]: DPDK is a suite of libraries and drivers designed to boost packet processing efficiency by bypassing the kernel and operating within user space. Instead of linking NIC ports to the kernel driver, DPDK uses a compatible driver that disconnects them from the kernel. Unlike traditional packet processing, which relies on kernel stack interrupts, the DPDK driver functions as a Poll Mode Driver (PMD), continually polling for incoming packets. This approach, along with kernel bypass, significantly improves packet processing performance. DPDK provides APIs for use in C programs. Initially developed by Intel, DPDK is now an open-source project with an expanding community. It supports all major CPU and NIC architectures from various vendors.
- eXpress Data Path (XDP)[1] and extended Berkeley Packet Filter (eBPF) [22]: While DPDK improves performance by bypassing the kernel, this results in the loss of networking functionalities provided by the kernel, requiring user-space applications to re-implement these features. XDP addresses this problem by integrating eBPF programs into the kernel's network stack. XDP introduces an early hook in the RX (receive) path, specifically within the NIC driver after interrupt handling. This hook allows the execution of a user-defined eBPF program, enabling decisions to be made before the Linux networking stack processes the

[1] https://www.iovisor.org/technology/xdp

packet.

- P4 Backends: Developing P4 programs is typically seen as easier than writing DPDK or BPF/XDP code. As a result, there have been initiatives to convert P4 programs into these other code formats. The P4 compiler (p4c) includes backends specifically for generating DPDK, BPF/XDP, and Userspace BPF (uBPF) code.

2.5.2.3 NIC Switch

The NIC switch manages QoS traffic control and directs traffic to the NIC execution engines. SmartNICs often use the Open vSwitch (OvS) specifications to implement this switch. OvS, initially designed to facilitate communication between VMs, comprises two main components: the control plane (ovs-switchd) and the data plane, also referred to as the datapath.

The OvS control plane is traditionally executed on the host in the userspace. With SmartNICs, the OvS control plane is executed on the CPU cores of the SmartNICs.

The standard OvS switch's datapath is situated in the kernel, which strains CPU resources and degrades the performance. To address these issues, many SmartNICs offer support for offloading OvS into their NIC switch. When this feature is utilized, the OvS datapath is moved to the hardware, resulting in superior performance compared to the software-based versions [53].

2.5.2.4 Vendor specific SDKs targeting ASIC-based and FPGA-based SmartNICs / Vendor agnostic

The following SDKs are proprietary and target ASIC-based SmartNICs. NVIDIA's DOCA: The Data Center-on-a-Chip Architecture (DOCA)[1] is a software development framework created by NVIDIA for its BlueField DPUs[2]. This framework includes a range of components such as libraries, service agents, and reference applications. Applications built with DOCA are developed in C and support DPDK, providing developers with access to all DPDK APIs for efficient packet processing. DOCA also features its own libraries to facilitate interactions with the SmartNIC's components. Other ones include:

[1] https://developer.nvidia.com/blog/programming-the-entire-data-center-infrastructure-with
[2] https://network.nvidia.com/files/doc-2020/pb-bluefield-2-dpu.pdf

OCTEON SDK[3], AMD Pensando SSDK[4], Intel P4 Studio (formerly Barefoot SDE)[5]. These SDKs facilitate programming, interaction with SmartNIC components, access to resources, among other functionalities. In general, they simplify the compilation and testing of programs, sometimes even including the ability to compile without the hardware and test using a provided simulator. Additionally, they often provide use cases and a Graphical User Interface (GUI) that, for example, in the case of Intel P4 Studio, offers a comprehensive insight into resource utilization, such as tables, hash, TCAM/SRAM, etc. In some cases, acquiring these SDKs requires signing an Non-Disclose Agreement (NDA) [53].

The following SDKs are proprietary and target FPGA-based SmartNICs: Vitis Networking P4[1], Intel P4 Suite for FPGA[2], Achronix Tool Suite[3], Napatech Link Toolkit[4] and Open FPGA Stack (OFS) with Open Programmable Acceleration Engine (OPAE) SDK[5]. Similarly, these SDKs simplify the process of creating programs and their compilation, including pre-built functions and in-system debugging, among other features [53].

The following are vendor agnostic abstraction frameworks: Open Programmable Infrastructure (OPI)[6], Infrastructure Programmer Development Kit (IPDK)[7], SONIC-DASH[8]. Instead of relying on vendor-specific SDKs, developers can use vendor-agnostic SDKs that can be used in the same way, abstracting the complexities associated with vendor-specific SDKs. Some functionalities may not be implemented, so it is still possible to complement them with vendor-provided functions. All of these SDKs are open source, focusing on various objectives such as infrastructure offload, management tasks, cloud

[3] https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-octeon-tx2-sdk-solutions-brief.pdf

[4] https://community.amd.com/t5/corporate/amd-pensando-dpu-software/ba-p/630282

[5] https://www.intel.la/content/www/xl/es/products/details/network-io/intelligent-fabric-processors/p4-studio.html

[1] https://www.xilinx.com/products/intellectual-property/ef-di-vitisnetp4.html

[2] https://www.intel.com/content/www/us/en/software/programmable/p4-suite-fpga/overview.html

[3] https://www.achronix.com/product/fpga-design-tools-achronix

[4] https://www.napatech.com/products/link-capture-software/

[5] https://www.intel.com/content/www/us/en/products/details/fpga/platforms/open-fpga-stack.html

[6] https://opiproject.org

[7] https://ipdk.io

[8] https://plvision.eu/offerings/sonic-dash-api-implementation

services, among others. Some of these platforms can also be used for other devices like switches or CPUs [53].

2.5.3 DPUs/GPUs/CPUs/FPGAs

Other devices that enable parallel data processing, including networking, machine learning, and general computation, and also allow offloading tasks from the CPU, include GPUs, DPUs, TPUs, and FPGAs.

Integrated circuits (ICs) are commonly referred to as "chips". They implement very small electronic circuits on a silicon substrate. CPUs, GPUs, and FPGAs are all types of integrated circuits[1].

A DPU is a new type of programmable processor that integrates three essential components. First, it features a system on a chip (SoC) that includes an industry-standard, high-performance, software-programmable, multi-core CPU, usually based on the widely used ARM architecture, which is closely integrated with other SoC elements. Second, it includes a high-performance network interface capable of parsing, processing, and efficiently transferring data at line rate—the speed of the broader network—to GPUs and CPUs. Third, it offers a comprehensive set of flexible and programmable acceleration engines designed to offload and enhance application performance in areas such as AI and machine learning, zero-trust security, telecommunications, and storage. These DPU capabilities are crucial for establishing an isolated, bare-metal, cloud-native computing platform which can contribute to substantially modifying the future of cloud-scale computing.

The DPU can function as an independent embedded processor, though it is more commonly integrated into a SmartNIC, serving as a key component in next-generation servers[2]. A DPU-based SmartNIC is a NIC that offloads processing tasks that the system CPU would normally handle. Using its own onboard processor, the DPU-based SmartNIC may be able to perform any combination of encryption/decryption, firewall, TCP/IP, and HTTP processing. SmartNICs are ideally suited for high-traffic web servers[3].

GPU-accelerated computing is the use of a GPU alongside a CPU to speed up the performance of applications in deep learning, analytics, and engineer-

[1] https://www.intel.la/content/www/xl/es/support/programmable/support-resources/fpga-training/getting-started.html

[2] https://blogs.nvidia.com/blog/whats-a-dpu-data-processing-unit/

[3] https://developer.nvidia.com/blog/choosing-the-best-dpu-based-smartnic/

ing. GPUs began as specialized ASICs designed for specific tasks, such as accelerating 3D rendering. Over time, these fixed-function engines have become more programmable and flexible. While their primary function remains focused on graphics and increasingly realistic visual elements in games, GPUs have evolved into more general-purpose parallel processors, handling a growing range of applications, including AI[4],[5]. The GPU excels at parallel processing and efficiently handles complex mathematical tasks that general-purpose CPUs struggle with without the need for emulation. These capabilities make GPUs particularly valuable for visualization tasks, such as computer games, math-intensive applications, and 3D rendering tools like AutoCAD. Since GPUs lack basic instructions, they are typically paired with CPUs within the same computer system. However, a GPU is not merely a CPU with additional instructions; it represents a fundamentally different approach to addressing specific computing challenges. The GPU's limited range of functions allows for much smaller cores, but its highly parallel architecture enables thousands of cores to tackle massive parallel computing tasks and achieve high data throughput. Despite this, GPUs are not well-suited for multitasking and generally have restricted memory access. GPUs are particularly effective for handling demanding machine learning (ML) models.

A TPU is a specialized application-specific integrated circuit (ASIC) created to expedite the high-volume mathematical and logical computations commonly associated with machine learning (ML) tasks. Unlike general-purpose processors, a TPU is more similar to an ASIC, offering a limited set of mathematical functions, mainly focused on matrix processing specifically for ML applications. It is known for delivering exceptional throughput and parallelism, similar to GPUs, but optimized and pushed to the limits in its design. TPUs are specially useful for machine learning, data analytics, edge computing and cloud computing[1].

All of these technologies enable task offloading, and which one to use will depend on the nature of the application. GPUs, DPUs and TPUs are more oriented towards ML, graphics, and similar applications. On the other hand, FPGAs are very useful for networking tasks.

[4] https://www.nvidia.com/es-la/drivers/what-is-gpu-computing/

[5] https://www.intel.la/content/www/xl/es/products/docs/processors/cpu-vs-gpu.html

[1] https://www.techtarget.com/whatis/definition/tensor-processing-unit-TPU

2.6 Applications

There are numerous applications where data plane programming can be useful due to the capabilities that it provides. These include QoS, Monitoring, Traffic Management and Congestion Control, Routing and Forwarding, Network Security, among others. In particular, there is a direct application in network monitoring. This can range from simple monitoring with statistics deployment, including stateful processing, to monitoring with security actions.

SmartNICs enhance the performance of a wide range of infrastructure applications, which can be broadly categorized into security, networking, and storage functions. They also accelerate various computational workloads, such as AI/ML inference and training, caching (like key-value stores), transaction processing, serverless functions, and more.

Below are a series of applications for both SmartNIC offload (based on the taxonomy presented at [53]) and data plane programmability, which often intersect in various ways. Each of these applications showcases how offloading to SmartNICs and leveraging data plane programmability can enhance network performance, security, and flexibility.

2.6.1 Security

The nature of data center traffic has significantly evolved with the advent of cloud-hosted applications and microservices. Traditionally, traffic patterns were dominated by North-South (NS) flows, which move between internal and external devices and are safeguarded by perimeter security appliances like firewalls, as shown in 2.33(a). However, over the past decade, there has been a shift towards East-West (EW) flows, which occur between devices within the data center and now constitute up to 80% of total data center traffic. Unlike North-South traffic, East-West traffic was largely unprotected. To address this, a common solution was to route EW traffic through a centralized security appliance for inspection, as depicted in 2.33(b). This approach causes the traffic to pass through intermediary devices, such as switches, twice, thereby increasing both network load and latency for the two hosts involved. This has led to the rise of Zero Trust and microsegmentation architectures, which focus on decentralizing security functions and bringing them closer to the resources that need protection. Data centers and cloud providers have transitioned to using software-based security functions to safeguard East-West traffic, as shown

in 2.33(c). While this approach offers benefits in terms of deployment ease and cost-effectiveness, it also has certain limitations:

- Performance: Packets must pass through the regular network stack to be processed by security functions on general-purpose CPUs, increasing latency and reducing throughput.
- Scalability: CPU cores often struggle to handle high traffic inspection rates, especially without software accelerators like DPDK, leading to increased packet drop rates.
- Isolation: All traffic, including potentially malicious traffic, is sent to the host, which poses security risks due to a lack of isolation.
- CPU usage: Security functions consume a significant portion of CPU processing power, particularly during high traffic periods, causing performance bottlenecks and service degradation for end-user applications.

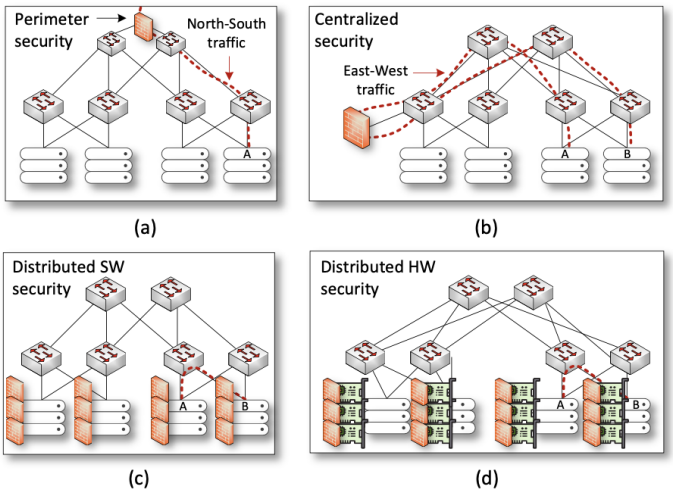


Figure 2.33: (a) Perimeter-based security: The appliance inspects only North-South traffic. (b) Centralized security: The appliance is capable of inspecting East-West traffic, but it results in significant bandwidth overhead. (c) Distributed software firewall: Software-based appliances are deployed on servers to inspect East-West traffic, but their performance is limited. (d) Distributed hardware firewall: The appliances are offloaded to SmartNICs on the servers, allowing high-performance inspection of East-West traffic [53].

To address these challenges, SmartNICs have been utilized to offload security functions from general-purpose CPUs, as shown in 2.33(d). Specifically, SmartNICs have been employed to offload firewall functions, IDS/IPS, DPI, and encryption for both data in motion and data at rest [53].

2.6.1.1 Firewall

A firewall monitors network traffic, both incoming and outgoing, and allows or blocks packets based on a set of predefined rules. Firewalls typically operate up to layer 4, handling basic Access Control List (ACL) tasks. This enables them to match traffic against network layer information (such as source and destination IP addresses) and transport layer information (like source and destination port numbers).

Software-based firewalls are widely used, particularly in cloud environments. They are often implemented alongside a virtual switch (e.g., OvS). In these setups, the traffic is inspected using the CPU cores of the host where the firewall is deployed, which can lead to reduced performance and increased CPU resource consumption.

As SmartNICs come with a programmable pipeline or an embedded switch, this allows the definition of match-action rules. This capability enables the direct implementation of firewalls with basic ACLs in hardware, operating at line rate. While developers can build firewall functionality from the ground up, this approach requires implementing several features, such as connection tracking for stateful inspection, flow caching, and aging.

Alternatively, the hardware-offloaded switch on SmartNICs can be used to implement firewall functions. In this approach, switch rules can be offloaded to the hardware transparently, and the developer only needs to specify traffic allow/block rules. The connection tracking feature of the switch can also be used to enable stateful inspection. For instance, VMware allows the offloading of firewall functions from its NSX distributed switch to the SmartNIC, specifically for L2-L4 inspection and firewalling [53].

In the case of pure data plane programmability, it enables customized packet processing, including the insertion of dynamic rules that can be modified at runtime, which is clearly useful in a firewall. Additionally, it allows for latency reduction by executing these programs on hardware devices rather than software, as is typically done, enabling inline packet inspection and filtering.

2.6.1.2 Intrusion Detection/Prevention System

IDS and Intrusion Prevention System (IPS) are cybersecurity tools aimed at protecting networks and hosts from unauthorized access, malicious actions, and security threats. An IDS observes and analyzes network or system events to

detect unusual patterns or anomalies, offering real-time alerts or logs for deeper analysis. An IPS not only detects these activities but also takes immediate action to prevent or block unauthorized actions in real-time.

IDS and IPS are typically run on the host's general-purpose CPUs, but SmartNICs have started offloading these functions to boost data processing. These include offloading the bypass function, Deep Packet Inspection (DPI), custom functions, IPSec offload, TLS offload and more. In the case of pure data plane programmability, it once again allows for the creation of dynamic rules and policies that can be changed at runtime, as well as customized traffic inspection. As will be discussed later, it also enables the integration of Machine Learning models for anomaly detection. Additionally, it supports inline filtering or anomaly detection, contributing to the decentralization of network tasks [53].

Other applications include Port Knocking, DDoS attack mitigation, connection security, between others [46].

2.6.2 Network Offloads

SDN and NFV are transformative technologies that have dramatically changed how networks are designed, deployed, and managed. Virtual switches are essential for providing the flexibility, scalability, and efficiency required by modern networks, particularly in connecting VMs. However, implementing networking functions as NFVs on servers places a significant burden on the CPU, especially in high-traffic networks. To address this, SmartNICs have recently been employed to offload these network functions from general-purpose CPUs, taking on tasks such as switching/routing, tunneling, measurement, telemetry, and more.

2.6.2.1 Switching

Virtual switching was developed to address the need for hypervisors to efficiently connect VMs to external networks. Initially, virtual switches operated within the hypervisor as software-based solutions. However, this method was demanding on CPU resources, leading to reduced system performance and inefficient bandwidth use. Software switches not only handle traditional layer 2 switching and layer 3 routing but also enable rule matching across different packet fields and support a variety of actions on packets, such as forwarding,

dropping, and marking.

SmartNICs use hardware components like lookups and ALUs to perform match-action functions for packet switching. Instead of creating new switching functions from scratch, many SmartNICs can offload the datapath from existing software switches, such as OvS or proprietary switches. Beyond packet switching, virtual switches on SmartNICs can also manage tasks like Network Address Translation (NAT), tunneling, and QoS functions, including rate limiting, policing, and scheduling [53]. Clearly, all this applies for data plane programming, allowing for aggregated functionalities to the pure switching/routing.

2.6.2.2 Tunneling and Overlay

Tunneling is a method that encapsulates one network protocol within another for transportation. It's frequently used in virtualized environments to establish isolated channels between VMs or between different parts of a virtualized network. This technique addresses the constraints of the physical network and allows for the creation of virtual networks that extend beyond physical limits. Several tunneling protocols, such as Virtual Extensible LAN (vxlan) among others, are employed in network virtualization. vxlan adds an extra layer of packet processing at the hypervisor level, which increases CPU overhead. As the number of flows grows, the CPU can become overloaded with encapsulation and decapsulation tasks, causing performance issues in throughput and latency. SmartNICs can offload these tunneling functions from the host CPU to their embedded NIC switch or programmable pipeline, reducing the load on the CPU. The control plane tasks, such as defining tunnels, are managed by the SmartNIC's CPU cores in software. This approach enhances throughput, reduces latency, and allows the host CPU to handle other tasks [53]. Data plane programmability enables the dynamic configuration of tunnels and the implementation of various (and potentially new) tunneling protocols, adapting to the requirements of each network.

