FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Accelerating network security tools using DPDK infrastructure
Student:	Filip Biľ
Supervisor:	Ing. Tomáš Čejka, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2023/2024

Instructions

Study modern technologies for high-speed network traffic monitoring and other security applications. Additionally, focus on Data Plane Development Kit (DPDK).
Study state-of-the-art applications that support DPDK to receive network packets at high-speed, such as ipfixprobe (flow exporter) [1] or Suricata (Intrusion Detection System) [2].
Design a software infrastructure based on DPDK that will filter (using Access Control Lists - ACL in DPDK) and distribute packets among a set of worker servers at full speed.
The aim is to distribute data load among multiple servers that run a performance-demanding security application so that the results would not be disrupted due to the data splitting, as it was discussed in the paper [3].
Implement the designed infrastructure including the documentation.
Evaluate the results using virtualized or physical infrastructure in cooperation with the supervisor.

[1] https://github.com/cesnet/ipfixprobe
[2] https://suricata.io/
[3] T. Čejka and M. Žádník, "Preserving Relations in Parallel Flow Data Processing," in Security of Networks and Services in an All-Connected World: 11th International Conference on Autonomous Infrastructure, Management, and Security, AIMS 2017.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 9 November 2022 in Prague.

Bachelor's thesis

ACCELERATING NETWORK SECURITY TOOLS USING DPDK INFRASTRUCTURE

Filip Biľ

Faculty of Information Technology
Department of Computer Systems
Supervisor: Ing. Tomáš Čejka, Ph.D.
May 6, 2023

Contents

List of Figures

List of Tables

List of Code Listings

I would like to express my great gratitude to those who have supported me throughout the creation of my bachelor thesis. First and foremost, I would like to thank my thesis tutor Ing. Tomáš Čejka, Ph.D. for his guidance, encouragement, ingenious feedback and constructive criticism. Secondly, I would like to thank to my parents for their invaluable support during my studies at the university. Lastly, I would like to thank the university that has provided me with the access to the knowledge and skills mandatory for shaping the direction of my studies and consequentially, this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 6, 2023 .....

Abstract

This thesis explores the problem of high-speed packet flow pre-filtering and processing on OSI layers 2 and 3 for the network security applications. The Data Plane Development Kit (DPDK) framework was chosen for the implementation of the proof-of-concept infrastructure. The goal of this thesis is to design an infrastructure that will solve the problem of preserving packet flows in the load-balancing between the specific network security applications in the high-speed networks in order of ∼100GiB with the access control lists (ACL). Using the DPDK infrastructure, desired speeds were reached with negligible packet drop rates. The results of this thesis enable to further design high-speed network load-balancers that will preserve the packet flows between the network security applications, thus helping these applications to more accurately analyze packet flows. Source codes used for the prefiltering infrastructure can be found as an attachment to this thesis, together with a sample packet flow generator.

Keywords DPDK, packet flow monitoring, network traffic pre-filtering, Suricata, ipfixprobe

Abstrakt

V tejto práci je skúmaná problematika spracovávania a predfiltrovania vysokorýchlostných sieťových tokov na vrstvách 2 a 3 OSI modelu pre bezpečnostné aplikácie. Pre implementáciu *proof-of-concept* infraštruktúry je použitá knižnica Data Plane Development Kit (DPDK). Účelom tejto práce je navhrnúť infraštruktúru, ktorá bude riešiť problém zachovávania tokov paketov pri rozkladaní záťaže medzi jednotlivé bezpečnostné aplikácie aj vo vysokorýchlostných sieťach v rádoch ∼100GiB s funkčnými filtrovacími pravidlami (ACL). Pomocou tejto DPDK infraštruktúry boli dosiahnuté požadované rýchlosti so zanedbateľnou *drop rate* paketov. Výsledky tejto práce umožňujú navrhnúť vysokorýchlostné sieťové zariadenia na vyvažovanie záťaže, ktoré budú zachovávať toky paketov medzi jednotlivé bezpečnostné aplikácie, čím pomôžu daným aplikáciám pri korektnejšej analýze tokov paketov. V prílohe sú uvedené kompletné zdrojové kódy pre predfiltrovaciu infraštruktúru spolu s jednoduchým generátorom sieťových tokov.

Kľúčové slová DPDK, monitorovanie tokov paketov, predfiltrovanie sieťovej premávky, Suricata, ipfixprobe

List of Abbreviations

API	Application Programming Interface
CLI	Command Line Interface
DPDK	Data Plane Development Kit
EAL	Environment Abstraction Layer
I/O	Input/Output
IPC	Inter-Process Communication
Lθ	OSI Layer θ
LTS	Long Term Stable
MTU	Maximum Transmission Unit
NIC	Network Interface Card
OS	Operating System
PMD	Poll Mode Driver
RSS	Receive Side Scaling
RTE	Run Time Environment
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
WAF	Web Application Firewall

Introduction

Internet and its related technologies are undoubtedly one of the inventions that have managed to shape the world in the 20th and the 21st century. Looking from the human perspective, the world has never been more interconnected than it is now — people manage to fulfill daily routine tasks such as shopping, working or searching for information just a few clicks away. That is an achievement that was unimaginable a century ago. We may argue whether it is a helpful or dreadful innovation, but it indisputably is something that was ahead of its time.

People have already experienced some of the concrete proofs that have shown us that the inventors of the internet (previously called ARPANET) did not anticipate its dramatic growth until 1980s. For instance, the so-called *IPv4 address exhaustion problem* is one of the examples where the internet scale showed us that the concepts the internet was built on were insufficient. Those drawbacks were fixed by resourceful hacks and are now being replaced with new technologies (such as *NAT* or *IPv6*), which is a tedious task to do.

Another great problem that has emerged with the tremendous amount of the users of the internet is the need for high-speed, high-availability networks. Nowadays, users demand the instantaneous reactions to their internet requests. This problem pressured the innovators to create high-speed transfer mediums, such as optical fibre cable, 40 Gbit/s or 100 Gbit/s Ethernet technology. But as new high-speed mediums were created, a new problem arose. The processing of the packets (e.g. packet forwarding, filtering or data integrity) hit the bottleneck in the operating system stack. Packet transfer between NIC and user-space applications, switching between kernel and user mode and packet in-memory copying are great examples which slow down the packet processing significantly. This became even a greater problem in network security, where each packet needed to be processed by multiple security devices (IDS/IPS, firewalls and/or L3 switches) whilst minimizing the drop rate. The overhead needed for packet inspection and/or forwarding became so essential that new solutions were mandatory.

One of the solutions is to use specialized hardware acceleration equipment, customized to fast packet processing or cryptographic operations. The equipment includes but is not limited to TCP offload engines, I/O acceleration technologies and so on. The main downside is the high cost and the limited customizability of the devices, which is almost always specific to the device vendor. On the brink of the millennium software packet processing libraries became favourable, as they offered the much needed customizability and scalability, providing the companies option to integrate the libraries in their existing production-ready solutions with the cost close to nothing. One of the modern packet processing frameworks is the Data Plane Development Kit (DPDK), used in this thesis.

One of the goals of this thesis is to analyze the capabilities of the DPDK framework and use this framework to design and develop a scaling infrastructure that can enhance performance limits of security applications in high-speed networks. Besides effective distribution of network packets with respect to relations of connections among a cluster of computation nodes, the

infrastructure is capable of high-throughput packet filtering. Therefore, security applications, more specifically *Suricata* and *ipfixprobe*, can be provided by the selected traffic that must be monitored or analyzed. The high-level overview of the proposed infrastructure is shown in Figure 1.

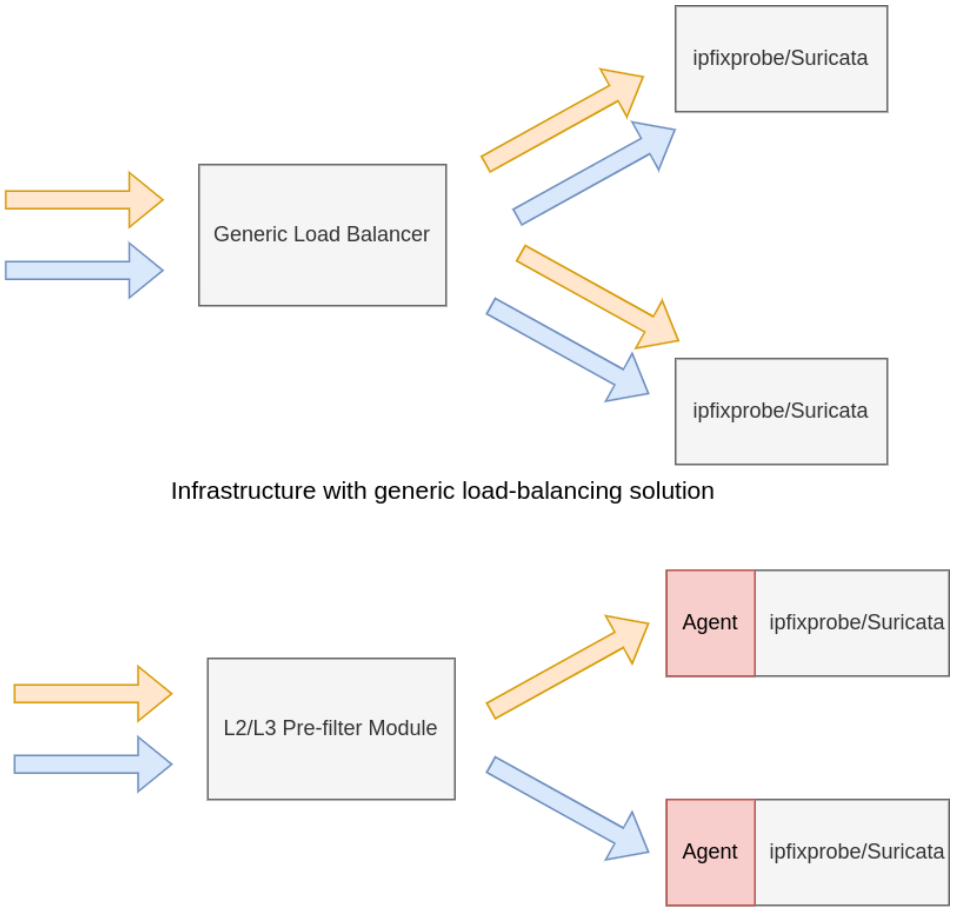

Figure 1 Generic Load Balancer vs. DPDK Pre-filter Module

This thesis is divided as follows: The DPDK framework is compared to other existing software libraries in chapter 1, and further explored in detail in chapter 2. After that, there are technical details about design and implementation of the proposed infrastructure and integration with Suricata and ipfixprobe in chapter 3. Finally, we measure and evaluate the results in chapter 4.

Chapter 1

State-of-the-art Works

Before we dive into the DPDK framework, we need to discuss different options available on the market, as there are plenty of powerful solutions that may be suitable in different scenarios.

1.1 Libpcap Library

The libpcap library was one of the first successful attempts that established the software packet processing. The library provided low-level interface for programmers to capture the network traffic, filter and/or forward the packets and ulitmately, process the data carried by the packets. The well-designed programming interface (pcap_t structure, unified naming convention, methods designed to achieve complex tasks and much more) was one of the reasons why the libpcap library became so popular (the history and further information about the libpcap library can be found in the book [1]).

Another great libpcap's strength was the fact that it was completely written in C. Most of the programmers that needed to write time-efficient/real-time code needed to work with low-level programming language, and C was an excellent choice. There are plenty of remarkable and popular projects that use libpcap library, namely Wireshark, Nmap, Snort or Suricata.

There are a few disadvantages when using this library though. First of all, "it doesn't address the issue of high-speed packet processing. If the application needs to process the packet data, it needs to copy its contents. For instance, when using `pcap_next_ex()` method, the packet data is not to be freed by the caller, and is not guaranteed to be valid after the next call to `pcap_next_ex()`, `pcap_next()`, `pcap_loop()`, or `pcap_dispatch()`; if the code needs it to remain valid, it must make a copy of it" [2]. Secondly, there is an issue of context switching — the packets captured by the NIC must traverse the whole operating system network stack before being processed by the user-space application. This creates an overhead that, as shown by modern libraries, can be diminished. Another problem with the libpcap library is its limited platform support. Although it works well on Unix-based environments, new peculiar problems may occur when operating on Windows/Mac OS. The libpcap library can also be compiled with the Visual Studio IDE, as mentioned in [3].

Overall, the libpcap library is a versatile tool that can be used as an introduction into packet processing for programmers. It also offers a better solution to network filtering and forwarding than the default C socket API. Regardless, it is not the most practical instrument when the goal is to achieve the biggest performance and network throughput while manipulating network traffic in order of gigabits per second.

1.2 WinPcap Library

WinPcap is a Windows-like alternative to the popular libpcap library created by the Riverbed Technology. As stated on the official WinPcap library webpage, "it has been recognized as the industry-standard tool for link-layer network access in Windows environments, allowing applications to capture and transmit network packets bypassing the protocol stack, and including kernel-level packet filtering, a network statistics engine and support for remote packet capture" [4]. The library consists of low-level drivers that behave analogously to Unix libpcap drivers, so the programmers can access the network stack more easily.

In the similar fashion, WinPcap was used to create Windows versions of the popular projects like Wireshark, Nmap or Ntop. It is also incorporated in the Windump tool, which provided a substitute for the tcpdump in the Windows environment. It is well-documented, tested and reliable library suitable for both the programmer and end user. It implemented the ease-of-access principle from the libpcap library, packet with sample programs that help developers understand the use cases more profoundly. It is licensed under BSD open source licence, so it can be integrated into both amateur-like and production-ready software.

