BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

USING METADATA TO OPTIMIZE THE SURICATA IDS/IPS

OPTIMALIZACE SYSTÉMU SURICATA PROSTŘEDNICTVÍM METADAT

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ANDREI SHCHAPANIAK**
AUTOR PRÁCE

SUPERVISOR **Ing. LUKÁŠ ŠIŠMIŠ**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

144390

Institut:	Department of Computer Systems (UPSY)
Student:	**Shchapaniak Andrei**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Using Metadata to Optimize the Suricata IDS/IPS**
Category:	Networking
Academic year:	2022/23

Assignment:

1. Study the DPDK library, DPDK Prefilter, and open-source Intrusion Detection System (IDS) / Intrusion Prevention System (IPS) Suricata.
2. Analyze the storing and reading options of packets' metadata.
3. Design a module for DPDK Prefilter software for metadata extraction of the network packets. Propose how metadata can be passed between the systems and used by the Suricata IDS/IPS.
4. Implement the proposed metadata design in Suricata and DPDK Prefilter. Test the created implementation.
5. Discuss achieved results and the possibilities of further improvements.

Literature:

- Dle pokynů vedoucího.

Requirements for the semestral defence:

- Points 1 to 3 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Šišmiš Lukáš, Ing.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	31.10.2022

Abstract

As the Internet continues to grow and evolve, cyber-attacks have become more harmful and severe. Therefore, it is important to have effective detection and prevention systems. The need for such systems is becoming more demanding because the damage caused by cyber-attacks can be devastating to individuals and organizations. This motivates the development of high-performance detection and prevention systems that are able to detect and prevent cyber-attacks in a more effective way. Specifically, this thesis is focused on one of such systems – Suricata. It is an open-source IDS/IPS system that is widely used in the industry due to its advanced capabilities and flexibility. The ultimate goal of the thesis is to design, implement and evaluate the transmission of per-packet metadata. Metadata could be added by smart network interface cards (SmartNICs). DPDK Prefilter was used in conjunction with Suricata to simulate the specialized hardware. It can simulate the transmission of metadata to Suricata. The impact of metadata and the results of the experimental evaluation will be discussed in detail at the end of this bachelor's thesis.

Abstrakt

Jak Internet stále roste a vyvíjí se, kybernetické útoky se stávají škodlivějšími a závažnějšími. Je tedy důležité mít účinné systémy detekce a prevence. Potřeba takových systémů se stává urgentnější, protože škody způsobené kybernetickými útoky mohou být pro jednotlivce i organizace katastrofické. Toto motivuje k vývoji vysoce výkonných systémů detekce a prevence, které jsou schopny efektivněji detekovat a bránit kybernetickým útokům. Tato bakalářská práce se konkrétně zaměřuje na jeden z těchto systémů – Suricatu. Jedná se o IDS/IPS systém s otevřeným kódem, který je v průmyslu široce používán díky svým pokročilým schopnostem a flexibilitě. Základním cílem této práce je navrhnout, implementovat a vyhodnotit přenos metadat pro každý paket. Metadata mohou být přidána pomocí chytrých síťových karet (SmartNICs). DPDK Prefilter byl použit společně se Suricatou k simulaci specializovaného hardwaru. On dokáže simulovat přenos metadat do Suricaty. Dopad metadat a výsledky experimentálního hodnocení budou podrobně popsány na konci této bakalářské práce.

Keywords

Suricata, AF_PACKET, DPDK, Prefilter, IDS, IPS, Suricata optimization, Metadata, eBPF, Network monitoring

Klíčová slova

Suricata, AF_PACKET, DPDK, Prefilter, IDS, IPS, Optimalizace Suricaty, Metadata, eBPF, Monitorování sítě

Reference

SHCHAPANIAK, Andrei. *Using Metadata to Optimize the Suricata IDS/IPS*. Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Lukáš Šišmiš

Rozšířený abstrakt

Sítě jsou nedílnou součástí moderního světa a hrají klíčovou roli v přenosu dat a informací. S nástupem internetu se výrazně změnil způsob, jakým lidé komunikují a získávají informace, což ovlivnilo mnoho aspektů života. Jednou z největších výhod modernizace sítí a internetu je zlepšení efektivity a rychlosti komunikace a přenosu dat. Lidé mohou snadno komunikovat s lidmi na druhé straně světa, sdílet informace a získávat nové znalosti a zkušenosti. Moderní sítě umožňují také vzdálenou práci a vzdělávání, což se v poslední době ukázalo jako velmi užitečné.

S narůstající závislostí na internetu se zvyšuje i počet hrozeb, které ohrožují bezpečnost uživatelů. Mezi tyto hrozby patří například škodlivý software, phishing, ransomware a další. Tyto hrozby mohou způsobit ztrátu dat, finanční škody a narušení soukromí uživatelů. Proto je důležité být opatrný a chránit se proti těmto hrozbám. Existují různé způsoby, jak se chránit proti hrozbám na internetu. Mezi tyto nástroje patří například firewally, antivirové programy nebo nástroje pro detekci podezřelých činností v sítích. Jedním z takových softwarových nástrojů jsou IDS/IPS systémy. Tyto nástroje slouží k detekci a prevenci útoků v počítačových sítích.

Tato práce je zaměřena na provedení experimentů se Suricatou. To je jeden z několika vysoce výkonných softwarových nástrojů s otevřeným kódem, který se využívá pro monitorování sítě a detekci hrozeb. Výhodou Suricaty je vysoká výkonnost, která je dosažena kvalitní multi-vláknovou architekturou. Suricata čelí významnému problému zpracování veškerého příchozího síťového provozu v reálném čase, což může vést k zmeškání hrozeb nebo falešných poplachů. K řešení tohoto problému stále probíhají různé výzkumy, které se zaměřují na zlepšení výkonu Suricaty.

Pokročilé síťové karty mohou obsahovat hardwarové offloady, které mohou pomoci Suricatě v rychlejším zpracování paketů. Metadáta, jako jsou extra informace o paketu, mohou být přenesena z hardwaru do Suricaty. Vývoj hardwaru je ale složitý a vyhodnocení, jaká metadata mohou Suricatě nejvíce pomoci, může být velmi užitečné. Avšak před tím, než se implementuje v hardware, je vhodné to otestovat v softwaru, protože to bude méně nákladné. Pro tyto účely existuje DPDK Prefilter, který umožňuje simulaci hardwarových funkcí v softwaru.

Cílem práce bylo navrhnout, implementovat a vyhodnotit použití metadat v rámci systému DPDK Prefilter. Tento nástroj má možnost simulovat použití hardwarových offloadů přímo v softwaru. Výsledky této práce mohou hrát důležitou roli při dalším rozšiřování možností Suricaty, které mohou být spojeny s metadaty.

V první kapitole bakalářské práce je představen systém Suricata. Na začátku je popsána jeho architektura a části, ze kterých se skládá. V další sekci je rozepsána dekódovací část podrobněji, protože na ni především byla zaměřena metadata. Pak je uveden krátký přehled o Internetu a síťových paketech, znalost kterých pomůže čtenáři porozumět účelu metadat. Poté se pokračuje strukturou detekčních pravidel. Detekční pravidla byla jedním z důvodů, proč byla metadata reprezentována jako informace ze síťových hlaviček. Další sekce obsahují informace o různých způsobech fungování vláken v Suricatě a modulech pro příjem paketů. AF_Packet a DPDK moduly jsou probrány do hloubky spolu s jejich navazujícími částmi. Také celá subsekce vysvětluje princip fungování eBPF filtru a jeho role v Suricatě.

V další kapitole je představen DPDK Prefilter. Hlavním účelem tohoto nástroje je simulace hardware funkcionality v softwaru. Navíc DPDK Prefilter je dobrým nahrazením eBPF filtru, protože řeší problém blokování sdílené eBPF mapy mezi Suricatou a eBPF programem. Na začátku kapitoly je krátce vysvětleno, co je DPDK Prefilter a jaké má výhody. Také je ukázáno jeho zapojení se Suricatou pomocí DPDK front. V první sekci

se probírá jeho architektura. Další sekce rozvádí téma o dekodovací části DPDK Prefiltru. V této sekci je také zmíněn proces dekódování paketů a bypasování celých toků na základě zpráv od Suricaty, kterými určuje, který tok má být bypasován. Třetí sekce popisuje způsob zapojení a meziprocesorovou komunikaci mezi Suricatou a DPDK Prefiltrem.

Třetí kapitola popisuje motivaci této práce, krátký návrh implementace a použití metadat. První sekce uvádí do tématu jak metadata mohou pomoci Suricatě a kde metadata mohou být uložena. Další sekce je věnována návrhu metadat, kde je ukázána jejich struktura a zdůrazněn důvod výběru metadat jako dat ze síťových hlaviček. Třetí sekce ukazuje proces ukládání metadat do paketů. V další sekci je popsán průběh dohody na stejných metadatech mezi dvěma systémy. To je nutný krok, aby nedošlo k nekonzistenci dat a dalším chybám. V poslední sekci jsou probírány konfigurační soubory pro Suricatu a DPDK Prefilter. To je nutné pro pochopení, co bude přidáno do těchto souborů a proč.

Poslední kapitola popisuje implementaci a testování metadat. V první sekci jsou popsány jednotlivé soubory, které byly přidány do projektu. Také celá subsekce je věnována implementaci způsobu procházení přes metadata a jejich nastavení. Další sekce je věnována integračním testům. Tyto testy slouží pro evaluaci, zda metadata neovlivnily detekční modul Suricaty. Pro tyto účely se používal program `suricata-verify`, pro který byly napsány další integrační testy. Poslední sekce popisuje testy na výkonnost a jejich výsledky. DPDK Prefilter byl spouštěn se Suricatou při různých konfiguracích. Pak se posílaly pakety s různou rychlostí. Na konci sekce jsou zobrazeny grafy pro porovnání.

Závěr práce je věnován sumarizaci výsledků. Na konci je krátce zmíněno o tématu, kterým se dá pokračovat v budoucnosti při rozšíření funkčnosti metadat.

Using Metadata to Optimize the Suricata IDS/IPS

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Lukáš Šišmiš. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

....
Andrei Shchapaniak
May 4, 2023

Acknowledgements

I would like to express my sincere appreciation to my supervisor Ing. Lukáš Šišmiš for his invaluable supervision and provided support to me duration of writing this thesis. I feel incredibly fortunate to have had him as my supervisor, and I am grateful for his contribution to both my academic and personal growth.

Contents

Chapter 1

Introduction

In the past, people used to communicate and exchange information via conventional methods. However, as society progressed, there was a growing demand for a more efficient and effective means of communication and information exchange. In 1969, a military-funded experiment has brought a global system that is used by a significant part of the world's population now. Over time computer networks continued improving, resulting in faster data transfer. However, these improvements have also brought negative consequences. With the increasing functionality of the Internet, new vulnerabilities for hackers are opening up. This has become the reason why creating a completely secure system is extremely difficult. No wonder the common sentiment in the field of cybersecurity is "No single system can be completely secure".

Before we describe methods of protection, we will define what types of attacks exist [19]. The most popular are:

- Denial-of-service (DoS) and distributed denial-of-service (DDoS) attacks. A DoS attack is aimed at overwhelming the system's resources and making them temporarily unavailable to users. A DDoS attack is also an attack on the system's resources, but it originates from many different sources, making it more powerful.
- Man-in-the-middle (MITM) attack. This is a type of attack where the hacker inserts himself between the communication of two parties who believe that they are directly communicating with each other.
- Phishing attack. This type of attack is more oriented toward humans. In essence, the attacker sends fake messages to trick a person into revealing sensitive information.

To be protected against different types of network attacks, people developed different tools to make the network secure. Examples of such systems are firewalls, antiviruses, intrusion detection systems (IDS), intrusion prevention systems (IPS), and others.

Now consider IDS and IPS in more detail, because this thesis is focused on one of such systems. An IDS is a system that monitors network events and analyzes them to detect security incidents and potential threats. An IPS provides more features because it not only detects security incidents but also prevents them [3].

Depending on the location of IDS in the entire network, the system is divided into two types: Network Intrusion Detection System (NIDS) and Host Intrusion Detection System (HIDS). NIDS monitors all network traffic on a specific segment of the network and can be placed at various locations within a network. It raises alerts about potential threats while the HIDS aims to protect individual devices and servers [23].

This thesis attempts to introduce the reader to the system Suricata and its potential to leverage other connected software to enhance its capabilities. It explores the architecture, modes, and settings of Suricata to provide a comprehensive understanding of the system's capabilities.

The goal of this thesis was to explore potential methods of transmitting information to Suricata, followed by the implementation of one of them. In the end, integration and performance tests were performed to evaluate the impact of metadata on the functionality of Suricata.

The results of the metadata evaluation can be used to expand the usage of metadata in the future. For instance, metadata can contain more valuable information, that optimizes packet processing in Suricata. Overall, this thesis provides a valuable contribution to the field of networks and highlights the potential of using metadata to improve the capabilities of Suricata.

Chapter 2

Suricata

Suricata[1] is an open-source network analysis and threat detection software used by most private and public organizations. It was developed by the Open Information Security Foundation (OISF), which is a nonprofit organization aiming at developing and supporting open-source security tools.

Suricata is a powerful IDS/IPS tool that can effectively detect and prevent cyber threats such as network intrusions, malware, and other malicious activity. It provides protection against various types of attacks such as DDoS attacks, Structured Query Language injections (SQLi), phishing, and others. One key advantage of Suricata is its ability to work in combination with other systems such as firewalls, Virtual Private Networks (VPNs), or Security Information and Event Management (SIEM) systems. Additionally, Suricata is compatible with various platforms such as Windows, Mac, and Linux, which makes it flexible to use.

