BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

ACCELERATING SURICATA WITH PATTERN-MATCHING METADATA

AKCELERACE SYSTÉMU SURICATA PROSTŘEDNICTVÍM VYHLEDÁVACÍCH METADAT

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DAVID TOBOLÍK**
AUTOR PRÁCE

SUPERVISOR **Ing. LUKÁŠ ŠIŠMIŠ**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

154375

Institut:	Department of Computer Systems (DCSY)
Student:	**Tobolík David**
Programme:	Information Technology
Title:	**Accelerating Suricata with pattern-matching metadata**
Category:	Networking
Academic year:	2023/24

Assignment:

1. Study the DPDK library and the Suricata and DPDK Prefilter systems.
2. Explore the possibilities of pattern matching in the network traffic.
3. Design a pattern-matching component into the DPDK Prefilter system and then propose the use of packet metadata in the Suricata system.
4. Implement the proposed solution and analyze its results under the selected traffic and a ruleset.
5. Discuss the results and propose further work based on the achieved results.

Literature:

- According to the instructions of the supervisor.

Requirements for the semestral defence:

- Fulfillment of goals 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:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	30.10.2023

Abstract

Suricata is a network monitoring application inspecting packets using a set of rules to detect malicious activity. One of the main detection mechanisms is pattern-matching, however, it is a resource-intensive process taking up most of the application processing time. This thesis focuses on designing a new component to help reduce the amount of pattern-matching in Suricata. The new component was implemented in an application called DPDK Prefilter used to simulate specialized hardware components using software implementation. It adds detection metadata to packets, which are used in Suricata to potentially skip patterns matching if the packet was checked and no patterns were found. The implementation utilizes DPDK for inter-process communication and sharing data, and Hyperscan was used as a pattern-matching engine. Different types of detection metadata were designed and implemented and some of them have shown improvements in the performance of Suricata by reducing the amount of pattern-matching.

Abstrakt

Suricata je aplikace pro monitorování sítí, která prohledává pakety pomocí sady pravidel pro rozpoznání vzorů v síťovém provozu a v případě, že detekuje podezřelou aktivitu, vyvolá upozornění. Pro porovnávání vzorů Suricata využívá pattern-matching, což je proces náročný na výpočetní zdroje a tvoří většinu času běhu aplikace. Tato práce se zaměřuje na návrh nové komponenty, která si klade za cíl snížit množství porovnávaných vzorů v systému Suricata pomocí přibližného vyhledávání vzorů v aplikaci zvané DPDK Prefilter, která slouží k simulaci specializovaného hardware pomocí softwarové implementace. Nová komponenta přidává vyhledávací metadata k paketům, která jsou v Suricatě použita k potenciálnímu přeskočení pattern-matchingu v případě, že byl paket zkontrolován v DPDK Prefilteru a nebyly nalezeny žádné vzory. Implementace využívá DPDK pro meziprocesovou komunikaci a sdílení dat, pro pattern-matching byl použit Hyperscan. V rámci práce byly navrženy a implementovány různé typy vyhledávacích metadat a některé z nich dokázaly vylepšit výkon Suricaty díky snížení množství pattern-matchingu.

Keywords

Suricata, IDS, IPS, Hyperscan, metadata, DPDK, DPDK Prefilter, pattern-matching, network monitoring, detection module

Klíčová slova

Suricata, IDS, IPS, Hyperscan, metadata, DPDK, DPDK Prefilter, vyhledávání vzorů, monitorování sítí, detekční modul

Reference

TOBOLÍK, David. *Accelerating Suricata with pattern-matching metadata.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Lukáš Šišmiš

Rozšířený abstrakt

Tato bakalářská práce se zabývá optimalizací systému Suricata, který slouží k monitorování síťového provozu. Avšak algoritmy pro vyhledávání vzorů v síťových datech jsou výpočetně náročné a zabírají většinu výpočetního času aplikace. Cílem této práce bylo nastudovat, jak funguje vyhledávání vzorů v Suricatě, navrhnout komponentu, která by urychlila tyto náročné výpočty a změřit dopad implementovaných změn.

Suricata je open-source software určený k detekci a prevenci hrozeb v síťovém provozu. Základním prvkem podobných systémů je uživatelem definovaná sada pravidel, která jsou porovnávána s monitorovaným provozem. V případě, že jsou nalezena data, která odpovídají některému pravidlu, tak může Suricata vyvolat upozornění, případně daný provoz zablokovat. Avšak část těchto výpočtů by mohla být přesunuta do specializovaného hardwaru, který by mohl porovnávání značně urychlit.

Protože vývoj funkcí, které mohou být přesunuty na specializované síťové karty nebo vývoj nových hardwarových komponent je náročný, je vhodné nejdříve implementovat a otestovat funkce v softwaru. Nová komponenta proto byla implementována v systému DPDK Prefilter, který umožňuje simulovat tyto funkce pomocí softwarové implementace. Při implementaci nové komponenty byla využita knihovna DPDK, kterou využívá i DPDK Prefilter i Suricata, a je používaná k optimalizaci vysokorychlostních síťových aplikací. Dále byl použit Hyperscan, což je knihovna optimalizovaná pro efektivní vyhledávání vzorů a regulárních výrazů v datech.

Jak již bylo zmíněno na začátku, porovnávání vzorů specifikovaných v pravidlech je náročný proces, proto Suricata implementuje několik optimalizací, aby snížila množství prohledávaných vzorů. Jedno pravidlo může obsahovat i několik vzorů, ale aby dané pravidlo mohlo být aplikováno na určitý provoz, musí v něm být nalezeny všechny vzory z daného pravidla. Suricata proto z každého pravidla vybere pouze jeden vzor, který prohledá jako první a podle výsledku se rozhodne, jestli budou prohledány i ostatní vzory. Právě na tuto část vyhledávání se zaměřuje tato práce a analyzuje způsoby, jakými by šlo přesunout část vyhledávání do DPDK Prefilteru.

Na základě analýzy byly zvoleny 2 typy vzorů, které byly vhodné pro předzpracování v DPDK Prefilteru. Následně byla navržena komponenta do Suricaty pro zpracování, ukládání a sdílení vzorů s DPDK Prefiterem. Prefilter byl rozšířen o komponentu, která přijme vzory ze Suricaty, inicializuje Hyperscan a následně za běhu prohledává obsah paketů a podle výsledků vyhledávání přidává metadata. Suricata pak kontroluje, zda v Prefilteru byly nalezeny nějaké vzory a je potřeba důkladnější inspekce a nebo se prohledávání může přeskočit.

První část práce se zabývá popisem základních pojmů a principů používaných pro detekci a prevenci síťových hrozeb v Suricatě. Detailněji je popsán detekční modul, který byl v rámci této práce rozšířen. Čtenář bude obeznámen s jeho funkčností a jednotlivými fázemi prohledávání síťového provozu, což je důležité pro pochopení nově navržené optimalizace. Dále je v práci krátce zmíněn DPDK Prefilter a jeho fungování v konfiguraci se Suricatou, jejich vzájemná komunikace a předávání dat.

Jedna kapitola je věnována Hyperscanu, který byl zvolen pro vyhledávání vzorů. Je popsán princip jeho funkčnosti, jsou vysvětleny jednotlivé režimy a relevantní rozšiřující nastavení. Hyperscan je používán i v Suricatě, jeho využití zjednodušilo implementaci a zajistilo kompatibilitu vyhledávání v DPDK Prefilteru a Suricatě. Pro vyhledávání vzorů byla využitá funkce předfiltrování dat, která se v Suricatě nevyužívá, ale byla vhodná pro předzpracování dat v této situaci.

Další kapitola se zabývá detailnějším popisem stávajícího stavu detekčního modulu. Rozbor rozebírá i implementační detaily a vysvětluje některá omezení, která měla vliv na návrh a implementaci nové komponenty.

Poslední kapitola popisuje konkrétní implementaci navržených změn, vysvětluje způsob výkonnostního testování a prezentuje dosažené výsledky. Měření výkonu bylo provedeno ve 2 různých konfiguracích Suricaty a DPDK Prefilteru na 2 typech provozu pro 4 různé varianty vyhledávacích metadat a variantu bez metadat, která sloužila k porovnání změn. Výsledkem výkonnostních testů bylo zjištění, že 2 z navržených variant dokázaly v některých případech vylepšit propustnost Suricaty o více než pětinu, v jiných naopak byly srovnatelné s variantou bez metadat. V rámci shrnutí výsledků je popsáno, pro jaké případy je nové vylepšení vhodné a ve kterých případech neposkytuje výrazné zlepšení.

Závěr práce shrnuje poznatky zjištěné v průběhu práce a rekapituluje dosažené výsledky. Obsahuje také návrhy na zlepšení nebo využití vytvořeného rozšíření.

Accelerating Suricata with pattern-matching metadata

Declaration

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

.....
David Tobolík
May 6, 2024

Acknowledgements

I would like to express appreciation to my supervisor Ing. Lukáš Šišmiš, for guiding me, sharing technical knowledge, providing valuable feedback to my work and helping me navigate through challenges.

Contents

Chapter 1

Introduction

Internet connects billions of devices worldwide through thoughtfully engineered infrastructure consisting of links, routers, switches, and other network devices that ensure connectivity between end devices. The number of devices will increase significantly with the concept of the Internet of Things. Some predictions (e.g. [19]) even say that IoT devices will make up to 75 % of connected devices. [11] With growing numbers of devices and connections grows also the amount of malicious traffic targeted at specific services trying to exploit weaknesses to gain unauthorized access, disrupt the functionality of the services, spread malware, or gain information for planning further attacks. Other factor contributing to the number of attacks is the availability of tools to create increasingly sophisticated attacks with requiring less technical knowledge [17].

2017 Black Hat Attendee Survey [3], answered by 580 IT security professionals, has shown the following results:

- *„About two-thirds of respondents think it's likely that their own organizations will have to respond to a major security breach in the next 12 months.“*
- *„60 % of respondents believe that a successful cyber attack on US critical infrastructure will occur in the next two years.“*

It is apparent that companies and individuals are exposed to various attacks. Because the majority of the attacks occur over the network, monitoring of the traffic can help detect suspicious activities or uncover the recon stage of potential attacks. Even though many modern attacks combine several sophisticated techniques including social engineering, which cannot be simply prevented, it is still important to monitor and analyze network traffic. This is where intrusion prevention and detection systems come into play. They perform a series of checks, like filtering and deep packet inspection, to find and report observed malicious activities and policy violations. However, with growing numbers of devices and connections, and increasing network bandwidths, it becomes challenging to process the high volumes of network traffic. That is why it is important to constantly improve existing solutions to not fall behind the evolution of computer networks.

This thesis focuses on one such monitoring system – Suricata and improving its performance. Chapter 2 describes the basic functionality of Suricata and contains a high-level overview of its architecture and a more detailed description of the module responsible for detecting patterns in inspected network traffic. It also introduces the DPDK library and DPDK Prefilter which are technologies used to implement a component aiming to improve performance of pattern detection in Suricata. The goal is to familiarize the reader

with the current state of Suricata and the technologies used in the implementation part of this thesis.

Chapter 3 describes Hyperscan, which is a high-performance pattern-matching engine used in Suricata. The chapter focuses on Hyperscan functionality and description of the API to help the reader understand how it can be used in the context of Suricata and DPDK Prefilter to implement an extension of both systems to improve the performance of pattern-matching.