2.6.2.3 Observability - Monitoring and Telemetry

Observability refers to the capability to gather and analyze telemetry data. During a network outage, effective observability aids in diagnosing and resolving issues. It also plays a crucial role in detecting malicious activities and

pinpointing network performance bottlenecks. Traditionally, packet observability solutions are implemented in hardware located outside the server.

SmartNICs support traditional packet observability methods. They also offer local monitoring and aggregation to reduce excessive traffic exports and can implement complex telemetry features, such as streaming algorithms and Bloom Filters, for detailed analysis. SmartNICs can provide additional telemetry data about the host system, including CPU, memory, and disk usage, enhancing overall system monitoring. SmartNICs can monitor traffic between VMs or containers within the same server, which traditional external approaches cannot. This offloads the monitoring burden from the host CPU, especially in high-traffic scenarios [53].

Data plane programming allows for detailed monitoring with customized inspection and granularity at the packet or flow level, adapting to specific needs. It also enables the inspection of non-traditional protocols. Monitoring applications within data plane programming, specifically P4, include: detection of heavy hitters, flow monitoring, in-band network telemetry, between others [46].

2.6.2.4 Load Balancing

Load balancers are essential in cloud environments for efficiently distributing network requests across servers in data centers. While traditional load balancers used specialized hardware, software-based solutions are now more common among cloud providers due to their flexibility and on-demand provisioning capabilities, despite higher operational and provisioning costs. Software-based load balancers offer more customization but increase expenses related to server acquisition and energy consumption. It is possible to offload load balancing to SmartNICs, including the distribution of incoming network traffic across multiple CPU cores (Receive Side Scaling) [53]. Data plane programmability allows for the implementation of load balancing functions directly in hardware, enhancing performance. It also enables dynamic application of load balancing, allowing changes at runtime [53].

2.6.3 Other categories for data plane programming

Other categories of applications for data plane programming, in particular using P4, include [46]:

2.6.3.1 Traffic Management and Congestion control

These include data center switching, congestion notification, traffic scheduling, traffic aggregation, Active Queue Management, and Traffic Offloading. The field of traffic management and congestion control takes advantage of three key features of P4: customizable packet headers, flexible header processing, and target-specific packet header processing functions. Data center switching typically focuses on parsing packet headers from standard protocols like IPv4/v6. However, more advanced protocols can be implemented using P4's flexible header processing capabilities. Target-specific packet processing functions are extensively used with many works leveraging externs such as metering and marking, though these may not be available on all hardware. A similar situation is observed where many approaches depend on priority queues. Since floating-point operations are not native to P4, some targets may provide externs to handle them. Many solutions bypass this limitation by using approximations or defining their own software-based externs [46].

2.6.3.2 Routing and Forwarding

These include Source Routing, Multicast, Publish/Subscribe Systems, between others. The field of routing and forwarding significantly benefits from P4's key features. First, the ability to define and utilize custom packet headers allows network administrators to tailor headers to specific use cases. Examples include source routing and multicast, where custom headers are used to implement lightweight mechanisms based on extra packet information not found in standard protocols. Although many of this kind of projects are developed for the BMv2 software switch, they should be easily portable to hardware platforms, as they do not rely on complex, target-specific operations.

Second, P4 enables flexible packet header processing based on the packet's header contents, which supports systems such as publish/subscribe networks, named data networks, and data plane resilience. Custom actions and conditional application of multiple match-action tables allow for adaptable packet processing tailored to specific scenarios. As with custom headers, most projects in this area were developed for BMv2 and should be transferable to hardware if target-specific actions are avoided.

Third, many works on data plane resilience rely on target-specific packet processing functions. For example, registers are often used to store the sta-

tus of egress ports (up/down) to trigger backup actions when needed. These projects are frequently implemented on the Tofino hardware platform, making the implementations highly target-dependent. Porting these to other platforms depends heavily on the hardware's capabilities and the externs used [46].

2.6.3.3 Advanced Networking

These include Cellular Networks (4G/5G), Internet of Things (IoT), Industrial Networking, Time-Sensitive Networking and NFV. Since the field of advanced networking encompasses a variety of topics, nearly all of P4's core features are utilized. The cellular networks domain heavily benefits from the ability to define and use custom packet headers, as many works rely on tunneling technologies. Additionally, flexible packet header processing enables the implementation of new 5G concepts. Some scenarios still require offloading tasks to specialized hardware or software using P4's target-specific packet header processing functions. NFV gains from the flexible development and deployment of network functions (NFs), allowing them to be replaced or relocated during operation. Furthermore, new protocols and extensions to existing ones leverage both custom packet headers and flexible packet header processing for their implementation [46].

2.6.4 Previous work use cases

In previous work [16], two use cases were proposed: monitoring and load balancing. In the monitoring case, a proof of concept was presented where a P4 program forwards packets appropriately while extracting useful data, exporting it to a P4Runtime controller to generate representative graphics with the obtained data. This use case successfully obtained useful data, demonstrating its easy accessibility, and enabled offline analysis (also testing the connection with the controller), generating graphical and representative results that contribute to the easy and quick understanding of the data. P4 allows for the complete "disassembly" of a packet, which makes this monitoring completely customizable and detailed. For example, statistics can be obtained by protocol (or other criteria) simply by inspecting this data in the packet header. Comparing this approach with a traditional one, such as the use of Simple Network Management Protocol (SNMP) [21], it can be seen that monitoring with P4 is not limited to existing protocols and can monitor networks that do not

use the Internet stack. Eventually, more personalized data inspection using SNMP may require more development, if the operating system is modifiable; otherwise, one must work with what the operator has implemented. However, SNMP allows the monitoring of some hardware aspects.

In the load balancing case, a proof of concept was presented where there are two P4 programs: a Round Robin (RR) load balancer and another load balancer that decides the destination according to the transport layer protocol (TCP or UDP). There is a P4Runtime controller that changes the P4 program running on the switch at runtime based on the average packet size. In the RR load balancing case, it was observed how state can be maintained within the switch between packets (keeping the last destination) and how this value can be queried and used for decision-making. This proof of concept demonstrated that it is possible to change both the P4 program and the control plane rules at runtime, showing the ability to make real-time decisions. Furthermore, these decisions can be based on monitoring data collected with the P4 program, in real time, and with the ease provided by a program written in Python (controller). Compared to a traditional approach, personalizing the forwarding algorithm would require implementing it in the operating system kernel if it is open-source. Otherwise, one would depend on the device manufacturer's implementation, meaning the desired algorithms would need to be incorporated. Lastly, it is not possible to change the execution algorithm in real-time on a traditional switch.

NDP [44] is a protocol that seeks to address the lack of transport protocols for data centers that provide high capacity and low latency. Creating a new protocol involves the need for network devices capable of handling packets defined under this new protocol. To achieve this without relying on vendor-provided solutions, network device programmability becomes highly valuable. This is what the developers of NDP did, providing an implementation of their protocol for network devices through data plane programming. The NDP protocol includes an implementation of an NDP switch using P4 and an NDP host using DPDK. In previous work [15], it was possible to run the implementation of an NDP host on Linux after making certain code modifications. This implementation uses DPDK to bypass the default stack and enable packet processing on the host, appropriate for the new protocol.

2.7 Machine Learning for networking

ML allows a system to analyze data, derive knowledge, and improve its understanding over time through experience. The latest developments in machine learning have increased the adaptability and robustness of these techniques, making them applicable across a wide range of real-world situations. The abundance of data in contemporary networks, which is expected to expand further with emerging networks like the Internet of Things (IoT), encourages the utilization of ML. This use goes beyond identifying hidden patterns, but also encompasses learning and understanding the processes responsible for generating the data. Classifying traffic is essential for network operators to carry out various tasks related to network operation and management. Recent advancements in machine learning have made these techniques adaptable and robust, allowing them to be applied to a wide range of real-world situations, from the exceptional to the ordinary.

Recent computing advancements provide the storage and processing power necessary for training and testing machine learning models on large datasets. However, network operations and management remain challenging, with faults often caused by human error. These faults can result in financial losses and damage to the reputation of network providers. As a result, there is significant interest in developing autonomic networks that are self-configuring, self-healing, self-optimizing, and self-protecting to enhance resilience. Although there is a critical need for cognitive control in network operation and management, it presents specific challenges for machine learning. Each network is unique, and the lack of standardization means that patterns effective in one network might not work in another. Additionally, the constantly evolving nature of networks makes it difficult to apply a fixed set of operational patterns. The rapid growth in the number of applications and types of connected devices makes manual network administration nearly impossible to maintain.

Recent technological advancements in networking, like SDN, enhance the potential for applying ML in this field. While ML has been widely used in areas like pattern recognition, speech synthesis, and outlier detection, its use in network operations and management has been limited due to challenges in collecting data from and controlling legacy network devices. SDN helps overcome these challenges by allowing network programmability. The insights gained from ML can assist in automating network management tasks, making

the application of ML to networking a complex yet promising research area. This requires a deep understanding of both ML techniques and networking issues.

There are four main categories of problems that can benefit from ML: clustering, classification, regression, and rule extraction. Clustering aims to group similar data points together while maximizing the separation between different groups. In classification and regression, the objective is to map new input data to a set of discrete outputs or continuous values, respectively. Rule extraction problems differ in that they focus on identifying statistical relationships within the data. ML is particularly well-suited for solving problems when there is a large representative dataset available. ML techniques are designed to uncover and leverage hidden patterns in data to: 1. describe outcomes by grouping data in clustering problems, 2. predict the outcomes of future events in classification and regression problems, and 3. assess the outcomes of a sequence of data points in rule extraction problems.

Networking issues can be framed as problems that benefit from machine learning. For instance, a classification problem in networking might involve predicting the type of security attack, such as Denial-of-Service (DoS), User-to-Root (U2R), Root-to-Local (R2L), or probing, based on network conditions. In contrast, a regression problem could involve predicting when a future failure is likely to occur. While there are various categories of problems that benefit from machine learning, there is a general approach to developing ML-based solutions. The essential components in designing these solutions for networking are: Data collection, which involves gathering, generating, or defining the dataset and the classes of interest. Feature engineering, that aims to reduce data dimensionality and identify key features that decrease computational costs and improve accuracy. Finally, ML techniques are used to analyze the complex relationships within the data and learn a model for predicting outcomes.

Machine learning encompasses four learning paradigms: supervised, unsupervised, semi-supervised, and reinforcement learning. These paradigms affect how data is collected, how features are engineered, and how ground truth is established. The goal is to infer an outcome based on a dataset, often referred to as training data. If the data description is known, labels are associated with the training data. The outcome is typically viewed as identifying membership in a particular class of interest. Supervised learning involves using labeled

training datasets to build models that identify patterns or behaviors based on known data. This method is typically applied to classification and regression problems to predict discrete or continuous outcomes. In cases where there is partial or incomplete labeling, semi-supervised learning techniques can be used. Unsupervised learning, on the other hand, uses unlabeled datasets to create models that differentiate between patterns in the data, making it suitable for clustering problems. For example, in networking, unsupervised learning can be used for detecting outliers or estimating density by grouping similar types of attacks. Reinforcement learning (RL) is an iterative, agent-based approach to decision-making problems. Unlike traditional learning methods that rely on training examples, RL involves an agent interacting with its environment, learning through exploration and feedback rather than predefined data. The agent receives rewards or penalties based on its actions, with the goal of discovering the best sequence of actions or "policy" to maximize cumulative rewards. This approach is well-suited for tasks like decision-making, planning, and scheduling, as it can prioritize long-term rewards over immediate gains. The choice of learning paradigm is closely tied to the nature of the training data and the problem at hand; if supervised learning isn't feasible due to insufficient data knowledge, other learning paradigms may be more appropriate [12].

2.7.0.1 Data collection

Machine learning techniques need representative, unbiased data to create effective models for networking problems. Data collection is crucial and can be done in two phases: offline and online. Offline data collection involves gathering a large amount of historical data for training and testing models, often from various repositories. Online data collection involves real-time network data, which can be used for model feedback or re-training. Monitoring and measurement tools are key for both types of data collection, offering control over aspects like sampling rate, monitoring duration, and location. These tools use network monitoring protocols and can be either active or passive. Active monitoring introduces additional traffic to the network for data collection, while passive monitoring avoids this overhead by analyzing existing network traffic, though it requires extra devices for data analysis.

After collecting data, it is typically divided into training, validation (or development), and test datasets. The training set is used to determine the

best parameters for a machine learning model, such as the weights in a neural network. The validation set helps in selecting the optimal model architecture or choosing among different models, though if the model and architecture are already chosen, validation may not be necessary. The test set is used to evaluate the model's unbiased performance.

Validation and testing can be done using either the holdout method, where a portion of the dataset is reserved for validation or testing, or k-fold cross-validation, where the dataset is split into k subsets, and each subset is used for validation or testing in turn, with results averaged across all rounds.

Common dataset splits include 60/20/20% for training, validation, and testing, or 70/30% if validation is not used. For larger datasets, more extreme splits are also valid. It's important to ensure that the training, validation, and test datasets are independent and represent the same distribution to avoid skewness, which can lead to model overfitting or underfitting [12].

2.7.0.2 Feature engineering

Raw data collected for machine learning can be noisy or incomplete, so it must be pre-processed to clean it before use. Feature extraction is another crucial step where features are selected to help in learning and inference. In networking, features can be categorized by their granularity: 1. Packet-level features: derived from individual packets, such as packet size statistics and time series information. These are robust to sampling variations. 2. Flow-level features: calculated from flow statistics like mean flow duration and packet counts. 3. Connection-level features: extracted from transport layer details, including throughput and TCP window size. These offer high-quality data but may involve computational overhead and are sensitive to sampling and routing changes.

Feature engineering involves both feature selection and extraction. Selection removes irrelevant or redundant features to reduce computational costs and prevent overfitting, while extraction creates new features from existing ones using methods like entropy, Fourier transform, and principal component analysis (PCA). Tools can aid in this process, but specialized methods—filter, wrapper, and embedded techniques—are often used for optimal feature selection. Filtering eliminates irrelevant features, wrapper techniques iteratively test different subsets, and embedded methods integrate selection into model

training.

When performing feature engineering, it's crucial to consider the specific characteristics of the task. For example, in network traffic classification to identify streaming applications, features such as average packet size and packet inter-arrival times are important. Average packet size tends to be consistent, and inter-arrival times help distinguish between bulk data transfers and streaming. However, factors like fragmentation, encryption, and queuing can affect these features. Additionally, streaming applications may resemble bulk data transfers in behavior. Thus, understanding the types of applications being classified is essential for selecting the most relevant features.

Last, it is crucial to choose features that align with the assumptions of the problem being addressed. In traffic classification, for instance, features from multi-modal application classes may exhibit non-Gaussian behavior, which can conflict with assumptions like independent, Gaussian-distributed feature distributions. Proper feature extraction and selection are essential to ensure the effectiveness of machine learning models [12].

2.7.0.3 Establishing ground truth

Establishing ground truth involves assigning formal labels to classes of interest, often through hand-labeling by experts, deep packet inspection (DPI), pattern matching (like application signatures), or unsupervised machine learning techniques. For traffic classification, ground truth can be established using application signatures based on features like average packet size and flow duration. These signatures need to be updated regularly, especially for encrypted traffic.

Alternatively, statistical and structural content models can describe datasets and infer classes. For example, these models can classify a protocol using the label from a single instance of that protocol and derive correlations from unlabeled training data. Accurate ground truth is crucial for machine learning model performance, as imbalances in training data across classes can affect model accuracy. Addressing class imbalance may require techniques such as under-sampling, over-sampling, or ensemble methods to ensure fair training and effective model performance [12].

2.7.0.4 Performance metrics and model validation

After building an ML model and establishing ground truth, it's essential to evaluate the model's performance. There isn't a single"best" learning algorithm, and error rates can't be universally compared across different applications. Performance metrics assess various aspects of the model, including reliability, robustness, accuracy, and complexity. Accuracy validation involves error analysis, where the difference between actual and predicted values is calculated.

For classification, error metrics like binary and categorical cross-entropy are used, while regression uses Mean Absolute Error (MAE) and Mean Squared Error (MSE). In classification tasks, accuracy is a common metric, defined as the proportion of correct predictions among all predictions. However, accuracy can be misleading with imbalanced data. To address this, metrics from a confusion matrix are used: 1. True Positive (TP): Correctly predicted positive instances. 2. True Negative (TN): Correctly predicted negative instances. 3. False Positive (FP): Incorrectly predicted positive instances. 4. False Negative (FN): Incorrectly predicted negative instances. These metrics help in evaluating and comparing different models, as well as tuning model parameters to balance recall and precision, leading to more reliable performance assessments [12].

2.7.1 ML applications in networking

There are various machine learning applications within computer networks, including: traffic prediction, traffic classification, traffic routing, congestion control, resource management, fault management, QoS and QoE management and network security. Network traffic prediction is essential for managing the complexity and diversity of modern networks. In this work, we will primarily focus on traffic classification, concentrating our analysis on this specific category.

2.7.1.1 Traffic Classification

Classifying traffic is crucial for network operators to perform various operational and management tasks. Traditionally, this has been done by associating Internet Assigned Numbers Authority (IANA) registered port numbers with applications. However, this simplistic method of using port numbers does not

support learning and has become outdated. Relying solely on port numbers is ineffective due to issues like dynamic port negotiation, tunneling, and the misuse of well-known application port numbers to disguise traffic and bypass firewalls. Despite this, many classifiers still use port numbers along with other techniques. Payload-based traffic classification serves as an alternative to port-based methods. Nevertheless, as it involves scanning the payload for recognized application signatures, it results in increased computational and storage expenses. Additionally, manual upkeep and adjustment of signatures becomes challenging due to the continuous growth in the number of applications and their dynamic nature. Moreover, concerns regarding security and privacy have led to the encryption of payload, with restricted access due to privacy laws. Consequently, deducing a signature for an application class using payload becomes intricate. The host behavior-based traffic classification uses the inherent behavioral traits of hosts within the network to anticipate the relevant classes. These classifiers operate on the premise that applications exhibit distinct communication patterns. For instance, a Peer-to-Peer (P2P) host might establish connections with multiple peers, employing a unique port number for each peer, while a web server could be accessed by various clients using the same port. The accuracy of traffic classification based on host behavior largely depends on where the monitoring system is placed. This is particularly important because routing asymmetries in the network core can influence the observed communication patterns. Differing from classifiers that rely on payload or host behavior for traffic analysis, classifiers based on flow features take a distinct approach. They adopt a broader viewpoint by examining a communication session, comprising a pair of complete flows. A complete flow denotes a one-way sequence of successive packets exchanged on the network between a port at one IP address and another port at a different IP address, utilizing a specific application protocol. The quintuple <srcIP, destIP, srcPort, destPort, protocol> identifies a flow. A feature is an attribute representing a unique characteristic of a flow, such as packet length, inter-arrival time, flow duration, and packet count. Flow feature-based classification uses these features to categorize flows, leveraging the unique traffic patterns generated by different applications. This technique has the potential to overcome several limitations of other methods, such as unregistered port numbers, encrypted payloads, routing asymmetries, and high storage and computational overhead. Fundamentally, traffic classification based on flow features takes advantage of the variety and discernible

attributes present in the traffic patterns produced by various applications. However, it remains to be seen if flow feature-based classifiers can match the accuracy of payload-based classifiers. Flow feature-based traffic classification is a well-studied technique that utilizes both supervised and unsupervised machine learning for classifying network traffic. In supervised learning, methods like kernel estimation, neural networks, and support vector machines (SVM) are used to achieve high accuracy. However, network operators often lack complete information about all network applications, making it unrealistic to have prior knowledge of every application for traffic classification. Therefore, unsupervised machine learning techniques are employed for practical traffic classification using flow features. Both hard and soft clustering techniques are explored in unsupervised learning. Given that application flow features can be quite similar, hard clustering is not suitable for detailed classification. Soft clustering, particularly density-based clustering, offers the required granularity and is faster to train than expectation-maximization (EM)-based soft clustering.