Suricata can run in different environments, including cloud-based, virtualized, and physical environments. It is configured using a YAML-based configuration file. It offers a set of different options for setting up the output format, inspected/decoded protocols, DPDK support and related options, and NIC configuration. The user can achieve high capture rates by configuring the previously mentioned options.

Suricata can be configured in two main modes [23]:

- `IDS`. Used for monitoring resources, analyzing traffic, or testing. In this mode, Suricata does not block traffic. Depending on the network connection Suricata can run in either Host-based IDS (HIDS) or Network IDS (NIDS) modes. The former mode is usually used for monitoring critical resources on a machine or testing. The latter mode is mainly used for analyzing traffic and raising alerts.

- `IPS`. This configuration provides the ability not only to monitor the network but also to block certain types of traffic.

Despite the fact that Suricata is widely used for network security monitoring, intrusion detection, and prevention, like any other system, Suricata has its advantages and disadvantages. The main ones are mentioned below [16]:

[1] https://suricata.io/

Advantages:

- Open-source software.
- Real-time traffic analysis and fast threat detection.
- Can be integrated with other systems.
- Regular updates.

Disadvantages:

- Resource extensive.
- Steeper learning curve.

2.1 Architecture

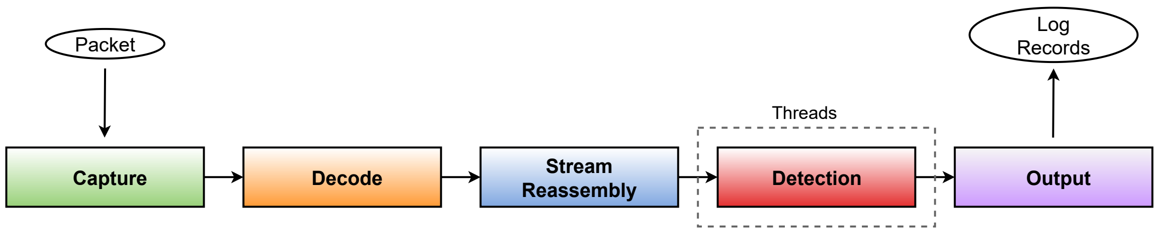


Figure 2.1: Suricata architecture.

- `Capture module` – receives packets from the NIC using packet capture libraries such as `AF_Packet`, `DPDK`, or others. Alternatively, it reads packets from Packet Capture (PCAP) files using the open-source cross-platform library `libpcap`[2]. The packets are captured in real-time and are passed to the Suricata next module for decoding.
- `Decode module` – is responsible for decoding incoming packets and extracting important data from the packets. This process involves multiple steps, including validating packet headers to confirm that they are not malformed and decoding the payload to extract additional information, such as file hashes and strings. Extracted data from the packet are put into Suricata's internal representation. This module is described in detail in Section 2.4.
- `Reassembly module` – is used to reassemble fragmented packets and reconstruct the original packet payloads. The packets are tracked in flows to keep information about the packets belonging to the same TCP stream. This module allows Suricata to detect stealthy attacks that bypass detection by using packet fragmentation.
- `Detection module` – identifies potential threats in the incoming packets against a set of rules. Suricata rules are described in Section 2.5. The module has the ability to parallelize detection processing for one packet simultaneously in IPS configuration mode. Additionally, the user can write custom Lua scripts for customizing the behavior of Suricata.
- `Output module` – is responsible for formatting and outputting the results of the detection module. The module reads events and alerts as input. As output, it prints statistics in different output formats such as JavaScript Object Notation (JSON), Eve-JSON, and Comma-Separated Values (CSV).

[2] https://man7.org/linux/man-pages/man3/libcap.3.html

2.2 Runmodes

To examine the available runmodes in Suricata, it is necessary to have a comprehensive understanding of its core building blocks. Suricata is a multi-threaded system consisting of three "building blocks": `thread`, `thread-module` and `queue`.

A thread is a lightweight process that can be scheduled and run concurrently with other threads by the operating system [14]. In Suricata, each thread contains thread-modules, which represent different functional parts of Suricata, and are linked to other threads via queues. Thread-module and its functional parts are explained in Section 2.1. Queues are used to transfer packets between threads. Suricata can operate in one or more threads, and each thread may consist of multiple thread-modules, with only one module running at a time. Although each thread-module can process only one packet at a time, Suricata typically consists of multiple threads assigned to a single processor. This allows the Suricata engine to process multiple packets simultaneously [23]. The configuration of these "building blocks" is known as the `runmode`. A thorough understanding of the runmodes, building blocks, and their interconnections is crucial for configuring Suricata to perform optimally and efficiently in different systems.

Suricata provides several runmodes such as:

- `Workers`. For the maximum performance it is recommended to use this runmode, because the NIC ensures the balanced distribution of packets between Suricata's processing threads. The scheme of this runmode is shown in Figure 2.2.

- `Autofp`. In order to process PCAP files, or in the case of certain IPS setups, autofp runmode is used. It contains one or more capture threads, that capture the packets and perform the packet decoding, after which it is passed on to the flow worker threads.

- `Single`. It contains a single packet processing thread. This runmode is mainly used for development or debugging. However, the number of management threads is unlimited.

2.3 Capture modes

In this section of the thesis, the focus will be on the two widely-used capture modes implemented in Suricata, namely `AF_Packet` and `DPDK`. It is aimed at providing a more comprehensive understanding of their roles in Suricata's performance, as well as their respective strengths and weaknesses. We will delve deeper into the technical details and key differences between these capture modes, which contribute to the overall efficiency and accuracy of Suricata's traffic analysis capabilities.

2.3.1 AF_Packet

AF_Packet[3] is a Linux socket used for both receiving and sending raw network packets directly from the network interface [23]. It is commonly used in Linux network programming to create high-performance applications because it allows bypassing the Linux kernel's networking stack and accessing the network interface directly.

[3] `https://man7.org/linux/man-pages/man7/packet.7.html`

Once the packet is gathered by the network interface, it is sent to the kernel. Then it is cloned into user space, but the original packet continues to be processed in the kernel. Application in the user space must ensure the handling of the received raw packet.

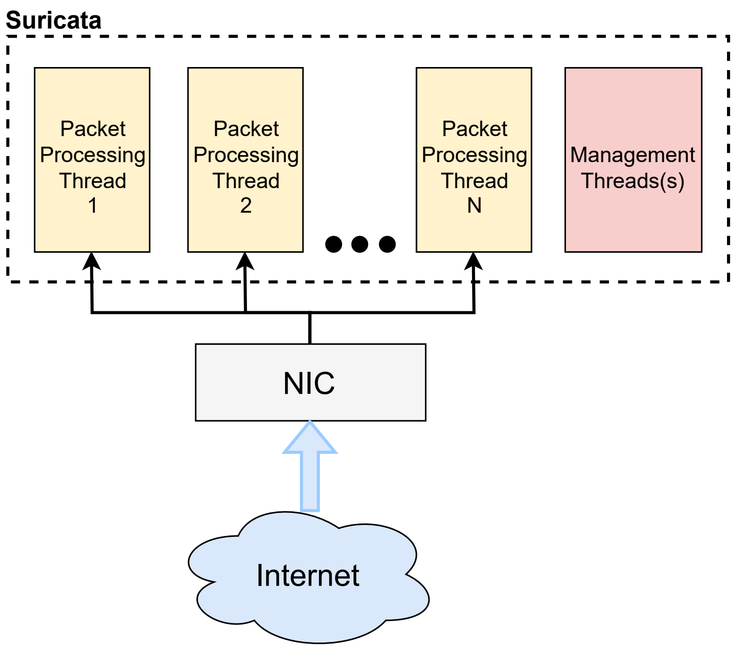

Suricata
Packet Processing Thread 1
Packet Processing Thread 2
Packet Processing Thread N
Management Threads(s)
NIC
Internet

Figure 2.2: Runmode `workers`.

Advantages:

- Can handle multiple packets in a single system call. It helps to improve performance and reduce the number of context switches.
- Can be bound to specific network interfaces. It gives applications more control over network performance.
- Provides versatility. Provides a level of abstraction from the kernel side, because it is hardware independent.

Disadvantages:

- AF_Packet has limited functionality (limited filtering capabilities, the application must have administrative privileges).
- Performance loss due to packet cloning.

2.3.2 eBPF and XDP in AF_Packet

The extended Berkeley Packet Filter (eBPF) and Express Data Path (XDP) are two powerful networking technologies widely used in the Linux kernel. They provide a flexible and efficient way to perform custom packets processing tasks, such as filtering, routing, and forwarding, with low latency and high throughput.

eBPF is a kernel technology that allows programs to run without the need to change the kernel source code or add additional modules. In other words, it is a lightweight, sandbox virtual machine (VM) inside the Linux kernel [7]. It was originally developed as an extension to the classic Berkeley Packet Filter (BPF) technology, which was used for packet filtering and monitoring in the networking stack. eBPF improves upon BPF by providing a more powerful and flexible programming model supporting advanced data structures, control flow constructs and system calls.

XDP is a networking technology that enables high-performance packet processing in the Linux kernel. It offers to execute the program on the packet that is just received from the receive rings and allows the eBPF program to define the fate of the packet at the beginning of the packet processing [9]. XDP can work in three modes within the packet processing pipeline: `hardware`, `driver` and `generic`.

It is worth noting that the best outcome in terms of XDP program performance can be achieved when running the program in `hardware`. In this case, CPU resources saving is possible because XDP is loaded to the NIC (currently supported by Netronome only [17]). XDP running in `generic` mode is more general-purpose and can be used when card selection is limited. However, Running XDP in `generic` mode could lead to less performance compared to operating in `hardware`. The choice of whether to run XDP in `hardware` or `software` mode depends on factors such as card selection and performance requirements.

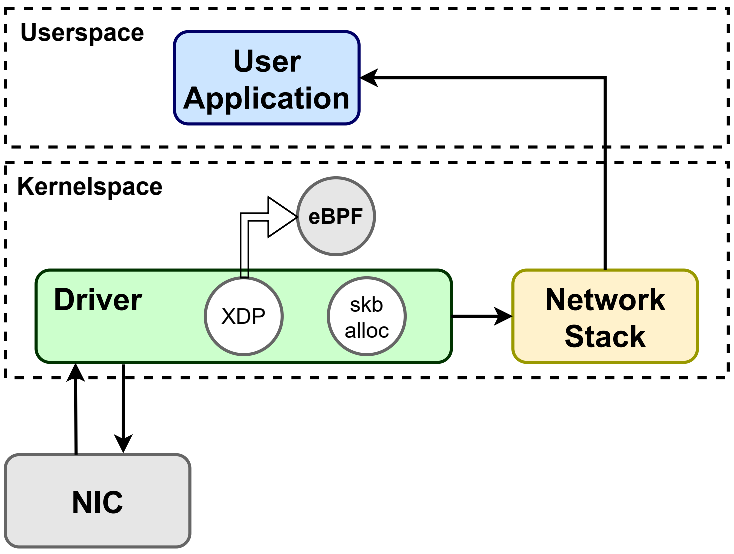


Figure 2.3: eBPF and XDP in driver mode [2].

Figure 2.3 shows eBPF and XDP in `driver` mode. That means that the XDP program runs within the driver itself and processes packets right after they are received off the NIC and before they are passed to the kernel.

Possible XDP actions [6]:

- `XDP_PASS` – pass packet further to the network stack.
- `XDP_DROP` – the simplest and the fastest action, packet is dropped.
- `XDP_TX` – send the packet back to the same NIC it arrived on.
- `XDP_ABORTED` – program error, packet is dropped.

eBPF in Suricata

Processing all captured packets can be a bottleneck for Suricata's performance. To solve this Suricata has an implemented eBPF mechanism in the `AF_Packet` interface which is described in Section 2.3.1. eBPF is used in Suricata for the next three purposes [20]:

- Supports any BPF filters.
- Provides load balancing.
- Loads XDP programs.

Using eBPF to filter packets can significantly improve Suricata's performance by delegating some of the filtering tasks to the kernel. This approach reduces the load on the user application and makes the use of system resources more efficient.

To address this issue, Suricata can leverage eBPF to filter out unwanted packets before they reach the user application. This is achieved through the use of eBPF hash tables, also known as `eBPF maps`. It is a data structure shared between user space and kernel space, which allows the passing of information between spaces [20]. eBPF maps are similar to the flow tables, where each entry contains details such as source and destination IP addresses and ports, and used the protocol. Suricata may check the map when a packet arrives to see whether it belongs to the flow that should be ignored. In addition to filtering, the flow table can also be modified dynamically based on the rules configured in Suricata.

2.3.3 DPDK

DPDK (Data Plane Development Kit)[4] is an open-source framework that enables the development of high-speed data plane applications. Since 2017, this project has been managed by Linux Foundation. DPDK provides a programming framework for x86, ARM, and PowerPC processors.

Advantages:

- Running NIC drivers in userspace instead.
- Using hugepages.
- Using Poll Mode Drivers (PMDs).

Disadvantages:

- Because DPDK bypasses the kernel, it may introduce security risks.
- DPDK is not compatible with all network interface cards.

Another significant advantage is that DPDK can be used in conjunction with NICs from different manufacturers. Major vendors, such as Intel or Mellanox, integrate native support of DPDK into the device driver of their cards. NICs of various vendors might require the installation of a custom device driver to work correctly with DPDK because the framework needs access to advanced parts of the NIC [24].

DPDK is developed based on the kernel bypassing approach, which consists of offloading packet processing from the OS kernel to userspace applications [4]. It helps to avoid time-consuming context switching between the application and the kernel and increases performance.

[4] https://www.dpdk.org/

Environment Abstraction Layer

The Environment Abstraction Layer (EAL) is responsible for gaining access to low-level resources and abstracting them to reduce the complexity of developing applications. It provides a generic interface that hides the environment specifics from the applications and libraries.