A high-level overview of the proposed design is discussed in Chapter 4, which also contains a more detailed analysis of the implementation of patterns in Suricata's detection module which was extended to move part of pattern-matching to the DPDK Prefilter.

Finally, Chapter 5 describes the implementation of a new module, which adds detection metadata to packets and enables Suricata to skip part of pattern-matching performed on the inspected traffic. The chapter also contains results of testing measuring the impact of implemented changes on performance.

Chapter 2

Suricata

Suricata is an open-source detection engine developed by the Open Information Security Foundation[1]. It is used to improve network security by monitoring and analyzing network traffic.

Suricata was developed as a modern multi-threaded intrusion detection system (IDS) and intrusion prevention system (IPS) to be backward compatible with Snort, another widely used network inspection software. On single-core machines, Snort outperforms Suricata in terms of performance and throughput. However, on multi-core machines with rulesets optimized for Suricata, Snort fell behind. [23]

Suricata has received attention of large companies, many of which have started using Suricata and some of them even contributed to the project, for example, Intel implemented the use of Hyperscan in Suricata [9]. Suricata has been adopted in various network environments, ranging from small enterprises to large-scale networks and data centers, including cloud-based ones such as Amazon Web Services [2].

This chapter focuses on explaining basic concepts used in Suricata to better understand how it processes packets from capturing it from NIC, through decoding, inspection, and producing outputs. Section 2.1 describes the Suricata architecture as well as the basic concepts used in different parts of Suricata, called thread modules. Section 2.2 focuses on the format and semantics of the Suricata rules, and describes which parts of the rules are important in pattern-matching. More attention is paid to the detection module in Section 2.3, which will help better understand the decisions made in the implementation part of this thesis.

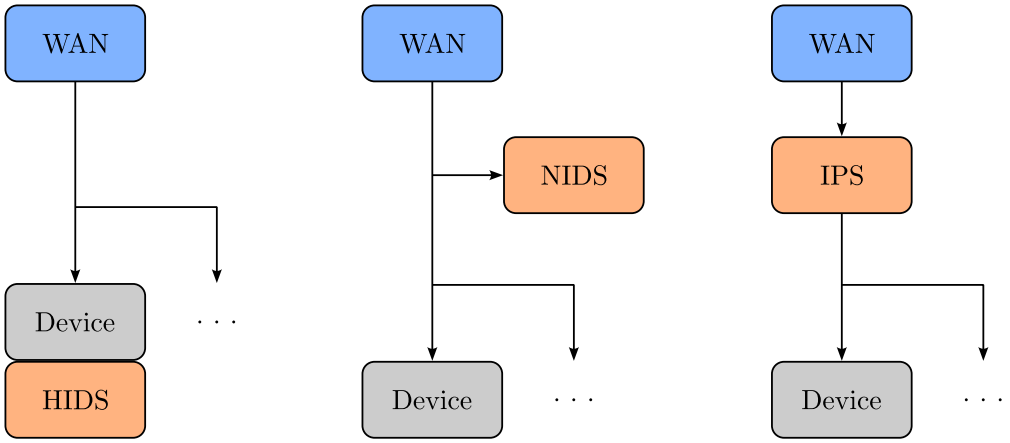
As was mentioned earlier, Suricata can be used both as IDS and IPS. Both modes are used to detect malicious activities on the network, however, there are some differences. As explained in [5] the characteristics are:

- **IDS** – Intrusion Detection Systems are designed to detect potential threats and anomalies. It is a way of passive monitoring of network traffic, which means no actions are taken to prevent potential threats. Instead an alert is generated and the action taken depends on the recipient. It is used mainly when high availability of the monitored system is required. IDS can be deployed to monitor a particular machine, called Host-based IDS (HIDS), or an entire network, called Network-based IDS (NIDS). The configuration of the deployment depends on the requirements for the system and the traffic that should be analyzed. In Figure 2.1a we can see HIDS deployment. In this case, the IDS system only inspects traffic on the end device on which it runs. This is used to monitor critical resources that run on the device. When the entire

[1] https://oisf.net/

network should be monitored, NIDS is used, as shown in Figure 2.1b. It does not stand in the traffic path, so the traffic has to be copied and sent to the IDS system.

- **IPS** – Intrusion Prevention Systems monitor and analyze the traffic like the IDS, but also have mechanisms to prevent malicious traffic from passing through. In the deployment of an IPS, all traffic must pass directly through the system, which can potentially lead to worse network throughput. Deployment diagram of Inline IPS can be seen in Figure 2.1c. All traffic going through the IPS is monitored and this also allows the system to drop or modify the packets.



(a) Host-based Intrusion Detection System (b) Network-based Intrusion Detection System (c) Inline Intrusion Prevention System

Figure 2.1: Comparison of IDS/IPS deployments, Source: [20]

2.1 Architecture

This section provides a high-level overview of Suricata architecture and explains the main principles of analyzing network traffic. Suricata uses a common architecture found in many Intrusion Detection and Intrusion Prevention systems. It consists of several modular threads, each having a specific role in packet processing. This modular design enhances the scalability and performance of Suricata but also allows adaptation to different use cases and deployment scenarios.

2.1.1 Thread modules

Suricata implements a multi-threaded design to handle different tasks needed in IDS/IPS applications. There are 2 main thread types – management and packet processing threads, also called worker threads or simply workers. Management threads take care of configuration, loading and updating rules used to detect patterns in inspected network traffic, managing and recycling flows, and collecting statistics. They interact with worker threads but are not directly involved in packet processing.

Worker threads interact directly with the processed packets and contain thread modules. Thread modules are elemental parts of Suricata's packet processing pipeline. Each of them plays an important role in the process and has one main responsibility. The order in which

the modules process the packet is shown in Figure 2.2. The output of each module is sent to the next one through a packet queue, and often the modules add metadata to the packet, which can be used further in the packet processing. The pipeline ends in the Outputs module, which generates alerts, logs, collects statistics, etc. Bellow are explained basic functionalities of each thread module as described in Section 12 of Suricata User Guide [14].

- **Packet capture** – The capture module acts as an entry point for network traffic and is responsible for acquiring packets from the Network Interface Card (NIC). It supports different packet acquisition methods, specified in configuration such as DPDK, AF_PACKET, PF_RING and others. Suricata also supports reading packet capture (pcap) files using libpcap[2] library.
- **Packet decoding** – This module decodes the contents of the packet based on the detected protocols and converts the data to an internal representation for further processing. This module also performs packet validation and maintains data about active streams and performs TCP reassembly.
- **Detection** – The detection module is the most resource-intensive because it performs deep packet inspection based on specified rules.
- **Output and alerting** – This module collects data from other modules and presents information about dropped packets, detected patterns, etc.

Figure 2.2: Suricata pipeline consisting of thread modules, Source: Section 12 in [14]

2.1.2 Runmodes

Suricata uses a multi-threaded approach, allowing several threads to run simultaneously. Each packet processing thread can run one or more thread modules, which are the building blocks of the packet processing pipeline. For a more detailed description of thread modules, see Subsection 2.1.1. Threads are connected with packet queues, allowing the packets to be passed between threads and thread modules, thus allowing the distribution of the work among the threads. A packet can be processed by only one thread at a time, and one thread can have only one thread module active. Threads can work with only one packet at a time, but if multiple worker threads are running, each of them can process packets independently, meaning that the number of packets processed at the moment is limited by the number of workers. How threads, thread modules, and packet queues are organized is defined by runmode, Suricata has 3 of them: [14]

- **Workers** – in workers runmode, there are several packet processing threads with all thread modules of the pipeline described in Subsection 2.1.1. The packets are distributed among the threads by a NIC driver. Each packet is fully processed only by one thread from capturing to outputs without passing between threads. It can

[2] https://www.tcpdump.org/

be seen in Figure 2.3a that the NIC delivers packets to multiple workers that contain all modules, with each group of modules representing one worker thread. This runmode is recommended to use, as it has the best performance in most cases.

- **Autofp** – this runmode uses two types of worker threads – capture and packet processing threads. Capture threads contain only the capture module and the rest of the pipeline is placed in the packet processing threads. The number of both thread types can be adjusted. The packets enter Suricata through the capture threads and are then passed to the packet processing threads. This also means that part of the load balancing is moved to Suricata, unlike in the workers mode. If one capture thread is present, it performs all load balancing, distributing the packets to the packet processing threads. When multiple capture threats are present, NIC acts as a load balancer for the capture threads, and from there Suricata takes care of distributing packets to worker threads. Autofp runmode with one capture thread can be seen in Figure 2.3b, the NIC forwards all packets to the capture thread running only the capture module and from there Suricata takes care of load balancing the traffic to the worker threads, which run the packet processing pipeline without the capture module. In Figure 2.3c we can see a similar configuration, but this time the autofp runmode has multiple capture threads configured. As was mentioned, the NIC distributes traffic to the capture threads and from there another load balancing takes place, this time in Suricata.
- **Single** – this runmode is used mainly for development purposes, it runs only one packet processing thread, to which all packets are forwarded. As we can see in Figure 2.3d the single thread contains the complete packet processing pipeline from capturing to decoding, detecting patterns, and finally outputs.

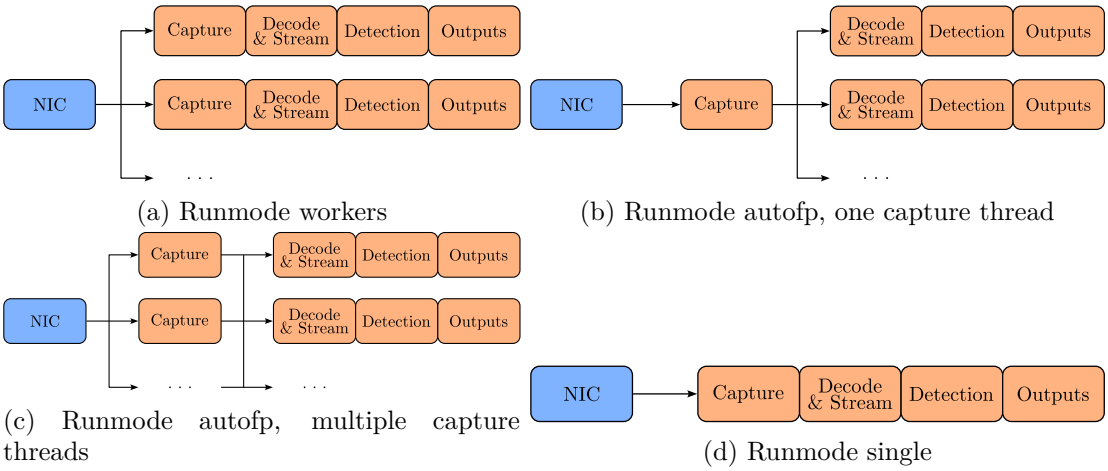


Figure 2.3: Comparison of Suricata runmodes, Source: Section 11 of [14]

2.2 Suricata rules

The core of IDS and IPS functionalities is made up of rules that define specific conditions, patterns, or behaviours of malicious activities in the analyzed traffic. These rules act as a set of instructions that allow the system to make decisions about network traffic. Each rule

is created to match against certain attributes within network packets to recognize known attack patterns or deviations from normal behaviour.

Suricata offers `suricata-update` tool along with OISF open-source rule databases to download and manage used rules. However, users can modify existing rulesets or even create entirely new rulesets using the rule definitions described Section 8 of the Suricata documentation [14].