Depending on the conclusion of a flow for traffic classification not only results in prolonged training time and increased memory usage for classifiers but also causes delays in making time-sensitive classification decisions. Consequently, early traffic classification can be done using the initial packets, rather than the complete flow.

Supervised machine learning provides high accuracy for traffic classification, while unsupervised techniques offer greater robustness. Consequently, combining supervised and unsupervised ML for traffic classification has proven successful. Semi-supervised classifiers are not only resilient but can also be easily adapted to detect zero-day traffic and retrained to improve accuracy against previously unknown applications. Recent advances in networking, such as SDN and NFV, present new opportunities for traffic classification, particularly in identifying applications and QoS classes. Although some initial studies in this area have achieved high accuracy, further evaluation is needed to assess their resilience, temporal and spatial stability, and computational overhead. It is also crucial to determine the feasibility of these technologies for making time-sensitive traffic classification decisions [12].

2.8 Final Remarks

This chapter outlined the theoretical foundations necessary for understanding and applying network programmability. It examined the enabling technologies and various hardware devices that support it, with a particular focus on the P4 language, concluding that P4 is the most suitable language for this work. Additionally, the chapter explored various applications of network programmability and provided an overview of machine learning in networking.

Chapter 3

Proof of concept

In order to test the complexity of applications that leverage network programmability, particularly using the P4 language, this chapter explores the implementation of a complex solution involving network programmability and machine learning. It presents a software solution along with its architecture, components, and corresponding tests. Subsequently, a scaled-down hardware version of this solution is introduced to demonstrate its potential real-world implementation. All the code developed is available at [17].

3.1 HALIDS - A Software solution

A promising idea is to do traffic classification as early as possible, that is, on network devices. Nevertheless, their resources have been traditionally constrained, encompassing limitations in terms of memory, processing capacity, available operations, and more. Consequently, these devices are traditionally treated as "dumb" regarding Network Traffic Monitoring and Analysis (NTMA), performing only the essential functions required for the network to operate. The emergence of new data plane architectures raises the hope that network devices will perform functions beyond simple traffic forwarding. By doing so, the burden on the control and management planes is alleviated, and a portion of the processing is decentralized. Additionally, processing within the network device occurs more expeditiously, reducing the need for offloading to the control plane.

In Figure 3.1, we can see a traditional deployment (in red) where decisions are made by an external server, adding latency and not leveraging the capabil-

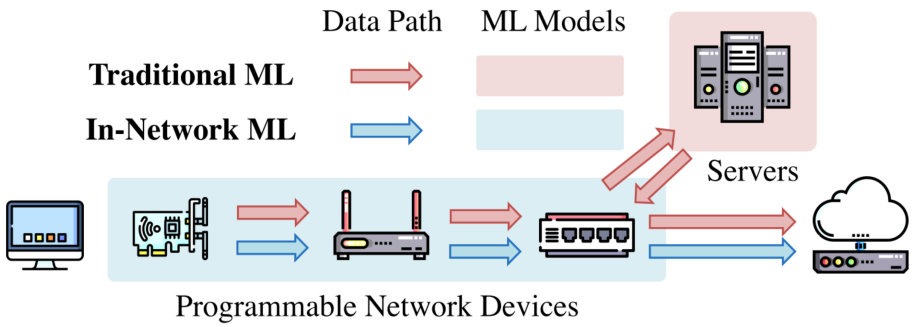


Figure 3.1: Traditional ML classification using external servers vs in-network classification using programmable hardware [96].

ities of a programmable network device. In blue, we can observe an in-network machine learning deployment, where the network device can make decisions at line rate.

It is here that traffic classification intersects with enhanced data plane and machine learning. We introduce *HALIDS*, a Hardware-Assisted Machine Learning IDS for in-Network Monitoring at forwarding speed. In this sense, the scenario presented in this application involves the creation of an IDS, wherein the network device can make a decision (regarding traffic) with a high level of confidence, or delegate this decision to an expert (oracle).

Both the network device and the oracle use AI/ML models for intelligent traffic classification. While the AI/ML model running at the network device is simpler, in order to fit the limited processing capabilities and memory of programmable devices, and runs at in-line rate, the oracle model is more complex and can be deployed off-band at an external server with significantly higher resources for AI/ML-driven analysis.

The HALIDS concept revolves around leveraging the fast processing power of switches to quickly process traffic at in-line rate, while also taking into account their limitations. In this case, the device is allowed to classify incoming traffic if the confidence in this decision is high. Otherwise, this classification is delegated to an "expert" (oracle), which, by having significantly more resources, can provide a more accurate decision. Specifically, this decision entails determining whether a packet is malicious or not, a task that the network device can perform to a limited extent due to the aforementioned constraints. The idea is that the network device can identify if a packet is evidently malicious. In the case of the device, the model is trained with fewer data points than the oracle's model due to memory limitations, privacy issues, between

other possibilities.

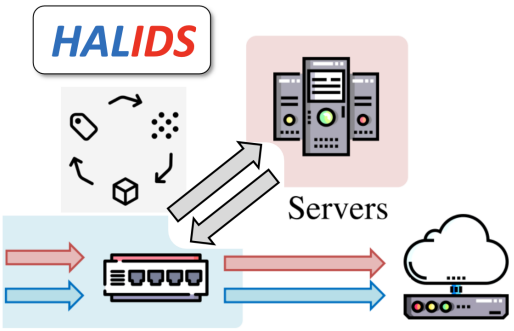


Figure 3.2: HALIDS general architecture. The programmable network device runs an ML model that classifies packets. If the confidence in the decision does not surpass a defined threshold, it queries an oracle (servers). If it is confident in its decision, it acts accordingly.

In Figure 3.2, we can see the general architecture of the HALIDS solution. We have a programmable network device that contains a ML model (in this case, a Random Forest (RF)) capable of making decisions at line rate when it has sufficient confidence in the model's classification, and act accordingly. If the classification does not surpass a confidence threshold, the switch can query an oracle, which has a more complex model and more training data. Once the oracle provides the classification, the switch can act based on it. When the oracle is queried, the original packet is sent to the oracle with a new header that includes the necessary features for class determination. Naturally, this introduces a delay in the action taken on the packet, as it must return from the oracle with a classification before any action can be performed. If the oracle returns a classification of malware, the flow is flagged, and appropriate actions are taken at the switch (e.g. drop), ceasing further queries to the oracle. Otherwise, the packets will continue to be classified at the switch and forwarded to the oracle in cases of low confidence.

P4 provides us with the ability program this behaviour in the network device. Following state of the art [1, 18, 60], HALIDS implements standard Decision Tree (DT)s and RF models within the programmable data plane, which easily map to the match-action pipeline by associating each level of the tree with a table. Ideally deployed on a server, the oracle's model can not only be significantly more complex and heterogeneous – e.g., relying on deep-learning, ensemble architectures, or even foundation models, but also trained with massive amounts of data to enhance the accuracy of its predictions.

HALIDS also relies on active learning principles [84] to enable model retraining at the programmable device, relying on the oracle model as tutor providing the required ground-truth labels. Model retraining is crucial for AI/ML-driven analysis to adapt to changing network traffic. Once a predefined retraining policy rule is reached, such as after offloading a specific number of packets to the oracle, the network device will undergo retraining using decisions made by the oracle. Consequently, as the traffic pattern evolves, the model at the network device is expected to reach decisions that may not be sufficiently reliable. By retraining with more accurate decisions from the oracle, the network device aims to dynamically adapt to the evolving traffic conditions. Since the match-action pipeline associates each level of the tree with a table, retraining the switch simply involves rewriting the tables from the control plane.

3.1.1 Related Work

Several works addressing different parts of the raised issue have been found, all using P4. Firstly, SwitchTree [60] proposes the integration of RF into the data plane for abnormal traffic identification. It extracts flow-level features with early detection (calculating the features in each packet) and incorporates RF as tables. In this case, the code is not automated and is limited to acting only on the switch. pForest [18] introduces a similar idea, but various RF are trained for different phases of the flow, additionally proposing a confidence percentage for decision-making. Despite seeming like an innovative idea, the code is not currently available. CML-IDS [39], the work most similar to ours, proposes an RF in the data plane, and an oracle used to provide more accurate decisions in case of a lack of confidence in the switch's decision. This work seems to have a strong focus on providing more powerful algorithms in the oracle and does not involve switch re-training. The aforementioned solutions are all implemented in software. Marina [85] is a hardware-based solution where the network device extracts the necessary (more complex than those used in other solutions) features and sends them to an ML server with a powerful prediction model. Flowrest [1] and Leo [51] propose a solution designed for hardware implementation, integrating RFs and DTs into the data plane. NetBeacon [97] proposes an idea similar to pForest, implemented in hardware.

Regarding background on Active Learning (AL) [84], the paradigm aims

to increase the performance of an AI/ML model by interactively querying an oracle – e.g., a human user or some other expert source of information, to label new data instances for subsequent model (re)training. In the context of communication networks, AL facilitates the practical implementation of AI/ML solutions by empowering users to make crucial administrative decisions and incorporating expert insights into the monitored data. When the model encounters unseen samples or lacks confidence in its predictions, it defers the decision to the expert administrator for further review and reclassification, using this additional knowledge for retraining. This process ensures that the model remains up-to-date and enables it to adjust to changes in the network over time, thereby addressing issues such as concept drift and zero-shot classifications (e.g., for zero-day attacks). Conventional AL methods generally focus on retraining the model after each queried sample, which can be impractical and inefficient. Indeed, continuously training and deploying new models may not be viable. To address this, batch-mode AL is usually applied, wherein the model is retrained after a pre-defined group of samples has been labeled by the oracle.

3.1.2 Solution

We choose SwitchTree [60] as the basis for HALIDS' programmable switch implementation, while the communication mechanism with the oracle is inspired by CML-IDS [39].

SwitchTree implementation comprises the P4 implementation of a Random Forest, along with the scripts (hardcoded) for training. For training the Random Forest, the UNSW-NB15 dataset[69, 70, 71, 83][1] is employed. This dataset categorizes attacks into 9 categories and encompasses 49 features. SwitchTree's focus is not on detecting the specific type of attack but rather on discerning whether it is a normal flow or not. To achieve this, 12 features are selected from the available set, as utilizing all features would demand excessive switch memory, especially for stateful features. To identify the most important features, the authors trained a Random Forest without depth or tree number limits, systematically discarding features based on impurity until achieving an F1-score$\geq$ 0.95. Concerning the number of trees, a higher count allows for greater generalization but necessitates more switch memory.

[1] https://research.unsw.edu.au/projects/unsw-nb15-dataset

Considering precision, recall, and F1 metrics, the authors conclude that the ideal number is 3 trees. Additionally, the authors constrain the tree depth to 10 levels [60].

SwitchTree consists of a P4 program responsible for extracting and calculating features for each packet and processing them with the previously trained RF. The classification is done by the RF, which is embedded in tables within the switch. In Figure 3.3, a Decision Tree embedded in a match-action pipeline is illustrated. Each level of the tree is mapped to a match-action stage, where a feature is checked at each level. Depending on whether the condition is satisfied or not, processing proceeds to the next level. As it is a Decision Tree, the obtained result is carried forward. This process continues until reaching a leaf, where a decision is made, and a class is assigned to the packet. Figure 3.4 exemplifies the table structure at the nth level within the P4 code. As shown, the table key comprises the unique node ID, the ID of the previously evaluated feature, and the previous result (whether the condition evaluated to true or false). The possible actions to be taken include:

- `CheckFeature`: Responsible for evaluating the condition described by the parameters received from the control plane. In this case, it receives the ID of the next node, the threshold to be evaluated (value at the node in the Decision Tree against which it is compared), and the ID of the feature with which it should be compared.
- `SetClass`: Tasked with assigning the corresponding class. This action maps to a leaf in the tree. In this instance, the control plane parameters include the ID of the next node (useful for advancing to the next Decision Tree in case of a Random Forest) and the class to be assigned.
- `NoAction`: The default action when a packet does not match any key.

For instance, considering the Decision Tree depicted in Figure 3.3, the initial examination involves the condition $dpkts \leq 0$ at level 1. Each feature evaluation should have a rule structured as follows:

`Key(node_id, prevFeature, isTrue)` $\rightarrow$ `CheckFeature(next_node_id, next_feature_id, threshold)`

Therefore, the level 0 will have the following entry:

`Key(node_id, 0, 1)` $\rightarrow$ `CheckFeature(next_node_id, dpkts_feature_id, 0)`

In this case, the initial state of the key has some particularities, such as the

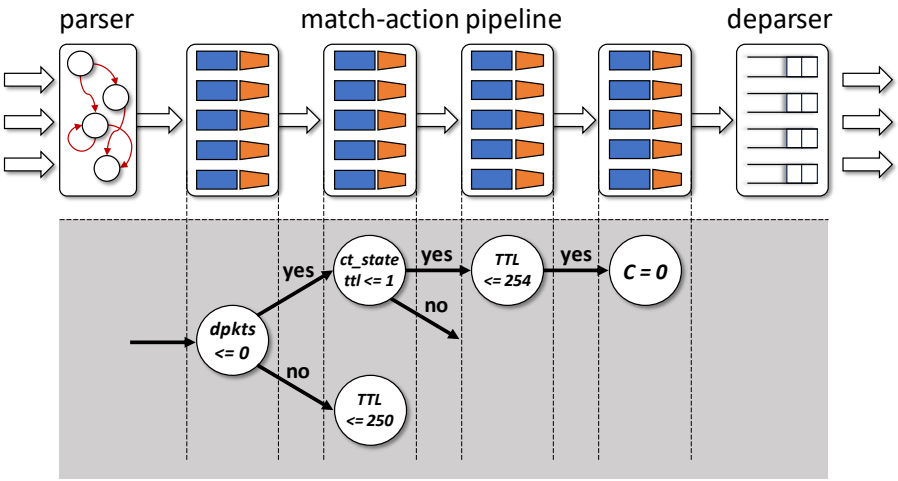


Figure 3.3: An instance embedding of a tree within the Random Forest is illustrated in the switch. The features evaluated include `dpkts` (number of packets sent from destination to source), `ct_state_ttl` (a function involving source TTL and destination TTL where TTL stands for time to live), `TTL` and class representing the ultimate packet classification [60].

```
table level_n {
    key = {
       meta.node_id: exact;
       meta.prevFeature: exact;
       meta.isTrue: exact;
    }
    actions = {
       NoAction;
       CheckFeature;
       SetClass;
    }
    size = 1024;
}
```

Figure 3.4: An illustration of a P4 table designed to encapsulate a node at the nth level of a Decision Tree [60]

previous feature being 0 (i.e., none), and it is marked as true. Subsequently, the condition to evaluate is $dpkts \leq 0$, as the next feature specified in the parameter is `dpkts` with a threshold of 0.

Next, the table at level 1 will have the following entries:

`Key(node_id, dpkts_feature_id, 1)` $\rightarrow$ `CheckFeature(next_node_id, ct_state_ttl_feature_id, 1)`

`Key(node_id, dpkts_feature_id, 0)` $\rightarrow$ `CheckFeature(next_node_id, TTL_feature_id, 250)`

The first rule represents the scenario where the condition $dpkts \leq 0$ evaluates to true (indicated by the `isTrue` value of 1), leading to the evaluation of

the upper part of the tree, where $ct\_state\_ttl \leq 1$ is assessed. The second rule illustrates the case where the condition $dpkts \leq 0$ evaluates to false (indicated by the `isTrue` value of 0), triggering the evaluation of the lower part of the tree, where $ttl \leq 250$ is examined.

Ultimately, upon reaching a leaf, the corresponding key will be generated to execute the SetClass action and assign the appropriate class. Each feature evaluation should have a rule structured as follows: `Key(node_id, prevFeature, isTrue)`→`SetClass(next_node_id, class_to_be_set)` In the example, after the condition $ttl \leq 0$ evaluates to false, the class 0 is set, resulting in the following rule at the last level: `Key(node_id, ttl_feature_id, 1)`→`SetClass(next_node_id, 0)`

The tables for each label with the aforementioned rules are illustrated in Figure 3.5.