The greatest disadvantage when working with the WinPcap library is its discontinued support. The last update the library received was on 8th March in 2013 — almost 10 years ago. The update came with a few bugfixes and added support for Windows 8 and Windows Server 2012. Since then, there has been no official major or minor development progress in the WinPcap library. According to the website, "WinPcap, though still available for download (v4.1.3), has not seen an upgrade in many years and there are no road map/future plans to update the technology" [4]. This may pose not only a performance, but more significantly a security issue, when developing a production-ready code. The fact that the WinPcap library is open source grants simple access for attackers to find and exploit possible vulnerabilities in the source code. This introduces security risk that needs to be considered when integrating the library.

To sum up, WinPcap was a widely-used library, bolstering multiple commonly-used tools like Wireshark or Nmap on Windows. However, because of its terminated support it is not recommended for real-world applications, as it may present a security threat to end users. It is worth mentioning that the library may and should be replaced with its modern alternative — Npcap, created by the Nmap project founder Gordon Lyon. More information about the Npcap project can be found on their official website [5].

1.3 IPFirewall

Another mentionable appliance which can be utilized as a packet forwarding and filtering software is IPFirewall (or ipfw). Included in the FreeBSD as a kernel loadable module, "it is comprised of several components: the kernel firewall filter rule processor and its integrated packet accounting facility, the logging facility, NAT, the `dummynet(4)` traffic shaper, a forward facility, a bridge facility, and an ipstealth facility" [6].

In short, it is a statefull firewall developed and managed by FreeBSD volunteer staff, configured by end users via the configuration file[1] — users may decide whether to work with the default firewall provided types (consisting of several different operating modes, each protecting the network, its workstations and eventually the network interfaces in various ways), or set up custom complex ruleset. Additionally, logging service can be turned on through `syslogd(8)`. All of these features make ipfw a spot-on utility, which can provide packet filtering and forwarding.

Nevertheless, ipfw does not deal with the concern of high-speed packet processing. It is also limited by its specific platform (FreeBSD) — there were numerous ports to different operating systems such as Linux, Windows or Mac OS X — all being replaced by modern alternatives. Also, ipfw is not a library — programmers cannot use it to modify or integrate the solution in their

[1] `/etc/rc.firewall`

code. For the reasons above, the ipfw is generally not considered as practical for implementation into existing high-speed packet processing solutions.

1.4 XDP

XDP (shortcut for *eXpress Data Path*) is a programmable, open-source, high-speed network processing framework for Linux kernel released in 2016. It utilizes eBPF technology, which allows applications to be encapsulated and executed with a privileged access with high effectiveness. Its core functionality comes from the ingenious implementation, as it captures the packets from the NIC RX queues before any allocations of the packet meta-data structures happen. It is supported by various big companies such as Intel, Microsoft or Mellanox [7].

From a practical perspective, there are few benefits when choosing the XDP framework. Firstly, it does not require any specialized hardware or any kernel bypassing mechanisms. Performance-wise, as shown in research [8], XDP is capable to drop around 25.7 mil. and redirect 7.9 mil. of packets from a single RX queue — there are limited number of libraries that offer these metrics. It is designed to be highly efficient and scalable, making it viable for fast-processing networking applications such as network function virtualization (NFV), software-defined networking (SDN), and packet filtering in firewalls and intrusion detection systems (such as Suricata).

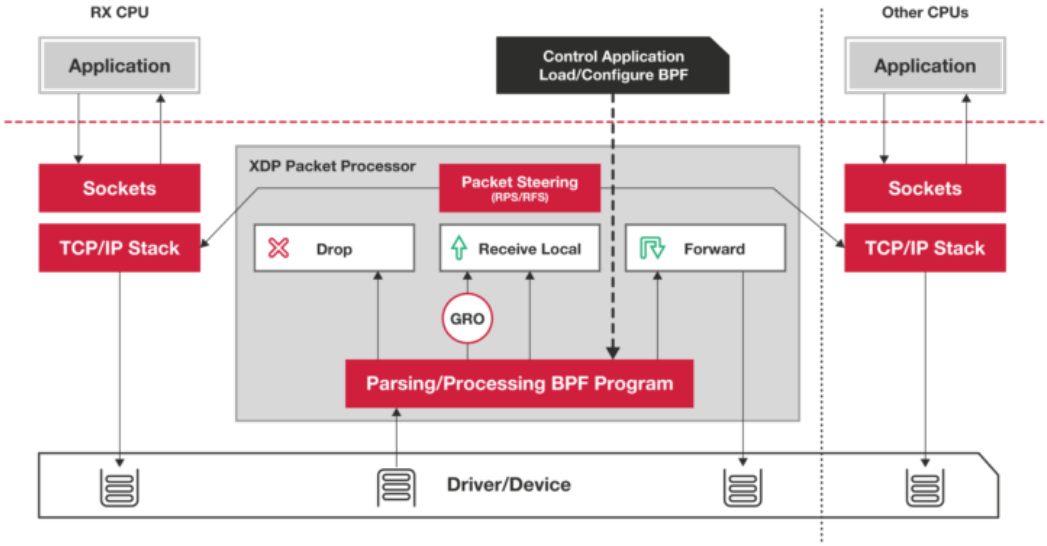

Figure 1.1 XDP Processing (taken from `https://www.iovisor.org/technology/xdp`)

The key difference between XDP and DPDK is that DPDK, being a user-space library, is programmed and compiled entirely in C language. Given the fact that XDP is compiled into eBPF bytecode (therefore making it less performant), DPDK has an upper hand when the processing speed factor is accounted for. Overall, "XDP is sometimes juxtaposed with DPDK when both are perfectly fine approaches" [9]. If used properly, it can do almost anything DPDK can achieve. However, as the DPDK slightly outperforms XDP in the packet processing speed, the DPDK was chosen.

1.5 PF_RING

Lastly, one of the state-of-the-art libraries that provide the capabilities to process packets in high-speed networks is PF_Ring. It is a top-shelf project created by the Ntop company, offering a set of libraries which programmers can integrate into their existing solutions. PF_RING effectively polls packets via Linux NAPI (new Linux API for networking devices) from the NICs, intercepting packets at the NIC driver level and processing them directly in the kernel [10].

There are multiple PF_RING products in the PF_RING framework — "Vanilla" PF_RING, PF_RING ZC (Zero Copy) and PF_RING FT (Flow Table), each with slight differences. Zero Copy version utilizes modern NIC hardware acceleration features in combination with Linux NAPI to achieve even higher processing and transmission rate. It minimizes the packet copying operations between kernel-space and user-space, speeding up the performance in some scenarios [11]. PF_RING Flow Table fuses the flow table mechanism in its pipeline, providing substantial performance increase when dealing with the packet flows [12].

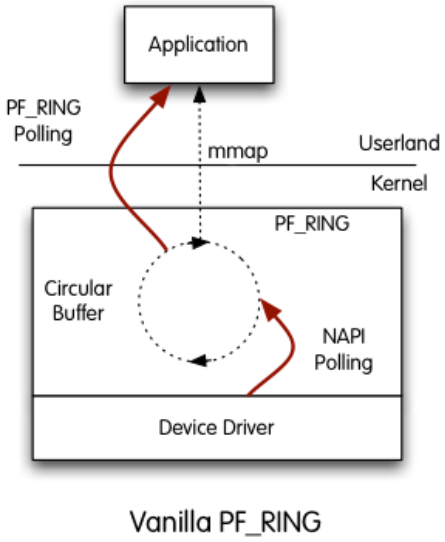

Figure 1.2 PF_RING polling (taken from `https://www.ntop.org/products/packet-capture/pf_ring/`)

The biggest downsides that play in favor of DPDK are the limited operating system support and slower packet processing rate. As stated in the previous paragraphs, PF_RING employs Linux NAPI (therefore it cannot be used in different operating systems), whereas DPDK is also supported on multiple operating systems like Windows or FreeBSD. Secondly, PF_RING is generally considered slower than DPDK, because of its architecture — PF_RING is a kernel-mode packet capturing framework, thus needing to pass the packets somehow to the user-space applications. This creates the overhead, which DPDK manages to bypass by operating in the user-space (explained in a greater detail in the next chapter).

For all those reasons, the proof of concept this thesis researches is based on the DPDK framework. Of course, there are a few advantages when deciding to work with the PF_RING framework — it is designed to be used with the standard Linux socket programming, hence having the shallower learning curve for programmers. DPDK also requires specific NIC drivers, which are mostly not supported by the older models. As always, the choice of the framework mostly depends on the use case and project specifics, and in certain cases, PF_RING might be the preferable option.

Chapter 2
Design

In the previous chapter, we have seen plenty of DPDK alternatives, each and every having its pros and cons. Now, let's explore what does the DPDK provide and how can it enhance packet processing and forwarding in combination with the applications such as Suricata or ipfixprobe.

2.1 DPDK Framework

The Data Plane Development Kit (further referred to only as **DPDK**) is a robust packet processing and forwarding set of libraries supported by most of the modern CPUs and NICs. Created by Intel in 2010, it is considered to be one of the top-notch solutions when developing any network appliance. It is licensed and distributed as open-source[1] software, making it possible for anyone to integrate it in their current solution.

The DPDK deals with the high-speed packet processing problems almost flawlessly. One of the challenges the packet processing poses is the data transfer speed bottleneck between the NIC, CPU(s) and the memory. The DPDK solves this with multiple utilities in its core: NUMA (non-uniform memory access) awareness, usage of SIMD instructions such as Intel SSE3 instruction set, lock-free structures in combination with memory and cache prefetching, but most importantly, the HugePage utilization.

The HugePage concept comes from the Linux kernel. Simply said, it enables the operating system employment of greater memory pages than the usual 4KiB, therefore reducing the demand for system resources needed to access page table entries by increasing the TLB hit ratio, both in 32-bit and 64-bit operating systems. The ideal hugepage size may vary between a few MiBs up to 1 GiB, depending on the hardware. DPDK also puts hugepages into effect when sharing memory pools between the primary and secondary DPDK applications, diminishing the data transition time between processes.

The concept of primary and secondary applications in DPDK was implemented with the purpose to ease the inter-process communication (IPC) between the different DPDK processes. Every DPDK standalone application runs as a primary application. The key difference is that the primary application initializes and manages the shared memory (hugepages) with full permissions, whereas the secondary application can only be "attached" to already existing shared memory. Everything is handled by the EAL in the method `rte_eal_init()` present in every DPDK application. During the creation of the primary process, DPDK saves the virtual addresses of the hugepages in the OS, current memory channels and so on. The secondary application is then created with the exact mapping as the primary application so that all the shared virtual addresses are valid. The option whether the application will be run as a primary or secondary is specified

[1] Open Source BSD License

via the command-line passed by the parameter *--proc-type=primary* or *--proc-type=secondary*.

Although there are numerous components in the DPDK framework, there are additional utilities worth mentioning shipped with the DPDK to help with setting up the environment or with debugging and testing hardware. For instance, in the latest official DPDK LTS build, developers may find plenty of detailed example applications such as *l2fwd* (L2 packet forwarder and classifier), *ip_fragmentation* (L3 fragmented packets forwarder and classifier) or a simple *helloworld*, which is commonly used to determine whether the DPDK framework works properly. Along with the DPDK example applications, developers frequently use *dpdk-devbind.py* script, which binds the DPDK-compatible drivers with the desired NIC.

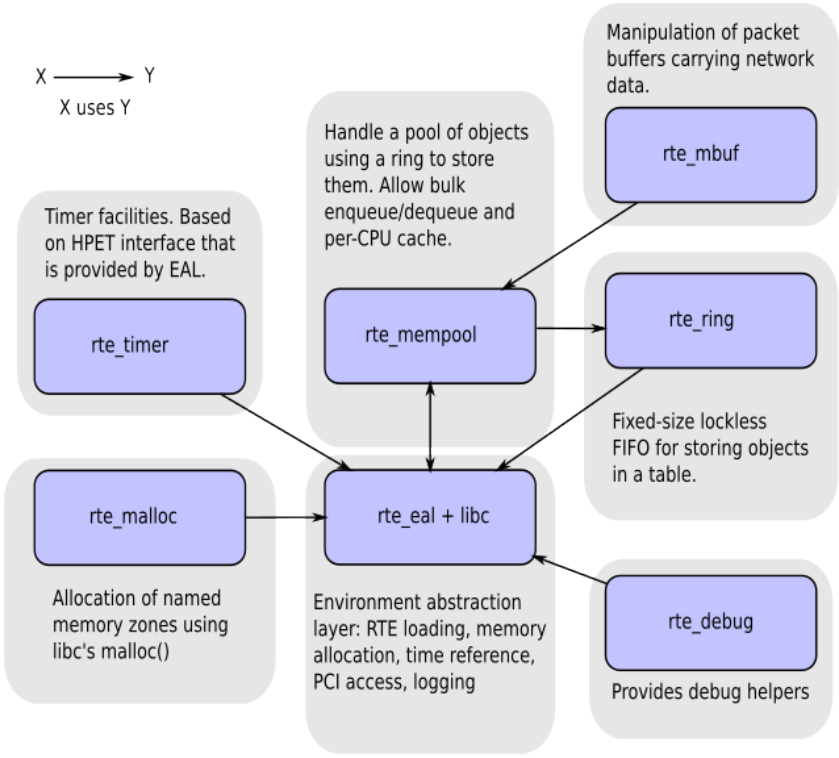

Figure 2.1 DPDK Core Components (taken from `http://doc.dpdk.org/guides/prog_guide/overview.html`)

The DPDK scope consists of multiple modules, each providing modern solution for specific programming task. The most important (and most commonly used by programmers) modules are shown in the figure 2.1. The modules include, but are not finite to:

rte_eal provides an API between the application and the DPDK core functionalities

rte_mempool handles dynamic memory allocations and deallocations, per-core caching and memory alignment

rte_ring implements a thread-safe, ring-like structure optimized for bulk procedures in the DPDK core

rte_mbuf manipulates packet-like structures and its accompanied metadata

rte_timer realizes precise time reference and asynchronous operations in the DPDK core

In this thesis, a few core components are used so frequently that deeper understanding is useful. Therefore, in the next part of this chapter, we discuss an overview of EAL, PMD and how does the DPDK handle memory efficiently.