Suricata rules, often referred to as signatures, specify patterns and anomalies for the IDS and IPS together with actions that should be taken for packets meeting the conditions. The rules have the following format [14]:

Action Header Rule options

All conditions in the rule header and rule options must be met to trigger the rule and take the specified action.

2.2.1 Action

Action determines what should happen with the packet that meets the conditions defined in the header and options sections. Possible actions are as follows:

- `alert` – generate an alert, which is further processed by an alert module
- `pass` – let the packet pass through the system
- `drop` – drop the packet, also generates an alert
- `reject`, `rejectsrc` – send TCP RST/ICMP unreachable to sender
- `rejectdst` – send TCP RST/ICMP error to receiver
- `rejectboth` – send TCP RST/ICMP error to both sender and receiver

2.2.2 Header

The header of the rule is used for filtering the traffic based on source, destination, the direction in which the packets are sent, and protocol.

Protocol Source IP Source port <>/-> Destination IP Destination port

- **Protocol** – Suricata provides 4 basic protocols to choose from – TCP, UDP, ICMP and IP, which matches all protocols from IP header. For more granularity, users can choose from many supported application layer protocols, to mention some of them: HTTP, FTP, TLS, DNS and more, all of them are listed in Sectio 8 of Suricata documentation [14].
- **IP addresses** – the rule is applied only if the packet source and destination match the specified IP addresses. These fields can contain special operators shown in Table 2.1.
- **Ports** – source and destination ports for transport layer protocols, the rule is only applied when these match. Ports can contain special operators as shown in Table 2.2.
- **Direction** – the filtered traffic can be either one directional (`->`) or bidirectional (`<>`).

IP format	description
address/prefix length	IP networks in CIDR notation
!	everything except the following ip rule
[.., ..]	list of ip rules
any	any IP address

Table 2.1: IP address operators

Port format	description
from:to	port ranges
!	negation of the following port rule
[.., ..]	list of port rules
any	matches all ports

Table 2.2: Port operators

2.2.3 Options

The rest of the rule is dedicated to options. Options can be used for fine-grained description of the packet, its payload, and header values. Options are often closely connected with specific network protocols and services, but also allow to filter flows, content, or even modify the payload, all of the supported options are described in detail in Suricata documentation [14].

Rule options are enclosed in parentheses and contain a list of options separated by a semicolon. Some options can contain additional settings specified after a colon, like this: `content:„403 Forbidden";`. Some options can even have multiple settings as in the following example: `flow:to_client, established`.

There are various types of options available in Suricata. Only a few of them are described to get an idea of how the rules are structured and to understand Suricata's capabilities. These keywords are taken from the Suricata documentation, where all available options can be found [14].

Meta keywords

These options don't affect the way Suricata searches for patterns but rather add metadata to the rules used mostly in logging and managing the rules. Here are some of the keywords with explanations:

- `msg` – adds a message to the rule describing its contents and possible alert.
- `sid` – gives the signature an ID by which it can be identified when an alert is generated.
- `requires` – the rule needs a specific Suricata version or features to be enabled, if these requirements are not met, the rule is ignored.
- `rev` – revision number of the rule, helps rule writers keep track of changes to the rule. If multiple rules with identical signature ID are present, Suricata takes only the one with highest revision number.
- `classtype` – specifies classification of the rule or alert. It is used to determine priority together with a classification configuration file, which contains keywords for classes,

their description, and priority. Priority is a bit of an exception because it tells Suricata in which order signatures should be evaluated, but it is still considered a meta keyword, as it does not contain any information about the packets themselves.

Modifier keywords

Some options affect other options or add special meaning to them, they are called modifier keywords and come in 2 types:

- **Content modifiers** – these options modify the preceding ones or can add meaning to them as shown in the examples below.
  - `content: „abc"; nocase;` here the `nocase` option means there will be no distinction between uppercase and lowercase characters.
  - `content:„index.php"; http_uri;` in this case the `http_uri` keyword says that the content should be found in the HTTP uri.
- **Sticky buffers** – keywords falling under the sticky buffers category are connected with the options after them, see the example below.
  - `ssh.proto; content:„2.0";` the `content` option is applied to the sticky buffer `ssh.proto`, meaning the rule will only match connections with SSH version 2.0.

Modifier keywords allow us to specify specific protocol header flags and values, offset and depth of patterns, and more. Many of these options exist as both content modifiers and sticky buffers, but the sticky buffer variant is the idiomatic way to write rules in Suricata.

Payload keywords

Payload keywords inspect the content of streams or packets. Some of these keywords are modifier keywords, which were described above. These options are mostly used as patterns used in the detection module to match against packet data.

- `content` – this option allows to specify byte sequences or strings to be found in the inspected traffic. Byte sequences are specified in hexadecimal format inside vertical bar characters such as `|61 0A|` and can be combined with regular characters. This allows us to search for non-printable symbols. Thus, the string `"Ab\nc"`, which cannot be used directly in the content option because of the line feed character, can be represented as `content:"Ab|0A|c";`.
- `nocase` – by default all patterns are matched as case-sensitive, this option enables case-insensitive search and is used as a content modifier. So `content:"abc"; nocase;` would match all the following strings: `"abc"`, `"aBc"`, `"ABC"`, `"AbC"`.
- `depth` – specifies the maximum of bytes from the beginning of the payload, which will be inspected for the pattern. In this way, the length of data to be inspected can be limites. Example: `content:"abc"; depth:3;` will match `"abcdef"`, but not `"aabcdef"`.

- `offset` – specifies how many bytes from the start of the payload should be skipped when matching the pattern. Example: `content:"def"; offset:3;` will match string `"abcdef"`, but not `"bcdefg"`, because the data will be searched from the fourth character, which is in this case `e`. It is often combined with `depth` option to specify a window in the payload to search. Example: `content:"def"; offset:3; depth:3;` will search only from fourth to sixth matching data, which can be expressed by the following regular expression `"...def.*"`.

- `pcre` – stands for Perl Compatible Regular Expression and allows to define searched patterns in a more advanced format than fixed strings and byte sequences provided by `content`. It is worth mentioning that regular expressions are not processed by Multi Pattern Matchers like Aho-Corasick or Hyperscan but by a separate pcre library. The format is the following: `pcre:"/<regex>/<options>";`. However regular expressions come with a high cost in terms of performance, so it is recommended to combine them with `content` keyword. In that case, the regular expression only gets evaluated when the content matches some data. Combination of these two could look like this: `content:"abc"; pcre:"/abc[0-9]{1,3}d?ef/i";`. The pcre is checked only if the string specified in the content is found, then it matches one to three digits, optional `d` character and mandatory `ef` all case-insensitively. However the content in this case is matched case-sensitively, so the case-insensitive search is only applied to the characters `d`, `e` and `f`. This can be solved by adding `nocase` after the `content` keyword.

Prefiltering keywords

Prefiltering options enable users and rule writers to gain better control of which parts of the signature are used as fast pattern in the prefiltering stage of pattern-matching. Both pattern-matching and fast pattern are further described in Subsection 2.3.1. In short, it enables Suricata to pick the least common pattern from the signature and possibly skip other patterns in the same signature. The fewer the patterns are found in traffic, the better. Suricata uses heuristics to determine pattern strength, but sometimes it is beneficial to specify fast pattern manually, when the pattern picked by Suricata would appear too often in the inspected network traffic. Suricata also allows to have only one fast pattern for each signature, so only one prefiltering keyword can be used.

- `fast_pattern` – acts as a content modifier and marks the preceding `content` as the fast pattern. Optionally, this option can have settings consisting of offset and depth in the following format: `fast_pattern:<offset>,<depth>;`, in that case only part of the content is used as a fast pattern.

- `prefilter` – by default `content` options are chosen as fast patterns for prefiltering. This option enables other options to be used as fast pattern, however, it should be used with caution. A good example of use case for the `prefilter` keyword is when all `content` patterns are only a few bytes long or when some other option is expected to rarely appear in regular traffic, such as the value of TTL[3] of zero.

[3] TTL (Time to Live) – value in IP header limiting the lifetime of a packet, when it reaches 0, the packet is discarded

2.3 Detection module

Suricata's detection module uses a range of techniques to analyze the characteristics of network packets. These techniques include rule-based detection, where predefined patterns associated with known attacks are matched against the packet payload, and behavioural analysis, which involves monitoring and identifying anomalies from regular patterns of normal network behaviour. The detection module operates in real time, allowing Suricata to quickly recognize and respond to potential threats.

The detection module is one of the most resource-intensive parts, and pattern-matching alone consumes up to 70% of processing time [7]. Therefore, many techniques are implemented to reduce the amount of traffic inspected. One of the straightforward ones is to match only some chosen parts of the rule first. As was mentioned in Section 2.2, all the conditions in the rule have to be met for the rule to be matched and the action to be taken. This means that if one of the conditions is false, the whole rule can be ruled out. This allows Suricata to take advantage of the rule header values to match protocol, source and destination IP addresses and ports and evaluate if they match the observed traffic before applying more complex parts of a rule.

2.3.1 Pattern matching

Pattern matching serves as a foundational mechanism for analyzing and identifying network traffic patterns to recognize predefined signatures, behaviours, or anomalies within the network data. Patterns in general can be strings, byte sequences, or regular expressions. By utilizing pattern-matching algorithms to check the network traffic against the predefined signatures, Suricata can decide what action should be taken next.

Pattern matching is a bottleneck not only of the detection module, but of the whole application. One of the most widely used pattern-matching algorithms is Aho-Corasick, also implemented in Suricata. However, it has one severe limitation – it can process only one input character in one cycle. This, together with the large memory footprint of the algorithm, presents a problem, where its efficiency is highly dependent on the memory speed [1]. Intel has implemented support for Hyperscan [9], which was shown to have ten times smaller memory footprint with the tested ruleset.

Suricata implements mechanisms to avoid pattern-matching completely, if possible. One of the ways is keeping track of whitelisted IP addresses and ports, if the packet contains either of them, it does not have to be inspected at all. Another way is keeping track of flows – if a flow is evaluated to pass through, all packets of this flow do not have to be inspected. Some flows only require the first few packets to be checked and all others have a pass. Suricata also keeps metadata about the applied rules and compares them with packet metadata. These metadata are designed in a way that if any metadata within a packet don't align with the rule metadata, the entire rule may be disregarded. These metadata include but are not limited to packets with specific TCP flags (SYN, ACK, RST), whether the packet has a payload or is empty, whether the packet or flow has triggered specific events during decoding or if the packet is part of a tracked flow.

Suricata performs pattern-matching in 2 stages – prefiltering and exact matching. Prefiltering utilizes fast patterns described below and matches the traffic against a group of patterns determined by the protocol specified in the IP header and server side port, see Subsection 2.3.2 to understand how rules are grouped. If any of the patterns match, the packet is marked for further inspection. If a rule is triggered on the packet, appro-

priate action is taken, this depends on the action specified in the rule itself (Section 2.2) and whether Suricata runs as IDS or IPS.

There are 2 types of pattern matchers used in the detection module – Multi Pattern Matchers (MPM) and Single Patterns Matchers (SPM). They are used as an abstraction for pattern matching engines used in Suricata, like Hyperscan or Aho-Corasick. MPMs are used with fast patterns in the prefiltering stage. They scan the data against multiple fast patterns and return all found matches. Based on the fast patterns that returned match in the prefiltering stage, the signatures they belong to are inspected further and for this type of pattern matching SPMs are used. They compare the data only to one pattern at a time. Note that MPMs and SPMs always match only string literals for better performance and searching for regular expressions is done with the PCRE library.