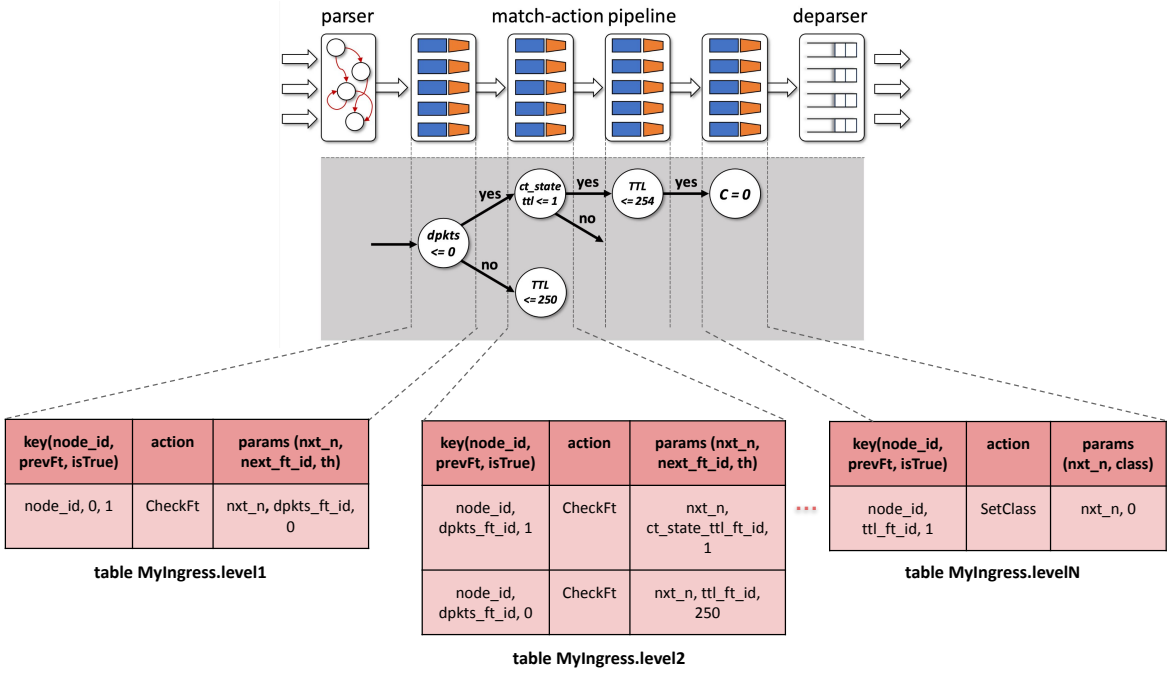


Figure 3.5: An illustration of the mapping between each table and the match-action pipeline. At each table it can be seen the key, action to be taken and the parameters for the action. prevFt refers to the key field prevFeature, params refers to the parameters for the action, where nxt_node refers to the next_node_id, next_ft_id refers to the next_feature_id, and th to the threshold to compare. CheckFt refers to the CheckFeature action and xxx_ft_id refers to the feature id for the corresponding feature.

In conclusion, multiple Decision Trees can be integrated into various match-action stages, functioning either as serial stages within a single pipeline or as

parallel stages across parallel pipelines, depending on the switch's capabilities. In the case of multiple DTs, meaning a RF, one would have tables for each level, for each tree. Figure 3.6 illustrates how to embed a RF in a pipeline. Similarly to the DT, we have a table for each level, but this time for also for each tree.

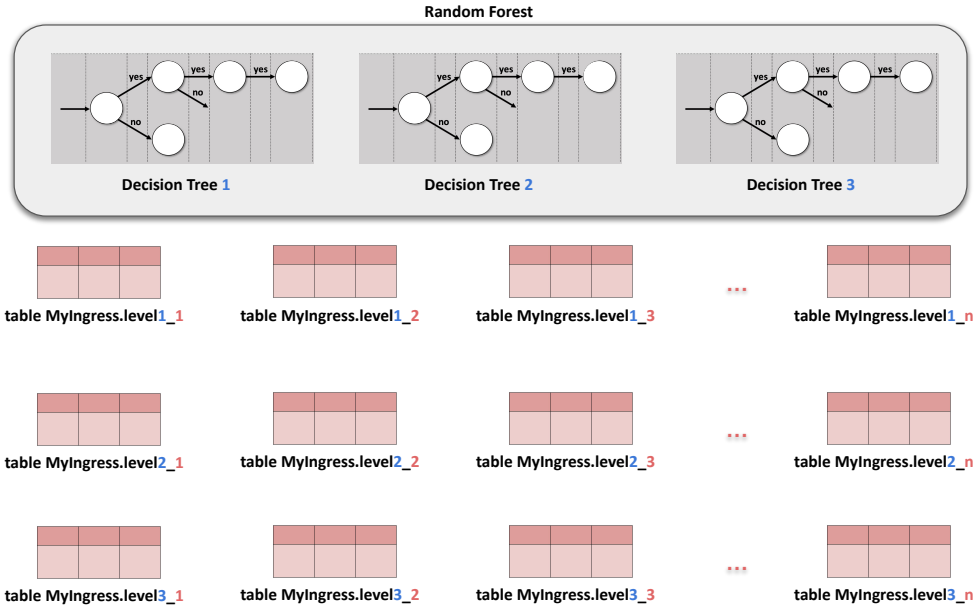


Figure 3.6: An illustration of a Random Forest embedded in a match-action pipeline.

A concluding stage assesses the outcomes from all the Decision Trees and, based on the implemented algorithm, generates the final classification. The SwitchTree implementation employs a voting algorithm, determining the final class as the one favored by the majority of the Decision Trees. Lastly, the implementation includes P4 code for both single-tree and three-tree scenarios within the Random Forest [60].

Regarding the features in SwitchTree, some are stateless (such as destination port), while others are stateful (such as flow duration), requiring the storage of information for each flow in the switch. Most of the latter are approximated since they involve floating-point operations or division, which P4 does not support. Traffic classification in SwitchTree occurs at flow level, performing early detection, meaning it does not wait for the flow to complete before classifying it. It computes features using packets received up to the current moment of the flow. To maintain flow records, registers and hashing

are employed, using the quintuple as a key to identify each flow.

For each packet arriving at the switch, a hash is applied using the standard quintuple <source_IP, destination_IP, source_port, destination_port, protocol> as a key to identify each flow. For every incoming packet, both type of features are calculated, and a decision is made for each packet, based on the flow information obtained up to that moment. As the flow progresses, more information is gathered (more packets arrive), and if a packet from the flow is eventually detected as malware, the remaining packets of the flow are marked accordingly. In Figure 3.7, we can see this behavior in action. First, a packet (pkt 1) from flow X arrives. For this flow (calculating the hash), we update the features; in this example, we update the flow duration. Then, we check for malware by running the model with the newly calculated features. If the flow is identified as malware, it is marked accordingly, and any subsequent packets from this flow will be considered malware, meaning no further feature calculation will be performed. Assuming the first packet is not marked as malware, the second packet from the flow (pkt 2) will arrive. With this new information, we update the features again, by subtracting the arrival time of the first packet from that of the second, to calculate the flow duration. Once all the features are updated, we check for malware and mark it accordingly. The same process applies to the next packet, packet 3 from the flow (pkt 3). We update the flow duration and other features and check again.

3.1.3 HALIDS System

Figure 3.8 illustrates the HALIDS architecture. We have made several improvements to SwitchTree. First, the entire process of training the switch is now automated. Mininet is no longer utilized, and virtual interfaces are adopted. This shift is necessitated by the fact that within the BMv2 variations, Mininet only launches the Simple Switch target (default version). In the proposed solution, we employ the SimpleSwitchGrpc [29], a version of the Simple Switch with P4Runtime support. P4Rutnime is used for the communication between the switch and the Oracle, encompassing packet exchange, configuration, and switch retraining. Therefore, the compilation and deployment of interfaces, switches, etc., are automated.

Second, we introduce the concept of an *oracle*. This Oracle is a Python program that implements a machine learning model, specifically a RF, which

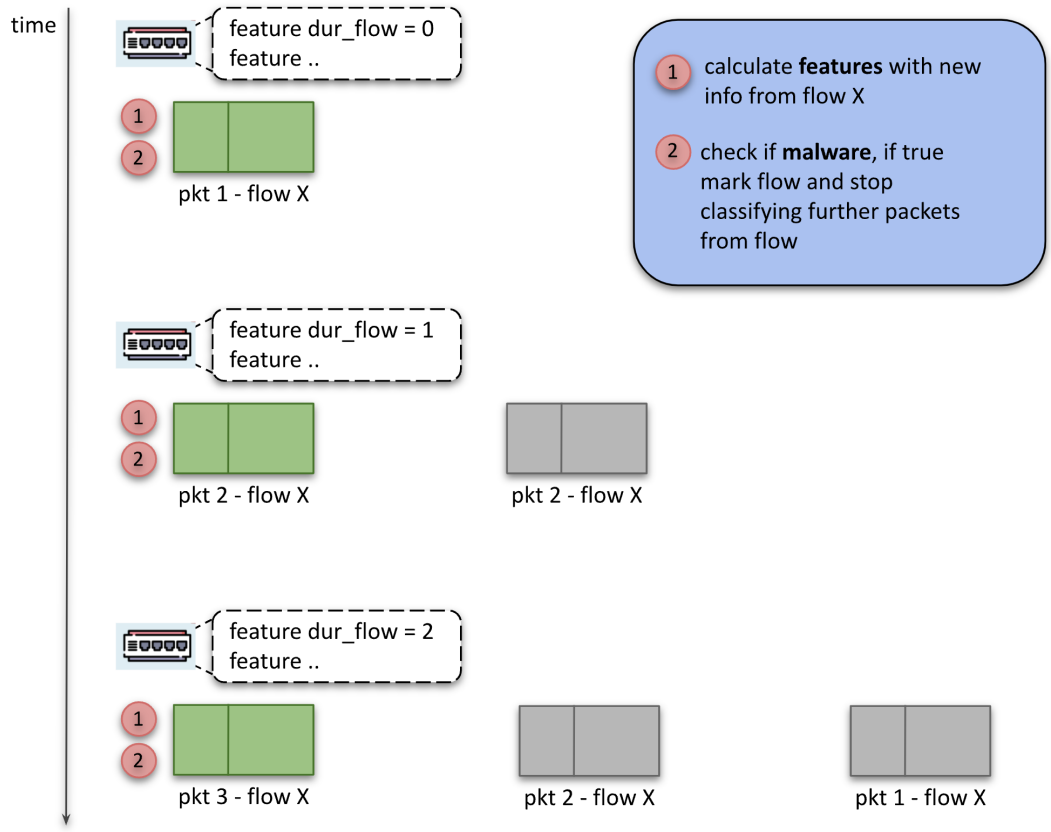


Figure 3.7: Illustration of how early detection at flow level for every packet is performed. Features are updated with every incoming packet from a flow, then it is checked for malware and marked accordingly.

initially conducts its training using the scikit-learn library[1]. This RF has a significantly greater number of levels and trees compared to the switch. Furthermore, it trains with all available data.

Subsequently, the Oracle establishes a connection with the switch via P4Runtime and installs the P4 program. It also trains the switch for the first time using a reduced set of available data and a smaller Random Forest. For this, it trains the model, generates the rules, and writes them into the switch's table. Then, for each packet received by the Oracle, it executes a function that predicts the label, explained later on. The Oracle is trained with all available data and designed with greater complexity than the switch model. When the oracle receives a packet, it extracts the features received in the packet and then processes them with the RF model to predict a label, which is then sent back to the switch. The workflow logic of HALIDS is explained in detail in

[1] https://scikit-learn.org/stable/

Figure 3.9.

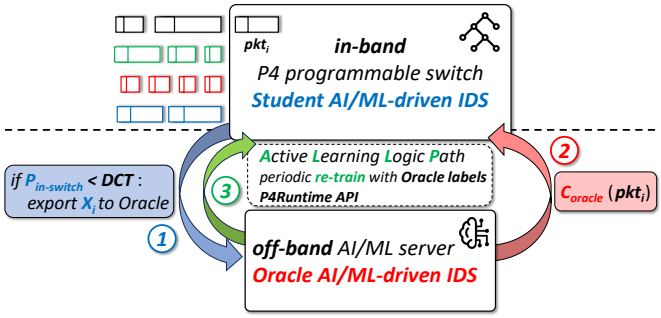


Figure 3.8: HALIDS architecture for in-band/off-band traffic classification, an eventual re-training through Active Learning. $P_{in-switch}$ refers to the decision confidence at the switch, X_i to the features, C_{oracle} to the packet classification at the oracle.

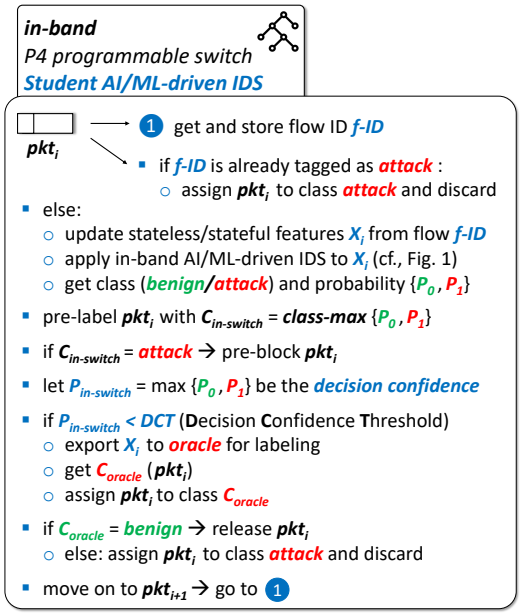


Figure 3.9: HALIDS traffic analysis workflow.

In the case of the switch, the confidence percentage of the decision made by the tree is incorporated. To achieve this, these values must be obtained during the training stage. These percentages are calculated as the number of samples of the label (0 or 1) divided by the number of samples reaching a leaf. Subsequently, these percentages are added to the rules (i.e., the tables), becoming an additional parameter of the `SetClass` action. In the scenario of having a single tree in the RF, the code of the `SetClass` action is modified to evaluate this confidence percentage and compare it with a defined threshold. If the

percentage surpasses the threshold, the normal forwarding is executed, taking into account the class predicted by the tree. If not, the packet is marked to be sent to the Oracle, allowing for a more secure class assignment. When three trees (or eventually more than one) are present, a `SetClass` action is designated for each tree (`SetClass1`, `SetClass2`, `SetClass3`). Each of these actions stores the confidence percentage and the class predicted by the corresponding tree. Instead of a simple majority vote, these percentages are weighted by the number of trees, and the classification is determined by the one with the highest value. The subsequent steps mirror those when a single tree in place to determine the packet's destination.

To send the packet to the Oracle, packet IO support from the P4Runtime shell [28] is employed. In this context, the goal is to transmit to the Oracle the necessary features for class determination, along with some data required for feature approximation. To achieve this, a header (of type `controller_header`) is added to the packet. The required data is then appended to this header, and the port indicating that it should be directed to the Oracle is set. The Oracle receives this packet, extracts the features, predicts the classification using the trained RF, and sends the classification back to the switch. It is essential to note that as the switch approximates some features due to its limited capabilities, the Oracle must also perform these approximations to obtain the classification. This can, of course, be improved in the future by sending different features or complementary information to achieve better approximations, as Python is not as limited in terms of operations. When the switch receives the packet from the Oracle (identifiable by the header), it simply sets the obtained class and proceeds to forward the packet to the appropriate destination based on the obtained class. This behavior is illustrated in Figure 3.10.

To enable retraining, the oracle stores the corresponding features and the classification obtained for each packet. Once a defined criteria is met, for example, when the number of packets received from the switch exceeds a certain threshold, retraining will proceed. For this, the model in the oracle is retrained using the desired training data. These can vary, for instance:

- all the training data can be used, with new classifications added;
- old data can be replaced with new data, using different metrics and techniques for replacement;
- only the new data can be used,

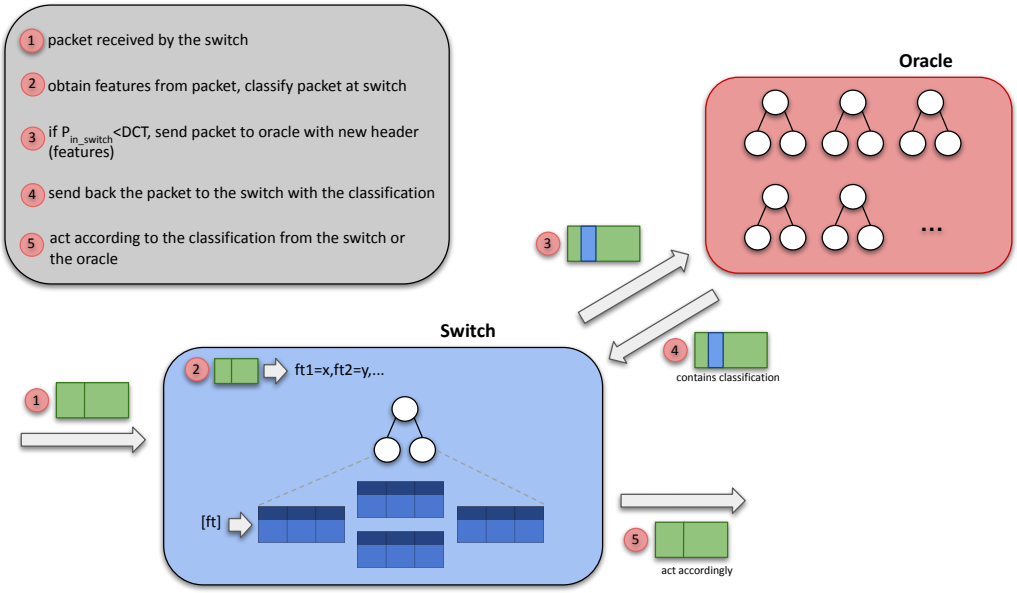


Figure 3.10: An illustration of a packet being sent to the oracle. P_{in_switch} refers to the confidence classification at the switch, and DCT to the threshold for comparing the confidence.

among other options. Once the model is retrained, P4Runtime is used to overwrite the tables with the corresponding retraining information during runtime. This behavior is illustrated in Figure 3.11. Once a packet arrives at the oracle, the corresponding counter is incremented. Then, the class is obtained, and the features and obtained class are saved for retraining. If the counter surpasses the threshold, we retrain the switch via P4Runtime. Note that the focus in this solution is to provide a closed loop, where a switch can make quick decisions with a certain degree of confidence using a simple machine learning model in the data plane. In cases where the confidence level is not sufficiently high, the decision is offloaded to the Oracle, which possesses a more powerful model. The Oracle returns the decision made about the packet to the switch, which acts accordingly. Simultaneously, with the decisions made by the Oracle, the switch is retrained to keep it adaptive to changing traffic. This completes the closed loop. Emphasis is placed on the approach of automating this entire process and communications, rather than seeking the best models, methods, performance, etc. This is beyond the scope of this work and can be considered as future work, given its significant importance. There is also a focus on generating an implementation adaptable to potential improvements.