From the services listed in [12], the typical ones are:

- `DPDK Loading and Launching`. The DPDK and its application are linked as a single application.
- `Core Affinity/Assignment Procedures`. Assigns execution units to specific cores.
- `CPU Feature Identification`. Determines at runtime if the current CPU supports the features.
- `System Memory Reservation`. Reserves different memory zones.

DPDK applications pass two parts of parameters. In the first part, they pass the parameters of the EAL library and the second part belongs to the application parameters. It looks like this:

```
./name_dpdk_app <EAL_params> -- <APP_params>
```

A list of EAL parameters is sourced here [11].

Hugepages

In computer systems, memory management plays a significant role in the efficient use of resources. When a process uses memory, the kernel divides Random Access Memory (RAM) into chunks of 4K bytes called pages, which can be swapped to disk. The management of virtual memory space by the CPU and the operating system becomes increasingly time-consuming as the number of pages grows. To optimize this process, current CPU architectures support larger pages, known as Huge pages on Linux, Super Pages on BSD, and Large Pages on Windows [25]. The main benefit of hugepages is its ability to manage huge-sized pages in memory in addition to the standard 4KB page size. By enabling Huge pages, the system has fewer page tables to deal with, and thus, there is less overhead to access/maintain them. This leads to enhanced kernel-level performance, which eventually boosts the application by reducing the number of mapping tables. Therefore, the use of hugepages can improve system performance for applications with high memory requirements [18].

Poll Mode Drivers

The standard packet path is depicted in Figure 2.4. Initially, NIC drivers use interrupt mechanisms to process incoming packets. When the NIC receives packets, an interrupt is triggered, and the kernel is prompted to process them. This method works well when packets arrive slowly because the delay of context switching can be ignored. However, when the packet arrival rate is considerable, like in high-speed networks, the processing performance is significantly reduced [4].

DPDK solved this problem by using the polling-based approach. It provides drivers called Poll Mode Drivers (PMDs). The main goal of these drivers is to poll network devices for packets. It requires also one of the kernel drivers that can be used as PMD, e.g. `vfio_pci`, `igb_uio`, and `uio_pci_generic`, to initialize the hardware device. Therefore, a NIC must be explicitly bound to one of the aforementioned drivers in order to be managed by DPDK [4].

It is recommended use PMDs only for high-load networks to reduce wasted CPU cycles.

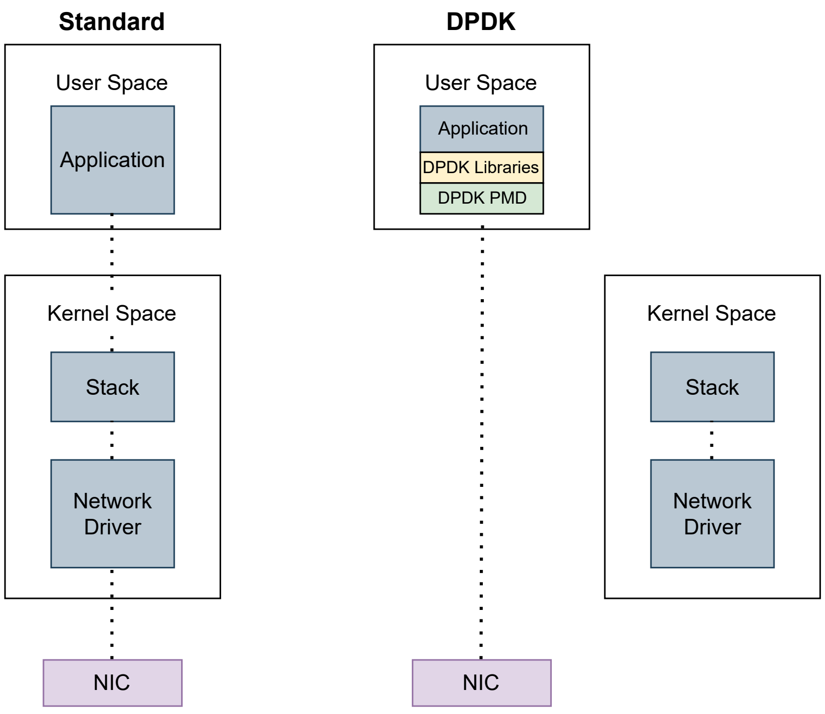


Figure 2.4: A comparison of the traditional way of processing packets in the operating system vs DPDK [8].

2.4 Decode module

The network is a complex system that consists of numerous protocols, each operating on a specific layer. The Open Systems Interconnection (OSI) reference model is a conceptual framework that describes the functions of the networking or telecommunication system independently from the underlying technology infrastructure [22]. This model defines seven layers that are described in Figure 2.5, each of these layers performs different functions in the network. To the right of each layer mentioned some protocols that belong to the corresponding layer.

Dividing the network into layers simplifies packet parsing for network applications. Packet parsing is a fundamental process of network applications that involves analyzing the headers of network packets and extracting relevant information to understand the contents of the packet.

As shown in Figure 2.6, a network packet is a basic unit of data that is grouped together and transferred over a computer network. The size and structure of a network packet depend on the underlying network structure [26]. Understanding these concepts is essential for developing network applications because it provides an awareness of how information is transferred over the network. To figure out more about the frame anatomy, size, and contents of different protocol headers, the following article is suitable for understanding the corresponding things [15].

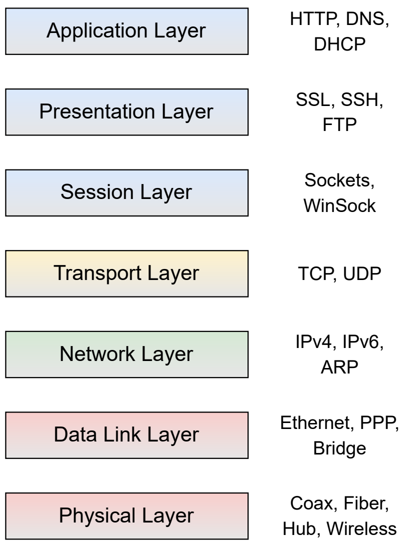


Figure 2.5: OSI model [22].

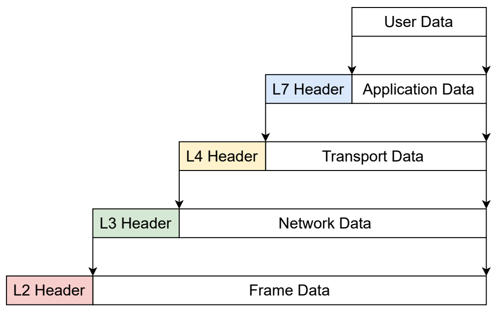


Figure 2.6: Structure of the network packet.

Packet parsing is an important stage that allows getting all necessary information from the packet for its next processing. Like other network parsers, Suricata starts from the second Data Link Layer.

- `Layer 2`. In addition to the Ethernet protocol, Suricata supports protocols such as Point-to-Point Protocol (PPP), Cisco High-level Data Link Control (HDLC), and others.
- `Layer 3`. On this layer, basic protocols such as Internet Protocol of version 4/6 (IPv4/6), Address Resolution Protocol (ARP), and others are supported. On this layer, Suricata also checks if the packet has set values for Virtual Local Area Network (VLAN).
- `Layer 4`. This layer does not contain a lot of the protocols, therefore Suricata parses only the main and frequently used protocols such as User Datagram Protocol (UDP), Transmission Control Protocol (TCP), Internet Control Message Protocol of version 4/6 (ICMPv4/6).
- `Layer 7`. Now on this layer, protocols such as Secure Sockets Layer/Transport Layer Security of different versions (SSL/TLS), Secure Shell (SSH), File Transfer Protocol (FTP), and others are supported. However, it is planned to add support for other protocols such as Simple Network Management Protocol (SNMP), Session Initiation Protocol (SIP), and others in the future.

Each packet goes through the decoding module, which is a significant part of Suricata. During this process, the internal structure of Suricata is filled in. It contains:

- Pointers to the beginning of headers.
- Information from headers such as ports, addresses, protocols, etc.

- Information about flow record.
- Other useful information which is used in the next modules.

Suricata defines internal structures (headers) for each supported network protocol. Up to the fourth layer, each layer contains the protocol ID of the following header. After Suricata must estimate the protocol by TCP payload. Therefore parsing of the packet is like tree traversal. In Figure 2.7 is shown the parsing of the basic TCP packet in Suricata. Dashed-dotted lines indicate possible combinations of protocols, and the green line indicates one of them in the case of `Ethernet-IPv4-TCP` packet.

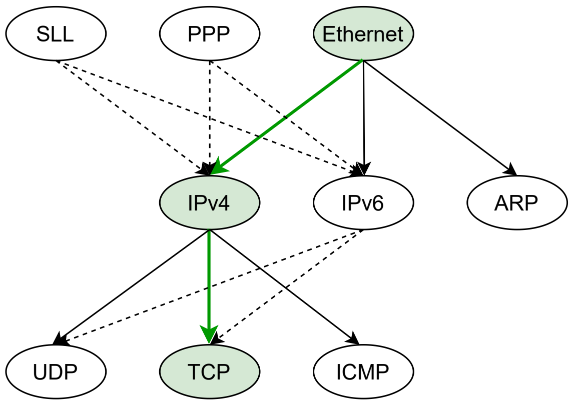


Figure 2.7: Example of packet parsing in Suricata in the form of a tree.

2.5 Detection rules

Suricata rule structure consists of 3 logical parts – ACTION, HEADER and RULE OPTIONS [21]. Each part of the rule is described in the subsections below.

2.5.1 ACTION

Specifies what will happen with a packet, if the rule matches.

- alert – generate an alert.
- pass – pass current packet without further inspection.
- drop – drop the packet and generate an alert (works in IPS mode only).
- reject[src|dst|both] – send RST/ICMP unreach error to the sender/receiver/both sides of the communication.

2.5.2 HEADER

Consists of four parts:

- protocol – specifies the protocol to match. Can be chosen one from the basic protocols: TCP, UDP, ICMP or IP (stands for 'all') or from application layer protocols: HTTP, FTP, DNS, or others. The whole list of application layer protocols can be found in the Suricata manual [21]. The user should enable protocols specified in the rules in `suricata.yaml`.

- source and destination IP addresses – specifies the source of the traffic and the destination of the traffic, respectively.

Example	Meaning
!10.10.10.1	Every IP address but 10.10.10.1
![10.10.10.1, 11.11.11.2]	Every IP address but 10.10.10.1 and 11.11.11.2
$HOME_NET	Setting of HOME_NET in `suricata.yaml`
any	Arbitrary IP address
10.0.0.0/24	Range from 10.0.0.0 till 10.0.0.255

Table 2.1: Example of defining network hosts.

- source and destination ports – specifies the source and destination ports which traffic goes through.

Example	Meaning
[121, 122]	Port 121, 122
[80: 443]	Range from 80 till 443
[1000:]	From 1000 till the highest port-number
any	Arbitrary ports
[5: 88, ![80, 81]]	Range from 5-88, except ports 80 and 81

Table 2.2: Example of defining network ports.

- direction – specifies in which way the signature has to match.

Example	Meaning
->	Only packets with the same direction can match
<>	Match rule both ways

Table 2.3: Example of defining directions.

2.5.3 RULE OPTIONS

Specifies additional details of the rule. These are enclosed by parenthesis and separated by semicolons. There are many different options, for example:

- msg – gives textual information about the signature.
- sid – defines signature's ID (must be unique).
- rev – picks the latest rule in case the same `sid` used by multiple rules.
- priority – defines what signature will be examined first.
- content – defines what the packet must contain to be matched.
- nocase – content search is case insensitive.
- ttl – checks for a specific IP time-to-live value in the header of a packet.

The rest of this list can be found here[5].

Note that the next signatures are examples and are provided for informational purposes to show the syntax of signatures.

1. Signature to detect Hypertext Transfer Protocol (HTTP) requests to the domain:

   ```
   alert http any any -> any any (msg:"HTTP request to
   malicious domain"; content:"GET"; nocase; http_host;
   content:"example.com"; http_uri; sid:1; rev:1;)
   ```

2. File access attempt:

   ```
   drop tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"File
   access attempt"; content:".bash_profile"; nocase;
   classtype:attempted-recon; sid:10222; rev:1;)
   ```

3. Server Message Block (SMB) protocol:

   ```
   alert tcp $EXTERNAL_NET any -> $HOME_NET 445 (msg:"SMB
   protocol"; content:"|aa ff 12 34 5f 1a 5e|"; depth:7;
   sid:12345; rev:2;)
   ```

[5] https://suricata.readthedocs.io/en/suricata-6.0.9/rules/index.html

Chapter 3

DPDK Prefilter

Network attacks are constantly evolving, enabling them to overload networks with a massive volume of packets. To prevent these large-scale attacks, firewalls or other network applications should process incoming traffic in the fastest way. One of these prevention systems is Suricata, which was described in detail in Chapter 2.

To handle more traffic, the network capture program's processing speed must be increased, or the amount of traffic passed to the application must be controlled. The latter method motivated the creation of DPDK Prefilter.

Figure 3.1 below shows the way how Suricata communicates with Prefilter. `RX` and `TX` are the receive and transmit channels respectively, that refer to the directions of data transfer. Only when the IPS mode is enabled the transmit channel is activated. `PKT` is a bulk of network packets. Also, the bypass function is implemented between two applications.