Fast pattern

Suricata rules can contain multiple patterns specified using the `content` keyword in the Options section of the rule, see Subsection 2.2.3. The rule is triggered only when all of those patterns are found in the inspected traffic. Suricata uses this to minimize the amount of pattern-matching, this process of picking the best patterns is called a fast pattern. The fast pattern is used to determine how unique the pattern is. The less likely the pattern will appear in regular traffic, the better. Fast patterns can be supplied manually by the user or automatically determined by Suricata during initialization.

When Suricata automatically chooses fast pattern, it breaks each rule into patterns and for each one determines its priority. The priority depends on the matched part of the packet, generally speaking, the lower-layer protocol sticky buffers have lower priority. This means that application layer sticky buffers tend to have the highest priority. After each pattern gets assigned priority, from each rule, the pattern with the highest priority is chosen as the fast pattern of the given rule.

If there are multiple patterns with the same highest priority, the longer pattern is used. In case multiple patterns have the same priority and length, for each one of them is calculated their pattern strength, which is a metric describing the diversity of characters present in the pattern. The way in which the strength of the pattern is calculated is shown in Algorithm 1. The pattern is processed character by character, and each of them increases the strength value. If the character appears for the first time in the pattern, it adds 3 to the strength if it is alphabetical. If the character is not alphabetical, but is printable or has a hexadecimal value of $0x00$, $0x01$ or $0xFF$, it increases the strength by 4, and in all other cases by 6. If a character with the same value is already present in the pattern, it always increases the strength only by 1.

If even after this step some patterns have equal pattern strengths, the registration order of the buffers they are looking into decides which will become the fast pattern, and if multiple patterns use the same buffer, the order of pattern in the rule is the deciding factor. [14]

2.3.2 Storing signatures

When Suricata defines a fast pattern for each signature, the signatures are sorted into groups called *Signature Group Head*. Rules are categorized into groups according to shared properties, more specifically, based on the protocol specified in the IP header, which is shown in Figure 2.4. TCP and UDP protocol groups are further split based on server-side ports. Each group can be bound by the minimum and maximum port numbers or is for 'any' port.

Algorithm 1 Pattern strength

```
1: function PatternStrength(pattern[0 ... length − 1], length)
2:     strength ← 0
3:     for character in pattern do
4:         if character occured in pattern for the first time then
5:             if character is alphabetical then
6:                 strength ← strength + 3
7:             else if character is printable or 0x00 or 0x01 or 0xFF then
8:                 strength ← strength + 4
9:             else
10:                strength ← strength + 6
11:            end if
12:        else
13:            strength ← strength + 1
14:        end if
15:    end for
16:    return strength
17: end function
```

During initialization, signatures are temporarily stored in a structure called `MpmCtx`. From there they are processed by the detection API into an internal representation, e.g. Hyperscan database, for use with the chosen pattern-matching engine.

2.3.3 Searching for patterns

Now that we know how Suricata organizes patterns during initialization, we can take a look at how they are used during detection. But before rules are checked for patterns using MPM, Suricata performs some checks before sending the packet to the pattern-matching engine. In the decode module, the packets are parsed and transformed to an internal representation containing metadata about the detected protocols, the direction of the packet, IP address, and ports. This information can be used to do IP and port matching before getting to expensive pattern-matching. There are several cases in which Suricata marks the packet to skip the pattern-matching part of detection, it can be one of the reasons described in Section 11 in [14].

- There is a pass or drop rule with no pattern-matching options. Suricata first evaluates the IP and port-only rules before the pattern-matching starts, meaning if a pass or drop rule is specified, the traffic meeting the requirements does not have to be inspected against other signatures.
- Suricata marks the flow for no inspection. There are cases in which Suricata inspects only the first few packets of a flow and can then pass or drop the rest of the flow. Packets are checked to be part of tracked flows before pattern-matching takes place, which means that in these cases it can be skipped.
- The packet is part of encrypted traffic. Suricata can be configured to stop processing traffic after the initial TLS[4] handshake and bypass the rest of the flow.

[4] TLS – Transport Layer Security

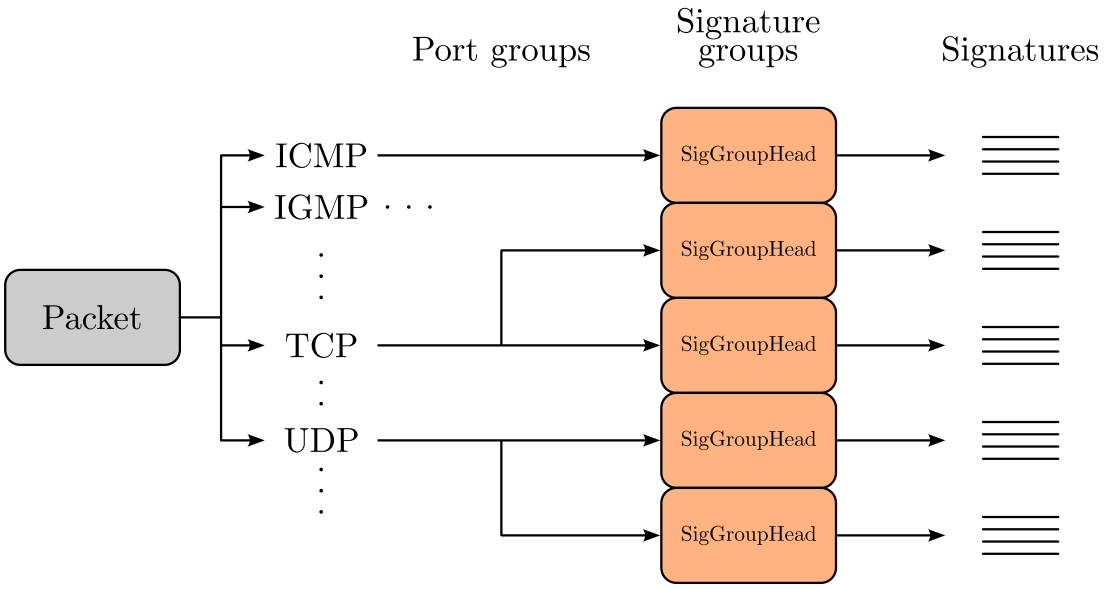


Figure 2.4: Rule groups in Suricata, Source: Section 12 in [14]

- Suricata has hit Exception Policy. Exception Policies are events triggered on special occasions like hitting a memory cap for certain services like flow tracking or stream reassembly, encountering errors when parsing application layers, etc. The action taken in such a case can lead to completely skip the pattern-matching, depending on the configuration. More details can be found in the Suricata documentation, Section 12 [14].

2.3.4 Prefiltering

Suricata uses fast patterns, described earlier in this section. This means the pattern is less likely to be matched against regular data and raise false positive alerts. Only after the fast pattern matches against some data in the traffic, other patterns from the same signature are matched against it. The process of scanning for fast patterns is called prefiltering as it rules out some signatures which don't have to be checked further. But those which match, have to be further evaluated to tell with certainty, if they match.

If the packet should be inspected, Signature Group Head is retrieved for the detected protocol and in case it is UDP or TCP also based on the port number. If there is no specific Signature Group Head for the port, the one for 'any' port is chosen. Then for all signatures the fast pattern is matched against the inspected data. Depending on the signature type, the searched data can be the packet payload, some header value, or even a reassembled stream of data. For all signatures that match, their ID gets stored in an array, which is used in the exact matching described in the next section.

2.3.5 Exact matching

If the packet is marked for further inspection, meaning one or more patterns matched in the prefiltering stage, Suricata will inspect the signature stored in the array from the prefilter. Only if all patterns from the signature match, the action specified in the rule is taken, rule format and semantics were described in Subsection 2.2.

Exact matching can take longer depending on the signatures it matched, mainly because Suricata has to go through all patterns from the matched rules, which can have various complexities of detection. It can involve searching for regular expressions, retrieving other parts of the flow the packet is part of, etc.

2.4 DPDK Prefilter

To improve the performance of Suricata, a DPDK Prefilter was introduced. The Prefilter is used to prototype and test extensions which could possibly be implemented into hardware. This way it allows to implement the changes in software first, measure the impact, and make better decision whether to implement it into hardware. It could be either hardware offloading into the NIC or a specialized hardware component. The Prefilter uses a powerful DPDK library used to accelerate packet processing applications. Prefiltering helps to better distribute the load and allows for better scalability.

2.4.1 DPDK

Data Plane Development Kit (DPDK)[5] is a powerful set of libraries and drivers designed to accelerate packet processing in network applications. Originally developed by Intel, DPDK provides a framework that enables efficient packet handling, making its use in many networking applications.

The primary reason for integrating DPDK into network applications is performance optimization. By using low-level abstractions and bypassing the kernel networking stack, applications can achieve significant improvements in throughput and latency, which are crucial for modern high-speed networks.

Poll mode drivers

Most operating systems schedule network tasks using interrupts, which provides balanced overhead and latency in cases where the network load is relatively low. However, in high-speed network applications, the interrupt-driven approach provides much worse performance and in some cases could even lead to livelock [13]. In this case, livelock means that the operating system uses all of the CPU time to handle interrupts, which come at a high rate, and little to none of the traffic gets to the desired application.

DPDK solves this issue with poll mode drivers, which directly connect network interface card and the DPDK application and bypass the kernel. This prevents context switching, occurring during interrupts, but the application has to actively poll the driver for new network data, however, this is standard procedure for high-speed network applications such as Suricata. [8]

Environment Abstraction Layer

Environment Abstraction Layer (EAL) is used to access environment-specific resources like memory space and hardware using a common DPDK interface. The EAL takes care of how these low-level resources are allocated and managed, giving users better control of the application behavior, while minimizing the need to understand implementation details.

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

DPDK has to be initialized by calling `rte_eal_init()`, which takes command-line arguments and prepares the system according to the specified options. This includes probing PCI devices, initializing hugepages if needed, launching threads on specified cores, and executing tasks provided by the user on these threads. The threads are often referred to as lcores. They are regular Linux/FreeBSD pthreads, but additionally they are managed by EAL.

When DPDK is used with multiple processes, one of them has to be initialized as a primary process. The primary process has full permissions to the shared memory and initializes it for any secondary processes. In addition, secondary processes can join only when the primary process is running. The type of process has to be specified during the initialization of the EAL for each process. In general, two primary DPDK processes cannot run simultaneously, however, it can be achieved by defining different namespaces. In that case, all processes in one namespace share memory and can communicate with each other, but this is not possible across namespaces. [8]

In Suricata, these arguments are set in the configuration file in the `dpdk` section. In the DPDK Prefilter, the ring specific configuration is set in the Prefilter configuration file and other arguments are passed on the command line: `prefilter <EAL args> -- -c <config file> <app args>`.

Hugepages

Hugepages are a feature implemented in many operating systems to allocate large chunks of memory. The default memory page size is on many platforms set to 4096 b. In general, for each page an application allocates a new entry to the page table is added. The page table is a structure that holds mappings between virtual memory addresses used by processes and actual physical memory addresses. To improve the performance of virtual memory translations, there is a hardware cache in processors called Translation Lookaside Buffer (TLB), which stores translation mappings for fast lookups. However, the TLB size is limited and cannot hold all entries in the page table. There are 2 scenarios of memory translations – either there is an entry in TLB for the mapped memory page, called TLB hit. In case there is no entry in the cache, it is called TLB miss, and it has to be retrieved from the page table, which is a more expensive operation.