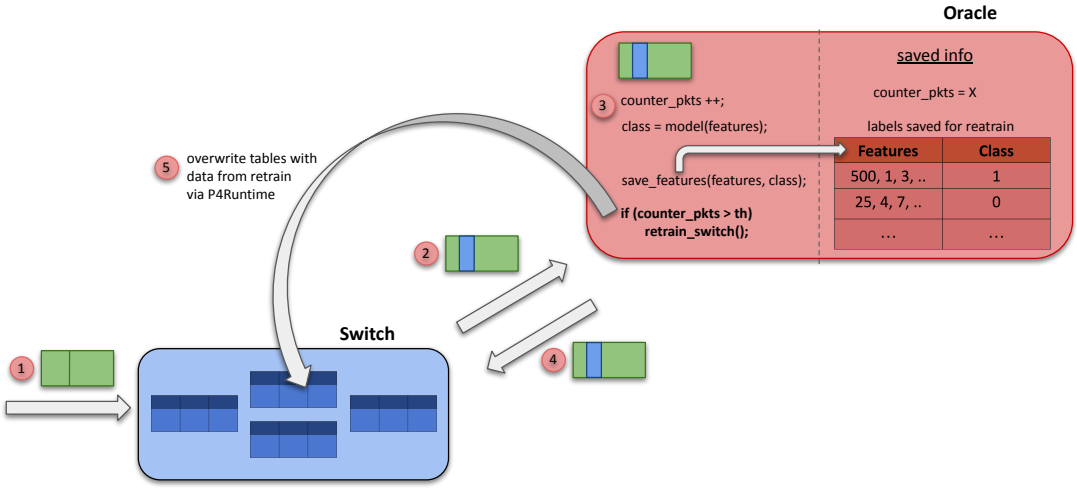


Figure 3.11: An illustration of how the switch is retrained by the oracle.

3.1.4 Evaluation

We implement and deploy HALIDS using the SimpleSwitchGrpc version of BMv2 [29], employing virtual interfaces. There are for virtual interfaces which will be associated with the switch at compilation. One of these interfaces will serve as the port through which incoming packets will arrive. Another will be designated for forwarding packets classified as malware, a third for those classified as normal, and the remaining interface for unclassified packets (non-IPv4 packets).

Traffic traces are injected using TCP-replay[1]. Training and validation of the ML models is done using the well-known UNSW-NB15 dataset, whereas testing is performed on top of actual packet traces from the same dataset. The data training consists of 700.000 data-labeled instances, each one with the 49 features from the dataset, and the corresponding associated attack. The packet trace used corresponds to data captured the day 22/01/2015 between 22-01-2015 11:49:36 and 22-01-2015 12:05:04 UTC. This capture has a total of 1.800.680 packets, with an average of 1900 packets/s approx. This dataset was chosen as it is a well known and widely used dataset, with a substantial

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

amount of data. It reflects modern traffic patterns, including a variety of different attack types and diverse features. It is also the dataset used by SwitchTree to obtain the features for the DT and for testing, so it is used also used in order to compare the results obtained.

It is important to note that the BMv2 is not designed to be a production grade switch, and therefore, it is not ideal to evaluate performance using this setup. Nevertheless, it can be observed that the packet loss is significantly high with such a large trace. Consequently, the debugging mode of the BMv2, which causes substantial packet loss, was disabled. Additionally, the testing environment was transitioned from a virtual machine to a physical machine, significantly reducing packet loss. Then, tests were conducted on a machine with 16GB RAM and 8 CPU cores.

3.1.4.1 Toy example: Small traffic and small model

We begin the analysis of HALIDS performance taking a so-called toy example, using as testing data a small traffic trace of 1.000 packets from the mentioned dataset.

In this study, we evaluate the operation of HALIDS under two distinct configurations. Initially, we confirm that the in-switch student model and the oracle model achieve identical performance when integrating the same model, ensuring the correct implementation of HALIDS. This scenario is called the baseline. The implementation is validated by processing all test packets either through the switch or the oracle. In both cases, the student and oracle machine learning models are trained on the entire training dataset using the same DT architecture with a depth of five levels. The detection of malware flows in this toy example is, in both cases (using only the switch or only the oracle) perfect, meaning 100% of malware flows are detected (in this case 10 malware flows). As of this, the results at packet level for both scenarios are the following: 1. True Positive (TP) packets: 85 2. False Positive (FP): 0 3. False Negative (FN): 99 4. True Negative (TN): 792 Particularly, there are less than 1000 packets, because of non-IPv4 packets. As expected, the results obtained at the switch and oracle are identical, verifying the implementation. This test was very useful, as initially, the results were not the same; there was a small difference between the True Positive and False Negative values. This issue was caused by a common problem in networks—a delayed response from the

oracle. In this case, the solution involved checking whether a flow had already been marked as malware when the oracle's response returned.

Building on the baseline detection performance, we then assess HALIDS as a functional detection system. Here, the in-switch student model is trained with only 50% of the training dataset while maintaining a tree depth of five, whereas the oracle is trained on the full training dataset using a significantly more complex architecture, consisting of 100 trees with a depth of 15. We then evaluate the detection performance at three different decision confidence thresholds (DCTs) of 80%, 90%, and 95%. In brief, a higher DCT results in more packets being classified by the oracle. Setting a confidence threshold of 80%, the results are the following: 1. TP: 61 2. FP: 0 3. FN: 123 4. TN: 792. In the case of setting the threshold to 90%, the TP packets increase to 93, and the FN decrease to 91. Similarly, when setting the threshold to 95%, the TP packets increase to 97, and the FN decrease to 87. This demonstrates a trend towards improved classification as the confidence threshold is raised, thereby delegating decisions to the oracle. Conversely, when lower confidence thresholds are set, such as 80%, all packets are classified at the switch, indicating that the confidence percentage set during training is elevated. Figure 3.12 shows the results, normalized to baseline performance. With DCT set to 80%, all packets are classified in-band at the switch, leading to a significant performance drop of nearly 30% due to the reduced training data compared to the baseline. As the DCT thresholds increase, more packets are sent to the oracle, and detection performance improves compared to the baseline—by approximately 10% for DCT = 90% and nearly 20% for DCT = 95%.

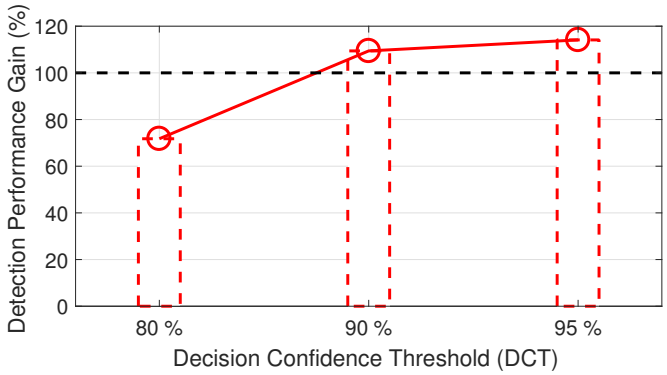


Figure 3.12: Detection performance gain using HALIDS.

3.1.4.2 Multi-model Assessment with Larger Traces

We then evaluate HALIDS using a larger testing dataset, consisting of the 1GB traffic trace mentioned before. Three different model implementations are deployed at the switch:

- model M_1, which is the same single Decision Tree of depth five, used in the toy example, but this time it will be tested with the bigger trace.
- model M_2, a small Random Forest composed of three Decision Trees, each with a depth of five. In this test, we evaluate an actual RF, since in the toy test we only evaluated a DT.
- model M_3, a single Decision Tree with a depth of ten. The goal of tests performed using this scenario is to evaluate how the depth of the tree affects performance, as there are more tables.

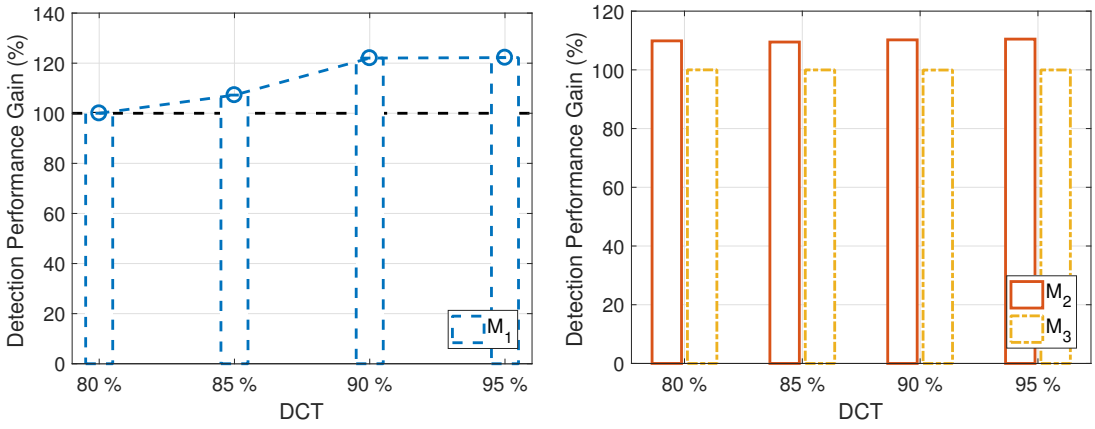
Once again, we train all in-switch models with only 50% of the training data and evaluate HALIDS using various DCT thresholds, including 80%, 85%, 90%, and 95%. The oracle is trained on the full training dataset. The baseline scenario involves classifying all packets at the switch without utilizing the oracle.

The percentage of correctly detected malware flows in the baseline scenario is around 97%. However, as the DCT increases, even though packet-level classification improves, counterintuitively, flow-level classification drops by two to three percentage points, depending on the case. This clearly doesn't make sense, as packet-level classification is improving and both models (switch and oracle) are trained on the same data. The flow detection rate should at least remain constant, because if it does but packet-level metrics improve, it would mean that the same flows were detected as malware with fewer packets. This is because once a packet is detected as malware, all subsequent packets corresponding to that flow are marked as malware.

Upon encountering this issue, we investigated and concluded that it is due to a problem with how flows are identified, meaning the hash. The way flows and packets are measured for correct classification is in the switch, using the hash id (flow id) and prior knowledge of what is malware and what is not. In this case, due to the large trace of flows, this hash mapping quickly becomes overwhelmed, requiring the release of entries for new flows. With so many packets being processed at such high speeds, this creates a delay when packets are sent to the oracle, while simultaneously flow data is being deleted from the

switch. In summary, the issue is that flow data is being overwritten with new flow data (with the same hash ID but different flow), which leads to nonsensical flow-level classification percentages. Attempts were made to fix this, but it was not possible as it requires a much deeper analysis. Some improvements were made, but they are not reflected in these tests. Therefore, all graphs and tests presented later are based on packet-level data.

Figure 3.13 presents the results for the three models. Figure 3.13(a) illustrates that the detection performance of M_1 remains stable at DCT = 80%, but improves by over 20% compared to the in-switch model baseline as the DCT increases, showing even better results than in the toy example.



(a) M_1 model.

(b) M_2 and M_3 models.

Figure 3.13: Detection performance gain for different in-switch models.

Figure 3.13(b) shows the results for M_2 and M_3. For both models, detection performance remains consistent across all DCT thresholds, indicating that they already exhibit strong classification performance and high confidence in their decisions. Notably, M_2 shows an 11% improvement over the baseline. However, M_3 maintains the same performance as the baseline, implying that oracle support has minimal impact. As shown later, approximately 0.3% of the testing samples (about 6,000 packets) are queried in the case of M_2, almost no packets are sent to the oracle when using M_3, which also supports the fact that both cases already have a high performance. Finally, Figure 3.14 presents the $F1$ scores for detecting malware packets for all three models at the highest DCT threshold of 95%. M_1 and M_2 initially perform poorly, but the oracle significantly boosts their performance by 20% and 10%, respectively. M_3 delivers the best results, nearing 100%, which explains the minimal number

of packets sent to the oracle.

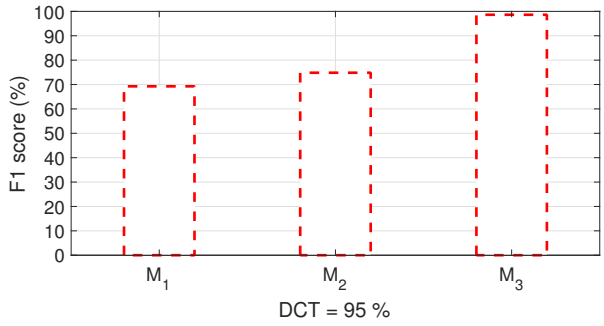


Figure 3.14: Detection performance using HALIDS models and highest DCT.

3.1.4.3 Querying Rate and Packet Loss

Figure 3.15 illustrates the fraction of packets sent to the oracle experienced by HALIDS during the testing of the three models. It is shown that the fraction of packets sent to the oracle is quite small across all three models, remaining below 0.4% for all models and DCT values. As expected, the oracle querying rate increases with higher DCT values. While M_1 and M_2 rely on the oracle for improved classifications, M_3 generates almost no queries when deployed at the switch.

Figure 3.16 illustrates the end-to-end packet loss experienced by HALIDS during the testing of the three models. Figure 3.16(a) demonstrates that packet loss remains consistent across different DCT values and matches the loss rates observed when no oracle is used, indicating that offloading to the oracle has minimal impact on packet loss. Packet losses for M_2 are notably higher due to the increased complexity of implementing the RF in the data plane, where tables are executed sequentially, leading to queue congestion and more packet discards. For completeness, Figure 3.16(b) presents packet loss results from additional tests conducted in a virtualized environment ('@virtual') instead of on physical hardware—all other tests in the paper were performed on a standard laptop with 16GB RAM and 8 CPU cores ('@hardware', as seen in Figure 3.16(b)). Although virtualization is not practical in this context, we observe an impact on resource utilization due to the oracle offloading logic. It is important to note that the BMv2 model is not intended for production-grade switching, making it less suitable for performance evaluation in a virtual

setup. As a practical tip, disabling the debugging mode on the BMv2 switch also helps reduce packet loss.

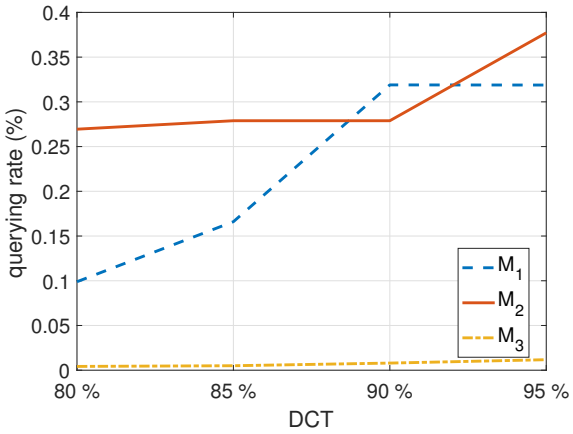
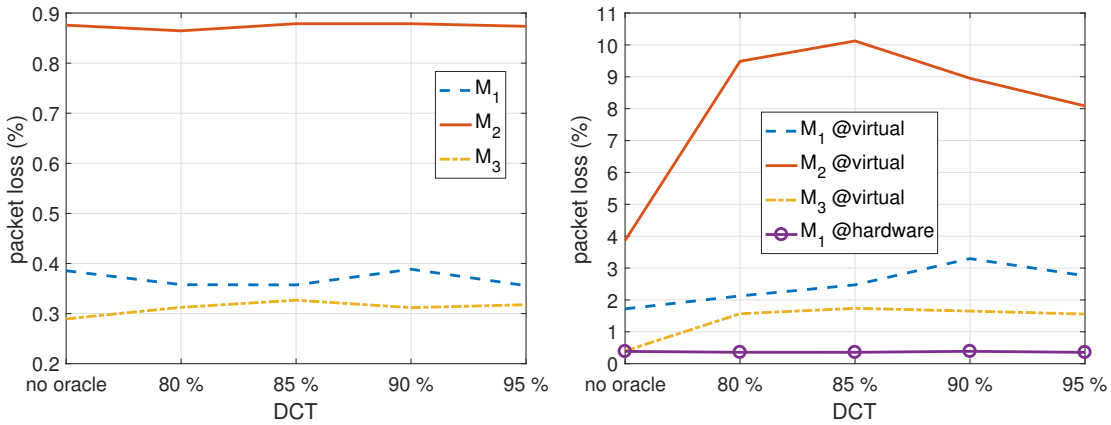


Figure 3.15: Oracle querying rate for the different in-switch models tested in HALIDS. M_1 and M_2 rely on the oracle for better classifications, but querying rates are very low, below 0.4% of the testing data.



(a) Packet loss.

(b) Packet loss compared to virtualized environment.

Figure 3.16: Packet loss for the different in-switch models tested in HALIDS. Packet losses for M_2 are significantly higher, but testing at better hardware deployments strongly reduces degradation.

We also compared the delay incurred when using the oracle, as the packet must be sent to the oracle and returned before taking action on it. First, the test was performed with the Toy Example, where the traffic takes 7.940433505 seconds when not using the oracle, and when forwarding all packets to the oracle, it incurs a time of 8.026289907 seconds. In other words, a difference of

less than 0.1 seconds. For the 1GB traffic capture, it was decided to compare the performance by passing all the traffic through the switch only, and then using a 90% confidence threshold, as one would typically use this threshold and never send 100% of the packets to avoid significant delay. In the first case, the time is 898.719299188 seconds, and in the second case (sending a total of 7,839 packets to the oracle), the incurred time is 899.004010317 seconds, meaning a difference of less than 0.3 seconds. Clearly, this measurement is not accurate, as the switch is not intended to be a production switch, and in a real environment, the controller could be located farther away, adding more latency. Nonetheless, it is interesting to analyze the incurred delay.

3.1.4.4 Model Retraining and Impact in Detection Performance

Concerning the retraining of the in-switch model, we tested a scenario where the M_1 model is trained with only 5% of the training data (approximately 30,000 packets), with the DCT set to 90%. The testing was conducted using the same 1GB trace as in previous evaluations. In this scenario, just over 5,000 packets are sent to the oracle for classification. The oracle receives packets from the switch and, upon reaching 5,000 packets, retrains the switch using the newly generated labels (i.e., oracle classifications). We tested three different retraining strategies: (i) adding the new 5,000 labels to the existing training data, resulting in *+17% new training samples* (increasing the training size to 35,000 packets); (ii) *replacing the first 5,000 labels* in the training dataset with the newest 5,000 classifications from the oracle (keeping the training size at 30,000 packets); and (iii) using only the newly generated 5,000 labels (i.e., *only 17%* of the original training data). The baseline scenario involves processing all packets through the switch without oracle support.

Figure 3.17 presents the results for these three retraining strategies. Incorporating new samples classified by the oracle into the in-switch model retraining significantly enhances detection performance, with an improvement of nearly 50%. The results are similar for both adding or replacing labels. This is a highly encouraging outcome, demonstrating that the HALIDS approach can greatly improve performance when a pre-trained AI/ML model faces high uncertainty in its decisions. A typical example might involve concept drift or the classification of previously unseen data distributions. However, retraining using only the newest samples, which excludes over 80% of the training data,

results in worse-than-baseline performance. Nonetheless, as strategy (ii) suggests, retraining with a broader time window of samples could be the optimal approach.