2.1.1 Environment Abstraction Layer (EAL)

The **Environment Abstraction Layer** is a key software library in the DPDK core that provides fundamental platform for the DPDK user-space applications to interact with the underneath operating system and hardware resources.

The EAL is initialized in the DPDK applications by the function `rte_eal_init()`. The full list of the operations done by this method is lengthy, but it can be summed up to a few steps:

- CLI parameters — EAL parses the command-line launch parameters to setup specific options in the DPDK core, such as core affinity, memory and device related options and so on[2]
- Threading — EAL initializes CPU cores affinity and worker threads via the *pthread* POSIX API
- Memory — EAL tracks and manages allocated hugepages as well as provides interface to allocate dynamic memory in runtime
- Hardware — EAL registers the interrupts from the hardware resources and handles the device discovery and further configuration
- Time — EAL provides an API for the programmers to accurately track time and set up asynchronous routines

Apart from the initialization phase, EAL is also responsible for certain tasks during the runtime. For instance, EAL registers and unregisters the IPC between the processes. This is done by the functions `rte_mp_action_register()` and `rte_mp_action_unregister()`, respectively. The EAL also encapsulates the process metadata, therefore providing the programmers the information about the current process via the functions `rte_eal_process_type()`, `rte_gettid()` or `rte_eal_primary_proc_alive()`.

There is a slight downside when using EAL as a memory manager — there is no way for the tools such as *Valgrind* to determine whether the used hugepages were used correctly, as the hugepages are not freed by the DPDK application. The configuration of the hugepages is done by the programmer in the operating system (commonly done by mounting in the `/dev/hugepages` directory) and the OS is responsible for cleaning up these resources. The problem occurs when the DPDK application accesses this allocated memory, as the memory tracking tools do not differ between the dynamic heap allocations and hugepages. This may result into unpleasant reports containing non-freed reachable memory blocks equal to the size of the referenced hugepages. Currently, there is no practical solution to this problem that is effective and feasible in the same time.

2.1.2 Poll Mode Driver (PMD)

The DPDK **Poll Mode Driver** is a software implementation of multiple high-performance virtual upstream drivers that allows programmers to process packets at gigabit speeds. The performance is boosted due to the fact that the drivers provide user-space API, thus bypassing the kernel stack completely. The API includes multiple packet receiving and sending methods, all of which are designed to enhance the performance of NIC. Linux users can test their PMD NIC integration with the tool `testpmd`, shipped with the DPDK framework.

[2]Common CLI parameters can be found in the documentation

The core concept of PMD is that it is not just a single common driver implementation for multiple NICs. The PMD is an abstraction for the set of specific implementations for each supported NIC (this is the main reason, why many of the old NICs do not support the DPDK — it is simply because the PMD does not include the driver for that particular NIC).

Nevertheless, with the modern NICs, the PMD can achieve the true spirit of high-speed packet processing. For instance, as stated in [13], the NVIDIA PMDs are "most advanced NICs on the market today, enabling multiple offloads in NIC hardware to provide maximum throughput at the lowest latency" that "reach 200GbE throughput with a single NIC port" and "provide a cost-effective solution that allows DPDK applications and non-DPDK applications to concurrently use the NIC." There is a variety of PMD implementations available for specific vendors, both for physical and virtual NICs. Some of the well known vendors include Intel igb PMD, Nvidia PMD, Cisco enic PMD or Microsoft mana PMD.

The PMD is operational in two modes - *polling mode* (default) and *interrupt mode*. However, the latter mode is used for debug and experimental uses only, as it goes against the essence of kernel bypassing (interrupt mode uses the traditional approach, where the NIC sends the interrupt to the CPU and the context switches are required to handle the interrupt, which may result in significant overhead). In the polling mode, the user-space application continuously polls for the packets, rather than waiting for the interrupt to happen. The polling requires the RX and TX "queues" that must be set up beforehand, so the received and sent packets can be placed into their respective queues. Some NICs even support multiple RX and TX queues, so the packet processing can be optimized with concurrent operations.

2.1.3 Memory and Caching

Efficient memory handling is one of the aspects that when used properly, it can hugely enhance the overall performance of any program. DPDK user-space applications are not different. There are few mechanisms integrated into the DPDK core that help programmers write thread-safe yet quick applications. We have already discussed the concepts of Hugepages in the section 2.1, but there are also other instruments that developers can utilize.

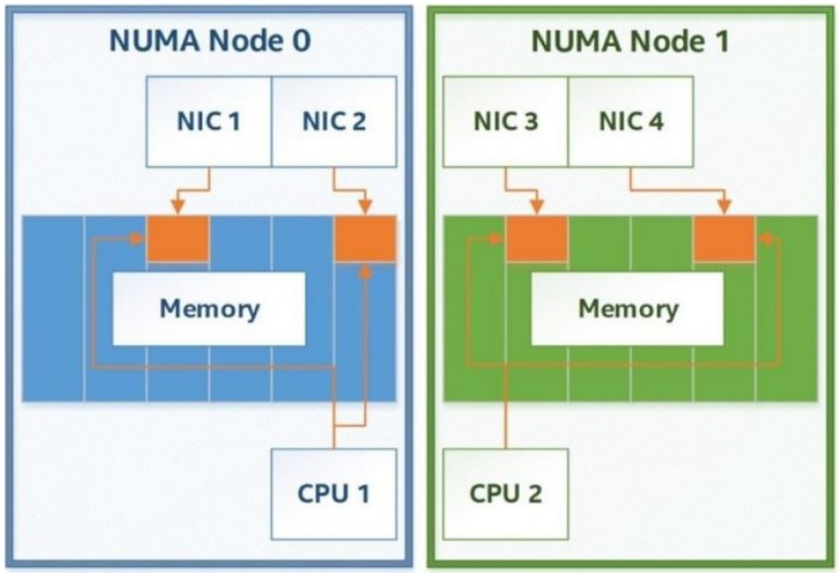

Figure 2.2 Ideal NUMA node allocation (taken from `https://www.dpdk.org/blog/2019/08/21/memory-in-dpdk-part-1-general-concepts/`)

One of the ways DPDK helps programmers with proper memory operations is the imple-

mentation of NUMA awareness. NUMA (non-uniform memory access) is a problem encountered when creating programs for multiprocessor systems – the consistency and coherency in the CPU caches must be taken into account, so the data is not corrupted or flawed in any way. This however comes with the upcost and overhead, when one CPU tries to access data located in another CPU's cache. One could argue that modern OS are built NUMA–aware so that cross–NUMA accesses are mitigated. This is true in most cases, but the DPDK core was built with NUMA-aware concepts for almost every operation.

As shown in Figure 2.2, developers can specify allocated NUMA nodes for various components. For instance, the parameter *socket_id* in `rte_mempool_create()` assures that the generic memory pool is allocated at the specified NUMA node. The *socket_id* parameter can be found in almost every method that creates objects in memory: `rte_pktmbuf_pool_create()`, `rte_ring_create()` and `rte_eth_rx_queue_setup()` are just a few of many examples. More detailed introduction into memory operations that DPDK implements can be found in [14].

2.2 Suricata

For the purpose of testing DPDK in network security applications, the **Suricata** (IDS/IPS)[3] was chosen as the best candidate. Created in 2009, Suricata is an open-source network monitoring solution, capable of inspecting high-volume network traffic. As shown in Figure 2.3, there has been an significant development throughout last 10 years. There is an integration into multiple software packet filtering frameworks, such as *AF_PACKET* or *PF_RING*. As of April 2023, the DPDK integration is unfortunately not yet integrated into the official Suricata version, but the solution is already in the merge request in the Suricata git repository[4]. In this thesis, we will be mostly taking advantage of the fact that Suricata is an open-source solution.

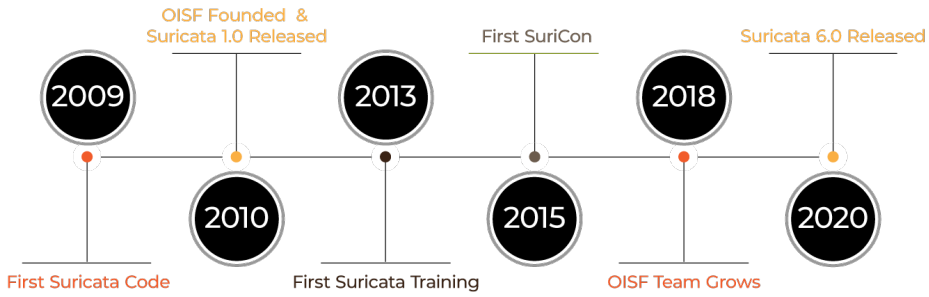

Figure 2.3 Suricata history (taken from `https://suricata.io/`)

There are plenty of handy features Suricata is capable of. For instance, when running in the IDS mode, "Suricata can log HTTP requests, log and store TLS certificates, extract files from flows and store them to disk. The full pcap capture support allows an easy analysis. All this makes Suricata a powerful engine for your Network Security Monitoring (NSM) ecosystem" [15]. There is also an enormous community built around the Suricata that helps with the development and integration with various 3rd-party applications. Thanks to that the Suricata is able to detect malicious L7 communication (such as the detection of various C2 channels). There are also integrations into various data mining tools like Splunk or Kibana, where the data visualisation and graphics can help network security analysts and CERT/CSIRT teams in their work.

[3] Suricata can be run both as Intrusion Detection System or Intrusion Prevention System
[4] https://github.com/lukashino/suricata/tree/bug/5923-dpdk-numa-api-v1

2.3 ipfixprobe

IPFIX flow exporter, conveniently called **ipfixprobe**, is another network monitoring utility developed by CESNET. The main objective of ipfixprobe is to categorize and group up similar packet traffic into coherent units called *flows*. There are a handful of flow categorizing mechanisms, from simple L3 and L4 identifiers classification (source and destination IP addresses, ports, ...) to advanced multi-layer behavioural analysis, utilizing known L4 and L7 protocol finite-state automata. The openness of the source code in combination with its API provides options for developers to come up with new flow classifier plugins utilized in ipfixprobe. When creating a plugin for DPDK, there is a robust script shipped in the official repository that is able to generate the template for development.

Before the implementation part of this thesis, there was no DPDK integration into ipfixprobe. As seen in the latter chapters, the collaboration ended fruitfully, as the ipfixprobe can now be run as a secondary DPDK application reading from the shared `rte_ring` structure allocated in the memory by the primary DPDK application. The name of the ring to be read from is simply passed as RING_NAME in an argument `-i 'dpdk-ring;r=<RING_NAME>'` when starting up ipfixprobe.

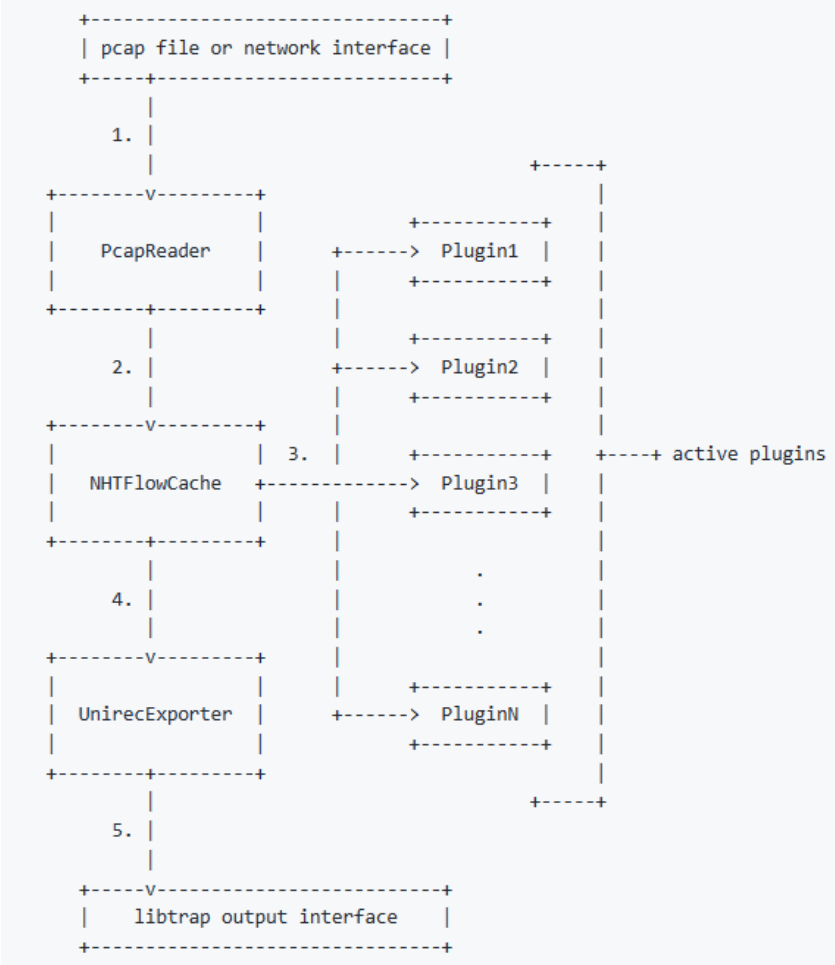

Figure 2.4 Simplified ipfixprobe function diagram (taken from `https://github.com/CESNET/ipfixprobe`)

Chapter 3
Implementation

DPDK framework provides a great amount of useful features that programmers can integrate into their environment. In this chapter, proof-of-concept infrastructure is designed and implemented, with an emphasis on performance of packet processing speed.