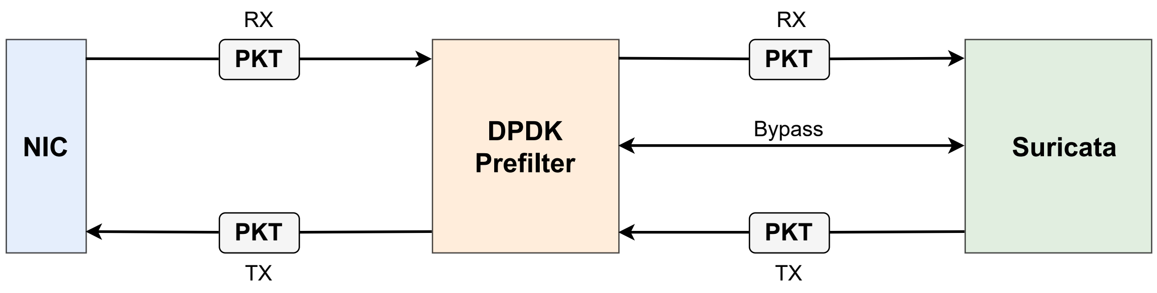


Figure 3.1: Connection of the DPDK Prefilter with Suricata.

Among the benefits, DPDK Prefilter has:

- Filter the incoming traffic.
- Provide asynchronous flow-based bypass before any of the packets of the flow reaches Suricata.
- Create an opportunity for quick prototyping of hardware functions in software.
- Interconnect systems to create more versatile architecture.
- Add metadata to packets passing through.

3.1 Operating stages

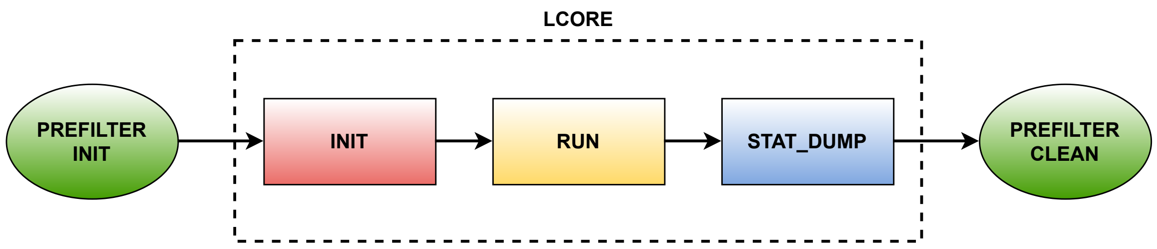


Figure 3.2: DPDK Prefilter operating stages.

- `PREFILTER_INIT`. It is the first part of the DPDK Prefilter. Here is a configuration of all necessary components. It includes:
  - The initialization of the device for packet capture and rings for packet forwarding to the next device.
  - Creation of more threads for capturing and decoding parts to accelerate packet processing.
  - The initialization of the Inter-process communication (IPC) between the DPDK Prefilter and another system.
  - Extraction of the necessary information about rings for each entry from the YAML-based configuration file. The configuration file contains information about the memory pool (pre-allocated chunk of kernel memory), NIC configuration (including Peripheral Component Interconnect (PCI) addresses of ports, promiscuous mode, support Receive Side Scaling (RSS), Maximum Transmission Unit (MTU)), and others details.
- `LCORE_INIT`. This stage is responsible for the configuration variables of the core and rings for receiving/sending packets. Also, there is necessary memory allocated for packet buffers and flow tables.
- `LCORE_RUN`. DPDK Prefilter gets bursts of packets and sends them to the module that tries to decode the packet and create a flow record. After decoding and possibly bypassing DPDK Prefilter enqueues packets to bursts and sends them to another system.
- `LCORE_STAT_DUMP`. Print to the standard output detailed statistics, including:
  - How many packets were captured from the Network Interface Card (NIC) and from another system (in case DPDK Prefilter operates in IPS mode).
  - How many packets were inspected for bypassing, bypassed, enqueued to another system, and other useful information.
- `PREFILTER_CLEAN`. The stage is executed on shutdown. DPDK Prefilter frees up memory that was allocated in previous stages.

3.2 Decode module

At the beginning, DPDK Prefilter tries to decode each packet and create a `FlowKey` record. This is used to determine if the packet is part of the traffic flow. Traffic flow is the sequence of packets from a source computer to a destination. Packets from the same traffic flow have a common five-tuple that consists of protocol, source and destination addresses, and source and destination ports.

DPDK Prefilter supports decoding of the four protocols only: IPv4, IPv6, TCP, and UDP. The process of the packet decoding is graphically depicted in Figure 3.3. DPDK Prefilter decodes each network layer, starting from the second `Data Link`, and fills up the necessary variables. If DPDK Prefilter is unable to determine which protocol has the next layer, it places the packet in the queue for the secondary application. If decoding is successful, it marks this packet as `"inspected"` and sends it to a function that determines if the packet can be bypassed or not. If the packet can not be bypassed, it is placed in the queue for forwarding to another system.

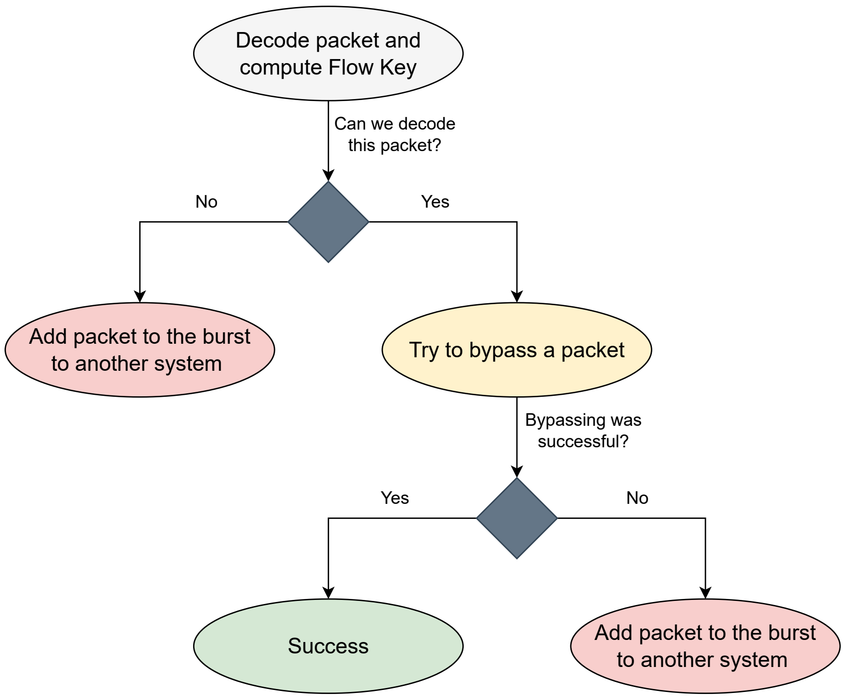

Decode packet and compute Flow Key
Can we decode this packet?
No
Yes
Add packet to the burst to another system
Try to bypass a packet
Bypassing was successful?
Yes
No
Success
Add packet to the burst to another system

Figure 3.3: Process of packet decoding in DPDK Prefilter.

3.2.1 Flow bypass

The Suricata system uses the eBPF program to efficiently bypass packets before they reach user space. As discussed in Subsection 2.3.2, this is achieved by using eBPF maps, which allow Suricata to communicate with other eBPF programs and define flows that should be dropped. However, a significant issue arises when a table lookup occurs, as the entire table becomes locked, and record insertion may be blocked. This can lead to delays and impact the system's performance. This standard way of communication is shown in Figure 3.4.

To eliminate the problem of a locked table, DPDK Prefilter replaces the XDP/eBPF program and provides another way of sharing information about flows. Suricata sends asynchronous messages to DPDK Prefilter about records that should be inserted into the table. DPDK Prefilter, in turn, utilizes an internal table to bypass packets. This new approach of communication between two systems is depicted in Figure 3.5. Packets are searched in the table after decoding and bypassed if necessary.

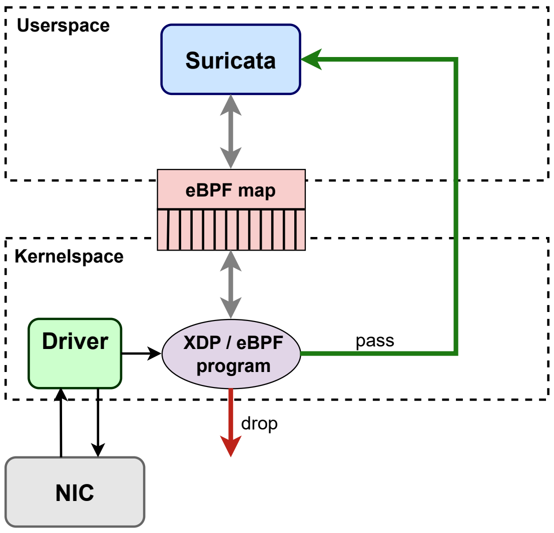


Figure 3.4: Communication between Suricata and another eBPF program.

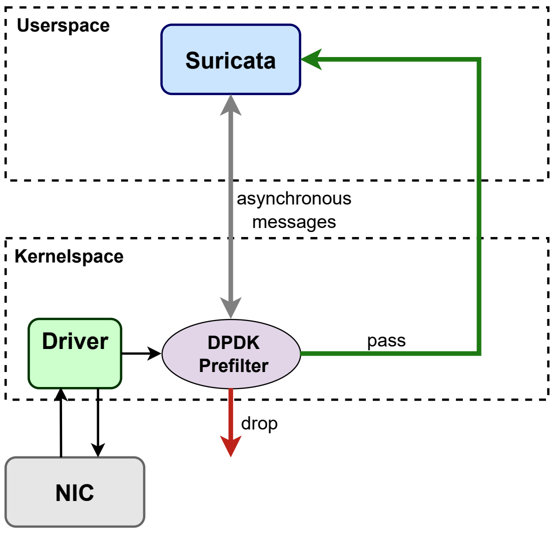


Figure 3.5: Communication between Suricata and DPDK Prefilter.

3.3 Communication with Suricata

To achieve maximum performance, it is crucial for two distinct systems to establish a communication channel between them in the most efficient way. DPDK Prefilter is developed using the DPDK framework, which offers the below-described techniques for effective system-to-system communication.

3.3.1 Packet rings

DPDK Prefilter uses DPDK rings to transfer packets to and from Suricata. DPDK rings are defined as a structure `rte_ring`. The implementation was forked from the FreeBSD ring buffer[1].

DPDK has a ring library, which provides a lockless, multi-producer/multi-consumer ring buffer implementation. This library is used to transmit and receive packets between the NIC and the application running in the user space without the need for locks or other synchronization mechanisms.

Advantages:

- Data is written to the buffer extremely quick because it requires a single 32-bit Compare-And-Swap instruction only.
- Versatile and flexible tool in network programming.
- Lesser number of cache misses when adding or removing a large number of objects from the queue. This is because pointers are stored in a table.

Disadvantages:

- The size of a DPDK ring is fixed at initialization and cannot be increased dynamically.
- Having many rings costs more in terms of memory than a linked list queue. The buffer always uses the maximum number of pointers, therefore an empty ring contains at least N objects [13].

The ring buffer consists of two pointers: a producer pointer and a consumer pointer. The NIC uses the producer pointer to enqueue packets into the ring buffer, while the application uses the consumer pointer to dequeue packets from the buffer. These pointers navigate through the ring buffer in a circular manner. When they reach the end, they get back to the beginning of the buffer [13].

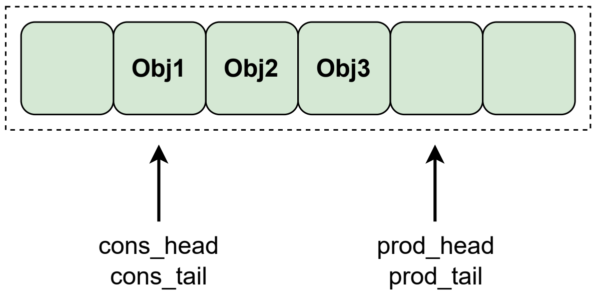


Figure 3.6: Ring structure.

DPDK provides several different types of rings such as [13]:

- `Multi-producer/Multi-consumer (MP/MC)`. The default mode of the ring. In this mode, multiple threads can enqueue/dequeue objects to/from the ring. It is usually the most suitable and fastest synchronization mode with one thread per core.

[1]https://svnweb.freebsd.org/base/release/8.0.0/sys/sys/buf_ring.h?revision=199625&view=markup

- `Single-producer/Single-consumer (SP/SC)`. In this mode, only one thread at a time is allowed to enqueue/dequeue objects to/from the ring.
- `MP/MC with Relaxed Tail Sync (RTS) mode`. The tail value is increased not by every thread that finished enqueue/dequeue, but only by the last one. That allows threads to avoid spinning on **ring tail** value and improve average enqueue/dequeue times on overcommitted systems.
- `MP/MC with Head/Tail Sync (HTS) mode`. In that mode enqueue/dequeue operation is fully serialized: at any given moment only one enqueue/dequeue operation can proceed.

3.3.2 Inter-process communication

Inter-process communication (IPC) is a flexible and powerful mechanism provided by the operating system that allows multiple processes to communicate with each other. This communication can be used for exchanging data, coordinating activities, or synchronizing execution.

Operating systems, network communication, and distributed computing are just a few of the many applications that make use of IPC. In modern operating systems, IPC enables processes to communicate with the kernel and other processes. In network communication, IPC is used for exchanging data between network devices. In distributed computing, IPC is used to coordinate the activities of processes running on different systems to work together toward a common goal.

Inter-process communication occurs between DPDK Prefilter and Suricata in different ways:

- `DPDK rings`. They offer a unidirectional data channel that enables data exchange between two or more processes. They are used for messages or packet transferring.
- `Shared memory`. The two processes share a common space or memory location known as a buffer. There the configuration is placed, which is available for two systems without the overhead of copying data.
- `Message passing`. In this method, processes communicate with each other without the aid of shared memory. Once a communication link has been established, they can start exchanging messages. It is well-suited for systems that require fault tolerance and scalability. It requires also careful design to ensure correct synchronization and message delivery.

DPDK Prefilter uses both asynchronous and synchronous requests to communicate with the secondary application. These requests are provided by the DPDK framework. Both types of requests trigger named callback on the receiver side, which is called from within a dedicated IPC or interrupt thread that is separate from EAL lcore threads [10].