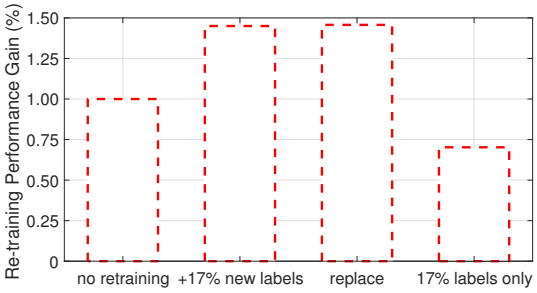


Figure 3.17: Detection performance using HALIDS retraining.

3.1.4.5 Conclusions and Future Work

We developed and evaluated HALIDS, a software prototype for adaptive, in-band AI/ML-IDS using P4, enhanced by off-band oracles that improve model classification and retraining. We conducted various tests, including setups with different resources and different densities of the models, evaluating both the performance of malware packet detection and packet loss, which is also an important factor to consider when deciding whether to use a solution of this kind or not. Overall, the evaluations suggest a promising direction for utilizing the combined in-band and off-band network traffic classification approach, leveraging active learning principles for model retraining.

A closed loop was successfully implemented, providing the network device with "intelligence" beyond the usual. Now, the device can make decisions when confidence thresholds are high, and if not, it can delegate the decisions to an oracle. Additionally, it can improve its future decisions through retraining based on the oracle's decisions.

The emphasis on this work is placed on the approach of automating the entire process and communications, while introducing the re-training capability, enabling dynamic adaptation of the model, rather than seeking the best models, methods, performance, etc. This is beyond the scope of this work and can be considered as future work, given its significant importance. There was also a focus on generating an implementation adaptable to potential improvements.

The toy tests served as proof of concept for HALIDS; still, upon more elaborate testing with more traffic, the solution was evaluated in greater depth,

determining the need to improve the implementation.

Within the future work, there are several possible enhancements. As mentioned earlier, the emphasis in this work was not on these improvements, but rather on ensuring their easy integration into the provided implementation. Particularly, the following clear enhancements are envisioned:

- Evaluate other models for the switch, where considerations must be taken regarding its capabilities, and especially for the Oracle. The decision was made to choose a simple model (but with more elements than the switch model) with the aim of focusing on the closed loop. In particular, the Oracle could feature a considerably more complex model or multiple models, as is the case with CML-IDS [39]. This would be straightforward to integrate since the Oracle is a Python program. For the switch, a model that naturally maps to the pipeline must be identified.
- Send all the features to the Oracle. In particular, one should evaluate the benefit of having these features for the prediction versus the cost of calculating them. Implementing this can be as simple as calculating the features and adding them to the header sent to the Oracle.
- Evaluate other criteria for switch retraining. Currently, a simple criterion is in place, such as the number of received packets, but the possibilities are endless for deciding when to retrain the switch. This is as straightforward as implementing in the Python program the algorithm that determines when to initiate retraining.
- Consider alternative confidence metrics for the RF labels. This is as simple as incorporating this data during the training phase.
- It is straightforward to add levels or quantities of trees both in the switch and the Oracle.
- Evaluate if a better approximation of some features can be achieved by sending different data to the Oracle. This could be due to the fact that the Oracle does not have limitations on operations or data types.

3.2 Experimenting with Netronome - Hardware

As part of a joint research project, it was decided to explore the possibilities of developing P4 code on hardware. Among the options considered, tak-

ing into account constraints such as cost, accessibility, and active support, the Netronome Agilio SmartNICs[1] were identified. As previously mentioned, SmartNICs can be programmed to meet user-specific needs. Specifically, the datapath of the Netronome SmartNICs can be programmed using eBPF, C, and P4.

The Agilio SmartNICs provide high-performance solutions for server-based networking tasks, including network virtualization, security, load balancing, quality of service, and telemetry. These SmartNICs utilize Netronome's Network Flow Processors (NFP-4000 and NFP-6000 series) to achieve their capabilities.

In particular, the chosen SmartNIC is the Agilio CX 2x10GbE SmartNIC. It has 2 10GbE interfaces, with NFP-4000 processors. A PC compatible with the SmartNIC must meet the following requirements:

- A motherboard with a PCIe slot compatible with Gen3 x8.
- Support for Single Root I/O Virtualization (SR-IOV). SR-IOV is a hardware reference that allows a single PCI Express (PCIe) endpoint to be used as multiple independent devices.
- Support for Alternate Routing ID (ARI).
- Virtualization Technology for Directed I/O (VT-d).

As illustrated in the figure 3.18, the Network Flow Processor (NFP) contains several internal blocks designed for networking datapath configuration and programmability. The NFP processing elements are organized into "islands" spread across the chip. The number of islands within an NFP varies depending on the chip model (NFP-4000 or NFP-6000).

The Flow Processing Cores (FPCs) are the primary programmable components of the NFP. Each island containing Flow Processing Cores (FPCs) can be programmed using one or more of the following languages: P4, C, or Microcode. These FPCs can handle packet classification and modification operations that extend beyond basic 5-tuple classification.

NFP-based Agilio SmartNICs support the following programming models:

- Host API-based Programming Model: Utilizing Agilio Software-supported APIs.
- User Datapath Programming Model: C-based programming with configuration APIs.

[1] https://netronome.com/agilio-smartnics/

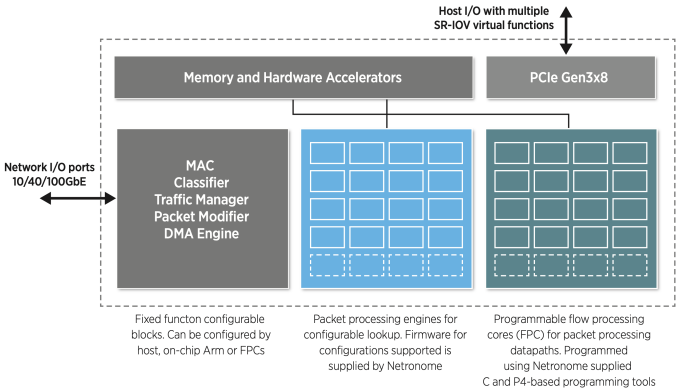


Figure 3.18: NFP programming architecture [73].

- User Datapath Programming Model: P4 and C-based programming with configuration APIs.
- User Datapath Programming Model: Incorporating a C (or P4) sandbox or plug-in application into the Agilio Software datapath.

P4 and C-based Programming with Configuration APIs programming model is designed for users who want to program the datapath in a hardware-agnostic manner, meaning they don't need to understand the specifics of the underlying NFP architecture. The packet-processing model introduced by the P4 language is illustrated in Figure 3.19. Users can write the datapath for a network device in P4 without needing any knowledge of the target hardware. The P4 toolchain, developed by the device vendor, converts the P4 program into device-specific firmware. Additionally, the P4 toolchain generates a runtime API (similar to the OpenFlow model) that allows for modification of match-action tables. To enable extensions to P4-based programmability, Netronome provides the ability to extend P4 datapath features with C-based custom applications. This is also referred to as application of C-based sandbox or plugins to a P4-defined datapath [73].

3.2.1 Netronome Programming Framework

Netronome was the first vendor to adopt P4 as the primary language for programming the SmartNIC data plane. While Netronome supports both $P4_{14}$ and $P4_{16}$ versions, much of the development framework is still centered around $P4_{14}$. The P4 development process on Netronome boards follows an architecture similar to the V1Model reference, enabling seamless translation of P4 code

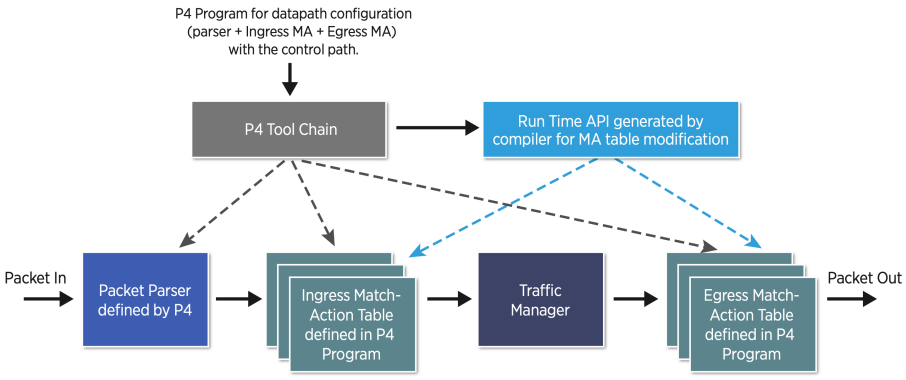


Figure 3.19: Packet processing model using P4 [73].

into the SmartNIC architecture. In addition to P4 programming, Netronome allows users to write low-level Micro-C code, which can be implemented as P4 externs or as standalone data plane applications.

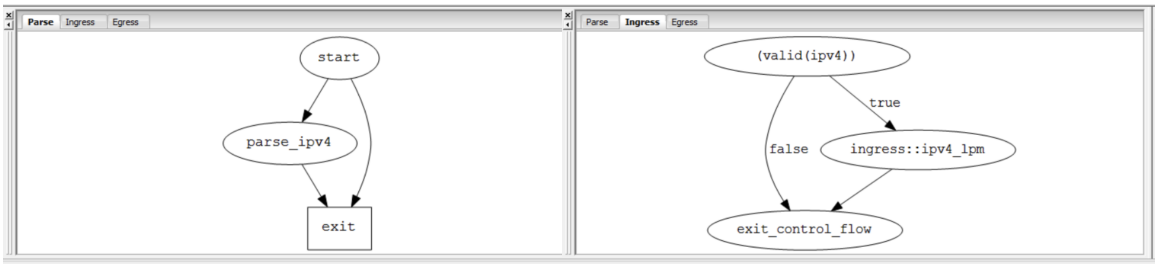
The Netronome SmartNIC is supported by a Linux Run Time Environment (RTE) service, which offers flexible interaction with the SmartNIC application from the operating system level. This allows tasks such as populating Match-Action tables or developing more advanced Control Plane applications.

Besides enabling the data plane firmware to be programmed from scratch, Netronome also provides Linux drivers with native support for eBPF/XDP. This feature allows for hardware programmability when offloading eBPF/XDP instructions to the SmartNIC for execution [61].

3.2.1.1 Compiling, Building and Running a P4 program

The NFP includes a Software Development Kit (SDK) that features a compiler and linker within an Integrated Development Environment (IDE) with a GUI for Windows OS. Additionally, the SDK provides a CLI for the same tasks. Once a P4 program is built, one can see the graphs that represent the parser, ingress and egress at the IDE. Some examples of these are shown in Figures 3.20 (a) and (b), showing the egress and ingress graphs of a l3 basic forwarding program.

After building, the IDE also provides an easy way for writing the tables rules. As seen in Figure 3.21, to the top left are the tables. When selected the rules are displayed at the bottom left (if there are any). When selected, we can write the values for the fields for matching, in this case the field is the ingress port, and the value given for it is port p0. Then below we can select the action



(a) Parser graph of a basic l3 forwarding.

(b) Ingress graph of a basic l3 forwarding.

Figure 3.20: Graphs from different stages of the pipeline created by the Netronome IDE.

to be taken, in this case forward, and fill the corresponding parameters for this action. In this case the only parameter is the port, and the value given is the port v0.0. One can also see other statistics of interest, such as how many packets matched each action, counters from the P4 program, system counters (RX/TX), data from the last packet seen (metadata and headers) and others.

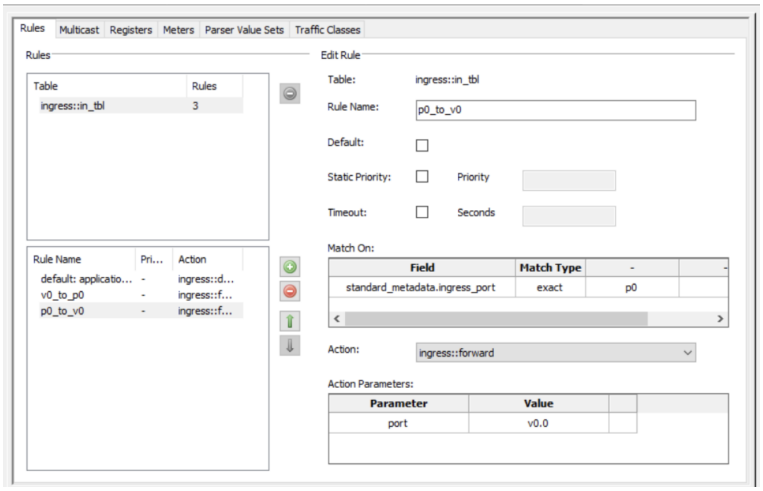


Figure 3.21: Viewing and editing tables in the Netronome IDE.

3.2.2 Testing basic programs

Basic tests were conducted on the hardware due to the adaptation curve required to effectively utilize the SmartNICs. The primary objective is to test some fundamental programs, leaving the evaluation of more complex programs as a task for future work.

The setup for testing these basic programs, consisting of two SmartNICs (with their own pc) connected, is shown in Figure 3.22. Each SmartNIC has two physical ports (p0 and p1 in the figure). Once the firmware is installed on the SmartNIC, the virtual interfaces can be viewed from the host operating system. Note that the number of virtual interfaces can be configured. It is also possible to use the SmartNIC in "non-programmable" mode, in which case the physical interfaces will be visible.

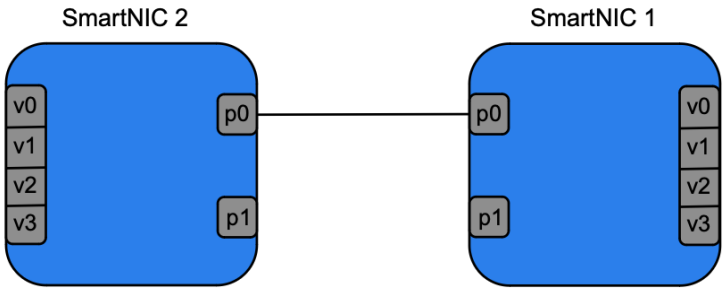


Figure 3.22: Setup for testing basic programs with two SmartNICs connected by physical port p0.

The programs tested included a basic wire program, a l3 basic forwarding, a l2 basic forwarding (namely a hub, a router and a switch). When attempting to develop more complex programs, it was discovered that the P4 architecture used (the SmartNIC employs the same architecture as BMv2, specifically the V1Model architecture) is restricted compared to its normal capabilities. For example, it is not possible to modify values associated with the device's queues, among other limitations. In conjunction with this, and knowing that it was possible to combine P4 code with C code (P4 Sandbox), it was decided to investigate how this integration works. Clearly, writing code in C is more complex than in P4, especially when it comes to accessing packet headers and information. An example of a C program that is called from a P4 program can be seen in the code below.

```
P4 program
control MyIngress(...) {
    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
            // C Sandbox
            primitive_basic();
        }
    }
}
```

Below is the code for the mentioned C program. When the function is called, it retrieves the ipv4 headers, then it accesses the value of the `diffserv` field, modifies it, and then returns to the normal execution of the P4 program. This way, it is possible to cover some functionalities for which the architecture is limited.

```
C program
// Required includes
uint8_t v_global = 4;

int pif_plugin_primitive_basic (EXTRACTED_HEADERS_T *headers, MATCH_DATA_T *match_data){
    PIF_PLUGIN_ipv4_T *ipv4_header;

    if (! pif_plugin_hdr_ipv4_present(headers)){
        return PIF_PLUGIN_RETURN_DROP;
    }

    ipv4_header = pif_plugin_hdr_get_ipv4(headers);
    if (v_global != ipv4_header->diffserv){
        PIF_HEADER_SET_ipv4___diffserv(ipv4_header, v_global);
    }
    return PIF_PLUGIN_RETURN_FORWARD;
}
```

3.2.3 Testing a basic Decision Tree

It was also decided to test a small version of a decision tree in hardware, as a proof of concept. Specifically, to maintain the general idea of HALIDS, a small three-level tree was implemented, which retrieves data from the incoming packet and classifies it. The tree embedded in the pipeline can be seen in Figure 3.23, along with the corresponding tables and rules for each level. At the first level, the protocol (hdr.ipv4.protocol) is evaluated, particularly if it is less than 6. In the case under evaluation (TCP or UDP traffic), this means that TCP packets will proceed to the upper part of the tree, while UDP packets will proceed to the lower part. Then, if the protocol condition evaluates negatively (UDP), the tree reaches a leaf where class 1 (malware) is set. If the protocol condition evaluates positively, the TTL (hdr.ipv4.ttl) is then evaluated, specifically if it is less than 250. If true, the traffic is classified as normal; otherwise, it is classified as malware.

To test the correct functionality of the program, only one SmartNIC was used, creating namespaces for each virtual interface. Traffic, both TCP and

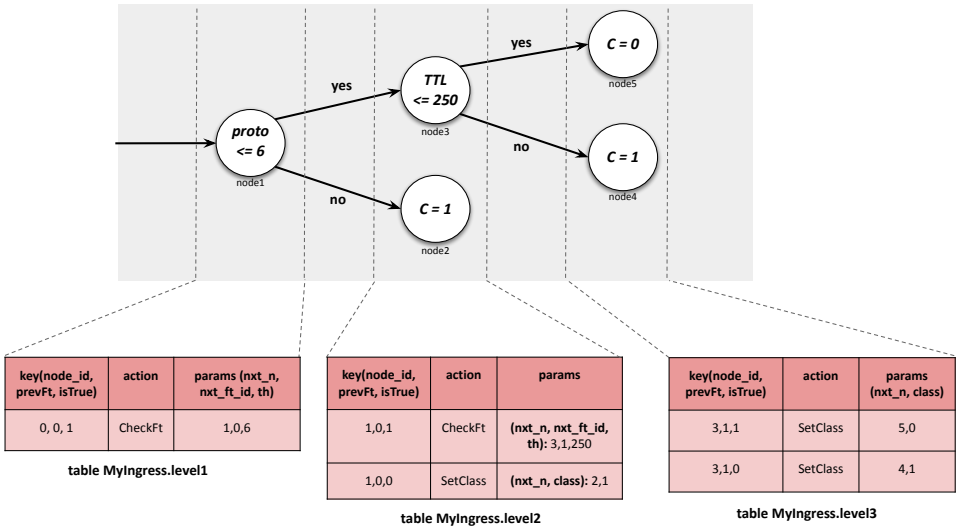


Figure 3.23: Small decision tree implemented in the Netronome SmartNIC.

UDP, was then generated from interface v1 to v0 using the iperf tool[1]. Afterward, a table was added where traffic classified as malware is forwarded out of the switch through interface v3, and normal traffic through interface v2.

As a first test, UDP traffic was generated from interface v1. Figure 3.24 shows the outgoing UDP traffic from interface 1 and how it is forwarded through interface 3, as it is classified as malware.