3.1 Proposed Infrastructure

The DPDK framework can be integrated into many different network architectures — from the most simple ones with one router and one switch device, up to the complex hierarchies divided into Core, Distribution and Access (CDA) layers and network planes. This fact can be justified simply because with the DPDK framework, programmers can implement fast and reliable solutions in different OSI layers — starting in the L2 broadcast domain, which splits bigger networks into smaller, easily manageable units with the MAC/EtherType pre-filtering and forwarding solutions, and closing with the L7 application solutions as IDS and Web Application Firewall (WAF), where the traffic inspection is correlated in multiple OSI layers. The DPDK offers solutions for any and every of the aforementioned use cases.

To introduce and explore the DPDK framework in the high-speed networks, both L2 and L3 pre-filtering were chosen as the proof-of-concept pre-filtering methods (the small differences between these infrastructures are described in the section 3.4). The L2 pre-filtering was chosen based on the following facts:

- Standard L2 frame headers include source and destination addresses + EtherType, so simple static access control lists can be defined for the traffic to be filtered with
- The L2 pre-filtering module source code is not as robust as it would be if another OSI layers were introduced, thus making the filtering parts clear for the reader and further research
- No encryption is required in this use case, therefore there is no need for additional hardware cryptographic modules (which would be necessary as it would otherwise create a bottleneck)

The frames, however, also come with slight disadvantages. First of all, frame headers do not include much data to differ various packet flows (for instance, there is no way to determine what application is this flow belonging to), so there is almost no way to implement powerful ACLs. Additionally, the sample network architecture must[1] contain the pre-filtering device in the same broadcast domain as the monitoring devices (Suricata, ipfixprobe, …). Both of these inconveniences can be bypassed in the further development of the pre-filtering device so the pre-filtering device would be operating on more OSI layers, as shown in the L3 pre-filtering example.

[1] The reason for this is explained further below

The development for multi-layer pre-filtering mechanism is advised, however, it may create a slight performance drawback — further research is needed.

The proposed infrastructure is depicted in the figure 3.1. There is no major difference between the infrastructure used in the L2 and L3 pre-filtering (both pre-filters need to be in the same broadcast domain as the endpoints with installed agents), therefore L2 pre-filter module is shown in this chapter as an example. The infrastructure consists of:

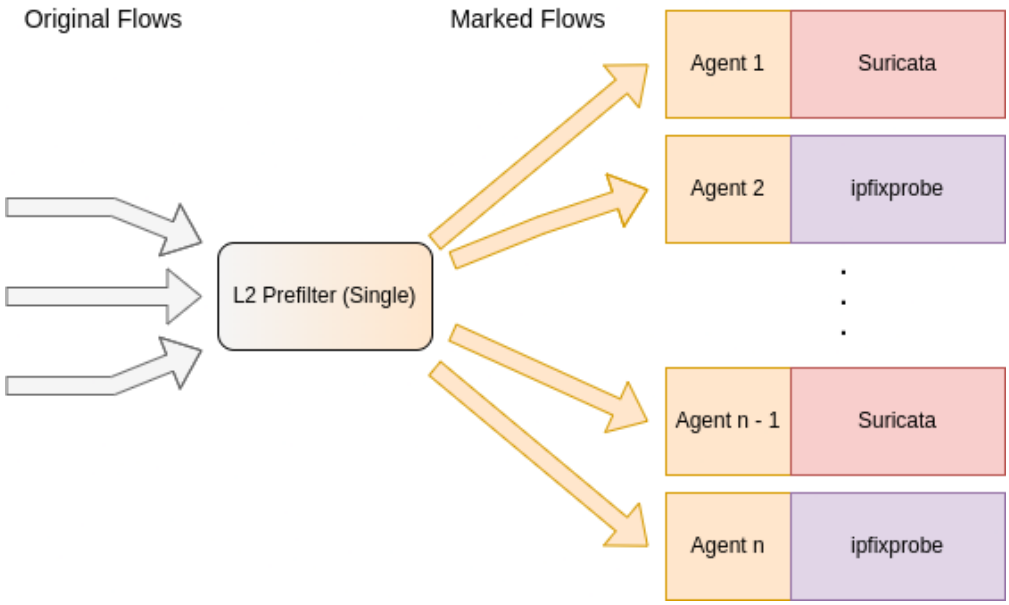

Figure 3.1 Proposed infrastructure (single L2 pre-filter module)

L2 Pre-filter Module is a pre-filtering device operating in the OSI layer 2, which filters, separates and forwards different flows into specified modules

Suricata/ipfixprobe modules (referred to as **agents**) are endpoint applications responsible for decapsulating the custom header.

Flow Generator (Optional) is used in development phase to generate stable flow of packets.

A more profound description of how each of the components of the infrastructure work can be found in the second part of this chapter.

Let's now discuss the reasoning behind the pre-filtering the data (frames) in figure 3.1. The high-speed network traffic that needs to be monitored is mirrored (copied) to the proposed infrastructure. The **original flows** of packets can come from multiple networking devices with different periods and load (as with the real traffic, different types of packets, both encrypted and plaintext, are expected). This behaviour simulates the real-world networks, where the network traffic generated by the users may vary throughout the day.

The network traffic is monitored by the single **L2 Pre-filter**. The L2 pre-filtering module is a core component in the infrastructure that aggregates the frames in the network and, based on the ACLs, either drops them or marks them (marking is further explained in the section 3.3). After that, the marked frames are then distributed evenly between the endpoint agent devices based on the flow they come from. As we operate in the OSI layer 2, the single L2 flow can be defined as a set of frames with the same source and destination MAC addresses (with the L3 pre-filtering module, we transition from MAC addresses to the IPv4 addresses that define L3 flows). Therefore, when two frames with the same source and destination MAC are received in

the pre-filtering module, they will be marked and forwarded to the same agent device (assuming they were not blacklisted in the ACLs).

The **marked flows** are then received by the agents, where they are stripped off the custom header and inserted into the network monitor applications such as Suricata or ipfixprobe. As this process differs from program to program, multiple implementations of the agent modules exist.

The overall idea is based on the existence of the minimal threshold of malicious data that needs to be captured in the different network monitoring applications. For instance, the IDS analyses the network flows and raises alert when one of the rules is matched. The rules commonly contain a specific threshold parameter, which triggers the alert when reached. This however creates a problem in large scale networks, where the resources of a single monitoring node are not sufficient, so the traffic is split between multiple monitoring devices. In general, there is no flow preservation, thus creating the possibility of splitting the malicious data into multiple network applications and ultimately not triggering the alert, as no threshold was reached.

In this thesis, the aspect of flow preservation it considered, so even with multiple agents the frames belonging to the one flow end up in the same agent. This method should eliminate the problem of traffic splitting in the L2/L3, but the overall reasoning can be also applied to higher OSI layers. This problem is more thoroughly discussed in the paper [16].

Lastly, the discussion about the multiple pre-filtering devices is in place. In the proposed infrastructure, the biggest obvious drawback is that the single pre-filtering module may create a bottleneck in the high-speed networks. As the hardware resources such as memory and CPU clock speed are limited, there is a reasoning whether to implement multiple pre-filtering modules into a *stack*. One of such infrastructures is shown in the figure 3.2, where there are 3 different L2 pre-filters, each bound to at least 2 agents and each with its custom marking mechanism.

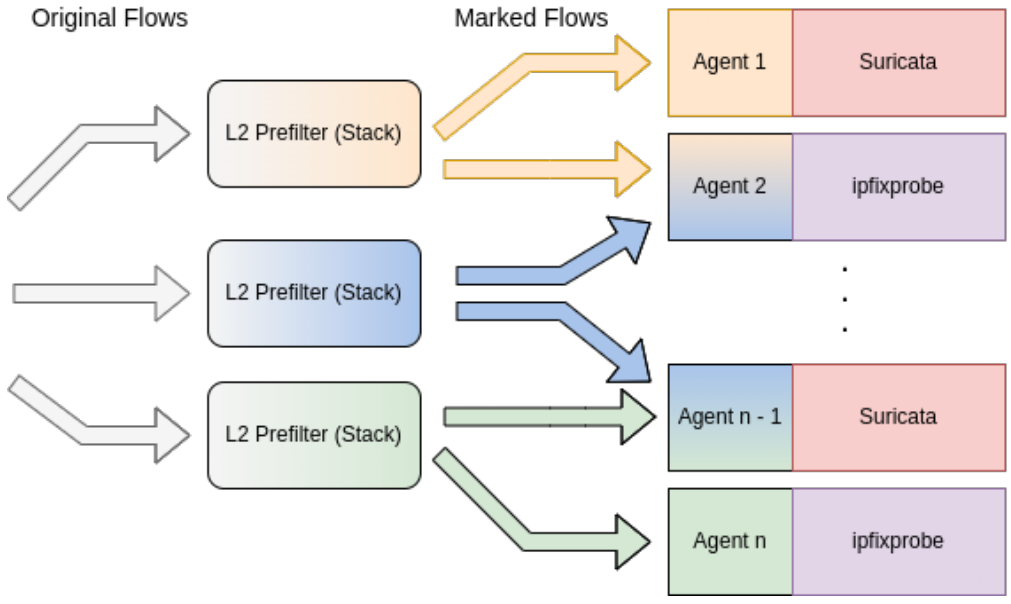

Figure 3.2 Sample infrastructure (L2 prefilter module stack)

There are plenty of combinations of module bindings, which in addition with the different frame marking mechanisms creates a limitless room for research (with the simplest being one pre-filtering module and one agent endpoint). Of course, the stacking comes with a few disadvantages:

- More complicated implementation of agents, as for each specific marking mechanism there must be a compliant agent implementation. The different agent implementations will be

unable to decapsulate the traffic from different prefilters, which may result into chaotic behaviour.

- The additional overhead of traffic mirroring into multiple prefilters may create a performance bottleneck on the existing network infrastructure.
- The different marking mechanisms may create a never-ending frame loops in some network topology. Imagine the situation:

  1. The L2 prefilters A and B are integrated into network with star/bus topology, thus placing the prefilters into the same broadcast domain.
  2. The prefilter A receives a frame, marks it with its specific header and sends it to the agent module.
  3. The prefilter B receives a marked frame from the prefilter A (as the prefilters monitor the whole traffic), marks it with its own specific header and sends it further to the agent module.
  4. The prefilter A receives the marked frame from the prefilter B, marks it and continues encapsulating the frame in the loop with the prefilter B.

  As there is no way for the prefilters to determine whether the frame has already been marked by the other one, they will keep creating malformed frames with numerous custom headers stacked onto each other, which in result will flood the agent(s) and eventually clog the whole broadcast domain, resulting into denial of service.
- The upkeep cost of multiple prefilters can exceed the cost of upgrading the single prefilter.

There is also a reasoning to integrate multiple pre-filtering mechanisms with the same marking methods. In this case, there will be no need for multiple agents implementations and no potential problems with compliance. As this is not in the scope of the research of this thesis, the further research about the viability of pre-filter stacking is advised.

3.2 Flow Generation

The saturated high-speed packet flow generation is mandatory for the pre-filtering modules to demonstrate their true power. However, in the testing environment, simulating real network packet flow is a highly difficult task (which is almost impossible to mimic without the access to the real-time data). Saturated flow generator is utilized in chapter 4, where the implemented infrastructure is evaluated more profoundly. For the development purposes however, there is no need to emulate high loads of saturated network traffic.

The simple packet flow generator can be developed using numerous software libraries such as *Scapy* or *ntopng*. The discussion is in place whether in the L2 pre-filtering, the L2 header contains sufficient data for categorization to create distinguishable flows, but as this proof-of-concept architecture filters the frames in the broadcast domain (this infrastructure can be simply[2] replicated with switches when the traffic is mirrored to L2 prefilter module), the source and destination MAC addresses will be sufficient. As this thesis uses the DPDK framework, both Scapy and custom DPDK flow generating application were used during the development phase.

Scapy can be employed as a python script that periodically sends the packets (these packets contain only Ethernet II and IPv4 header + null bytes in the payload to satisfy the minimal Ethernet II frame length) into the network. Sample implementation of such a script called *generator.py* is shown in the Code listing 3.1. The `generate_flows()` method returns an array of Scapy packet-defining structures with random source and destination MAC/IPv4 addresses.