Asynchronous requests are helpful when a program does not require a response to keep working. This means that the program can continue to perform other tasks while waiting for the response. In contrast, synchronous requests are used when the program needs a response from another system to keep functioning. Synchronous requests block the caller thread until the response is received. The differences between synchronous and asynchronous requests are illustrated in Figure 3.7.

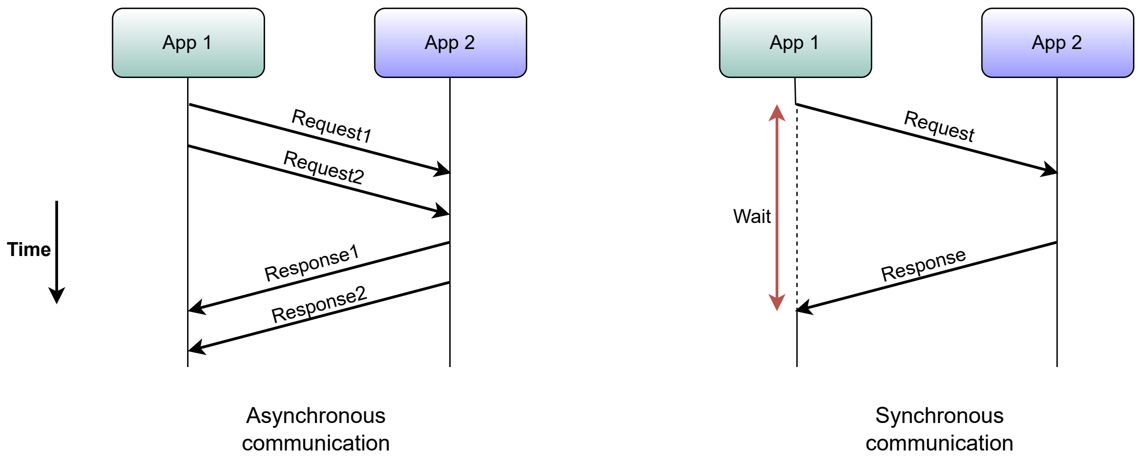


Figure 3.7: Comparison of asynchronous and synchronous requests.

Chapter 4

Per-packet metadata

This chapter is focused on a comprehensive overview of metadata usage in Suricata. The first section of the chapter describes the motivation behind this thesis and its primary objectives, setting the stage for further topics. The following sections describe metadata implementation. That involves design of metadata structures, method for metadata storage, negotiation process between two applications, and configuration of metadata. These sections offer a detailed account of the various aspects required for the successful integration of metadata into the Suricata system. The final section of this chapter presents the proposal for the metadata structure. This section provides a brief overview of the fundamental structures and functions needed for the effective implementation of metadata in the system.

4.1 Motivation

There are several approaches to how applications can be optimized. It can be the elimination of bottlenecks thanks to code profiling tools such as `perf` and `Intel VTune`, using appropriate data structures and algorithms, managing memory well, or parallelizing processes. Applications can be further improved by e.g. hardware acceleration. `SmartNICs` can aid the software application by filtering out unwanted traffic. They can also add auxiliary data to individual packets to reduce the processing path within the application.

However, creating the hardware is an expensive and time-consuming process. DPDK Prefilter was developed to simulate the potential performance gains of hardware acceleration. The goal of this thesis is to extend this concept by designing and implementing metadata that can be passed between DPDK Prefilter and Suricata.

Metadata support can cover a broad set of problems. To test and evaluate metadata support, the use case of extracted information from network headers was selected. The main reason for this choice is the Suricata rules described in Section 2.5. Each rule contains matching values such as protocol, port, IP address, and other details. This information can be found during packet processing, and therefore, the decoding module plays a crucial role in correctly matching rules with each packet.

Since metadata is unique for each packet, it makes sense to include them with each packet. One possible method of transmitting metadata with packets is shown in Figure 4.1. As shown in this Figure, metadata is generated by DPDK Prefilter described in Chapter 3 and is sent to Suricata and vice versa. The only difference from the standard architecture, shown in Figure 3.1, is sending packets supplemented by metadata instead of raw packets. How metadata is stored will be discussed in the next sections.

The use of metadata can provide additional insights into packet processing, which can be beneficial in optimizing network performance. The proposed investigation of metadata transmission to Suricata can help to expand this topic and transmit more valuable data. This can help to improve and make the processing and filtering of packets more efficient.

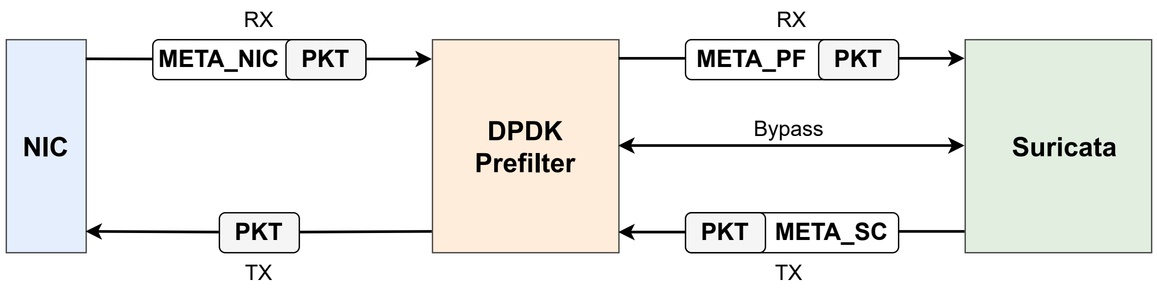


Figure 4.1: Adding metadata to packets.

4.2 Design

The aim of this thesis is to explore the usefulness of metadata generated by DPDK Prefilter. For these purposes, a framework `DPDK` is used along with its internal structures for processing network traffic. For instance, the `rte_mbuf`[1] data structure is used to represent the network packet. It contains a private data space that can be allocated during the creation of the memory pool using the `rte_mempool_create()`[2] function. The size of the private data space will be statically defined in the configuration file of DPDK Prefilter. This private space is a great place for metadata since the application defines this space while creating the mempool, and no external interference occurs as it is statically allocated.

As mentioned earlier, four fundamental network protocols have been selected as metadata values for Suricata. These protocols include `IPv4`, `IPv6`, `TCP`, and `UDP`. The offloads associated with each of these protocols are illustrated in Figure 4.2.

- **Network Layer protocols**. The primary offloads for protocols `IPv4` and `IPv6` consist of IP addresses. Additionally, the IPv4 protocol includes IP options like Loose Source Route (LSR), Record Route, Strict Source Route (SSR), and others.
- **Transport Layer protocols**. The primary offloads for protocols `IPv4` and `IPv6` consist of port numbers, byte length for rapid access to the payload, and payload length. Moreover, the TCP protocol includes TCP options like Maximum Segment Size (MSS), Window Scaling, Selective Acknowledgment (SACK), and others.

In the absence of metadata, Suricata relied on packet parsing to extract required values from network headers. These values were used by the detection module for matching rules. However, with the inclusion of metadata, Suricata no longer needs to parse packets as the necessary values are transmitted by DPDK Prefilter as metadata. This means that packet

[1] https://doc.dpdk.org/api/structrte__mbuf.html
[2] https://doc.dpdk.org/api/rte__mempool_8h.html#a503f2f889043a48ca9995878846db2fd

decoding is now performed only once in both systems, rather than being repeated for each individual application. This approach allows the elimination of redundant parsing, which can reduce the time it takes to process packets.

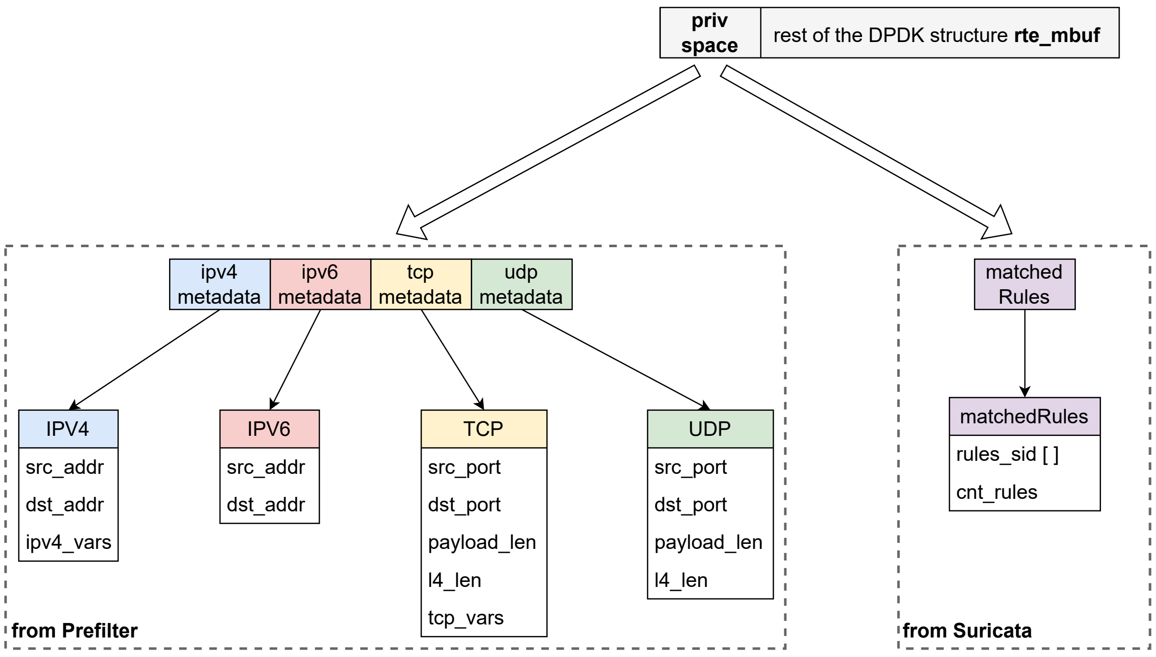
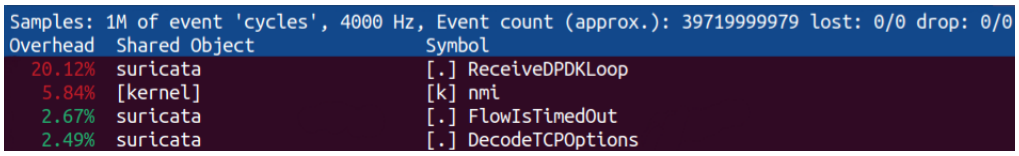


Figure 4.2: Metadata design.

Also, a motivation to choose `tcp_vars` as metadata is shown in Figure 4.3. It is an output of the tool **perf top**. The function `DecodeTCPOptions()` uses **2.49%** of CPU resources. Using metadata it is possible to almost completely eliminate the load of the function to **0%**.



```
Samples: 1M of event 'cycles', 4000 Hz, Event count (approx.): 39719999979 lost: 0/0 drop: 0/0
Overhead  Shared Object               Symbol
  20.12%  suricata                    [.] ReceiveDPDKLoop
   5.84%  [kernel]                    [k] nmi
   2.67%  suricata                    [.] FlowIsTimedOut
   2.49%  suricata                    [.] DecodeTCPOptions
```

Figure 4.3: CPU load under TCP-focused traffic.

4.2.1 Metadata in IPS mode

In the IPS mode, packets are transmitted back to DPDK Prefilter. It is essential for blocking or preventing malicious traffic from entering the network. This is achieved through a process called packet dropping, where packets that match certain rules are discarded before they can reach their intended destination. Suricata can assign metadata for the secondary application, which is DPDK Prefilter in this case. It is depicted on the right side of Figure 4.2. This metadata contains a total count of matched rules for the current packet and the array of the corresponding signature IDs, which Suricata gets during the detection process.

Adding metadata to the system allows Suricata to provide useful information to other applications about network packets or related data. Specifically, DPDK Prefilter can use this information to perform further traffic analysis.

4.3 Configuration

These systems, as mentioned in Chapters 2 and 3, use YAML-based configuration files for the setup of all the necessary variables for the next correct work. A section of `metadata` is added to the configuration file, which is divided into three subsections:

- What metadata DPDK Prefilter can send to Suricata. In case when metadata is extracted from network headers, the user can specify what protocols data should be sent to Suricata from.
- What metadata Suricata can send to DPDK Prefilter. Here, the user can specify what data can be sent to DPDK Prefilter back. This option is used only in IPS mode.
- What size of private space does the system need.

```
metadata:
    offloads-from-pf-to-suri:
        IPV4: <yes/no>
        IPV6: <yes/no>
        TCP: <yes/no>
        UDP: <yes/no>
    offloads-from-suri-to-pf:
        matchedRules: <yes/no>
    private_space_size: <u_int>
```

Both DPDK Prefilter and Suricata extract parameters from the specified configurations and load them into their internal structures. Each offload is represented by a single bit, and the collective arrangement of these bits forms a number that corresponds to the desired system configuration.

This binary representation enables an efficient method of configuring the system. This approach allows a high degree of flexibility and adaptability. Moreover, using binary representation simplifies the process of comparing and evaluating different configurations.

4.4 Agreement

Before DPDK Prefilter starts receiving packets and transmitting them to Suricata, both of them must reach a consensus on the used offloads. This synchronization is a necessary step when one system wants to transmit some information to another. This process is illustrated in Figure 4.4.