This has one main implication – applications using a lot of memory will have a lot of entries in the page table and TLB miss will occur more often when accessing different parts of memory. This is especially problematic with memory fragmentation, when memory is allocated in many small chunks spread across the memory. Hugepages attempt to solve this problem by allowing applications to allocate larger pages of memory instead of the default 4 kiB ones, resulting in allocating continuous memory segments and more efficient memory translation lookups. [4] [16].

DPDK takes advantage of this technology and, unless this functionality is explicitly disabled, allocates memory using hugepages including memory for IPC communication and shared memory [8].

DPDK IPC messages

IPC stands for Inter Process Communication and is used to share information and/or trigger events between processes. IPC messages can be sent from and to both primary and secondary processes. Primary processes can only send broadcast messages to all secondary

processes, and secondary processes can send only unicast messages to the primary process. There are three different types of IPC messages in DPDK:

- Message – one-way message, where no response is expected.
- Synchronous request – blocking two-way communication mechanisms, the sender waits until a response arrives.
- Asynchronous request – non-blocking way of communication, where response is expected but is processed asynchronously.

Both synchronous and asynchronous requests require the timeout to be specified. After the timeout expires the sender will no longer wait for a reply from the receiver(s).

For the messages to be processed by the receiver, there has to be a defined way to handle the message, this is done with action registration. Each message has a name specified, and for each name, there can be a registered callback in the receiver process to handle this kind of message, if no callback for the given message name is registered, the message is ignored. Callbacks are handled by a special thread IPC or interrupt thread, which is not part of EAL lcore threads, but shares the same memory space. This enables to modify the state of the application threads. The same applies to asynchronous request callback. [8]

Shared memory

DPDK needs to ensure that the memory resources are properly distributed among the processes of the multi-process applications. DPDK uses shared memory between processes to manage Inter Process Communication, create shared queues, etc. When the application is launched in the primary process, DPDK records the details of the memory configuration to memory-mapped files. When the secondary process starts, the file is read and the EAL recreates the same memory configuration in the secondary process, so that all memory zones are shared between the processes, the memory pointers are valid, and refer to the same objects on both sides.

Since memory is shared at the start of the application, all processes have access to it, so if memory needs to be shared between them, they need to make sure they access the same part of it. This can be done by allocating either a memory pool or a memory zone. Memory pools are used to allocate memory for fixed-size objects and allow for better control over how objects are stored in the memory, have built-in caching, and more. Memory zones are named continuous parts of memory, but how it is used is up to the user. [8]

2.4.2 Architecture

DPDK Prefilter runs as the primary DPDK process and as was mentioned in Subsection 2.4.1 this means that the Prefilter has to be run first, initialize itself and then Suricata in DPDK runmode can be started as the secondary process.

The configuration of the DPDK Prefilter is shown in Figure 2.5. The Prefilter has two main communication interfaces, one with the NIC and one with the main Suricata application. The Prefilter retrieves packets from the Network Interface Card (NIC) using polling, which was described in Subsection 2.4.1.

After the packets are retrieved from the NIC, they are processed by the prefilter. The main tasks include decoding, checking if the packet can be matched against some

of the rules or requires further inspection, and attaching metadata, which are then sent with the packet to Suricata.

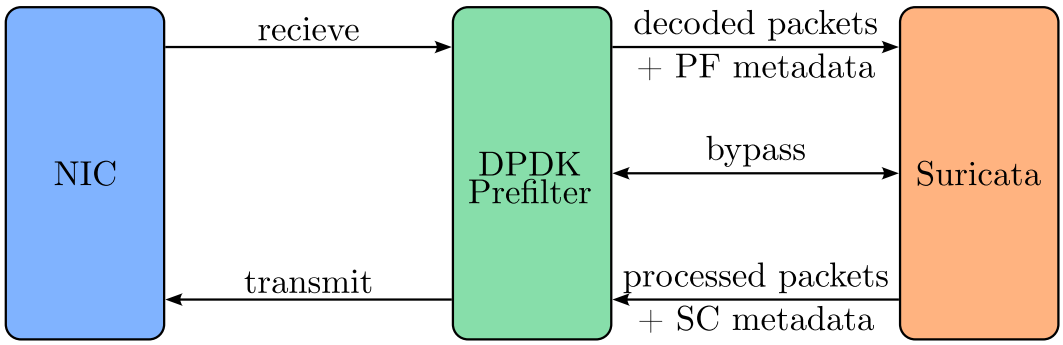


Figure 2.5: Suricata with DPDK Prefilter, Source: [18]

DPDK Prefilter operates in several stages, each thread has its state describing what action should be taken next. Each state is connected with initializing part of the prefilter, after it is done, the state is set to the following one until it reaches the „running“ state in which the thread enters the main loop, where it processes network traffic. The states are the following:

- `LCORE_INIT` – the thread initializes its variables, attaches rings, etc.
- `LCORE_OFLOADS_INIT` – sends information about used metadata configuration
- `LCORE_RUN` – enters the main app loop
- `LCORE_DETACH` – means Suricata has detached and the thread enters an idle state, if the Prefilter should be stopped the thread deinitializes
- `LCORE_STAT_DUMP` – collects statistics and prints them

2.4.3 Communication with Suricata

During initialization, the DPDK Prefilter and Suricata exchange IPC messages to synchronize the process and exchange information about the configurations of both applications. IPC messages are also used when shutting down to exchange statistics and deinitialize both applications.

- `LCORE_ACTION_{ATTACH, DETACH}` – sent from Suricata to the DPDK Prefilter after DPDK is initialized to attach the Prefilter and on termination to detach it
- `LCORE_ACTION_{START, STOP}` – signals when Suricata is listening for data from prefilter
- `LCORE_ACTION_BYPASS_TLB_DUMP_{START,STOP}` – manages exchange of statistics
- `LCORE_ACTION_OFFLOADS_SETUP` – initializes exchange of information about enabled metadata

DPDK Prefilter uses packet rings as queues to send and receive packets from Suricata. Rings or circular buffers in DPDK are used for communication or sharing data between applications, DPDK also uses them internally for managing IPC communication. It allows access from multiple consumers and producers without using locks while being thread-safe.

Chapter 3

Hyperscan

Hyperscan[1] is an open-source (BSD license) high-performance pattern-matching library developed by Intel written in C/C++. The algorithms used in Hyperscan are optimized to use Intel SIMD[2] instructions, therefore it is meant to run on x86 platforms. Its high performance makes it suitable for use in firewalls, IDS and IPS systems such as Snort and Suricata [21].

Hyperscan supports multi pattern matching for regular expressions using PCRE[3] syntax, however, it does not support all of the constructs. It matches the scanned data against multiple patterns at once and returns all matched patterns with additional information [15]. It is achieved by processing the data into a pattern database and using various regular expression matching techniques including deterministic finite automata, discovery of literal, decomposition of regular expressions, Glushkov Non-deterministic Finite Automata and more [12][22].

3.1 Pattern database

For Hyperscan to search for patterns, they have to be compiled from regular expressions into a pattern database. The compilation performs pattern analysis and optimizes them according to specified flags and/or adjusts the pattern database to the CPU architecture. The compiled database must be created before pattern-matching and remain untouched during the whole scanning process. This step moves part of the computing before the main part of the application and performs optimizations of the provided patterns, which leads to faster pattern-matching down the line.

Patterns supplied to a database contain an ID, flags specifying how the pattern should be matched, and optionally extended options used with some of the flags.

The whole database can be compiled in 3 different modes, which affect how Hyperscan scans data, we will have a look at all of these modes. We will have a look at some flags often used with patterns, a full list of pattern options can be found in Hyperscan documentation. [10]

- `HS_FLAG_CASELESS` – Hyperscan by default searches all patterns case-sensitively, this enables case-insensitive search for the given pattern.

[1] https://github.com/intel/hyperscan
[2] SIMD – Single Instruction Multiple Data
[3] PCRE – Perl Compatible Regular Expressions

- `HS_FLAG_SINGLEMATCH` – returns only the first match for each pattern ID as opposed to the default behaviour of reporting all matches. This can slightly reduce matching time, especially when multiple patterns share the same ID.
- `HS_FLAG_PREFILTER` – enables prefiltering mode of Hyperscan described in Subsection 3.1.1. It is used to approximately match the provided patterns and can lead to improved performance.
- `HS_FLAG_MULTILINE` – by default Hyperscan matches regular expression tokens `^` as begining of the data and `$` as end of the searched data. This option extends this to the start and end of a line respectively.

3.1.1 Prefiltering mode

Prefiltering is a special mode of Hyperscan that searches the provided patterns approximately. It can be achieved by providing a special flag when compiling the patterns which results in a pattern database that will match a superset of the provided patterns. The pattern database is optimized for performance, so even though the result isn't exact, it can be received relatively quickly opposed to matching against a regular pattern database.

This feature is often used in applications as the first stage of pattern-matching, which for exact results has to be followed by regular scan calls. However, if none of the patterns is matched, regular scanning can be skipped entirely, because none of the patterns from the database, which are a superset of the original patterns did not match, therefore the original patterns are not contained in the scanned data.

3.2 Runtime resources

Scanning uses the compiled pattern database to search data in memory, which has severe performance benefits, but to manage the internal state, Hyperscan needs two other structures – scratch space and stream state. However, both can be allocated in advance to further improve performance. Their size depends strongly on the pattern database, so it can only be allocated after the pattern database is compiled.

The scratch space is used to store internal data during scanning, after which it can be freed from memory. It is required for each thread that calls the scan function. If using multiple pattern databases, only one scratch space is necessary, however, it is needed to call `hs_alloc_scratch()` on each database that will be used with the scratch space, this will ensure that the scratch space size is sufficient to be used with any of them.

The stream state is only used in the stream mode, further discussed in Subsection 3.3, to store metadata, which helps to find patterns that are spread across multiple chunks of data. Similarly to scratch space, these metadata only persist for one open stream and are no longer needed when the stream is closed.

3.3 Hyperscan modes

Hyperscan can run in 3 different scanning modes – block, stream and vectored. Each one is indented for different use based on the characteristics of the data that should be scanned. The mode in which Hyperscan will run must be determined at the time of pattern database compilation. A pattern database has to be used only with API functions intended for the mode which it was compiled for. All scanning modes are described in the next sections.

Block mode

Block mode searches one continuous block of data and compares it with a pattern database. Data must be provided in a single block of memory, and for each block `hs_scan()` function has to be called. Because all pattern-matching is done in one call, no state has to be stored unlike in stream mode (see Subsection 3.3).

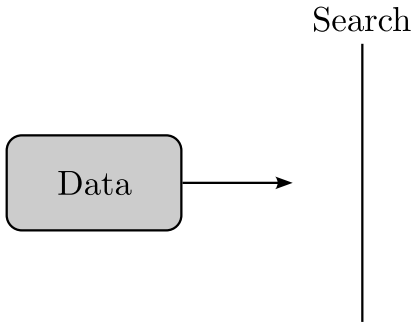


Figure 3.1: diagram of Hyperscan block mode

Vectored mode

Vectored mode accepts multiple blocks of data and searches them all at once as if it were a continuous block, compared to block mode, where data are scanned one by one. The `hs_scan_vector()` function takes an array of pointers to the scanned data, a size array with sizes of each data block, and the length of the previous arrays. This is useful when patterns are spread across multiple data blocks because in block mode these patterns would not be found.