[2]Most of the modern switches such as Cisco or MikroTik offer the option of port mirroring

Code listing 3.1 *generator.py* - Scapy traffic generator

```
if __name__ == '__main__':

    random.seed()
    flows = generate_flows(MAX_FLOWS)
    print(str(MAX_FLOWS) + " flows generated.")

    while True:
        try:
            for i in range(MAX_FLOWS):
                sendp(flows[i], count=40)

            flows[random.randint(0,MAX_FLOWS - 1)] = generate_flows(1)[0]
            time.sleep(1)
        except KeyboardInterrupt:
            print("Shutting down.")
            break
```

After that, there is an infinite loop (simple Ctrl-C breaks the loop) in which 40 packets are constructed and sent via the NIC. After each iteration, one of the flows is replaced with the new combination of source and destination MAC addresses, effectively creating the randomizing pattern. The script can be launched simply by executing "`python3 ./generator.py`" in the script's directory.

The script generates a small number of testing packets, therefore providing an excellent solution when testing the infrastructure with small volume of flows. It is not an effective solution when a large volume of network traffic is required, as the Scapy module is not optimal for high-speed packet creation in order of hundreds of gigabytes (for instance, packets originating in the application do not bypass the OS stack and therefore create a unnecessary delay). Therefore, for an artificial high-speed frame flows more specific framework is needed — luckily, one of the frameworks capable of generating a large volume of flows is DPDK.

With DPDK, we can achieve periodic flow generation in our own custom flow generating application simply by using a few methods DPDK offers. However, the DPDK framework is really robust, which is impractical for small tasks such as packet generation. In this instance, L2 frames with randomized payload (16 bytes) are periodically generated. The particular steps include:

1. Initialize the EAL with correct CLI parameters (such as the amount and affinity of cores, TX port number or specific debugging options) with `rte_eal_init()`.
2. Allocate the array for newly created frames with `rte_pktmbuf_pool_create()`.
3. Configure and start the NIC and its TX queues/descriptors. This includes the process of verifying whether the provided port number is correct, reading and configuring custom NIC configuration, adjusting the number of TX descriptors and setting up the TX queue(s).
4. Periodically fill the allocated memory with newly created frames.
5. Send the created frames using the NIC TX queue with `rte_eth_tx_burst()`
6. Free up the memory using `rte_pktmbuf_free_bulk()` so new frames can be created.

The DPDK traffic generating binary is called *generator.out*. The difference between the DPDK and Scapy version is mainly in the speed when the large volume of traffic is generated. During the development, the Scapy implementation was used to test the scalability with multiple

different flows which is achieved thanks to the random addressing. The DPDK is used mainly for the higher loads of the frames in a few particular predefined L2 flows specified as arguments passed to the application. With an underlying random payload after the frame header, which standard network monitoring applications such as *Wireshark* may mark as malformed (this is however an expected behaviour with random bytes instead of a proper layer 3 OSI packet header).

The DPDK traffic generating application can be launched using the CLI command "`sudo ./generator.out --no-telemetry -l 0 -- -f <DSTMAC>`". The `--no-telemetry` and `-l 0` parameters are EAL options[3], which specify the logging level and CPU core map (in this case, we suppress most of the debugging logs and assign the CPU core 0 to our DPDK application). The `-f` parameter specifies the destination MAC addresses, so up to 10 flows[4] can be configured (the source MAC is always set as the sending NIC's address).

3.3 L2 Pre-filter Module

The backbone component in the proposed infrastructure is undoubtedly the **L2 pre-filter module**, as it performs most of the core functionalities that are necessary in the high-speed flow monitoring. Before the introduction of the main parts of the L2 pre-filter module, the goals and main capabilities need to be set. In this case, the main purposes of the L2 pre-filtering module were:

1. Monitor the high-speed network traffic in the broadcast domain of the pre-filter without the increased frame drop rate.
2. Filter the blacklisted frames using access control lists based on their source and destination MAC addresses.
3. Create an efficient method of transferring the processed frames into the network monitoring applications running at the different endpoint devices in the same broadcast domain.

The high-speed processing of larger volumes of frames is one of the use cases the DPDK was designed for. As stated in chapter 2, the DPDK uses multiple tricks to amplify the RX and TX speeds while minimizing the overhead in the operating system. Almost every modern NIC now supports the *promiscuous mode*, which effectively causes the NIC to capture every frame received by the medium (NICs discard the frames by default if the destination address does not correspond to their MAC address). In DPDK, this behaviour can be invoked by the `rte_eth_promiscuous_enable()` method. This method is put into effect right after the hardware initialization part in the *pipeline* submodule, where the NIC configuration is located.

The frame filtering in the DPDK can be achieved in multiple ways. The most common way of handling the ACLs in DPDK is to use the designated ACL structures and their corresponding methods. This method supports hardware optimizations for fast flow filtering, such as Intel SSE or AVX512X16 vector operations, which enable multicore parallel flow filtering [17]. The disadvantage that discouraged the use of ACL structures in L2 infrastructure is its complexity with L2 filtering. In these structures, there are multiple mechanisms built for complex L3 and L4 ACLs, yet they lack the straightforward solutions for L2 filtering. The L2 filtering integration using predefined ACL structures in DPDK is unnecessarily obnoxious, therefore another solution for the L2 pre-filtering was needed to be found.

Instead of DPDK ACLs, simpler yet effective way was chosen for the L2 filtering – the custom implementation of the ACL structures. As described later, the custom ACL structures in the *ACL submodule* provide stable foundation for the L2 pre-filtering module to filter the blacklisted flows. The ACLs do not use the hardware accelerated operations as the DPDK ACLs do, but in the order of tens of simple ACLs, this should not create a significant overhead.

[3] The end of the EAL parameters and the start of the application parameters is denoted by the `--` mark

[4] This behaviour can be changed by modifying the predefined macro in the source header file *constraints.h*

Lastly, an efficient transferring method implementation is mandatory for the fluent pre-filtering mechanism. One possible method of transport is the **substitution method**, which is based on the MAC address replacement. In this method, the source address is replaced by the pre-filter TX NIC MAC address and the destination address is replaced by the agent's MAC address. The disadvantage is that the original L2 flow is not preserved – the information about the source and destination addresses is inevitably lost. This may pose a problem in the L2 broadcast domain monitoring, as there is no way for the monitoring applications like *Suricata* and *ipfixprobe* to determine the initial addresses. For the reasons above, this thesis does not use the substitution method.

The proposed method for the transport of processed frames by the L2 pre-filtering module is the **encapsulation method**, shown in the Figure 3.3. In this process, whole frame is encapsulated into another valid Ethernet frame header, so it will be properly switched in the broadcast domain. The default Ethernet header consists of 3 fields – the destination MAC address, the source MAC address and the EtherType (EtherType is a field in the Ethernet header indicating expected protocol in the next layer). By prepending the Ethernet header in front of the original frame we achieve the preservation of the original flow, as no information needs to be modified. In the new header, the source address is set to the L2 Prefilter MAC address (as this is the device that performs the encapsulation) and the destination address is set to the designated agent address (therefore, any L2 switch can forward the frames to the right endpoint device). The EtherType field is set to the hexadecimal constant 0xF1F0 (this arbitrary constant can be changed to any non-used EtherType), so the endpoint devices can precisely determine whether the received frame carries another encapsulated information and thus needs to perform the frame decapsulating. The frame decapsulating is a trivial process of checking whether the EtherType was set to the constant 0xF1F0, and if so, the first 14 bytes[5] are skipped.

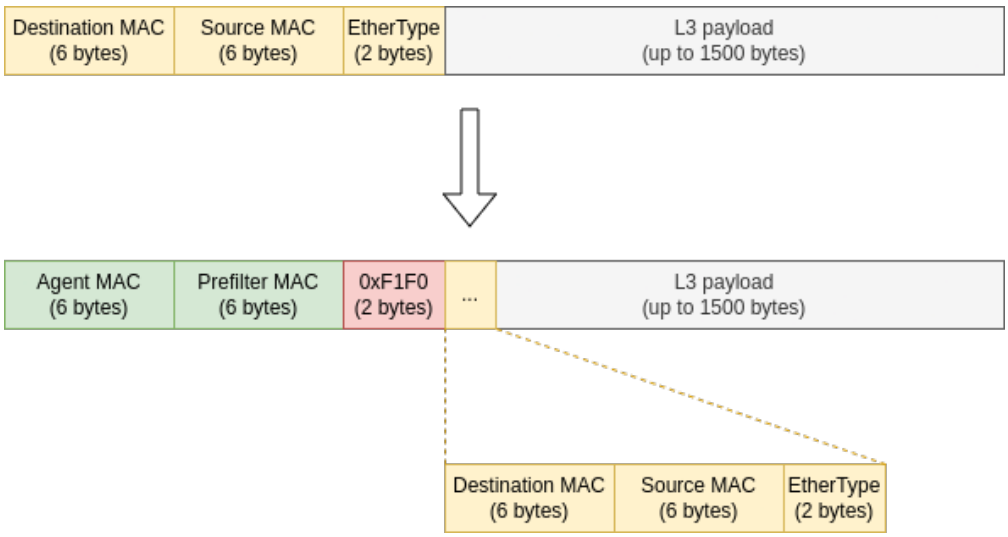

Figure 3.3 L2 prefilter module frame encapsulation

One small issue might arise, when the maximal MTU (Maximal Transmission Unit) frame is processed by the L2 pre-filter module. The default MTU for the Ethernet II frame is currently set in the RFC 894 [18] to the 1518 bytes – 1500 bytes of payload + 14 bytes of standard Ethernet header + 4 bytes of FCS[6]. When this frame is encapsulated by the L2 pre-filter module, another 14 bytes are prepended to the frame. This creates a potentially invalid Ethernet II frame, which results into undefined behaviour in most of the state-of-the-art switches. Depending on the

[5] 6 bytes of source MAC + 6 bytes of destination MAC + 2 bytes of EtherType = 14 bytes of header
[6] Frame Check Sequence

vendor and the product, switches may drop or forward the frame. This problem can be bypassed by using the *Ethernet jumbo frames*, which can effectively carry up to 9000 bytes of payload (even more than 9000 bytes in one frame are possible, as described in [19]).

Let's now introduce the components (submodules) of the L2 pre-filtering module and their associated tasks. The core submodules are shown in Figure 3.4

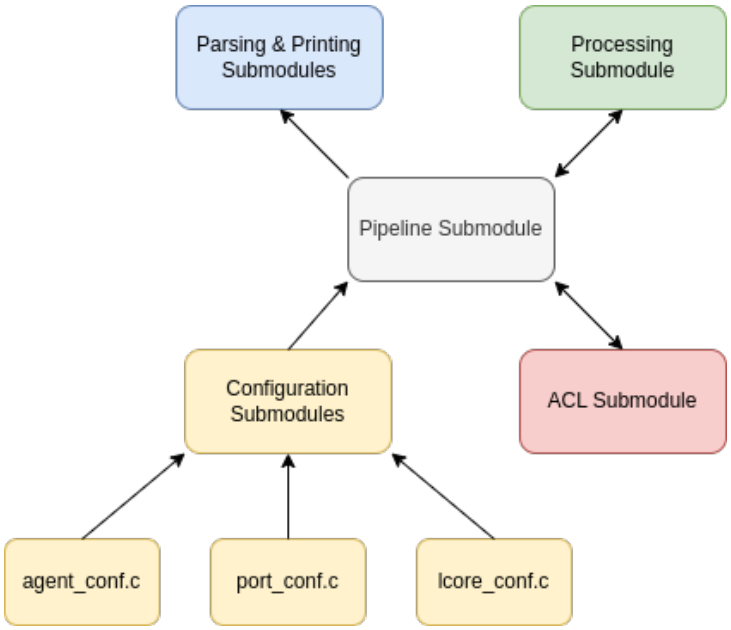

Figure 3.4 L2 pre-filter submodules

Pipeline Submodule is responsible for initializing the hardware resources and the multithreading model. It also interconnects other submodules into one ecosystem.

Configuration Submodule contains structures responsible for handling the custom configuration data. It consists of:

- agent configuration, which handles the endpoint agents traffic distribution algorithm and associated MAC addresses.
- port configuration, which manages the NIC RX and TX ports, queues and descriptors.
- logical core configuration, which tracks the designated CPU configuration, socket alignment and threading affinity.

ACL Submodule filters the flows using primitive blacklists for source and destination MAC addresses specified via the CLI.

Processing Submodule handles frame flow creation, hashing and frame encapsulation

Parsing & Printing Submodules are responsible for CLI arguments parsing and user I/O.

All of these modules altogether provide a robust system that manages to filter the frames efficiently.

The pipeline submodule is designed with the separation of concerns principle in mind. The pre-filtering process is separated into three tasks: frame retrieval, frame processing and frame sending. The frame retrieval task consists of an infinite loop that polls the frames from the RX queue of the NIC. After the frames are received, the frame processing takes place, where

Code listing 3.2 *pipeline.c* - pipeline_thread_sender() method

```
while (rte_atomic32_read(&pipe->finished) == 0)
    {
        cur_tsc = rte_rdtsc();
        diff_tsc = cur_tsc - prev_tsc;
        tx_pkts = 0;

        if(unlikely(diff_tsc > drain_tsc)) {
            tx_pkts += rte_eth_tx_buffer_flush( pipe->port_conf->tx_port,
                                                queue_id,
                                                pipe->tx_pool);
            prev_tsc = cur_tsc;
        }

        rx_pkts = rte_ring_dequeue_burst(   pipe->ring_sender,
                                            (void *) pkts,
                                            BURST_SIZE,
                                            NULL);
        if(likely(rx_pkts != 0))
            tx_pkts += rte_eth_tx_burst(pipe->port_conf->tx_port,
                                        queue_id,
                                        pkts,
                                        rx_pkts);

        rte_pktmbuf_free_bulk(pkts, rx_pkts);
        rte_atomic64_add(&pipe->sender_seen, rx_pkts);
        rte_atomic64_add(&pipe->sender_dropped, rx_pkts - tx_pkts);
    }
```

the frames are classified and processed (encapsulated). After all the frames are processed, the sending task is initiated, so that all the successfully processed tasks are sent to the designated agent endpoints. Each of these tasks can be run on different worker threads, therefore maximizing the efficiency of the CPU and I/O. Small problem that might arise is the question whether an efficient method exists for thread-safe data passing between the different threads. The DPDK framework has a beautiful solution — the `rte_ring` structure implementation contains both multi-consumer and multi-producer equivalents. Specific implementation is decided during the initialization of the ring structure via the ring-specific flags. For instance, in the scenario with one frame retrieval thread and multiple processing threads, the combination (logical OR) of RING_F_SP_ENQ and RING_F_MC_RTS_DEQ flags[7] is used. One of the many instances where the reading packets from the ring is happening is shown in the Code listing 3.2.