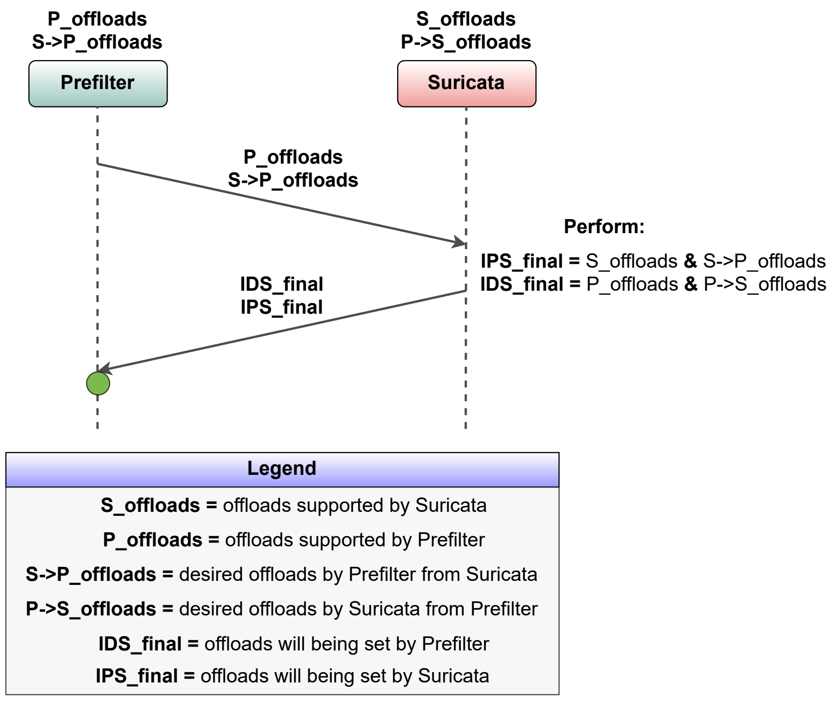

P_offloads
S->P_offloads
Prefilter
S_offloads
P->S_offloads
Suricata
P_offloads
S->P_offloads
Perform:
IPS_final = S_offloads & S->P_offloads
IDS_final = P_offloads & P->S_offloads
IDS_final
IPS_final
Legend
S_offloads = offloads supported by Suricata
P_offloads = offloads supported by Prefilter
S->P_offloads = desired offloads by Prefilter from Suricata
P->S_offloads = desired offloads by Suricata from Prefilter
IDS_final = offloads will being set by Prefilter
IPS_final = offloads will being set by Suricata

Figure 4.4: Process of agreement on metadata between Suricata and DPDK Prefilter.

In the beginning, each system only knows information from its respective configuration file. Consequently, DPDK Prefilter transmits this information to Suricata. Suricata then computes the final offloads based on the received data. As was described in Section 4.3, individual bits are responsible for specific offloads. Therefore, the most effective method for determining the same offloads for both systems is the application of the `logical and (&)`. This process is a simple and rapid two-way handshake.

To facilitate this stage, IPC communication is used, as described in Section 3.3.2, and a shared context is created. The shared context is a structure containing common variables that are used by both systems. Subsequently, each system assigns the corresponding variables to its internal structures.

4.5 Processing

In the previous sections, a comprehensive examination of what metadata looks like, its purpose and application, and the architectures of systems that utilize metadata were covered. The next step is to propose an accelerated processing method for metadata to avoid bottlenecks in the code. The most interesting stages of metadata processing will be considered in the following subsections.

4.5.1 Filling in metadata

The following structure represents the metadata from DPDK Prefilter shown in Figure 4.2.

```
struct MetadataToSuri {
    <uint32_t> metadata_set[CNT_METADATA_TO_SURI];
    <metadata_ipv4_t> metadata_ipv4;
    <metadata_ipv6_t> metadata_ipv6;
    <metadata_tcp_t> metadata_tcp;
    <metadata_udp_t> metadata_udp;
    <PacketEngineEvents> events;
}
```

The first variable in this context is an array called `metadata_set`. It is used by Suricata to read metadata. It helps to avoid accessing non-filled offloads. DPDK Prefilter fills in this array after packet decoding. The following four variables correspond to metadata for each protocol, respectively. During packet decoding, these variables are filled in with relevant information. The last variable in this structure is designated as `events`, which is used to detect any errors that occur during packet decoding. If a packet error is detected, the corresponding flag is set within this variable.

4.5.2 Traversing

This subsection focuses on a method designed to optimize both the reading and writing of metadata. To illustrate this point, let's consider a scenario where Suricata has sixteen different offloads available, but only one is enabled for transmission between systems. In order to find and set this specific offload, it would be necessary to iterate through all sixteen offloads to identify the required one. It is a time-consuming and inefficient process. In case of high-speed network traffic, unnecessary conditional statements can affect the performance and throughput of the system. Figure 4.5 below illustrates a method that minimizes the need for excessive searching and looping.

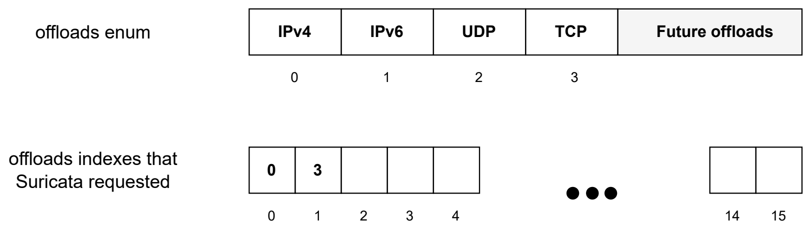


Figure 4.5: A scenario where metadata is only filled for `IPv4` and `TCP` protocols.

An array of indexes is created for both DPDK Prefilter and Suricata after the final offloads between the systems are determined. The array contains values from `offloads enum`. Using these values it is possible to have quick and efficient access to the offloads, without the need to iterate through the entire array and waste time searching for the required offload.

Chapter 5

Implementation and evaluation

Chapter 4 introduces a proposed design for metadata, building upon the knowledge gained in the previous chapter. This serves as a foundation for further development. The subsequent chapter delves into the integration tests, which are primarily aimed at verifying that Suricata's behavior remains consistent with or without metadata. This section also described the principles of the performed tests and the tool used to perform them.

The final section primarily focused on performance tests, wherein Suricata and DPDK Prefilter will be running with different configuration settings to see if the integration of metadata has not degraded the performance of Suricata. Additionally, this section provides comprehensive details regarding the testing environment, tools, and PCAP files used throughout the process.

5.1 Implementation

5.1.1 Added source files

- `src/metadata.{c,h}` – contains functions responsible for the detection of the next protocol.
- `src/metadata-l3-ipv4.{c,h}` – decodes information from the **IPv4** protocol, such as IP addresses, next protocol, and IP options.
- `src/metadata-l3-ipv6.{c,h}` – decodes information from the **IPv6** protocol, such as IP addresses.
- `src/metadata-l4-udp.{c,h}` – decodes information from the **UDP** protocol, such as ports, and payload header length.
- `src/metadata-l4-tcp.{c,h}` – decodes information from the **TCP** protocol, such as ports, payload header length, and TCP options.

5.1.2 Brief summary

This section offers a succinct summary of the process which applications write and read metadata through. Graphically, this process is depicted in Figure 5.1. When applications agree on the same metadata, DPDK Prefilter initiates packet capturing. The packet going through the program reaches the function `FlowKeyExtendedInitFromMbuf()`, which is discussed in detail in Section 3.2. If the packet remains undecoded, it is enqueued into the packet burst for Suricata without metadata. Otherwise, if the packet is successfully decoded, the function `SetMetadataToMbuf()` is called. This function is responsible for extracting information from network headers and filling in the metadata into the packet as well. Then this packet is enqueued to the packet burst also. The function `ReceiveDPDKLoop()` is responsible for receiving and translating raw packets into the internal structure. If the metadata is stored in the packet, Suricata processes it using the function `ReadAndFillMetadata()` and fills in the corresponding variables. Suricata uses a flag variable, which enables it to skip decoding functions in cases when information from network headers has already been obtained.

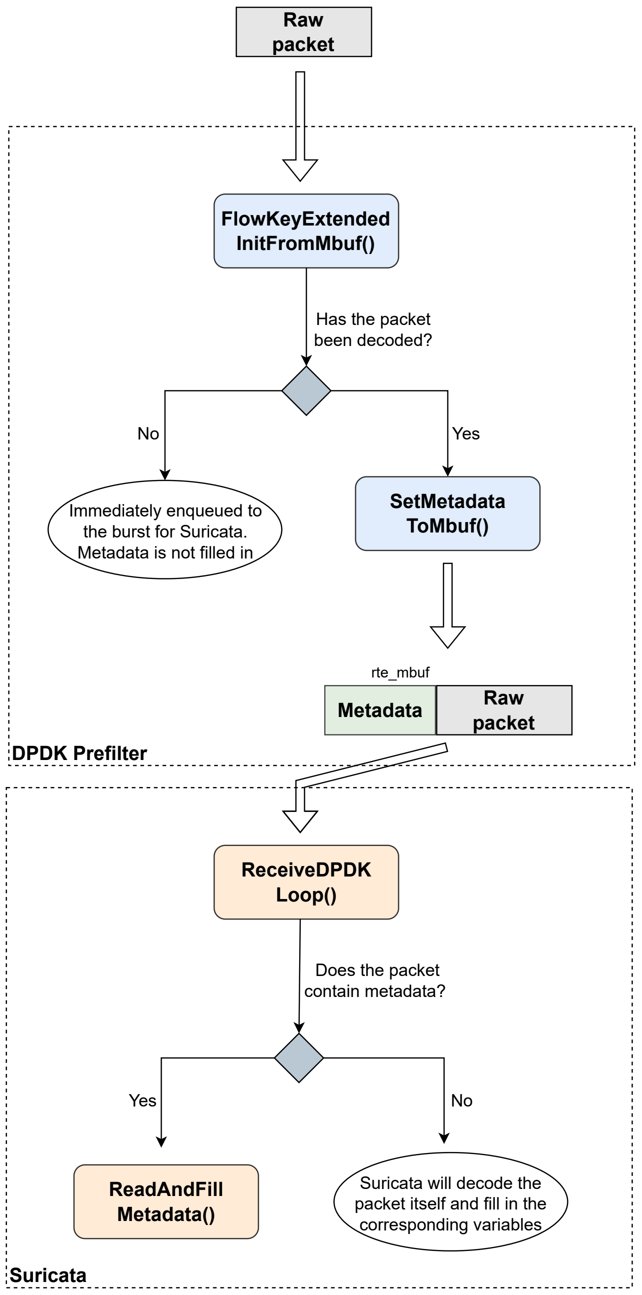

Raw packet
FlowKeyExtended InitFromMbuf()
Has the packet been decoded?
No
Yes
Immediately enqueued to the burst for Suricata. Metadata is not filled in
SetMetadata ToMbuf()
rte_mbuf
Metadata
Raw packet
DPDK Prefilter
ReceiveDPDK Loop()
Does the packet contain metadata?
Yes
No
ReadAndFill Metadata()
Suricata will decode the packet itself and fill in the corresponding variables
Suricata

Figure 5.1: Process of filling in metadata into the packet.

5.2 Integration tests

Before the changes are included in the program, it is necessary to make sure that it works properly. For these purposes, developers use integration tests. They evaluate whether different components of a software system can work together correctly. Their goal is to identify and isolate issues that may arise from the interactions between the components, and to ensure that the system works as expected. `Suricata-verify`, described in Subsection 5.2.1, is a tool to perform Suricata integration tests.

5.2.1 Suricata-verify

`Suricata-veriy`[1] is command-line tool that comes with Suricata. Its main purpose is to validate and verify the state of Suricata after configuration changes. This helps ensure that new versions of Suricata do not contain errors. If Suricata generates expected alerts, the test is considered successful; otherwise, the test fails.

Each test consists of the next parts:

- `test.yaml` – is a configuration file used by suricata-verify to define a test case. It contains information about the test, including the name of the test, arguments for Suricata, expected output, possible shell code that will be run after the test, and any additional parameters useful for the test. Expected output is defined as `label: data`, which should match with statistics in the `eve.json`.
- `test.rules` – contains rules in Suricata's format described in Section 2.5. It is used to trigger specific behaviors or events that are expected to occur when Suricata is functioning correctly.
- `input.pcap` – packet capture file in the PCAP format that contains network packets for Suricata testing.
- `suricata.yaml` – the main configuration file for Suricata. It contains various configuration options that define how Suricata operates, such as the network interface to listen on, the logging options, runmode, and more.

Extension for DPDK Prefilter

In order to integrate DPDK Prefilter into `suricata-verify`, several changes had to be made. These changes included modifying the default arguments for Suricata, adding two additional parameters to the main function **run()**, and adjusting the test running process. These changes enable validation and verification of the functionality of the metadata, ensuring that it works as estimated.

```
./run.py --testdir <path_to_prefilter_tests> --prefilter [--metadata] -j 1
```

- `--prefilter`
  - Instead of reading packets from a PCAP file, it runs Suricata in the DPDK runmode.

[1] https://github.com/OISF/suricata-verify

  - DPDK Prefilter, Suricata and tcpreplay[2] are run using **subprocess.Popen()**[3] function.

- `--metadata` – defines which configuration file DPDK Prefilter will use. There are two configuration files **prefilter_metadata.yaml** and **prefilter_no_metadata.yaml** in the folder with a test.

5.2.2 New prefilter tests

As described in the previous subsection, these tests are performed by comparing the expected results of an application with the actual results obtained after running it. To generate the expected responses, DPDK Prefilter and Suricata were executed without metadata using identical configurations and PCAP files.

Tests that check the functionality of metadata are provided below. Each test includes a brief description, the rule used for generating the alert, and the network packet created using `Scapy`[4] python library.

IPV4

- **Description**: This test is aimed at IPv4 metadata. The rule will match if the metadata contains correct information about IP addresses and options. `IPOption(bytes([0x83, 0x03, 0x04]))` sets the Loose Source and Record Route (LSRR) option into the IP header. In this case, `0x83` represents the type of option, `0x03` is the length of the route data and `0x04` is the pointer into the route data [1].