Figure 3.2 shows the vectored scanning mode. All data is scanned at once as if it were one continuous block of data. From the user's perspective, the output is equal as if the data were copied into one place in memory and scanned using block mode or scanned one by one using stream mode in one stream. For a description of stream mode, see Subsection 3.3.

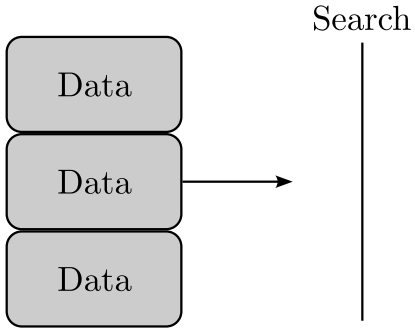


Figure 3.2: diagram of Hyperscan vectored mode

Stream mode

Stream mode allows pattern-matching across multiple blocks of data. The first call is function called `hs_open_stream()` which creates a Hyperscan stream ready to be filled with data and allocates memory for the stream state, which is used to store information

between scan calls. Then any number of `hs_scan_stream()` can occur, each of these calls writes the data to the stream and returns all new matches. The stream is closed by calling `hs_close_stream()`, which finishes stream scanning and releases all allocated memory needed to maintain the stream state.

This mode is useful when all data are not available at once and come at time intervals, this allows one to scan part of the payload and wait for the rest to come, thus distributing the load over time. However, if all data are available at once, block or vectored mode, depending on the case, should be used, since each call of `hs_scan_stream()` has to manage the state of Hyperscan, which is unnecessary if the scanning can be done in one scan call, therefore no additional memory except for scratch space is needed.

As shown in Figure 3.3 the data blocks are supplied for scanning one by one in several distinct calls. Before the first data are searched, a stream has to be opened, and then any number of data blocks can be searched. When the stream is closed, the scanning is finished and another stream can be opened. All data blocks supplied in one stream are searched as if they were one continuous chunk of data.

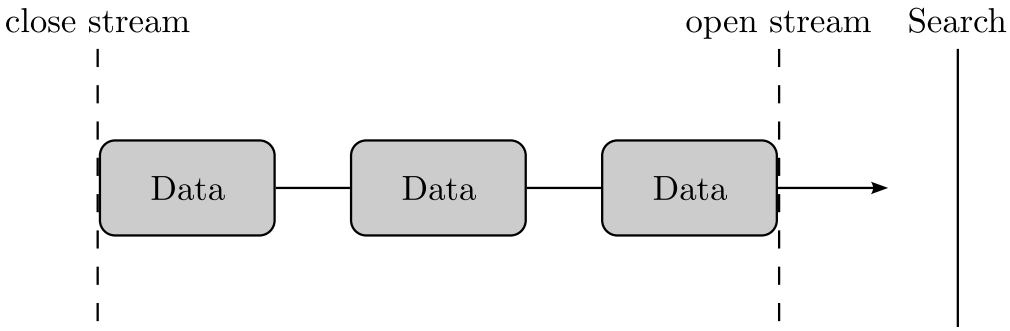


Figure 3.3: diagram of Hyperscan stream mode

3.4 Scanning

When the pattern database is compiled and the scratch space is allocated, Hyperscan can begin searching for patterns. How the scanning is executed depends on the chosen mode of Hyperscan. In block and vectored Hyperscan searches the supplied data and returns all matches. In stream mode a stream has to be opened, then data can be supplied multiple times and the stream is then closed. Note that Hyperscan can return matches even during stream closing because there can be signatures matching against the end of the stream, which cannot be evaluated until it gets closed.

Hyperscan returns all matches found in the callback given to the scan function. The callback contains the pattern ID, indexes where the pattern was found, and context also given to the scan function, which is used to distribute the information about the match, but the exact process is up to the user.

During the scanning process, Hyperscan does not allocate any memory and uses only the scratch space, eventually stream state, and reads data from the pattern database. [10]

Chapter 4

Detection metadata

This chapter contains analysis of how Suricata performs pattern-matching and explores the possibilities of moving part of the pattern-matching to the DPDK Prefilter. Based on this information, a high-level overview of the designed extension to both systems is described in Section 4.2.

The implementation of the DPDK Prefilter and infrastructure to send metadata to Suricata has opened the door for various experiments of what else can be moved away from the main application and pre-processed to increase performance and throughput. As was mentioned in Section 2.3, the detection module takes up a significant amount of system resources, and because the packets are captured, decoded, and partially matched against the rules, another part of the detection module could be moved to prefilter. The goal of this thesis is to implement such a solution and measure the impact on the performance of Suricata in various configurations and scenarios.

4.1 Current state

In Section 2.3 was described a high-level overview of how the Suricata detection module searches for patterns specified in signatures. However, the implementation of this is a little bit more complicated, and to properly design the new component, it is essential to understand the internals of the detection module. This knowledge will serve as a basis for the proposed design.

4.1.1 Detection API

Suricata uses a common API for all multi-pattern matchers (e.g. Hyperscan, Aho-Corasick). All MPMs register their API functions to a shared global table from which they can be called. The detection API is used to create an abstraction of the detection engine and provide a common interface to use them. It is mainly used to provide signatures for the detection engine and then to scan for patterns. The detection engines are used only to match all provided signatures exactly. The detection API does not have any prefilter functionality, even though some detection engines, like Hyperscan, have support for prefiltering. All the prefiltering is done in Suricata by choosing fast patterns for and the rest is used for exact matching as was described in Subsection 2.3.4.

The API uses the `MpmCtx` structure, which is used for storing the state, to communicate with the MPM engines and store metadata about the memory used by the MPM. It also contains metadata about signatures, which Suricata uses during detection. Generally

`MpmCtx` contains a group of signatures with shared properties. In Subsection 2.3.2 was mentioned that the signatures are grouped into structures called `SigGroupHead`. However, these contain other smaller groups of signatures, `MpmCtx` being the smallest one of them. The relation of these two structures is described further in this chapter.

The main functionality of the API consists of these functions:

- `InitCtx, DestroyCtx` – initialize `MpmCtx`, which is shared among all threads
- `InitThreadCtx, DestroyThreadCtx` – initialize `MpmThreadCtx`, which must be allocated for each thread using the MPM engine, for each `MpmCtx` new `MpmThreadCtx` has to be created.
- `AddPattern, AddPatternNocase` – used during initialization to fill `MpmCtx` with patterns.
- `Prepare` – this function processes all patterns added to `MpmCtx` to generate internal tables. After this call, all previously added patterns are cleared and no new patterns should be added to `MpmCtx` from this point.
- `Search` – scan for patterns previously loaded into `MpmCtx`. This function should be called only after `Prepare` is called on the `MpmCtx`.

Figure 4.1 shows the order of detect API calls from Suricata that have to be made for each `MpmCtx`. First, the internal context of each pattern-matching engine is initialized. Then all patterns are supplied to the context and the engine stores them in the internal context. After all patterns are loaded, they are processed and prepared for scanning. When the `MpmCtx` is ready, thread contexts can be initialized. Then each initialized thread can scan for patterns. During Suricata deinitialization, thread contexts are freed first and then the `MpmCtx` can be freed too.

4.1.2 Preparing signatures

During initialization, `SigGroupHead` contains all signatures for the given pair of protocol and server-side port. These signatures are divided into structures called `MpmStore` based on the protocol and direction. The divison of signatures from `SigGroupHead` can be seen in Figure 4.2. Each group head has the signatures divided into several mpm stores based on direction. As was described in Subsection 2.3.2 each `SigGroupHead` already contains signaures only with one protocol and port number group. The This step is mostly done to reuse `MpmStore` with multiple Signature Group Heads that contain the same set of rules with the same protocol and direction. Each `MpmStore` has its own `MpmCtx` initialized together with the `MpmStore`. In case `MpmStore` is reused by another `SigGroupHead` both of these structures are shared.

During the initialization of `MpmCtx`, it is also passed to the detection engine, all signatures from the `MpmStore` are added and then the detection engine processes them into its internal representation.

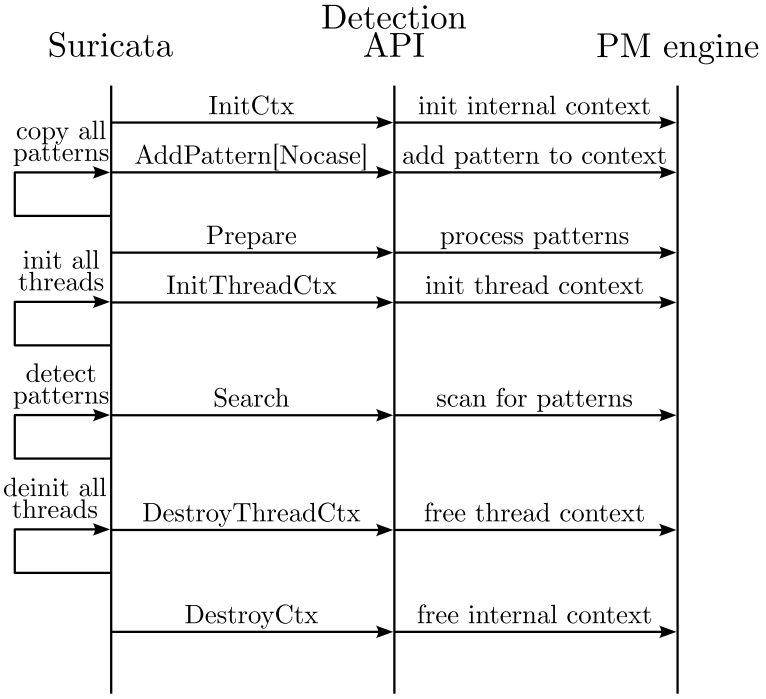


Figure 4.1: Sequence diagram of interactions with detection API

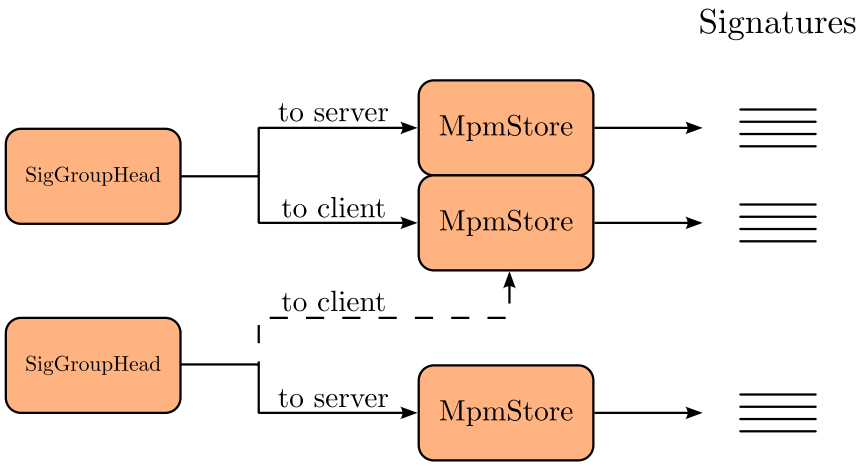


Figure 4.2: Division of signatures from `SigGroupHead`

4.1.3 Prefiltering

In Subsection 2.3.2 was described how signatures are grouped, based on protocol and server port. However, Suricata divides them further based on their type. Each `SigGroupHead` contains 4 lists of prefilter engines. The prefilter engine is a structure that contains its context, in this case `MpmCtx` and prefilter function. The 4 types of prefilter engines in Signature Group Head are:

- **Packet engines** – used to inspect specific part of the packets, like protocol layer fields. Their prefilter function first retrieves the buffer from the packet, which is then inspected using MPM.