Apart from the variables declaration, this is the concrete implementation of the sender thread used in the pipeline submodule. The body of the function is encapsulated in the while loop (that is ended when the `SIGSEGV` or `SIGINT` interrupts happen), where the packets are polled from the shared ring using the `rte_ring_dequeue_burst()` method. After some of the packets have been polled (the amount of packets is indicated by the return value of the dequeue method), they are inserted into the TX queue and ultimately freed (as the TX queue caches the packets locally, they can be freed immediately after the insertion into the TX queue). There is also a timer set up that periodically flushes the TX buffer, so the packets are sent even if the threshold for the flush is not yet reached (the timer makes sure that the packets are sent as soon as possible, even when small volume of packets is received). The frequency of the timer is based on the CPU ticks tracked by the TSC register (present on all x86 processors starting from Intel Pentium [20]),

[7] SP_ENQ — Single Producer Enqueue; MC_RTS_DEQ — Multi-consumer, Relaxed Tail Sync Dequeue

where the formula is as follows:

$$drain\_tsc = \frac{\texttt{rte\_get\_tsc\_hz()} + uSPERS - 1}{uSPERS} * 100$$

and the $uSPERS = 10^6$ (uSPERS indicates the amount of microseconds in one second). There are also thread-safe operations to update the statistics of the pre-filter, namely the number of the successfully sent packets and the number of all the packets seen in this thread.

The DPDK `rte_ring` structure provides safe mechanism for the threads to interchange the data without the need of synchronisation primitives (they are properly implemented in the `rte_ring` source code, so the developers need not to worry about the problems that may occur from the race conditions). The L2 pre-filtering module can therefore be launched in two separate threading modes: **generic** and **specialized**. The generic mode uses every worker thread (every logical thread apart from the main thread, which handles I/O) in the retrieval, processing and sending phase. During the development, it was documented that the RX retrieval functions do not provide thread-safe mechanisms for frame receiving from the same RX queue. There is no mention about this behaviour in the DPDK official documentation, but by experimenting, users may experience undefined behaviour. In the pipeline submodule, the spinlocks are used to solve the race condition between the worker threads – the number of worker threads generic mode starts up can be deduced from the equation:

$$worker\_threads = min(num\_rx\_queues; num\_tx\_queues)$$

This implementation is based on the fact that the DPDK indexes the available NIC queues as [0; *max*], where the *max* is stored in the `rte_eth_dev_info` structure (fields max_rx_queues and max_tx_queues respectively).

The specialized mode uses the separation of concerns approach in the worker threads distribution. One worker thread is designated for the frame retrieval from the queue 0 (which is always present in NIC) and one worker thread is designated for the packet sending, also from the queue 0. Every other worker thread is set for the packet processing, as this task is considered to be the computationally harder than the packet RX/TX. With the correct implementation, this approach employs effectively every core without the need for any thread-safe signalization (apart from the `rte_ring` structure setup). The utilization of the *pipeline-like* model in combination with the DPDK zero-copy mechanism creates a balanced approach with even load distribution between the worker threads.

Lastly, the flow hashing mechanism is used to further accelerate the processing speed. The DPDK provides the hash table interface via the `rte_hash` structure. During the initialization of hash table, programmers can tweak multiple options to their specific needs. For example, the proposed infrastructure modifies the generic field *extra_flag*, in which it sets up the property for concurrency mechanisms. Generally, there are plenty of optional features that DPDK hash table offers, such as specifying the used hash function (the default is set to Cuckoo hashing algorithm). As stated in the official documentation [21], one of the proper use cases for which the DPDK hash tables were designed for is flow classification. This is exactly what the hash table is used for in the L2 pre-filtering module — it enables the high-speed packet classification, so the designated agent endpoint address can be determined quickly. The key is set to be the combination of the MAC addresses (IPv4 addresses in the L3 pre-filtering module) and the data is the MAC address of the designated agent endpoint. The designated agent can be determined by the load-balancing algorithm, which in this thesis was set to be a simple round-robin algorithm.

It is possible to use hashing even more efficiently in some specific scenarios. Assuming that the RX NIC supports the RSS, the DPDK offers methods on using the hash computed by NIC instead of computing custom hashes for the hash table. With proper hardware, this method may accelerate the frame processing task even further.

3.4 L3 Pre-filter Module

As stated in the previous section, the L2 pre-filtering module has its drawbacks. For instance, it is limited to monitoring the broadcast domain of the network it is currently deployed in. This limitation was effectively removed by modifying the pre-filtering module to create the flows from the IPv4 header, instead of the Ethernet II header. The internal structure of the submodules is exactly the same as in the L2 pre-filtering module. The only differences are as follows:

- The flows are generated using the source and destination IPv4 addresses. The main change is that it enables the traffic monitoring even from the different networks, as the source and destination addresses are not changed in transit by default (of course, there are numerous malicious networking attacks such as IPv4 spoofing). One of the use cases may be the use in high-speed transit networks, described later in the section.

- The frames with different *EtherType* than 0x0800 (IPv4) are dropped during the processing task, as the flows require the IPv4 addresses for the hashing mechanism. This behaviour may sound strict, but it was decided to be the best effort/speed implementation for this use case. Other implementations may include the IPv6 extension for the IPv6 protocol support, either with the custom hash table (one for IPv4 and one for IPv6) or with the same hash table and IPv4 address translation (the IPv4 address would be translated to the IPv6 address space based on the algorithm described in [22]).

- The ACLs used in the L3 pre-filtering module are much more robust than in L2 pre-filtering module, mostly due to the fact that the DPDK offers native ACL structures and macros such as `rte_acl_ctx` and `RTE_ACL_RULE_DEF()`. The ACLs use dynamic structures and variety of techniques to accelerate the packet filtering, such as ACL categories, rule templates and ACL memory alignment. The ACLs are managed by the ACL context (`rte_acl_ctx` structure), that needs to be "built" each time the ACL rules are updated. This does not pose a problem in the proposed infrastructure, as the ACL rules are defined once via the startup CLI arguments. The complexity of the ACLs allow the users to further specify the filtered hosts in the *A.B.C.D/MM* notation (the mask is in the CIDR notation). For example, to filter traffic from a single host with the IP address 1.2.3.4, the pre-filter is launched with the "-T 1.2.3.4/32" CLI argument.

The versatility of the L2 pre-filtering module code led to the easy implementation of the L3 pre-filtering module. The reusability of the structures and processes is obvious from the similarities found in the source code. With enough time and resources, it is possible to completely integrate both of the solutions into one complex solution, where the desired OSI layer would be specified.

In conclusion, both of the L2 and L3 pre-filtering modules have their advantages and disadvantages. There is difference between their functionality that is reflected in their use cases. The topology of the monitored network is a significant factor that affects the desired choice. L2 pre-filtering module is useful when the users want to monitor specific broadcast domain, such as one VLAN subnet, in which the L2 addresses could be straightforwardly correlated to the endpoints. On the other hand, L3 pre-filtering module can be used in the transit networks where there is a high-speed transport medium between two routers. In this topology, the L3 pre-filtering module could *sniff* the packets and forward them to the desired agent(s). This would not be possible with the L2 pre-filter, as the source and the destination MAC addresses would belong to the two routers at the ends of the high-speed medium. However, with the IPv4 addresses correlation in the L3 pre-filtering module, it is possible to pre-filter the traffic for the monitoring appliances, such as Suricata and ipfixprobe.

3.5 Agent Modules

To achieve fast transport of the packets from the pre-filtering mechanism to the network security applications, an encapsulation process happens on the pre-filtering module. However, as the encapsulated packets do not form traditional frames (the encapsulation creates an L2 tunnel between the pre-filter and endpoint device), they must be decapsulated on the endpoint device before the network security application processes them. The difficulty of this task consists of two main problems:

- The RX and decapsulation process must be at least as rapid as the TX speed of the pre-filtering device. If these requirements are not met, the bottleneck is expected to be created at the endpoint device.
- After the decapsulation process, an efficient way to pass packet data to the network security applications must exist. As there are numerous of third-party applications, an integration into each of the designated applications must exist (as seen later in this chapter, there are two main ways of integrating the DPDK framework with third-party applications).

The problem of rapid RX and decapsulation process can be solved by creating an **agent** application, running at the same endpoint as the network security application. The sole purpose of an agent application is to form another side of the L2 tunnel with the pre-filtering device. This however may pose a problem, as a universal implementation of agent module is impossible, simply due to the different source codes and APIs of the network security applications. Instead of this, customized agent modules are used for every network security application. The agent module can be integrated into an existing solution by two methods: **standalone agent** and **agent plugin**. The core difference of these two solutions is in the amount of running applications on the endpoint device, as shown in Figure 3.5.

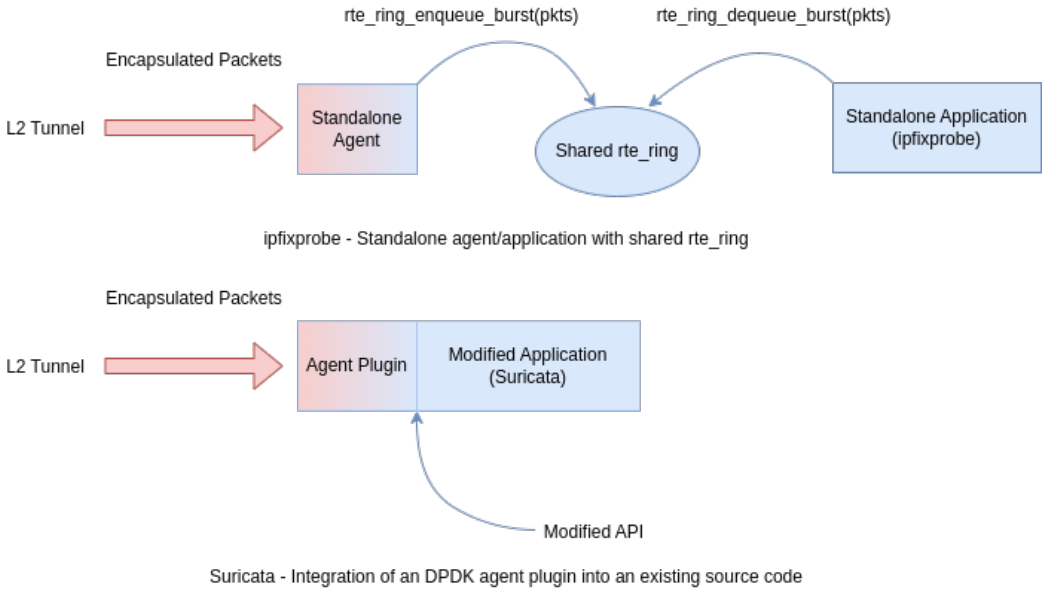

Figure 3.5 Agent integration - red flow contains marked packets, blue flow is after decapsulation

When creating a standalone agent application, one of the applications (agent module or network security application)[8] is started as a primary DPDK process, while the other one is

[8]The decision depends mainly on the developers, as there is no real benefit in running the application as primary

started as a secondary process. This way, the EAL is initialized only once and the memory structures are shared between the processes. As with almost every memory structure in DPDK (`rte_pktmbuf`, `rte_hash`, …), corresponding lookup method exists, so the secondary application can find the structure by its name allocated by the primary application.

In this proof-of-concept infrastructure, the transport of the decapsulated packets for the *ipfixprobe* application is realised via the shared `rte_ring` structure. When launching the agent module, you can specify the ring name with the parameter `-r <NAME>`. The agent module was set to be launched as a primary DPDK application, therefore after the launch a ring structure is created in the shared memory. After the ring is created in the memory, ipfixprobe can be launched as an secondary application with the input interface set to be the DPDK ring structure via the CLI parameters `-i 'dpdk-ring;r=<NAME>;e= --proc-type=secondary'`.

When both standalone applications are operational, encapsulated packets received by the RX NIC interface bound with the DPDK driver undergo the following process:

DPDK Agent Module

1. Packet burst is received in the RX queue of the NIC by the method `rte_eth_rx_burst()`.
2. Each packet is validated – the *EtherType* field in the L2 is checked to determine, whether the received packet contains encapsulated payload. If this is not the case, the packet is dropped, as it does not belong to the DPDK interface.
3. All of the valid packets have first 14 bytes of the L2 header removed by the method `rte_pktmbuf_adj()`.
4. Trimmed packets are then inserted into the shared `rte_ring` structure allocated by the agent through the method `rte_ring_enqueue()`.

ipfixprobe

5. Burst of packets is received from the shared `rte_ring` structure with the name specified in the CLI argument by the method `rte_ring_dequeue_burst()`.
6. Packets are parsed by the ipfixprobe method `parse_packet()`.
7. Packets are processed by the ipfixprobe kernel.