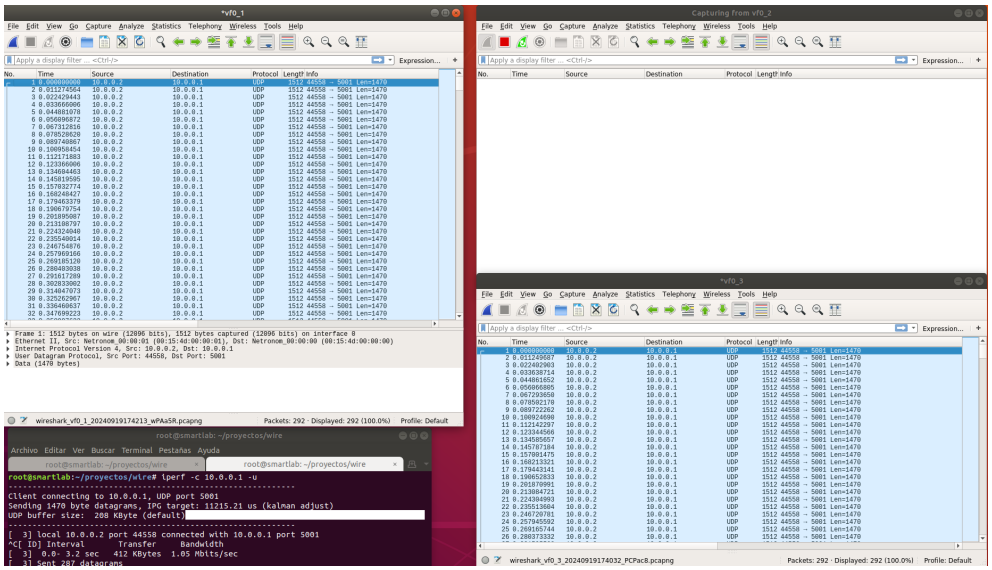


Figure 3.24: Traffic captures on interfaces v1, v2, and v3. Outgoing UDP traffic from interface v1 is forwarded to interface v3 as it is classified as malware.

[1] https://iperf.fr

Then, TCP traffic was generated from interface v1. All these packets have a TTL of 64, meaning it is classified as normal traffic. Figure 3.25 shows the outgoing TCP traffic from interface 1 and how it is forwarded through interface 2.

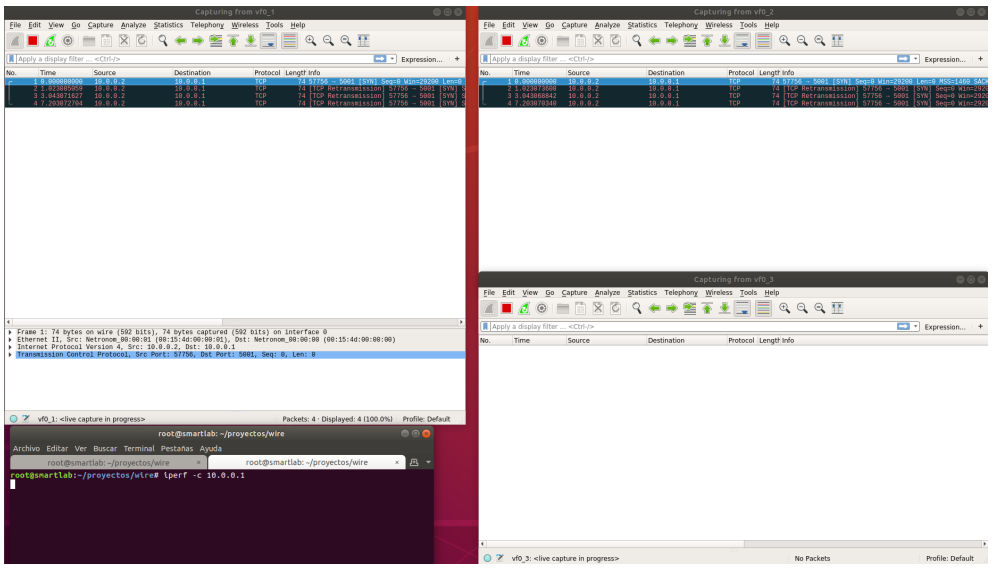


Figure 3.25: Traffic captures on interfaces v1, v2, and v3. Outgoing TCP traffic from interface v1 is forwarded to interface v2 as it is classified as normal traffic, because the TTL does not surpass the predefined threshold.

To test the negative evaluation of the TTL, the corresponding parameter in the table rules was modified. Specifically, the condition was changed to **TTL** $\leq$ **30**. In this case, the traffic should then be classified as malware. Figure 3.26 shows the outgoing TCP traffic from interface 1 and how it is forwarded through interface 3, as it is classified as malware.

Lastly, it was decided to test changing the TTL evaluation parameter at runtime. First, it is evaluated against 250, and then this value is changed to 30. As shown in Figure 3.27, outgoing TCP traffic from interface 1 is first forwarded through interface 2 (as TTL $\leq$ 250 evaluates true), so the traffic is classified as normal. Then, the condition is changed and traffic starts to be forwarded through interface 3 (as TTL $\leq$ 30 evaluates true), so the traffic is classified as malware.

3.2.4 Conclusions and Future Work

The goal of this proof of concept was to successfully execute P4 code, no matter how basic, on hardware. Although P4 promises to be a simple and

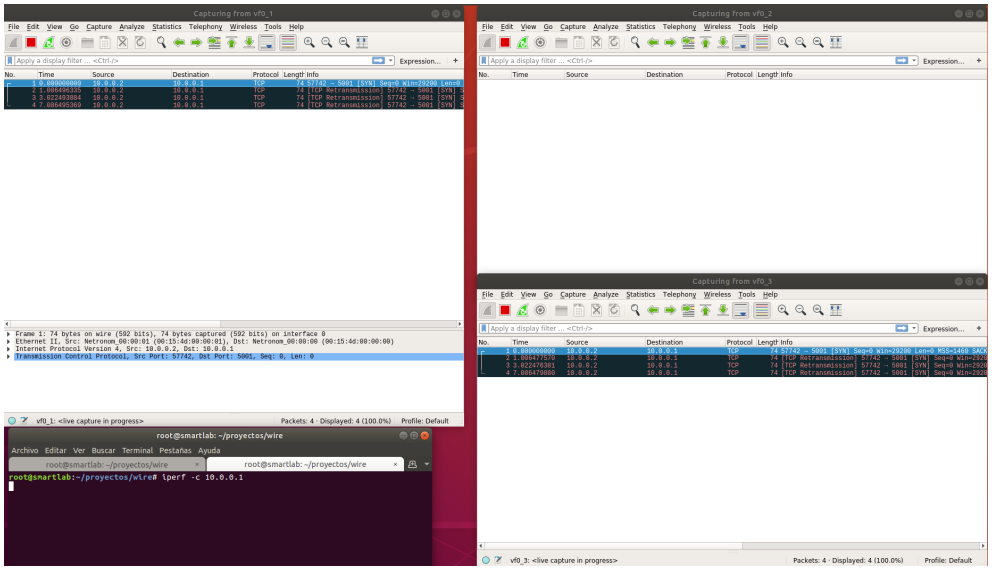


Figure 3.26: Traffic captures on interfaces v1, v2, and v3. Outgoing TCP traffic from interface v1 is forwarded to interface v3 as it is classified as malware, because the predefined threshold for compraing the TTL was lowered.

flexible language tailored to the user's needs, the available targets (especially hardware) are quite limited. Many network devices either do not offer programmability with P4 or only offer partial implementations. This creates a significant gap between what P4 promises and what can actually be achieved on real devices. Additionally, the most promising hardware devices tend to be difficult to access, both in terms of availability and cost. This is why testing on these SmartNICs was promising, as they are relatively affordable and also allow the functionalities of P4 to be complemented with C code.

By performing the basic tests mentioned, it was at least possible to ensure the basic programmability of these SmartNICs with P4. It was also possible to test a small version of a decision tree in hardware, demonstrating how these can be properly adapted to the programmable hardware pipeline. This suggests that it would be feasible to increase the complexity of the features used (to the level of HALIDS complexity), with the main limitation being the architecture of the SmartNIC. Additionally, it was shown that it is indeed possible to change the switch rules at runtime on a real device, observing how these changes take effect on live traffic without needing to restart the device. An interesting observation that emerged from this implementation is that the most complex process is writing tables, which was automated in the full version of HALIDS.

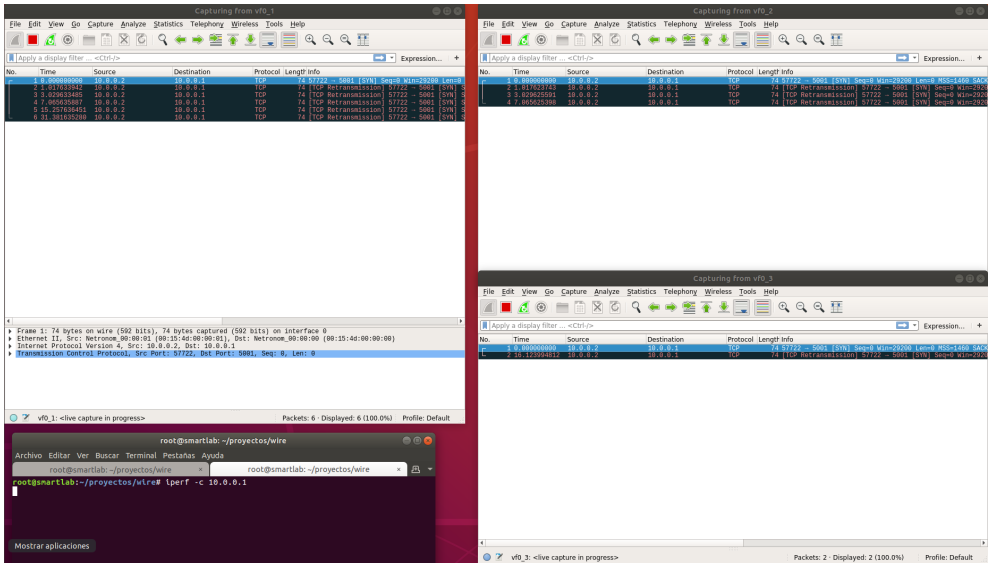


Figure 3.27: Traffic captures on interfaces v1, v2, and v3. Outgoing TCP traffic from interface v1 is first forwarded to interface v2. At runtime the threshold for the TTL was lowered, so then the traffic begins being forwarded to interface v3.

Additionally, once the learning curve of installing the board and running the programs was overcome, we are left with a highly useful IDE, facilitating the easy installation of programs and rules on the board, as well as debugging more complex programs. It was observed that the implemented architecture is restricted, as is often the case with devices that implement programmability with P4. However, it is promising to be able to complement these limitations with C code.

As future work, more complex programs need to be implemented to explore the limitations within the architecture provided by the SmartNIC more thoroughly. Additionally, it is necessary to investigate the possibility of using P4Runtime to potentially implement HALIDS in hardware in the future. Lastly, further testing of the integration with C programs is needed, as the scope of the tests conducted was very limited, focused solely on achieving integration between P4 and C code. It remains to be explored how powerful packet inspection in C can be and to evaluate any performance degradation.

Chapter 4

Final Remarks

Firstly, a thorough survey of network programmability was conducted, along with the technologies and devices that enable it, providing a comprehensive understanding of its potential and the tools necessary for effective implementation. In particular, the analysis of various data plane programming languages led to the conclusion that the best path forward is to use P4. Not only is it the most widely used language, with a strong community backing it, but it is also one of the few, if not the only, that is actively maintained, easy to deploy, and highly functional. Its versatility and widespread adoption make it the preferred choice for the development of modern network applications.

Various applications of data plane programmability were also investigated, including its use in network optimization, implementation of customized security policies, enhancement of traffic performance, and enabling new functionalities such as network segmentation and dynamic resource management. This research highlights the potential of data plane programmability to transform how modern networks are designed and managed.

It was also possible to analyze the different devices that enable network programmability, such as programmable switches, SmartNICs, FPGAs, DPUs, etc., identifying the differences between them, their benefits, and the various applications for which they are most suitable. This analysis provides a clear understanding of how each device can be leveraged to meet different needs and scenarios within network infrastructure.

Various packet acceleration, processing, and filtering techniques such as DPDK, XDP, and BPF were also analyzed, and how they complement network programmability by providing additional capabilities to enhance performance,

flexibility, and efficiency in data handling at the network level. Each of these technologies offers specific advantages that can be used in conjunction with network programmability to optimize packet processing and adapt networks to particular needs.

Lastly, the applications of Machine Learning in networks were analyzed, exploring how these techniques can enhance traffic management, anomaly detection, performance optimization, and automated decision-making. It was highlighted how can Machine Learning be integrated with network programmability to develop smarter and more adaptive solutions that dynamically respond to changing network conditions.

Then, two proof-of-concept demonstrations were presented to illustrate the applicability of data plane programmability. Specifically, in the software prototype, HALIDS was implemented: a prototype for adaptive, in-band AI/ML IDS using P4, enhanced by off-band oracles that improve model classification and retraining. Various tests were conducted to demonstrate its effectiveness, and future work for this prototype was outlined. This proof of concept highlights the power of network programmability, showing how it can empower network devices and decentralize computing. In this way, network devices can contribute to network management, alleviating loads on other devices. A closed-loop system was successfully implemented, adaptable to changes. Additionally, network programmability was complemented by an emerging technique, Machine Learning, opening doors to new possibilities.

Next, a small proof of concept was also successfully implemented in hardware with P4, aimed at bringing some of the proposed ideas into the real world. This test included running basic programs and an additional test to evaluate integration with C code. The full potential of these implementations remains to be determined due to the limitations of the used architecture. A smaller version of a decision tree was successfully tested in hardware, demonstrating its adaptability to the programmable hardware pipeline. This indicates that it is feasible to scale up the complexity of features (to HALIDS levels), with the primary constraint being the SmartNIC architecture. Furthermore, it was confirmed that switch rules can be modified at runtime on an actual device, allowing for real-time changes to be observed in live traffic without requiring a device restart. It will be necessary to assess whether it will eventually be possible to run more complex applications as a whole, such as HALIDS, on this hardware setup.

Network programmability was driven by the growing need for adaptability in today's applications. It provides unprecedented flexibility in network management, allowing users—whether network operators, researchers, enterprises, application developers, among others—to create customized solutions that quickly adapt to ever-changing needs. In addition to enabling near-complete customization, it achieves this without the need to involve manufacturers, reducing time and costs, and often improving performance.

Combining Machine Learning techniques with data plane programmability introduces new opportunities for network automation and real-time adaptation. This synergy promises the development of more intelligent networks that can learn and adjust to shifting conditions instantly, enhancing both their resilience and operational efficiency. In conclusion, data plane programmability not only revolutionizes network infrastructure but also establishes a new benchmark for innovation and efficiency in network management, becoming a vital element for the future of telecommunications and distributed computing.

However, it is necessary to provide an objective critique of both network programmability and the P4 language in particular. Network programmability faces significant limitations due to the lack of the devices that enable it. Programming these devices without high-level programming languages involves considerable difficulty and a steep learning curve. Therefore, it is ideal to use programmable devices that support data plane programming languages. However, these devices are limited in number and have restricted support.

In the specific case of P4, there is a notable gap between the promised capabilities and what can actually be achieved. Many devices do not implement or only partially implement P4 architectures, and several that previously did have ceased support and distribution. The most promising devices are often very expensive and difficult to access, due to restrictive licenses, high associated costs, or limited availability in smaller markets. Given this clear lack of maturity in real-world devices, using SmartNICs, FPGAs, and similar devices for network programming is considered as an alternative, taking into account the associated difficulties and limitations of these options.

Although network programmability and P4 offer a promising vision for the future of networking, they also face a number of significant challenges. Collaboration between academia and industry is essential to realize the effective implementation of these technologies. Academia can contribute through research and the development of new techniques and approaches, while industry

can provide the infrastructure, resources, and practical knowledge needed to implement and scale these solutions. Together, they can address current limitations, advance standardization, and facilitate the widespread adoption of network programmability, thus achieving a significant evolution in how networks are designed and managed.

Bibliography

[1] Aristide T.-J. Akem, Michele Gucciardo, and Marco Fiore. "Flowrest: Practical Flow-Level Inference in Programmable Switches with Random Forests". In: *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications* (2023), pp. 1–10. URL: https://api.semanticscholar.org/CorpusID:260237623.

[2] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.

[3] Carolyn Anderson et al. "NetKAT: Semantic Foundations for Networks". In: vol. 49. Jan. 2014, pp. 113–126. DOI: 10.1145/2578855.2535862.

[4] gRPC Authors. *Introduction to gRPC*. https://grpc.io/docs/what-is-grpc/introduction/. [Online; Accessed: July 2024].

[5] Ilya Baldin et al. "FABRIC: A National-Scale Programmable Experimental Network Infrastructure". In: *IEEE Internet Computing* 23.6 (2019), pp. 38–47. DOI: 10.1109/MIC.2019.2958545.

[6] Tom Barbette, Cyril Soldani, and Laurent Mathy. "Fast Userspace Packet Processing". English. In: *Proceedings of ANCS 2015*. F.R.S.-FNRS - Fonds de la Recherche Scientifique. Oakland, United States - California, 2015. DOI: 10.1109/ANCS.2015.7110116.

[7] Hao Bi and Zhao-Hun Wang. "DPDK-based Improvement of Packet Forwarding". In: *ITM Web of Conferences* 7 (Jan. 2016), p. 01009. DOI: 10.1051/itmconf/20160701009.

[8] Giuseppe Bianchi et al. "OpenState: Programming Platform-Independent Stateful Openflow Applications inside the Switch". In: *SIGCOMM Comput. Commun. Rev.* 44.2 (May 2014), pp. 44–51. ISSN: 0146-4833. DOI: 10.1145/2602204.2602211. URL: https://doi.org/10.1145/2602204.2602211.

[9] Roberto Bifulco and Gábor Rétvári. "A Survey on the Programmable Data Plane: Abstractions, Architectures, and Open Problems". In: June 2018, pp. 1–7. DOI: 10.1109/HPSR.2018.8850761.

[10] Julien Boite et al. "Statesec: Stateful monitoring for DDoS protection in software defined networks". In: *2017 IEEE Conference on Network Softwarization (NetSoft)* (2017), pp. 1–9.

[11] Pat Bosshart et al. "P4: Programming Protocol-Independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: https://doi.org/10.1145/2656877.2656890.

[12] R. Boutaba et al. "A Comprehensive Survey on Machine Learning for Networking: Evolution, Applications and Research Opportunities". In: *Journal of Internet Services and Applications* 9 (May 2018). DOI: 10.1186/s13174-018-0087-2.