- **Rule**:

```
alert ip 211.56.70.12 20 -> 192.168.0.100 80 (msg:"Test IPv4 metadata
options"; ipopts:lsrr; content:"Contains LSRR option in the IP header";
sid:140502; rev:1;)
```

- **Packet**:

```
L3 = IP(dst="192.168.0.100",src="211.56.70.12",
     options=IPOption(bytes([0x83, 0x03, 0x04])))
L4 = TCP(sport=20, dport=80)
DATA = Raw("Contains LSRR option in the IP header")
```

IPV6

- **Description**: This test is focused on IPv6 metadata. As Suricata rules do not support the IPv6 address format, we can use the `content` keyword to ensure that the addresses are correctly transmitted using metadata.

[2]https://tcpreplay.appneta.com/
[3]https://docs.python.org/3/library/subprocess.html
[4]https://scapy.net/

- **Rule**:

```
alert ip any any -> any any (msg:"Test IPv6 metadata"; ipv6.hdr;
content:"|fe c2 00 00 00 00 00 00 00 00 00 00 00 00 10 12 fd da
cf d2 00 00 00 00 00 00 00 00 00 00 12 34|"; sid:112233; rev:1;)
options"; ipopts:lsrr; content:"Contains LSRR option in the IP
header"; sid:140502; rev:1;)
```

- **Packet**:

```
L3 = IPv6(dst="fdda:cfd2::1234", src="fec2::1012", nh=58)
L4 = ICMPv6EchoRequest()
```

UDP

- **Description**: This test is focused on UDP metadata. Since this protocol does not have options, the test is aimed at verifying the correctness of the transmitted metadata such as ports and the length of the UDP header. It can be verified by searching a substring in the data, whose location is determined by the keywords `offset` and `depth`.

- **Rule**:

```
alert udp 127.0.0.1 8080 -> 192.168.0.1 1234 (msg:"Test UDP
metadata"; content:"Tim Berners-Lee"; offset:43; depth:15;
sid:123456; rev:1;)
```

- **Packet**:

```
L3 = IP(dst="192.168.0.1", src="127.0.0.1", proto=17)
L4 = TCP(sport=8080, dport=1234)
DATA = Raw("The first website was published in 19991 by
            Tim Berners-Lee")
```

TCP

- This test is aimed at TCP metadata. The test focuses not only on verifying the correctness of the transmitted ports but also on TCP options as metadata, which play an important role in network traffic and are often set in the TCP header. The variable `tcp_options` contains the Maximum Segment Size (MSS) option with a value of `1460` [5]. This information is required for matching a Suricata rule.

- **Rule**:

```
alert tcp 211.56.70.12 3223 -> 192.168.0.100 80 (msg:"Test TCP
metadata options"; window:8192; tcp.mss:1460; tcp.hdr;
content:"|02 04|"; sid:777777; rev:1;))
```

- **Packet**:

```
tcp_options = [(TCP_OPTION_MSS, struct.pack('>H', 1460)),
               (TCP_OPTION_NOP, bytes([])),
               (TCP_OPTION_WNDSCL, bytes([])),
               (TCP_OPTION_WNDSCL, bytes([0x00]))]

L3 = IP(dst="192.168.0.100", src="211.56.70.12")
L4 = TCP(sport=3223, dport=80, flags=TCP_SYN_FLAG,
options=tcp_options)
```

5.3 Performance tests

Performance tests evaluate how well an application performs under specific conditions. The goal of performance testing is to identify and isolate any performance-related issues, and to measure the overall speed of the system.

The performance tests in this section are primarily focused on measuring throughput. It's essential to conduct tests whenever changes are made to accurately assess their impact on the program. With the addition of metadata, Suricata no longer needs to decode some protocols, as the necessary information is readily available within the metadata. However, including metadata and utilizing it has added conditional statements to Suricata's code, making it difficult to predict the effects on its performance.

5.3.1 Tool for packet replay – Trex

Trex[5], also known as Traffic Generator, is a powerful open-source tool used for generating network traffic and performance testing. It can be used to simulate various types of network traffic, such as:

- `Stateless replay` – network communication where each individual packet is handled independently because the network device does not keep any record or state information about the previous communication.

- `Stateful replay` – network communication where the network device keeps track of the state of the communication. It makes an impact on the handling of received packets. In summary, this type of network traffic is used for testing specific network scenarios in a controlled and repeatable environment.

- `Real-world traffic emulation` – network communication including traffic with different packet sizes, traffic with varying inter-packet gaps, and traffic with different protocols. Usually, it is helpful for evaluating the performance of network devices and applications under typical network usage scenarios.

Trex is a powerful tool that can generate traffic at high rates and handle millions of packets per second on a single server, making it suitable for testing high-speed networks. It is useful for testing the performance of network devices.

[5]https://trex-miner.com/

Python was chosen as the scripting language for tests because it has a vast number of third-party libraries available. Additionally, Python has a dedicated Application Programming Interface (API) for Trex, known as `trex-api`[6]. To replay packets using this tool is necessary to perform the following steps:

- Create an object of `CTRexClient` class. It defines the client side of further interaction with TRex and connects it to the listening daemon-server. It provides methods for starting and stopping traffic, configuring traffic profiles, and retrieving traffic statistics.
- Create an object of `STLClient` class. It allows using functions for stateless mode. It provides methods for configuring traffic profiles, generating traffic, and retrieving traffic statistics. Stateless mode is selected because it is the only one which packets can be replayed on the port of specified NIC in.
- Call the function `push_remote()`[7]. It pushes PCAP files to the remote device.

5.3.2 Testing environment

Server specification:

- `OS`: Oracle Linux Server 8.6
- `CPU`: 6 cores Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz
- `RAM`: 64 GB
- `NIC`:
    1. `2 port Intel Corporation 82599ES 10-Gigabit SFI/SFP+ Network Connection` – used by Trex to replay packets.
    2. `2 port Intel Corporation Ethernet Controller X710 for 10GbE SFP+` – used by DPDK Prefilter to receive packets.

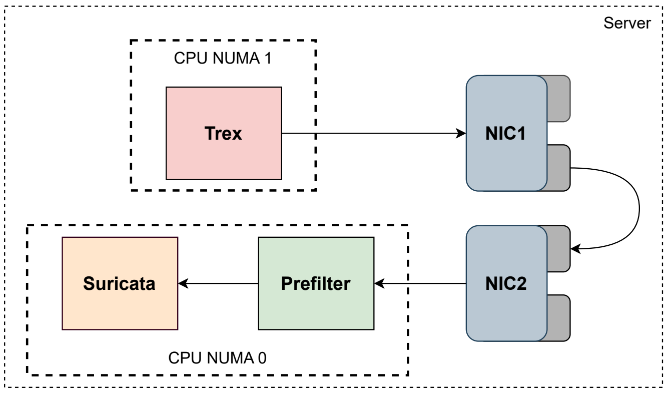


Figure 5.2: Testing scheme.

[6]https://pypi.org/project/trex-api/

[7]https://trex-tgn.cisco.com/trex/doc/cp_stl_docs/api/client_code.html?highlight=push_remote#trex.stl.trex_stl_client.STLClient.push_remote

The testing scheme is illustrated in Figure 5.2. In this setup, Trex is used to generate packets and send them to one NIC. The packets are then forwarded to another NIC, where they are read by DPDK Prefilter.

5.3.3 Tests description

Tests work in the following way. DPDK Prefilter runs with Suricata in two configurations:

1. **1** thread worker DPDK Prefilter + **1** thread worker Suricata
2. **1** thread workers DPDK Prefilter + **3** thread workers Suricata

Each configuration has four subconfigurations:

1. Suricata without rules + DPDK Prefilter with metadata.
2. Suricata without rules + DPDK Prefilter without metadata.
3. Suricata with a set of rules + DPDK Prefilter with metadata.
4. Suricata with a set of rules + DPDK Prefilter without metadata.

Dividing the configurations is necessary because Suricata's detection module requires a significant amount of CPU resources, which in turn depends on the number of detection rules. When testing Suricata without rules, there is a chance of observing a minor advantage in utilizing metadata as it aims to transfer information from network headers. However, this advantage may not hold when testing with a large set of detection rules, which can consume a significant amount of resources.

Each subconfiguration contains an array of packet transmission rates, which are used to replay packets to DPDK Prefilter. The tests are repeated three times for each transmission rate, and the average value of processed packets with that speed is calculated. This helps to ensure that the results are consistent.

5.3.4 PCAP file

It is crucial to have easily repeatable traffic for performance tests. This traffic should be diverse and simulate real-time traffic, as it contributes to more accurate test results. Ideally, real traffic gathered from a network device should be used for this purpose. For metadata tests, the following PCAP file was used. The output of the `capinfos` tool is shown below. This is a command-line utility that provides various information about PCAP files.

```
File encapsulation:  Ethernet
Packet size limit:   file hdr: 65535 bytes
Number of packets:   1,000
File size:           776 MB
Data size:           760 MB
Capture duration:    1.123194 seconds
Data byte rate:      676 MBps
Data bit rate:       5,413 Mbps
Average packet size: 760.03 bytes
Average packet rate: 890 kpackets/s
```

5.3.5 Tests results

The following subsection presents the results of a series of tests conducted to evaluate the impact of metadata on the performance of Suricata. The primary objective of these tests is to ascertain whether the integration of metadata degrades the performance of Suricata, as well as to identify any potential performance improvements that may be realized through its use.

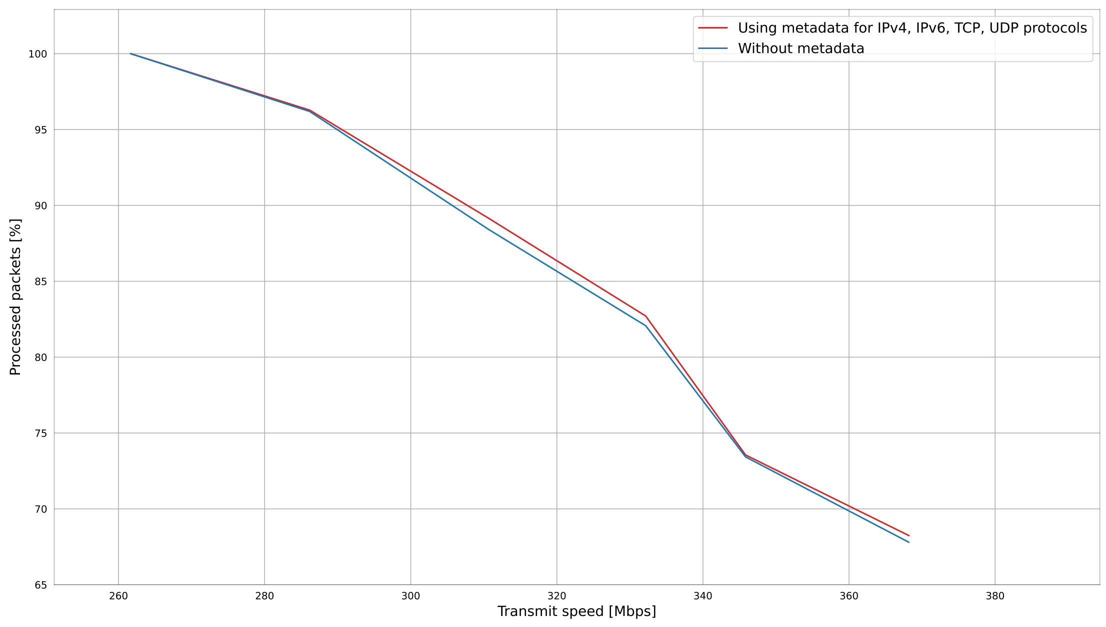


Figure 5.3: Enabled rules, 1 Prefilter worker – 1 Suricata worker.

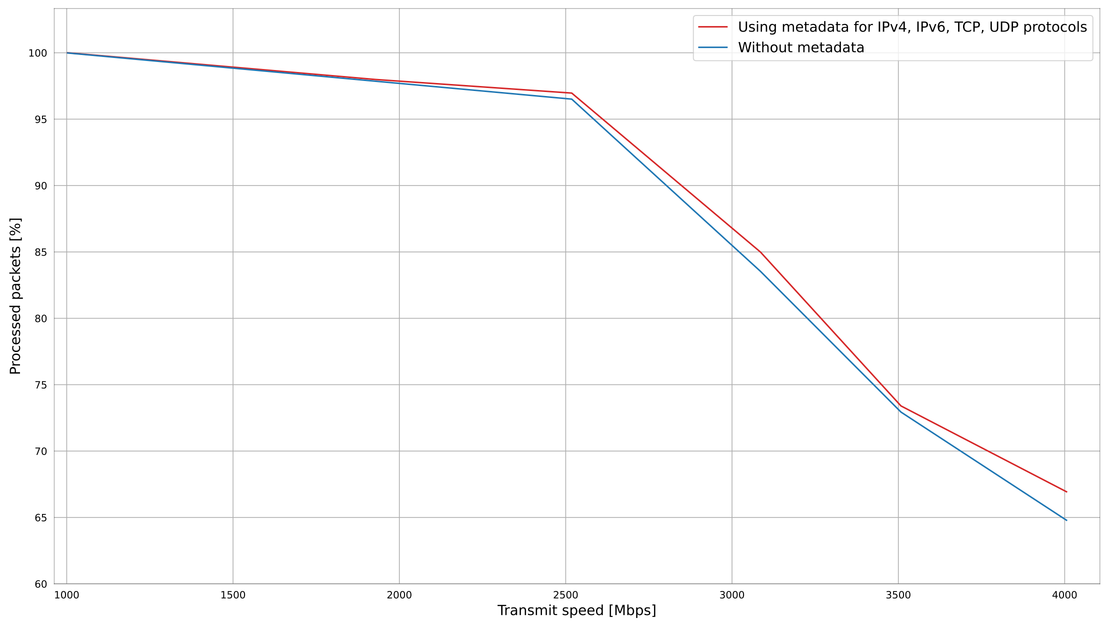


Figure 5.4: Disabled rules, 1 Prefilter worker – 1 Suricata worker.