By using the shared DPDK `rte_ring` structure, there is a minimal number of copy operations necessary when passing the packet data between the two applications — it only requires a single 32-bit Compare-and-Swap operation and is also optimized for the bulk operations to reduce the cache miss rate (more about the DPDK ring structure can be found in the documentation [23]).

The standalone agent integration requires a development of a standalone application, which can be tedious and unnecessary in some use cases. Sometimes it is more convenient to change a few lines of a source code of a third-party application, rather than developing a brand new agent module. This option is shown in the bottom part of Figure 3.5, where the whole process takes place in the enhanced version of the network security application. The retrieval of the marked packets and the adjacent decapsulation is in the code supplied by the DPDK developer, and the decapsulated packets are then forwarded to the application's kernel via the API. Depending on the difficulty of the process integration into existing application ecosystem, the creation of a DPDK agent plugin may be a preferable choice over the development of a standalone DPDK application.

The integration of a DPDK interface into *Suricata* is an upgrade of a recent proof-of-concept implementation. As proved in the thesis [24], the DPDK framework can significantly improve the overall performance of the Suricata `AF_PACKET` interface. For example, the DPDK interface with 8 NIC queues and no Suricata ACLs has a drop rate of only around 2% with a throughput of 40 Gbps. Without the DPDK integration, the drop rate of an `AF_PACKET` interface is much higher with a substantial 23%. This performance improvement is even more remarkable when 16 worker

Code listing 3.3 *source-dpdk.c* - Modified ReceiveDPDKLoop() method

```
if( rte_pktmbuf_mtod(p->dpdk_v.mbuf, struct rte_ether_hdr *)->ether_type
    == rte_be_to_cpu_16(0xf1f0)) {
    PacketSetData(  p,
                    rte_pktmbuf_mtod(p->dpdk_v.mbuf, uint8_t *) + 14,
                    rte_pktmbuf_pkt_len(p->dpdk_v.mbuf) - 14);
} else {
    PacketSetData(  p,
                    rte_pktmbuf_mtod(p->dpdk_v.mbuf, uint8_t *),
                    rte_pktmbuf_pkt_len(p->dpdk_v.mbuf));
}
```

threads (8 physical CPU cores and 8 hyperthreaded cores) are employed with 16 queues. "In DPDK, 16 queues allow better data reception and thus handle speed of 40 Gbps with practically zero packet loss."

Since Suricata version *7.0.0-beta1*, an initial support for DPDK framework exists in both IDS and IPS mode [25]. This integration supports the standard methods for packet RX via the DPDK framework. For this thesis, a modification of the existing DPDK code in Suricata is necessary – the packets retrieved by the NIC queues need to be decapsulated before they are processed by the Suricata kernel. This slight modification must not interfere with other non-marked packets, as it would disrupt the network traffic received from other sources. The changes made to the Suricata code are shown in the Code listing 3.3. This specific code is a modified version of a standard DPDK retrieval loop of `ReceiveDPDKLoop()` method located in the *source-dpdk.c* file. As the packets are received by the RX NIC queue, the *EtherType* field is checked to determine whether the frame header contains an encapsulated payload. If this is the case, first 14 bytes are trimmed from the packet, so that only the original headers and payload exist. Otherwise, the packet is simply forwarded to the Suricata kernel unchanged. This is a deliberate behaviour as we want to receive not only the marked packets but also the generic traffic flow (this behaviour can be simply changed in the *else* clause of the code, where the packet would be dropped).

In this thesis, the Suricata and ipfixprobe applications were chosen intentionally to show the difference in the DPDK integration into various network security applications. Depending on the resources and the application source code, developers can choose if they want to create a standalone agent application or to use a simple DPDK agent plugin.

Chapter 4

Testing and Evaluation

In chapter 3, a DPDK-based infrastructure and its particular implementation were introduced for scaling security applications. Let's discuss the efficiency and resource utilization of the proposed infrastructure in the defined tests and evaluate the result dataset.

Firstly, let's look at the results evaluation with a small number of packets. This test was conducted to determine whether the L3 pre-filtering module is working correctly in both generic and specialized mode. The following use cases were tested:

1. Table 4.1 – L3 pre-filtering module with no ACLs (full forwarding)
2. Table 4.2 – L3 pre-filtering module with one ACL
3. Table 4.3 – L3 pre-filtering module with 16 ACLs (maximum of ACLs currently available)

This use cases were chosen so both the pre-filtering and forwarding parts of the L3 pre-filtering module were tested. Testing with different subnets was not mandatory, but it further proves the correctness of the L3 pre-filtering module functionality. In all of the use cases, the infrastructure consisted of:

- Single custom Scapy packet generator *tester.py* (modified version of a generator.py introduced in section 3.2)
  - OS: Ubuntu 22.04 Jammy (virtualized)
  - CPU: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
  - vCPU cores used: 1
  - RAM: 1 GiB
  - NIC: 1x DPDK-complaint e1000 (+ 1 used for SSH)
- Single L3 pre-filtering module
  - OS: Ubuntu 22.04 Jammy (virtualized)
  - CPU: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
  - vCPU cores used: 4 (1 RX, 1TX, 1 processing and 1 CLI)
  - RAM: 4 GiB
  - NIC: 2x DPDK-complaint e1000 (+ 1 used for SSH)

- Single endpoint with ipfixprobe and its associated agent
  - OS: Ubuntu 22.04 Jammy (virtualized)
  - CPU: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
  - vCPU cores used: 2
  - RAM: 2 GiB
  - NIC: 1x DPDK-complaint e1000 (+ 1 used for SSH)

To make the packet dataset universal, in all of the use cases tested, the source address of the packets belonged in the subnet `10.0.0.0/28` (IP addresses ranging from `10.0.0.1` up to `10.0.0.14`) and the destination address was constantly set to be `10.0.1.1`. All of the tests use the Scapy generator in combination with *tcpreplay* to generate a stable 10^5 pps stream[1] of 512 B L3 packets (plus headers). The total number of bytes generated in the timespan of 10 seconds is therefore equal to:

$$10^5 \text{ pps} * 10 \text{ s} * (14 \text{ B L2} + 20 \text{ B L3} + 512 \text{ B payload}) = 10^6 \text{ pkts} * 574 \text{ B} = 574 \text{ MB}$$

Table 4.1 L3 pre-filtering module with no ACLs applied

Pre-filter Mode	Specialized Mode			Generic Mode		
Total Packets Generated	10^4	$5 * 10^4$	10^5	10^4	$5 * 10^4$	10^5
Received pkts	10,000	50,000	100,000	10,000	50,000	100,000
Processed pkts	10,000	50,000	100,000	10,000	50,000	100,000
Sent pkts	10,000	50,000	100,000	10,000	50,000	100,000
Agent received pkts	10,000	49,384	97,855	10,000	49,929	99,584
ipfixprobe pkts	10,000	49,384	97,855	10,000	49,929	99,584
Pre-filtering drop rate	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Transfer drop rate	0.00%	1.23%	2.10%	0.00%	0.14%	0.41%
Agent drop rate	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

As shown in Table 4.1, with no ACLs included, the performance of the L3 pre-filtering module is terrific. The drop rate of a sole L3 pre-filtering module is 0% when small amounts of data are received. The same goes for the intercommunication between the DPDK agent module and the ipfixprobe running as a secondary application. Packet drop rate was experienced only in the transfer between the virtual machines – this drop rate varied between the runs, which may indicate that the measurements on the virtual machines are not stable for this metric (the transport medium is heavily dependent on the hypervisor machine, which may drop some packets even though DPDK applications can handle the load). This can be correlated with the increased drop rate of packets when using multi-threaded (specialized) mode of the L3 pre-filtering module, where the transfer drop rate is at 2.1% for 10^5 packets.

In Table 4.2, results are shown from the measurements where one sample ACL is applied in the L3 pre-filtering module. There is no significant drop in the performance metrics evident from the data. However, it was observed that the transfer drop rate had increased for both generic and specialized mode when testing 10s run (flow rate was approximately 460 Kbps) — the drop rate for specialized mode had risen to 6.87% ($\approx$ 450% increase from no ACLs) and for the generic mode to 1.39% ($\approx$ 240% increase from no ACLs). This may be due to the virtualized infrastructure, as the drop rate varied in the different runs the same way as in the no ACLs testing. Nonetheless, the pre-filtering drop rate and the agent drop rate stood still at 0.00%, meaning that the L3 pre-filtering infrastructure is capable of handling higher bandwidth even

Table 4.2 L3 pre-filtering module with one ACL applied

Pre-filter Mode	Specialized Mode			Generic Mode		
Total Packets Generated	10^4	$5 * 10^4$	10^5	10^4	$5 * 10^4$	10^5
Received pkts	10,000	50,000	100,000	10,000	50,000	100,000
Processed pkts	10,000	50,000	100,000	10,000	50,000	100,000
Sent pkts	10,000	50,000	100,000	10,000	50,000	100,000
Agent received pkts	10,000	49,464	93,163	10,000	49,878	98,605
ipfixprobe pkts	10,000	49,464	93,163	10,000	49,878	98,605
Pre-filtering drop rate	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Transfer drop rate	0.00%	1.07%	6.87%	0.00%	0.24%	1.39%
Agent drop rate	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

with a single ACL applied. Furthermore, the desired ACL was successfully applied, as the filtered packets did not show up in the ipfixprobe application.

For the last test with a smaller bandwidth speed, the highest number of ACLs that the L3 pre-filtering module supports were applied. The ACLs applied were chosen deliberately to preserve some addresses from the original flow, so the ipfixprobe would still receive a fraction of generated packets. The filtered addresses were chosen accordingly:

- The addresses from range `10.0.0.3` up to `10.0.0.7` were filtered using separate ACLs like "`-F 10.0.0.3/32`"
- The addresses from the subnet `10.0.0.8/28` were filtered altogether with one ACL.
- 5 ACLs that filter the IP addresses from the network `192.168.0.0/16` were chosen randomly to provide more realistic results.
- 5 ACLs that filter the IP addresses to the network `172.16.0.0/12` were chosen randomly to provide more realistic results.

The generator was set to always generate $\frac{1}{5}$ packets in each of the flows `10.0.0.1` $\rightarrow$ `10.0.1.1` and `10.0.0.2` $\rightarrow$ `10.0.1.1`. Other $\frac{4}{5}$ packets were generated from the random filtered flows to generate a load at filtering submodule. Because of this behaviour, *sent pkts* row contains the amount of non-filtered, sent packets.

Table 4.3 L3 pre-filtering module with 16 ACLs applied

Pre-filter Mode	Specialized Mode			Generic Mode		
Total Packets Generated	10^4	$5 * 10^4$	10^5	10^4	$5 * 10^4$	10^5
Received pkts	10,000	50,000	100,000	10,000	50,000	100,000
Processed pkts	10,000	50,000	100,000	10,000	50,000	100,000
Sent pkts	2,000	10,000	20,000	2,000	10,000	20,000
Agent received pkts	2,000	10,000	20,000	2,000	10,000	20,000
ipfixprobe pkts	2,000	10,000	20,000	2,000	10,000	20,000
Pre-filtering drop rate	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Transfer drop rate	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Agent drop rate	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

From Table 4.3 we can see that the filtering submodule fluently and correctly handles the task using one worker vCPU core with 0% drop rate. Only the desired packets reached the ipfixprobe

[1] pps stands for packets per second

agent, proving that the ACLs work precisely (in fact, the filtered packets did not even reach the TX queue of the NIC). This is an indicator that the DPDK ACLs are being used efficiently, even at lower speeds. This fact supports the statement that the DPDK framework is a viable choice not only for high-speed networks, but also for networks with lower bandwidths.

In this second half of the chapter, the true nature of DPDK is explored — the efficiency in the high-speed networks. For this thesis, the testing was done in collaboration with *CESNET* association, which supplied the infrastructure necessary for high-speed network testing. The specifications of the infrastructure used in the testing are following:

- High-speed traffic generator *Spirent TestCenter* — robust solution capable of generating data ranging from the L2 to L7 with 100 Gb/s
- Single L3 pre-filtering module
  - OS: Oracle Linux 8
  - CPU: Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz
  - vCPU cores used: 64
  - RAM: 186 GiB + 31 GiB swap
  - NIC: Customized FPGA Card

Two use cases were tested: the amount of processed packets with no ACLs applied to determine the throughput and forwarding part of the L3 pre-filtering module, and the amount of processed packets with one inbound and one outbound ACL to test the filtering part (there is also listed the amount of filtered packets by the ACLs). For both of these tests a saturated 100 Gbps packet stream was supplied as a source of packets. The specialized mode was used for this use case, as it does not make sense to test the results on one processing thread. The measurements were taken for different packet lengths, ranging between 64 and 1024 bytes. All of the packets contained the L2, L3 (IPv4) and L4 (UDP) header with payload. The results are shown in Table 4.4.