- **Payload engines** – inspect the whole packets. The prefilter function can prefilter either a single packet payload or retrieve a stream buffer and perform pattern-matching on the reassembled data of the stream.
- **Transaction engines** – used to inspect transactions made using specific application layer protocols such as HTTP request and reply.
- **Frame engines** – first retrieve the frame buffer and match the patterns on it. Frame in Suricata refers to a generic part of network data, in this case, it is part of a UDP or TCP stream. It allows to create a window of the stream buffer and even differentiate between headers and data as described in Section 27 in [14].

The signature Group Head is assigned these prefilter engines during initialization along with their `MpmCtx` containing the patterns and the prefilter function used later during prefiltering.

During prefiltering the packet, each of the pattern engines is run, unless the packet is marked to not be inspected. Each prefilter engine picks the right part of the packet or flow to be inspected and runs the MPM with its own `MpmCtx` designed specifically for use with prefiltering.

4.2 Design

The goal of this thesis is to implement pattern-matching metadata to use in the DPDK Prefilter. To design this component, I needed to take into consideration how prefiltering and multi-pattern matching works in Suricata. In general, the component should do the following:

- Retrieve relevant signatures from Suricata during the initialization.
- Compile its own set of Hyperscan databases.
- Scan captured traffic and attach detection metadata about the patterns found.
- Suricata receives the packet with metadata and will decide whether to run pattern-matching on the packet based on the results from the DPDK Prefilter.

Figure 4.3 shows a high-level overview of the configuration of the DPDK Prefilter and Suricata. DPDK Prefilter retrieves packets from NIC as usual. After the packets are captured, they are scanned against the compiled databases using the new Hyperscan component (HS in the figure) in the prefilter, and results of the pattern-matching are stored in the packet metadata, used by Suricata to determine if it can skip detection for the packet or it should be further inspected. However, to run pattern-matching, pattern databases need to be compiled first. This ensures new Compile data component (CD in the figure), which is part of Suricata and its main function is to collect patterns in Suricata, turn them into internal representation, and share them with the prefilter.

When designing the new component, I needed to decide where in Suricata the signatures would be shared and how the metadata from the Prefilter would be used. As was explained in Subsection 4.1.3 the prefiltering consists of retrieving `SigGroupHead` for the packet and running all prefilter engines contained in this structure. When we have a look at the prefiltering engines and prerequisites to run them, we find out the frame and transaction engines both need stream reassembly and the transaction engine additionally requires

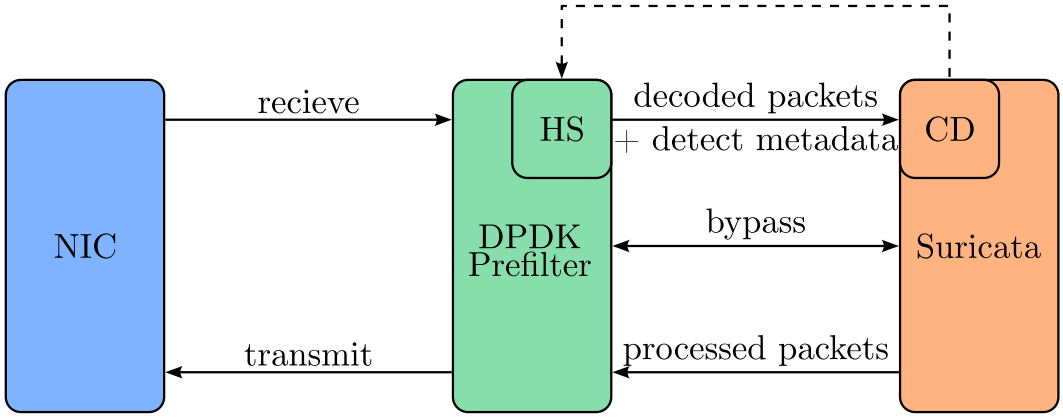


Figure 4.3: DPDK Prefilter with Hyperscan component

application layer inspection. However, this functionality is not available in the DPDK Prefilter and would require the addition of the stream reassembly module. On the other hand, the packet and payload engines both inspect only a single packet, so porting them to the Prefilter should be doable without moving too much of the functionality from Suricata.

Both prefiltering functions for payload and packet engines take the packet and `MpmCtx` to be inspected as arguments, this will be the deciding point whether to run pattern-matching or not, based on the metadata received from the prefilter. This means that the `MpmCtx` must contain information on whether it was loaded into the prefilter, and what type of prefiltering patterns it contains to compare it with the detect metadata and determine if the packet possibly contains some patterns from the context or not.

4.2.1 Detection engine

As a pattern-matching engine was chosen Hyperscan as it is the recommended option to use with Suricata if it is supported by the hardware used. Since the Prefilter should only preprocess the data to be later used by Suricata, the best option would be to use the Hyperscan prefiltering functionality. In this way, it can run a fast approximate evaluation if the scanned data contain some of the patterns.

As was mentioned in Section 2.4.2, the DPDK Prefilter already has the infrastructure to send metadata to Suricata. The results of Hyperscan prefiltering can be sent with the packets to the main application, which will help to eliminate pattern-matching on packets, where no patterns are matched. This takes advantage of the Hyperscan prefilter property, which matches the superset of the provided rules, meaning that it can produce false positives but not false negatives.

4.2.2 Signature sharing

During the initialization stage, Suricata shares patterns from rules with the DPDK Prefilter. In the prefilter, these signatures are processed and compiled into the Hyperscan databases. Refer to Section 4.3 to see why there may be multiple databases.

To figure out where to retrieve the patterns from Suricata and send them to the DPDK Prefilter, it is important to understand how the patterns are initialized in Suricata. Earlier in this section, it was described that there are 2 prefilter engines with relevant patterns

to send to the DPDK Prefilter. Each of these prefilter engines has its own `MpmCtx` which will contain the information on whether it was processed by the DPDK Prefilter and whether to check for packet metadata to possibly skip engine execution. The patterns in the `MpmCtx` are available only during initialization from `MpmStore` as was described in Subsection 4.1.2 before they are sent to the detection API, where they are processed to internal representation and cannot be modified further.

Figure 4.4 shows how the metadata are set in the DPDK Prefilter. The packet is matched against the pattern database, if any pattern is found, the match callback sets detection metadata to mark the packet for further inspection in Suricata.

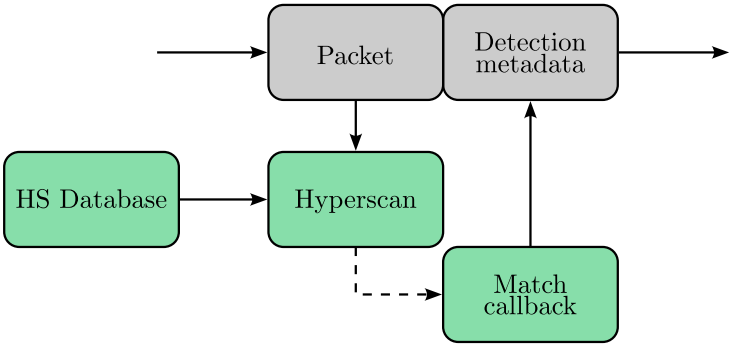


Figure 4.4: Searching packets in DPDK Prefilter

4.3 Metadata variants

To demonstrate the usefulness of adding pattern-matching metadata, 4 variants of metadata were designed. They all share the basic structure but differ in which signature metadata are enabled and/or how are patterns compiled.

- **Joined metadata** – in this variant all patterns are combined into one Hyperscan database, which will be compared to incoming traffic. Also in this variant, there is no difference between payload and packet metadata, so either both prefilter engine types will be run in Suricata or none of them.
- **Separate metadata** – in this case each prefilter engine, packet and payload, have its own pattern database and the packets are scanned against both of them. Matches from each one of them set metadata about which type was matched, this way Suricata can decide which prefilter engine to run and which can be skipped.
- **Payload metadata, Packet metadata** – these variants mainly aim to measure the impact of each prefilter engine on performance and will have either payload or packet metadata enabled.

Chapter 5

Implementation and performance testing

This chapter describes the changes made to the current implementation to incorporate the designed components to both Suricata and DPDK Prefilter. The following sections focus on important extensions made to Suricata and DPDK Prefilter and how it works in the context of current implementation.

5.1 Implementation

This section focuses on implementation of the detection metadata, more specifically Subsection 5.1.1 describes where in Suricata patterns are obtained and how the compile data exchange process works. Subsection 5.1.2 explains how the DPDK Prefilter responds to signal to initialize the Hyperscan component, and finally Subsection 5.1.3 sums up how the Hyperscan component is used to search for patterns and set metadata for Suricata.

5.1.1 Loading patterns to prefilter

Suricata processes rules before the DPDK Prefilter is attached and before the DPDK is initialized in Suricata itself, so a shared DPDK memory zone cannot be created to share rule at the time they are processed. DPDK IPC messages cannot be used either, because the DPDK has not connected the two applications. This means a different approach to sharing signatures had to be taken.

In Subsection 4.2.2 was mentioned that the patterns will be retrieved from `MpmCtx`. However, in this part of pattern preparation in Suricata, there is no way to tell which prefilter engine the `MpmCtx` belongs to, so it has to be distributed from a different part of the code where this information is available. A convenient way is storing the information about the type of prefilter engine in `MpmStore` as it is coupled with the `MpmCtx` and exists before it is created. In fact, the initialization of the `MpmStore` depends on the prefilter engine with which it is intended to be used, so this value can be set there.

When the `MpmCtx` is ready to be passed to the detection API to process its patterns, it can be sent to the DPDK Prefilter. However, from the design of the component, all patterns for one prefilter engine type should be combined into one pattern database, and this way we only get one `MpmCtx` patterns at a time. That is why there was implemented a module in Suricata that temporarily stores and combines all provided patterns and processes them into an internal representation called compile data. In this way, all patterns are temporar-

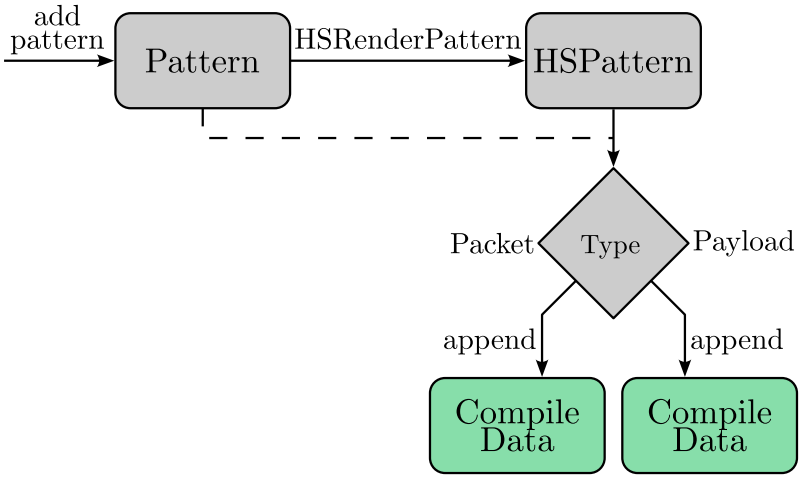


Figure 5.1: Adding patterns to compile data in Suricata

ily stored in the new module, and only after all detection module initialization is done in Suricata, the compile data can be sent to the DPDK Prefilter.