[13] B. Brandino, P. Casas, and E. Grampin. "Detecting Attacks at Switching Speed: AI/ML and Active Learning for in-Network Monitoring in Data Planes". In: *The 32nd IEEE International Conference on Network Protocols. The Workshop on Intelligent Classification of High-Speed Network Traffic (ICNT)*. 2024 in press, pp. 1–6.

[14] B. Brandino et al. "HALIDS: a Hardware-Assisted Machine Learning IDS for in-Network Monitoring". In: *2024 8th Network Traffic Measurement and Analysis Conference (TMA)*. 2024, pp. 1–4. DOI: 10.23919/TMA62044.2024.10559083.

[15] Belén Brandino. "Baja latencia y alta performance en datacenters: el protocolo NDP". Proyecto Taller de Sistemas Ciber Físicos. Universidad de la República (Uruguay). Facultad de Ingeniería, 2020. URL: https://eva.fing.edu.uy/pluginfile.php/324214/mod_folder/content/0/Baja%20latencia%20y%20alta%20performance%

20en%20datacenters%20-%20el%20protocolo%20NDP%20-%20BelenBrandino%20-%20202002.pdf?forcedownload=1.

[16] Belén Brandino. "Dispositivos de Red Programables". Tesis de grado. Universidad de la República (Uruguay). Facultad de Ingeniería, 2022. URL: https://hdl.handle.net/20.500.12008/31562.

[17] Belén Brandino. *HALIDS: a Hardware-Assisted Machine Learning IDS for in-Network Monitoring)*. https://gitlab.fing.edu.uy/bbrandino/halids. [Online; Accessed: October 2024].

[18] Coralie Busse-Grawitz et al. "pForest: In-Network Inference with Random Forests". In: *CoRR* abs/1909.05680 (2019). arXiv: 1909.05680. URL: http://arxiv.org/abs/1909.05680.

[19] Antonio Capone et al. "Detour planning for fast and reliable failure recovery in SDN with OpenState". In: *2015 11th International Conference on the Design of Reliable Communication Networks (DRCN)*. 2015, pp. 25–32. DOI: 10.1109/DRCN.2015.7148981.

[20] Carmelo Cascone et al. "SPIDER: Fault resilient SDN pipeline with recovery delay guarantees". In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. 2016, pp. 296–302. DOI: 10.1109/NETSOFT.2016.7502425.

[21] Jeffrey D. Case et al. *Simple Network Management Protocol (SNMP)*. STD 15. http://www.rfc-editor.org/rfc/rfc1157.txt. RFC Editor, May 1990. URL: http://www.rfc-editor.org/rfc/rfc1157.txt.

[22] Cilium. *BPF and XDP Reference Guide*. https://docs.cilium.io/en/stable/bpf/. [Online; Accessed: July 2024].

[23] P4 Language Consortium. *$P4_{16}$ Language Specification*. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html. [Online; Accessed: July 2024].

[24] P4 Language Consortium. *BEHAVIORAL MODEL (bmv2)*. https://github.com/p4lang/behavioral-model. [Online; Accessed: July 2024].

[25] P4 Language Consortium. *P4 Language Tutorial*. http://bit.ly/p4d2-2018-spring. [Online; Accessed: July 2024].

[26] P4 Language Consortium. *p4c*. https://github.com/p4lang/p4c. [Online; Accessed: July 2024].

[27] P4 Language Consortium. *P4Runtime Specification*. https://github.com/p4lang/p4runtime. [Online; Accessed: July 2024].

[28] P4 Language Consortium. *Packet IO*. https://github.com/p4lang/p4runtime-shell/blob/main/usage/packet_io.md. [Online; Accessed: July 2024].

[29] P4 Language Consortium. *SimpleSwitchGrpc - a version of SimpleSwitch with P4Runtime support*. https://github.com/p4lang/behavioral-model/blob/main/targets/simple_switch_grpc/README.md. [Online; Accessed: March 2024].

[30] P4 Language Consortium. *The P4 Language Specification*. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf. [Online; Accessed: July 2024].

[31] P4 Language Consortium. *v1model.p4*. https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4. [Online; Accessed: July 2024].

[32] The P4 Language Consortium. *P4 Portable NIC Architecture (PNA)*. https://p4.org/p4-spec/docs/PNA.html. [Online; Accessed: July 2024].

[33] Google Developers. *Protocol Buffers*. https://developers.google.com/protocol-buffers. [Online; Accessed: July 2024].

[34] doc.dpdk.org. *Overview*. https://doc.dpdk.org/guides-16.04/prog_guide/overview.html. [Online; Accessed: July 2024].

[35] Mazdak Fatahi et al. *Open Source Routers: A Survey*. 2022. arXiv: 2203.01701 [cs.NI].

[36] FD.io. *The Technology Behind FD.io*. https://fd.io/technology/. [Online; Accessed: July 2024].

[37] FD.io. *VPP/What is VPP?* https://wiki.fd.io/view/VPP/What_is_VPP%3F. [Online; Accessed: July 2024].

[38] FD.io. *What is the Vector Packet Processor (VPP)*. https://s3-docs.fd.io/vpp/23.06/index.html. [Online; Accessed: July 2024].

[39] Pegah Golchin et al. "CML-IDS: Enhancing Intrusion Detection in SDN Through Collaborative Machine Learning". In: *2023 19th International Conference on Network and Service Management (CNSM)*. 2023, pp. 1–9. DOI: 10.23919/CNSM59352.2023.10327863.

[40] Intel Corporation— Data Center Group, Samsung Electronics - Networks Business, and Samsung Research. *Samsung Achieves 305 Gbps on 5G UPF Core Utilizing Intel® Architecture*. Tech. rep. 2020.

[41] The P4.org API Working Group. *P4Runtime Specification*. https://p4lang.github.io/p4runtime/spec/main/P4Runtime-Spec.pdf. [Online; Accessed: July 2024].

[42] The P4.org Architecture Working Group. *$P4_{16}$ Portable Switch Architecture (PSA)*. https://p4.org/p4-spec/docs/PSA.html. [Online; Accessed: July 2024].

[43] Sangjin Han et al. *SoftNIC: A Software NIC to Augment Hardware*. Tech. rep. UCB/EECS-2015-155. EECS Department, University of California, Berkeley, May 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html.

[44] Mark Handley et al. "Re-architecting datacenter networks and stacks for low latency and high performance". In: *SIGCOMM* (2017), pp. 29–42.

[45] F. Hauser et al. "A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research". In: *ArXiv* abs/2101.10632 (2021).

[46] F. Hauser et al. "A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research". In: *ArXiv* abs/2101.10632 (2021).

[47] Anxiao He et al. "Hummingbird: Dynamic Path Validation With Hidden Equal-Probability Sampling". In: *IEEE Transactions on Information Forensics and Security* 18 (2023), pp. 1268–1282. DOI: 10.1109/TIFS.2023.3236806.

[48] Stephen Ibanez et al. "The P4->NetFPGA Workflow for Line-Rate Packet Processing". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 1–9. ISBN: 9781450361378. DOI: 10.1145/3289602.3293924. URL: https://doi.org/10.1145/3289602.3293924.

[49] SUN MICROSYSTEMS INC. *NIT(4P); SunOS Reference Manual*. http://www.bitsavers.org/pdf/sun/sunos/4.1/800-3827-10A_SunOS_Reference_Manual_Vol1_199003.pdf. [Online; Accessed: July 2024].

[50] Intel. *Intel® Tofino™ 2*. https://www.intel.es/content/www/es/es/products/network-io/programmable-ethernet-switch/tofino-2-series.html. [Online; Accessed: July 2024].

[51] Syed Usman Jafri et al. "Leo: Online ML-based Traffic Classification at Multi-Terabit Line Rate". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, May 2024, pp. 1573–1591. ISBN: 978-1-939133-39-7. URL: https://www.usenix.org/conference/nsdi24/presentation/jafri.

[52] Chih-Heng Ke and Shih-Jung Hsu. "Load Balancing Using P4 in Software-Defined Networks". In: *Journal of Internet Technology* 21 (2020), pp. 1671–1679. URL: https://api.semanticscholar.org/CorpusID:230656678.

[53] Elie F. Kfoury et al. "A Comprehensive Survey on SmartNICs: Architectures, Development Models, Applications, and Research Directions". In: *IEEE Access* 12 (2024), pp. 107297–107336. DOI: 10.1109/ACCESS.2024.3437203.

[54] Somayeh Kianpisheh and Tarik Taleb. "A Survey on In-network Computing: Programmable Data Plane And Technology Specific Applications". In: *IEEE Communications Surveys & Tutorials* PP (Jan. 2022), pp. 1–1. DOI: 10.1109/COMST.2022.3213237.

[55] Eddie Kohler et al. "The Click Modular Router". In: *ACM Trans. Comput. Syst.* 18.3 (Aug. 2000), pp. 263–297. ISSN: 0734-2071. DOI: 10.1145/354871.354874. URL: https://doi.org/10.1145/354871.354874.

[56] James Kurose. *Redes de computadoras*. Ciudad de México: Pearson Educación, 2017. ISBN: 978-84-9035-528-2.

[57] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. 7th ed. Boston, MA: Pearson, 2016. ISBN: 978-0-13-359414-0.

[58] The Berkeley NetSys Lab. *BESS Overview*. https://github.com/NetSys/bess/wiki/BESS-Overview. [Online; Accessed: July 2024].

[59] Sándor Laki et al. "P4Pi: P4 on Raspberry Pi for Networking Education". In: *SIGCOMM Comput. Commun. Rev.* 51.3 (July 2021), pp. 17–21. ISSN: 0146-4833. DOI: 10.1145/3477482.3477486. URL: https://doi.org/10.1145/3477482.3477486.

[60] Jong-Hyouk Lee and Kamal Singh. "SwitchTree: In-network Computing and Traffic Analyses with Random Forests". In: *Neural Computing and Applications* (Nov. 2020). DOI: 10.1007/s00521-020-05440-2.

[61] M. Caggiani Luizelli et al. "SmartNICs: The Next Leap in Networking". In: *SMARTNICS AT NETSOFT AND SBRC 2024*. Brazil: IEE, 2024.

[62] Pilar Manzanares-Lopez, Juan Muñoz-Gea, and Josemaria Malgosa. "Passive In-Band Network Telemetry Systems: The Potential of Programmable Data Plane on Network-Wide Telemetry". In: *IEEE Access* PP (Jan. 2021), pp. 1–1. DOI: 10.1109/ACCESS.2021.3055462.

[63] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-Level Packet Capture". In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX'93. San Diego, California: USENIX Association, 1993, p. 2.

[64] Nick McKeown et al. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: https://doi.org/10.1145/1355734.1355746.

[65] J. Mogul, R. Rashid, and M. Accetta. "The packer filter: an efficient mechanism for user-level network code". In: *SIGOPS Oper. Syst. Rev.* 21.5 (Nov. 1987), pp. 39–51. ISSN: 0163-5980. DOI: 10.1145/37499.37505. URL: https://doi.org/10.1145/37499.37505.

[66] Gordon E. Moore. "Progress in digital integrated electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]" In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 36–37. DOI: 10.1109/N-SSC.2006.4804410.

[67] Robert Morris et al. “The Click modular router”. In: *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*. SOSP ’99. Charleston, South Carolina, USA: Association for Computing Machinery, 1999, pp. 217–231. ISBN: 1581131402. DOI: 10.1145/319151.319166. URL: https://doi.org/10.1145/319151.319166.

[68] Masoud Moshref et al. “Flow-Level State Transition as a New Switch Primitive for SDN”. In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN ’14. Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 61–66. ISBN: 9781450329897. DOI: 10.1145/2620728.2620729. URL: https://doi.org/10.1145/2620728.2620729.

[69] Nour Moustafa, Gideon Creech, and Jill Slay. “Big Data Analytics for Intrusion Detection System: Statistical Decision-Making Using Finite Dirichlet Mixture Models”. In: *Data Analytics and Decision Support for Cybersecurity: Trends, Methodologies and Applications*. Ed. by Iván Palomares Carrascosa, Harsha Kumara Kalutarage, and Yan Huang. Cham: Springer International Publishing, 2017, pp. 127–156. ISBN: 978-3-319-59439-2. DOI: 10.1007/978-3-319-59439-2_5. URL: https://doi.org/10.1007/978-3-319-59439-2_5.

[70] Nour Moustafa and Jill Slay. “The evaluation of Network Anomaly Detection Systems: Statistical analysis of the UNSW-NB15 data set and the comparison with the KDD99 data set”. In: *Information Security Journal: A Global Perspective* 25.1-3 (2016), pp. 18–31. DOI: 10.1080/19393555.2015.1125974. eprint: https://doi.org/10.1080/19393555.2015.1125974. URL: https://doi.org/10.1080/19393555.2015.1125974.

[71] Nour Moustafa and Jill Slay. “UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)”. In: *2015 Military Communications and Information Systems Conference (MilCIS)*. 2015, pp. 1–6. DOI: 10.1109/MilCIS.2015.7348942.

[72] NetFPGA. *SimpleSumeSwitch Architecture (v1.2.1 and Earlier)*. https://github.com/NetFPGA/P4-NetFPGA-public/wiki/SimpleSumeSwitch-Architecture-(v1.2.1-and-Earlier). [Online; Accessed: July 2024].

[73] Inc. Netronome Systems. *Programming Netronome Agilio® SmartNICs*. Tech. rep. Netronome Systems, Inc., 2018.

[74] Orange. *P4rt-OVS: Programming Protocol-Independent, Runtime Extensions for Open vSwitch using P4*. https://github.com/Orange-OpenSource/p4rt-ovs. [Online; Accessed: July 2024].

[75] Tomasz Osiński. *P4-OvS - Bringing the power of P4 to OvS!* https://github.com/osinstom/P4-OvS. [Online; Accessed: July 2024].

[76] Tomasz Osiński. *p4c-ubpf: a New Back-end for the P4 Compiler*. https://opennetworking.org/news-and-events/blog/p4c-ubpf-a-new-back-end-for-the-p4-compiler/. [Online; Accessed: July 2024].

[77] Shoumik Palkar et al. "E2: A Framework for NFV Applications". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: Association for Computing Machinery, 2015, pp. 121–136. ISBN: 9781450338349. DOI: 10.1145/2815400.2815423. URL: https://doi.org/10.1145/2815400.2815423.

[78] Aurojit Panda et al. "NetBricks: Taking the V out of NFV". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 203–216. ISBN: 9781931971331.

[79] Salvatore Pontarelli et al. "Flowblaze: Stateful Packet Processing in Hardware". In: *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*. NSDI'19. Boston, MA, USA: USENIX Association, 2019, pp. 531–547. ISBN: 9781931971492.

[80] Pluginized Protocols. *xBGP: making BGP truly extensible*. https://pluginized-protocols.org/xbgp/. [Online; Accessed: July 2024].

[81] Y. Rekhter, T. Li, and S. Hares. *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271. http://www.rfc-editor.org/rfc/rfc4271.txt. RFC Editor, Jan. 2006. URL: http://www.rfc-editor.org/rfc/rfc4271.txt.

[82] Luigi Rizzo. "Netmap: A Novel Framework for Fast Packet I/O". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, p. 9.

[83] Mohanad Sarhan et al. “NetFlow Datasets for Machine Learning-Based Network Intrusion Detection Systems”. In: *Big Data Technologies and Applications*. Springer International Publishing, 2021, pp. 117–135. ISBN: 9783030728021. DOI: 10.1007/978-3-030-72802-1_9. URL: http://dx.doi.org/10.1007/978-3-030-72802-1_9.

[84] Burr Settles. “Active Learning Literature Survey”. In: (July 2010).

[85] Michael Seufert et al. “Marina: Realizing ML-Driven Real-Time Network Traffic Monitoring at Terabit Scale”. In: *IEEE Transactions on Network and Service Management* PP (Jan. 2024), pp. 1–1. DOI: 10.1109/TNSM.2024.3382393.

[86] Muhammad Shahbaz et al. “PISCES: A Programmable, Protocol-Independent Software Switch”. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM ’16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 525–538. ISBN: 9781450341936. DOI: 10.1145/2934872.2934886. URL: https://doi.org/10.1145/2934872.2934886.

[87] Anirudh Sivaraman et al. “Packet Transactions: High-Level Programming for Line-Rate Switches”. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM ’16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 15–28. ISBN: 9781450341936. DOI: 10.1145/2934872.2934900. URL: https://doi.org/10.1145/2934872.2934900.

[88] Haoyu Song. “Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane”. In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN ’13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 127–132. ISBN: 9781450321785. DOI: 10.1145/2491185.2491190. URL: https://doi.org/10.1145/2491185.2491190.

[89] SPAN. *BESS Berkeley Extensible Software Switch*. http://span.cs.berkeley.edu/bess.html. [Online; Accessed: July 2024].

[90] MICHELE DI STEFANO. *BERKELEY PACKET FILTER: theory, practice and perspectives*. https://amslaurea.unibo.it/19622/1/berkeleypacketfilter_distefano.pdf. [Online; Accessed: July 2024].

[91] Marcos A. M. Vieira et al. “Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications”. In: *ACM Comput. Surv.* 53.1 (Feb. 2020). ISSN: 0360-0300. DOI: 10.1145/3371038. URL: https://doi.org/10.1145/3371038.

[92] Thomas Wirtgen et al. “The Case for Pluginized Routing Protocols”. In: *2019 IEEE 27th International Conference on Network Protocols (ICNP)*. 2019, pp. 1–12. DOI: 10.1109/ICNP.2019.8888065.

[93] Andrej Yemelianov. *Introduction to DPDK: Architecture and Principles*. https://selectel.ru/blog/en/2016/11/24/introduction-dpdk-architecture-principles/. [Online; Accessed: July 2024].

[94] Chaoqun You et al. “Hierarchical Multiresource Fair Queueing for Packet Processing”. In: *IEEE Transactions on Network and Service Management* 20.1 (2023), pp. 726–740. DOI: 10.1109/TNSM.2022.3197747.

[95] Yang Zhang et al. “ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining”. In: *Proceedings of the Symposium on SDN Research* (2017). URL: https://api.semanticscholar.org/CorpusID:3966367.

[96] Changgang Zheng et al. “Planter: Rapid Prototyping of In-Network Machine Learning Inference”. In: *SIGCOMM Comput. Commun. Rev.* 54.1 (Aug. 2024), pp. 2–21. ISSN: 0146-4833. DOI: 10.1145/3687230.3687232. URL: https://doi.org/10.1145/3687230.3687232.

[97] Guangmeng Zhou et al. “An Efficient Design of Intelligent Network Data Plane”. In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6203–6220. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/zhou-guangmeng.