Table 4.4 L3 pre-filtering module with 100 Gbps saturated flow

Packet length	Processed pkts (no ACL)	Processed pkts (2 ACLs)	Filtered pkts
64 B	17,607,242	12,617,598	44,635
128 B	15,481,715	14,187,004	49,690
256 B	13,413,720	13,582,721	50,471
512 B	11,575,206	12,694,365	61,117
1024 B	10,358,616	10,935,624	58,204

From the statistics in the aforementioned table, we can determine the throughput of the NIC by using the formula

$$throughput\ (\text{Gbps}) = \frac{pkts * size * 8}{time * 10^9}$$

which for the pre-filtering without the ACLs equals

$$\frac{17607242 * 64 * 8}{10 * 10^9} = 0.901\ \text{Gbps}$$

$$\frac{15481715 * 128 * 8}{10 * 10^9} = 1.585\ \text{Gbps}$$

$$\frac{13413720 * 256 * 8}{10 * 10^9} = 2.747\ \text{Gbps}$$

$$\frac{11575206 * 512 * 8}{10 * 10^9} = 4.741 \text{ Gbps}$$
$$\frac{10358616 * 1024 * 8}{10 * 10^9} = 8.486 \text{ Gbps}$$

and for the pre-filtering with ACLs applied equals

$$\frac{12617598 * 64 * 8}{10 * 10^9} = 0.646 \text{ Gbps}$$
$$\frac{14187004 * 128 * 8}{10 * 10^9} = 1.453 \text{ Gbps}$$
$$\frac{13582721 * 256 * 8}{10 * 10^9} = 2.782 \text{ Gbps}$$
$$\frac{12694365 * 512 * 8}{10 * 10^9} = 5.200 \text{ Gbps}$$
$$\frac{10935624 * 1024 * 8}{10 * 10^9} = 8.958 \text{ Gbps}$$

These results are also nicely summed up in Figure 4.1.

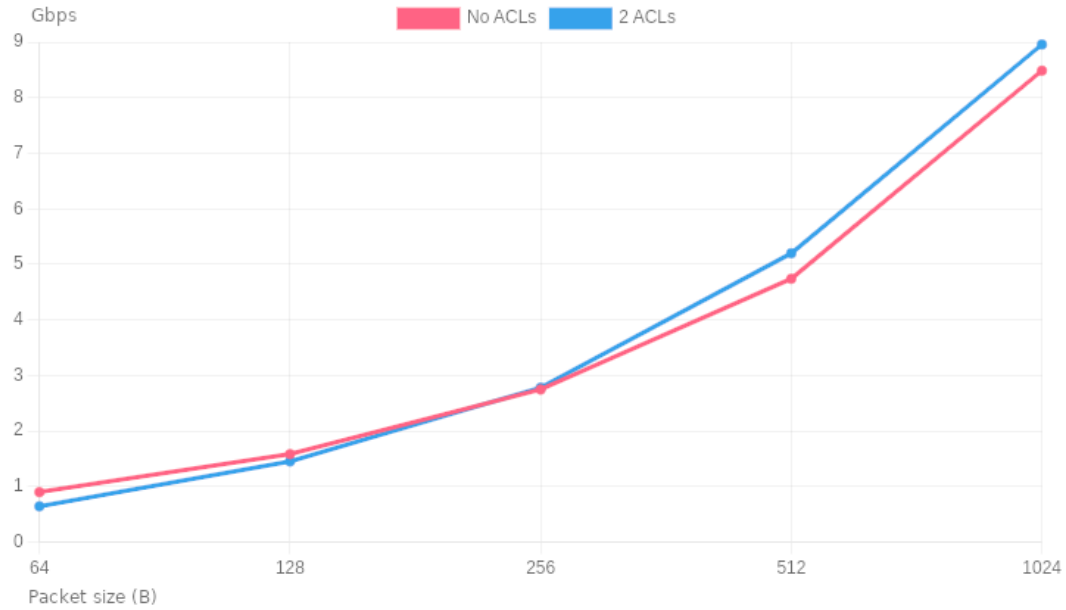

Figure 4.1 L3 pre-filtering module performance with 100 Gbps flow

From the measurements we can see that the L3 pre-filtering module performs better with the packets of size closer to MTU. This is an expected behaviour, as there is more data (L2 and L3 headers) to be processed when receiving smaller packets than there is in the longer ones. The crucial fact in this tests is that all of these measurements were taken using only 1 RX and 1 TX queue. As the NIC supports the RSS[2], NIC's throughput scales additively with the amount of RX and TX queues used. We can therefore safely assume that the full throughput of the NIC with 8 RX and TX queues for packets of size 1024 B is then equal to $8.958 * 8 = 71.664 \approx 72$ Gbps. As the packet size rises to the MTU size, so does the throughput of the L3 pre-filtering module, which ultimately converges to 100 Gbps. This proves that the DPDK framework utilizes the NICs at their full potential.

[2]Receive Side Scaling

Lastly, the agent processing speed can be further increased by running multiple instances of the agent application on the endpoint. The ipfixprobe application supports multiple input sources and therefore can manage multiple DPDK ring structures, created by the agent application(s). There is however a slight issue with this reasoning – only one of the agent applications can run as a primary DPDK application, as the EAL can not be initiated and freed multiple times at the same time. This limitation can be bypassed by using the EAL arguments when launching the agent applications. Even better solution is to use one agent application that will scale the DPDK ring structures regarding to the desired amount of NICs.

Chapter 5
Conclusion

The goal of this thesis was to design a concrete proof-of-concept infrastructure using the DPDK framework that would pre-filter the packet flows between the network security applications evenly while preserving the packet flows. The infrastructure should also handle the network traffic correctly even in the high-speed networks.

The study was conducted whether the DPDK framework is a viable choice for this thesis. Plenty of software libraries that handle packet processing and filtering were found, but none of them fit the assignment of the thesis better than DPDK. As for the facts provided in the earlier parts of this thesis, the DPDK offers a handy solution when high-speed network applications need to be developed.

The proof-of-concept infrastructure was designed to filter L2 or L3 traffic using the access control lists. The infrastructure consists of the pre-filtering module, which processes the packet flows, filters the traffic and ultimately encapsulates them and sends them to the designated endpoints. Depending on the network security application, different agent modules receive the encapsulated packets, process them and forward them to the designated application. In this thesis, both ipfixprobe and Suricata were chosen as examples for network security applications.

The proposed infrastructure was tested in two different use cases, both using the ipfixprobe as an endpoint network security application. The small data throughput use case was tested to determine the correctness of the filtering and forwarding module with small flow rate. The testing of this use case was successful, as there seems to be no problem with both generic and specialized mode of the L3 pre-filtering module. The processing drop rate of the L3 pre-filtering module and the agent module were both 0%, proving that the DPDK framework does not cause halting problems with small data. In the second use case, the tests were conducted on the L3 pre-filtering module with and without the ACLs when 100 Gbps needed to be processed for 10 seconds. The tests with high-speed networks showed that the single NIC queue implementation can handle $\sim$ 10 Gbps. By using all NIC queues, it is achievable to get to desired speeds in order of 100 Gbps.

Due to the limited scope of the thesis, sample proof-of-concept infrastructure was implemented, leaving room for a potential improvement of both the pre-filtering module and the agent module. For instance, in the aforementioned pre-filtering module's implementation, only one of the potentially many available RX and TX queues are used (the specific number of queues depend on the NIC). The RSS hashing functions could also be used to decrease the load on the processing threads by using already calculated hashes from the RSS in the flow classification. Additionally, in high-end CPUs where more worker threads are available, this creates a room for experimenting with various combinations with multiple frame retrieving, processing and sending worker threads. The overall performance and distribution is expected to be even better than the standard pipeline approach used in this thesis, but further research is required.

In summary, all of the goals set for this thesis were achieved, with no result being inconclusive. The proposed proof-of-concept infrastructure proved that the DPDK framework is a feasible solution for flow-preserving, load-balancing applications.

Bibliography

1. BEALE, Jay; FOSTER, James C.; POSLUNS, Jeffrey; CASWELL, Brian. *Snort Intrusion Detection 2.0*. Rockland, MA: Syngress, 2003. ISBN 1-931836-74-4.
2. *PCAP_NEXT_EX(3PCAP) MAN PAGE* [online]. 2022. [visited on 2023-03-03]. Available from: `https://www.tcpdump.org/manpages/pcap_next_ex.3pcap.html`.
3. *Building libpcap on Windows with Visual Studio* [online]. 2021. [visited on 2023-03-31]. Available from: `https://github.com/the-tcpdump-group/libpcap/blob/master/doc/README.Win32.md`.
4. *WinPcap Official Website* [online]. 2018. [visited on 2023-03-03]. Available from: `https://www.winpcap.org/default.htm`.
5. *Npcap Official Website* [online]. 2021. [visited on 2023-03-03]. Available from: `https://npcap.com/`.
6. *FreeBSD Documentation, Chapter 32. Firewalls* [online]. 2023. [visited on 2023-03-04]. Available from: `https://docs.freebsd.org/en/books/handbook/firewalls/#firewalls-ipfw`.
7. *BPF Features by Linux Kernel Version* [online]. 2023. [visited on 2023-03-12]. Available from: `https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#xdp`.
8. *XDP benchmark baseline* [online]. 2023. [visited on 2023-03-12]. Available from: `https://github.com/tohojo/xdp-paper/blob/master/benchmarks/bench01_baseline.org#initial-data-from-jespers-runs`.
9. *XDP Packet Processing Overview* [online]. 2023. [visited on 2023-03-12]. Available from: `https://www.iovisor.org/technology/xdp`.
10. *PF_RING™* [online]. 2023. [visited on 2023-03-17]. Available from: `https://www.ntop.org/products/packet-capture/pf_ring/`.
11. *PF_RING ZC (Zero Copy)* [online]. 2023. [visited on 2023-03-17]. Available from: `https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/`.
12. *PF_RING FT (Flow Table)* [online]. 2023. [visited on 2023-03-17]. Available from: `https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-ft-flow-table/`.
13. *NVIDIA PMDs* [online]. 2023. [visited on 2023-03-31]. Available from: `https://developer.nvidia.com/networking/dpdk`.
14. *Memory in DPDK, Part 1: General Concepts* [online]. 2023. [visited on 2023-03-31]. Available from: `https://www.dpdk.org/blog/2019/08/21/memory-in-dpdk-part-1-general-concepts/`.

15. *Suricata Features - Official Website* [online]. 2023. [visited on 2023-04-01]. Available from: https://suricata.io/features/.

16. ČEJKA, Tomáš; ŽÁDNÍK, Martin. Preserving Relations in Parallel Flow Data Processing. In: *Autonomous Infrastructure, Management and Security*. 2017.

17. Packet Classification and Access Control - DPDK Programmer's Guide. In: [online]. 2015 [visited on 2023-04-13]. Available from: https://doc.dpdk.org/guides/prog_guide/packet_classif_access_ctrl.html.

18. RFC 894 - A Standard for the Transmission of IP Datagrams over Ethernet Networks. In: [online]. 1984 [visited on 2023-04-08]. Available from: https://www.rfc-editor.org/rfc/rfc894.

19. *Ethernet Jumbo Frames*. 2009-11. Tech. rep. Ethernet Alliance. Available also from: http://www.ethernetalliance.org/wp-content/uploads/2011/10/EA-Ethernet-Jumbo-Frames-v0-1.pdf.

20. WIKIPEDIA CONTRIBUTORS. *Time Stamp Counter — Wikipedia, The Free Encyclopedia*. 2023. Available also from: https://en.wikipedia.org/w/index.php?title=Time_Stamp_Counter&oldid=1137210613. [Online; accessed 19-April-2023].

21. PAGH, Rasmus. *Cuckoo Hashing for Undergraduates*. 2006. Tech. rep. IT University of Copenhagen. Available also from: https://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf.

22. RFC 7915 - IP/ICMP Translation Algorithm. In: [online]. 2016 [visited on 2023-04-08]. Available from: https://www.rfc-editor.org/rfc/rfc7915.

23. Ring Library - DPDK Programmer's Guide. In: [online]. 2018 [visited on 2023-04-21]. Available from: https://doc.dpdk.org/guides/prog_guide/ring_lib.html.

24. ŠIŠMIŠ, Lukáš. *Optimization of the Suricata IDS/IPS*. Brno, 2021. MA thesis. Brno University of Technology, Faculty of Information Technology, Department of Computer Systems (DCSY).

25. ChangeLog - Suricata git repository maintained by the OISF. In: [online]. 2022 [visited on 2023-04-22]. Available from: https://github.com/OISF/suricata/blob/master/ChangeLog.

Content of the attached media

- `bi-bap/` — Git repository folder
  - `README.md` — Brief description about the media content
  - `agent-ipfixprobe/` — Folder with ipfixprobe application and its associated agent
    - `src/`
    - `Makefile`
    - `dpdk-setup.sh` — Script used to set up DPDK environment on VMs
  - `agent-Suricata/` — Folder with Suricata modified files (Suricata not included)
    - `suricata/`
    - `dpdk-setup.sh` — Script used to set up DPDK environment on VMs
  - `generator/` — Folder with packet generator application files
    - `src/`
    - `Makefile`
    - `dpdk-setup.sh` — Script used to set up DPDK environment on VMs
    - `generator.py`
    - `tester.py`
  - `l2filter/` — Folder with L2 Pre-filtering Module application files
    - `src/`
    - `Makefile`
    - `dpdk-setup.sh` — Script used to set up DPDK environment on VMs
  - `l3filter/` — Folder with L3 Pre-filtering Module application files
    - `src/`
    - `Makefile`
    - `Doxyfile`
    - `dpdk-setup.sh` — Script used to set up DPDK environment on VMs
  - `text/` — Thesis PDF and LaTeX source files
    - `thesis.pdf` — This document