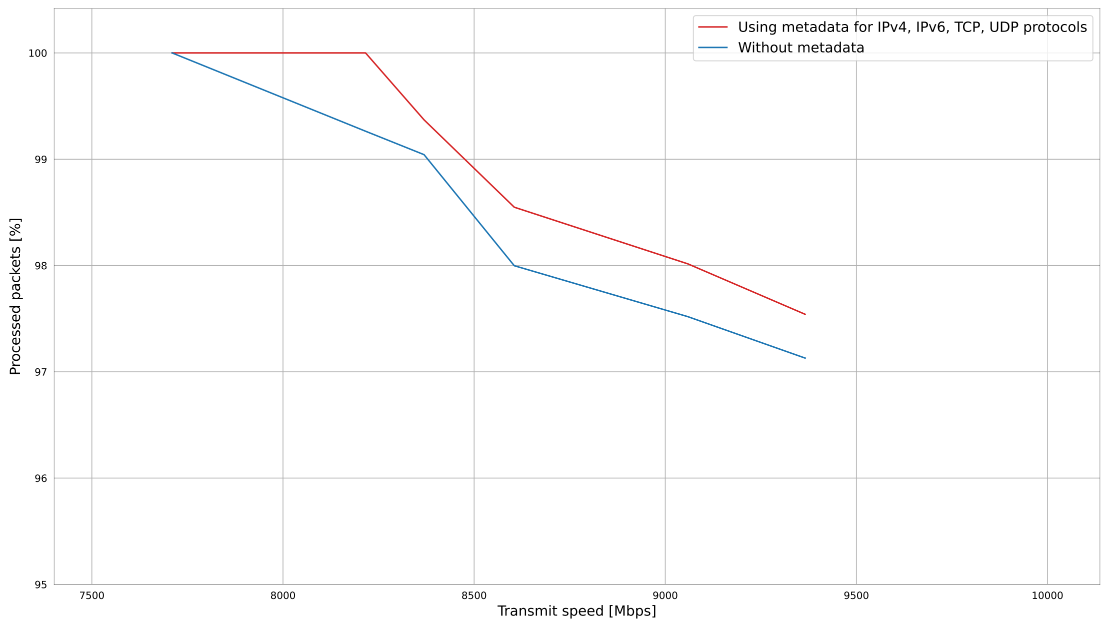


Figure 5.5: Disabled rules, 1 Prefilter worker – 3 Suricata worker.

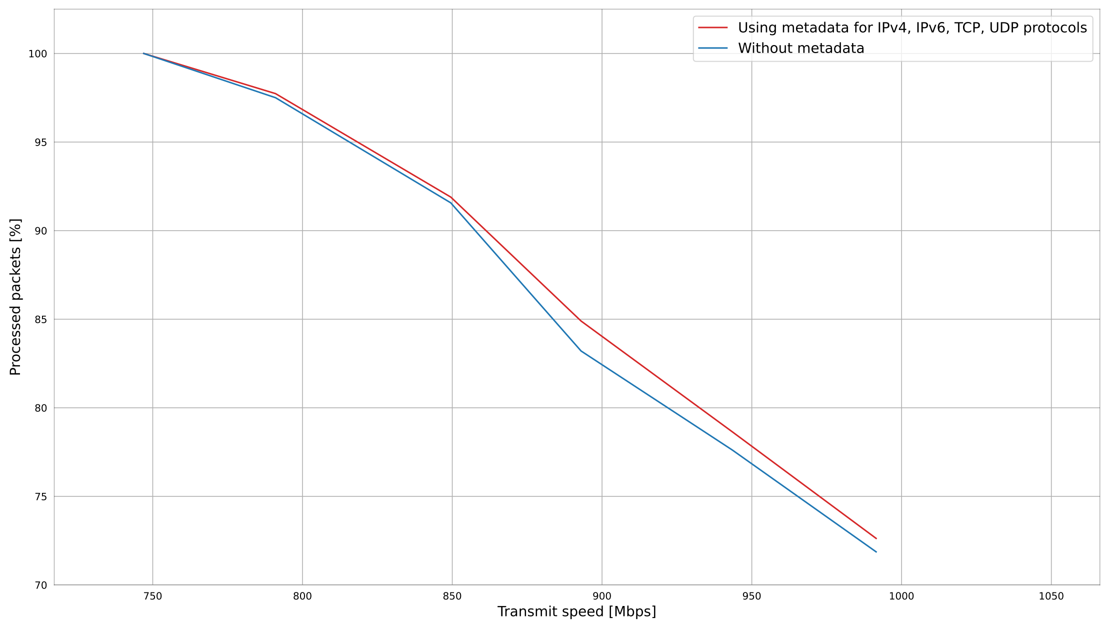


Figure 5.6: Enabled rules, 1 Prefilter worker – 3 Suricata worker.

The aforementioned four figures provide valuable insights into the impact of metadata on Suricata. It is evident that the integration of metadata does not degrade the performance of Suricata. In some cases, a minor performance improvement was even observed, further demonstrating the feasibility of metadata support in Suricata. These results serve as a strong foundation for future research and development in the field of metadata within Suricata.

An improvement is more noticeable on the graphs representing Suricata's operation without rules compared to those with rules enabled. The reason for this is that when the rules are enabled, Suricata allocates a lot of resources to the detection module for tasks such as pattern matching. In the absence of rules, resources are distributed more uniformly throughout Suricata, enabling it to process more incoming packets. However, it is important to understand that network header parsing is not the main part of Suricata. The purpose of these graphs is to demonstrate the successful implementation of metadata in Suricata, rather than to emphasize its impact on overall performance.

Chapter 6

Conclusion and future work

This work is aimed at Suricata. Suricata is a high-performance open-source software used for network monitoring and threat detection. Suricata faces a significant problem of processing all incoming network traffic in real-time, which can lead to missed threats. Transmitting additional information or reducing the amount of analyzed traffic can be used to accelerate Suricata and solve this problem. Advanced network cards can help Suricata in faster packet processing using the former method. However, hardware development is complex, and it may be helpful to evaluate what additional information is useful before it is implemented in hardware. Testing it in software before implementation can be less expensive. DPDK Prefilter can be used for these purposes, allowing the simulation of hardware functions in software.

The goal of this bachelor's thesis was to come up with a possible approach for transmitting metadata to the Suricata IDS/IPS system. After that, the next steps were implementing metadata support for both the Suricata and DPDK Prefilter applications, composing integration and performance tests, and evaluating the impact of different metadata parameters on the overall packet processing of Suricata.

Additional work was conducted in the form of integration and performance tests, which enriched the results of this study. Although not originally included in the assigned task, these extra tests served as proof of the proposed method's successful functionality.

The integration tests fully covered the added metadata support for Suricata. The test results demonstrate that the metadata was accurately integrated into Suricata without compromising the system's integrity.

Since packet header information is not a complex task in Suricata, it was not expected that it brings huge performance benefits. The implementation of metadata is versatile, enabling the straightforward addition of new metadata for both DPDK Prefilter and Suricata. Metadata can be therefore added on a per-packet basis without any significant cost. This statement is supported by the data from performance tests. This proves the value of metadata that can be transmitted to Suricata encapsulated in each packet. The implementation of metadata is versatile, enabling the straightforward addition of new metadata for both DPDK Prefilter and Suricata.

The results of performance tests highlight the potential of metadata as a method of transmitting additional information with network traffic to Suricata. The extensible implementation of metadata enables the addition of new metadata offloads, facilitating future work that could involve various experiments with different metadata. This would help in identifying the information that Suricata requires to accelerate the processing of network packets.

One potential extension could involve transmitting information for string pattern matching. With a large number of rules in place, Suricata demands significant CPU resources to identify patterns among all rules. Performing a portion of pattern matching in hardware and subsequently supplying metadata for Suricata could play a significant role in optimization.

Another potential extension could involve encrypted traffic. When Suricata encounters encrypted traffic, it cannot identify the protocol from the Application Layer (seventh layer) and instead relies on machine learning-based estimation techniques. If metadata included information about the Application Layer along with the packet, it could potentially enhance packet bypassing based on the protocol.

Bibliography

[1] *Internet Protocol* [RFC 791]. RFC Editor, september 1981. DOI: 10.17487/RFC0791. Available at: https://www.rfc-editor.org/info/rfc791.

[2] Amaral, T. A. Navarro do, Rosa, R. V., Moura, D. F. C. and Esteve Rothenberg, C. Run-Time Adaptive In-Kernel BPF/XDP Solution for 5G UPF. *Electronics*. 2022, vol. 11, no. 7. DOI: 10.3390/electronics11071022. ISSN 2079-9292. Available at: https://www.mdpi.com/2079-9292/11/7/1022.

[3] Ashtari, H. *Intrusion Detection System vs. Intrusion Prevention System: Key Differences and Similarities* [online]. March 2022 [cit. 2022-12-24]. Available at: https://www.spiceworks.com/it-security/network-security/articles/ids-vs-ips/.

[4] Belkhiri, A., Pepin, M., Bly, M. and Dagenais, M. Performance analysis of DPDK-based applications through tracing. *Journal of Parallel and Distributed Computing* [online]. 2023, vol. 173, p. 1–19, [cit. 2023-02-19]. DOI: https://doi.org/10.1016/j.jpdc.2022.10.012. ISSN 0743-7315. Available at: https://www.sciencedirect.com/science/article/pii/S0743731522002271.

[5] Borman, D. *TCP Options and Maximum Segment Size (MSS)* [RFC 6691]. RFC Editor, july 2012. DOI: 10.17487/RFC6691. Available at: https://www.rfc-editor.org/info/rfc6691.

[6] Brouer, J. D. *Prototype Kernel documentation - XDP Actions* [online]. [cit. 2023-02-28]. Available at: https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/implementation/xdp_actions.html.

[7] Chockalingam, L. *What Is eBPF and Why Does It Matter for Observability?* [online]. [cit. 2023-03-02]. Available at: https://newrelic.com/blog/best-practices/what-is-ebpf.

[8] Coders, P. *What is DPDK?* [online]. [cit. 2023-02-12]. Available at: https://www.packetcoders.io/what-is-dpdk/.

[9] Foundation, L. *BPF and XDP Reference Guide - Program Types* [online]. [cit. 2023-02-28]. Available at: https://docs.cilium.io/en/stable/bpf/progtypes/#xdp.

[10] Foundation, L. *Getting Started Guide for Linux: 43. Multi-process Support* [online]. [cit. 2023-02-25]. Available at: https://doc.dpdk.org/guides/prog_guide/multi_proc_support.html.

[11] Foundation, L. *Getting Started Guide for Linux: 9. EAL parameters* [online]. [cit. 2023-01-2]. Available at: https://doc.dpdk.org/guides/linux_gsg/linux_eal_parameters.html.

[12] Foundation, L. *Programmer's Guide: 4. Environment Abstraction Layer* [online]. [cit. 2023-02-12]. Available at: https://doc.dpdk.org/guides/prog_guide/env_abstraction_layer.html.

[13] Foundation, L. *Programmer's Guide: 8. Ring Library* [online]. [cit. 2023-02-20]. Available at: https://doc.dpdk.org/guides/prog_guide/ring_lib.html.

[14] GeeksforGeeks. *Thread in Operating System* [online]. [cit. 2023-02-12]. Available at: https://www.geeksforgeeks.org/thread-in-operating-system/.

[15] Goss, M. 12 common network protocols and their functions explained. [online]. august 2020, [cit. 2023-02-19]. Available at: https://www.techtarget.com/searchnetworking/feature/12-common-network-protocols-and-their-functions-explained.

[16] Help, S. T. *Top 10 BEST Intrusion Detection Systems (IDS)* [online]. [cit. 2023-02-27]. Available at: https://www.softwaretestinghelp.com/intrusion-detection-systems/.

[17] Jakub Kicinski, N. V. EBPF Hardware Offload to SmartNICs: cls bpf and XDP. [online]. september 2019, [cit. 2023-03-16]. Available at: https://legacy.netdevconf.info/1.2/papers/eBPF_HW_OFFLOAD.pdf.

[18] KernelTalks. *What are the huge pages in Linux?* [online]. [cit. 2023-02-13]. Available at: https://kerneltalks.com/services/what-is-huge-pages-in-linux/.

[19] Melnick, J. *Top 10 Most Common Types of Cyber Attacks* [online]. May 2018 [cit. 2022-12-24]. Available at: https://blog.netwrix.com/2018/05/15/top-10-most-common-types-of-cyber-attacks.

[20] OISF. *Suricata documentation - 19.4 eBPF and XDP* [online]. [cit. 2023-02-28]. Available at: https://suricata.readthedocs.io/en/latest/capture-hardware/ebpf-xdp.html.

[21] OISF. *Suricata documentation - 6.1 Rules format* [online]. [cit. 2023-02-15]. Available at: https://suricata.readthedocs.io/en/suricata-6.0.0/rules/intro.html.

[22] Raza, M. OSI Model: The 7 Layers of Network Architecture. [online]. june 2018, [cit. 2023-02-19]. Available at: https://www.bmc.com/blogs/osi-model-7-layers/.

[23] Šišmiš, L. *Optimization of the Suricata IDS/IPS*. Brno, CZ, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/23479/.

[24] Šuráň, J. *Evaluation of rte_flow Network Interface Cards Support*. Brno, CZ, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/24818/.

[25] Wiki, D. *Hugepages* [online]. [cit. 2023-02-15]. Available at: https://wiki.debian.org/Hugepages.

[26] Yasar, K. and Zola, A. Network packet. [online]. july 2022, [cit. 2023-02-19]. Available at: https://www.techtarget.com/searchnetworking/definition/packet.