In Figure 5.1 we can see how patterns are added to compile data. A pattern is passed along with `MpmCtx` type it belongs to. The pattern itself is converted to Hyperscan pattern and appended to either packet or payload compile data. The compile data update their metadata like number of patterns or memory size and converts Suricata detection flags to Hyperscan flags.

Another solution to this would be sending the patterns or processed compile data as they are received and letting DPDK Prefilter deal with managing them. However, at the time the patterns are prepared in Suricata, DPDK is not initialized yet and the Prefilter is not attached to Suricata, meaning it could not be used to share the data. That is why the compile data are sent to the DPDK Prefilter during DPDK initialization in Suricata after the Prefilter is attached.

The whole process of sharing patterns can be seen in the sequence diagram in Figure 5.2. It is important to mention that the Compile data module is part of Suricata, DPDK enables inter process communication and memory sharing between Suricata and DPDK Prefilter, and Hyperscan is, in this case, module in the DPDK Prefilter. When Suricata initializes the patterns in the detection module, before they are passed to the detection API, it calls `AddCompileData` with a `MpmCtx` filled with patterns and the type of prefiltering engine the `MpmCtx` will become part of. Compile data component processes the patterns into the compile data structure. After Suricata finishes the initialization of the detection module, it tells the Compile data component to share them with DPDK Prefilter by calling `DpdkIpcBuildHsDb`. This function allocates memzone in shared memory space using DPDK and moves all compile data to this memzone. When the shared memory is initialized, it sends synchronous request to the DPDK Prefilter, which reads the shared memory and uses the Hyperscan to compile all pattern databases. After the initialization is done, DPDK Prefilter sends a reply to Suricata, which then frees the shared memzone, and with this the pattern sharing is done.

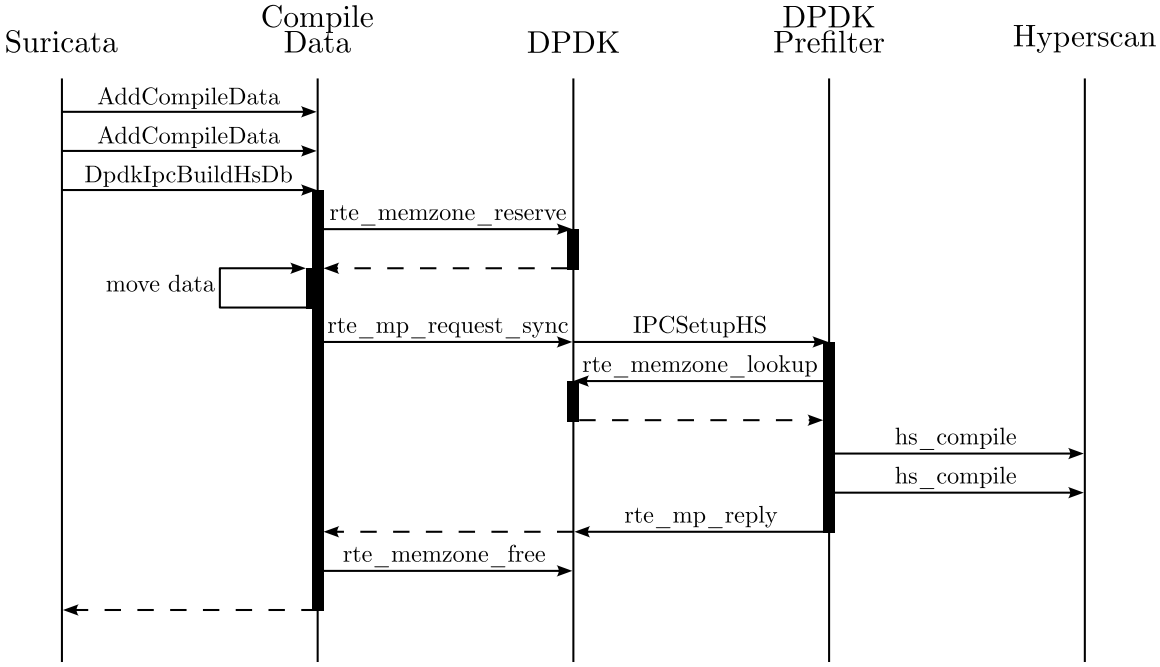


Figure 5.2: Sequence diagram of sharing compile data with DPDK Prefilter

5.1.2 Prefilter initialization

For the needs of this extension, thread states of the DPDK Prefilter, explained in Subsection 2.4.2, have been expanded with the following states:

- `LCORE_HS_DB_INIT` – compile data from Suricata are ready in shared memory and can be compiled into pattern databases
- `LCORE_HS_DB_DONE` – indicates the compiled Hyperscan databases are ready
- `LCORE_SCRATCH_INIT` – allocate scratch space for all compiled databases
- `LCORE_SCRATCH_DONE` – thread has allocated scratch space

An IPC action was added to the Prefilter to handle the Hyperscan synchronization messages from Suricata called `IPC_ACTION_HYPERSCAN_SETUP`.

When the DPDK Prefilter receives an IPC request to initialize Hyperscan, one thread is chosen to retrieve the compile data from shared memory, process it, and compile it into databases. Because we want to know only if some pattern from the database is present in the packet or not, Hyperscan can stop scanning after the first match. To enable this functionality, all pattern IDs have to be set to the same number and each of these rules must have flag `HS_FLAG_SINGLEMATCH` enabled, Hyperscan flags were discussed in Section 3.1. After this step, all threads are set to state `LCORE_SCRATCH_INIT` to allocate scratch space for the compiled databases.

The scratch space is created only by one thread, which calls `hs_compile` on all created databases to ensure that the scratch space is large enough to be used with all the compiled Hyperscan databases. When the first thread creates and allocates the scratch space, it shares it with other threads that copy the created scratch space using `hs_clone_scratch`. After the thread has allocated its own scratch space, it sets its state to `LCORE_SCRATCH_DONE` to indicate that it is ready to scan for patterns.

After the Hyperscan database compilation is done and scratch space is allocated for all threads, an IPC reply is sent to Suricata with a return code in case the compilation or any part of the preceding DPDK Prefilter initialization is not successful. At this point, all threads are ready to receive packets and perform pattern-matching on them.

5.1.3 Pattern matching and use of metadata

When everything is prepared and the DPDK Prefilter starts receiving data, each packet is scanned against the Hyperscan databases. In case there are any matches, the match callback sets metadata about the type of database against which the packet was scanned. Then Hyperscan stops scanning because of the singlematch flags set during database compilation.

When Suricata receives the packet and is about to run the prefilter engines, it checks if any match was found for the prefilter engine in the DPDK Prefilter. In that case, it runs the prefilter engine to confirm whether some pattern is actually present or if it was a false positive. In the other case, when no pattern was found in the DPDK Prefilter, we can safely assume there are no patterns present, and execution of the prefilter engine can be skipped. The rest of the detection continues as described in Subsection 2.3.5.

5.2 Performance testing

To test the performance of the system with the new changes, Trex[1] traffic generator was used to replay traffic from pcap files, Trex is described in the next section. The configuration of Trex, DPDK Prefilter, and Suricata is shown in Figure 5.3. Traffic from Trex is sent to one NIC, which is connected to the second one, this way DPDK Prefilter reads the traffic directly from hardware to make the testing as close to real deployment as possible. DPDK Prefilter and Suricata were deployed in IDS mode, meaning the packets captured by the Prefilter are sent to Suricata, where they are inspected and dropped, opposed to IPS where they would be sent back to DPDK Prefilter and NIC. Additionally, the server used to run the tests has 2 NUMA nodes, one was used for Trex and the other for Suricata and DPDK Prefilter.

For measuring, the main 2 metrics are packet drop rate and throughput. Throughput was measured directly in Trex, which measures the real rate of generated traffic. The drop rate was measured between the DPDK Prefilter and Suricata to measure the impact of the implemented changes on Suricata.

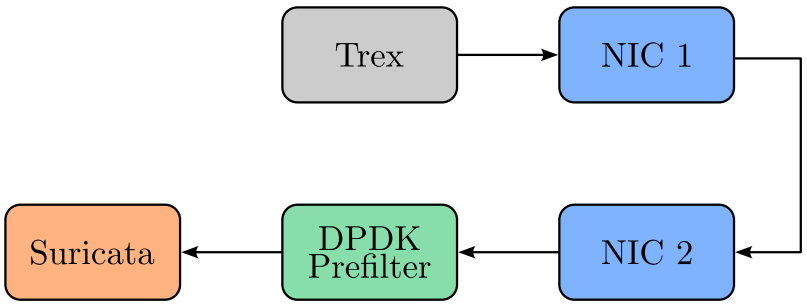


Figure 5.3: Testing configuration of Trex and Suricata with DPDK Prefilter

Environment:

- **OS** – Oracle Linux Server 8.8

[1] https://trex-tgn.cisco.com/

- **CPU** – Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, 2 sockets, 6 cores per socket, 2 threads per core, 2 NUMA nodes
- **RAM** – 64 GB
- **NIC**
  - Intel Corporation 82599ES 10-Gigabit SFI/SFP+ Network Connection (rev 01) for Trex
  - Intel Corporation Ethernet Controller X710 for 10GbE SFP+ (rev 02) for Suricata and DPDK Prefilter
- **Suricata** – 8.0.0-dev, DPDK Prefilter branch[2] with implemented changes
- **Trex** – v2.83

5.2.1 Trex

Trex is an open-source tool for generating traffic, developed by Cisco. It uses the DPDK library for improved performance. It allows not only to generate network traffic, but also to capture and analyze it. Trex supports 2 modes – statefull and stateless.

In statefull mode, Trex works on 2 interfaces, where one acts as client side and the other as server side, these communicate with each other to emulate connection between clients and servers. It allows to emulate L7 applications, provides better control of flows, offers per-flow statistics, and other advanced features. However, it does not support all packet types, needs each flow separated in different pcap file and special configuration for each flow.

Stateless mode, on the other hand, supports replaying of full pcap files even with multiple flows, has simpler configuration, and supports all kinds of packets. This comes at a cost of only per-interface statistics, little control of separate flows, and it does not compare outcoming and incoming traffic to emulate communication between server and client.

Trex provides interactive python API for both modes, which was used for automating tests and collecting statistics, the python script can be found in the attachment. The stateless mode of Trex was used for all tests mainly because it supports replaying pcap files with all kinds of packets without the need to separate flows. [6]

5.2.2 Testing scenarios

To measure the performance of the implemented changes, all versions of the metadata, described in Section 4.3, were tested in different configurations and with different variants of traffic and compare all of them to a version without metadata. To better test the impact on Suricata, the DPDK Prefilter is set up the same for each test and performs pattern matching on all patterns from Suricata. The only difference between the scenarios is the type of metadata sent to Suricata. E.g. in a test with only packet metadata, patterns for payload metadata are also searched, but only packet metadata are set. This way there is always the same configuration of the DPDK Prefilter and allows to isolate Suricata and measure its performance.

The tests were done in two configurations, first one had 1 worker thread in each DPDK Prefilter and Suricata. The second has 1 worker thread in DPDK Prefilter and 4

[2] https://github.com/lukashino/suricata/tree/feat/5203-primary-app-v14

worker threads in Suricata, different configurations did not show significant performance gain and the results were best visible in these two. For all testing scenarios, ruleset `emerging-all.rules`[3] (downloaded on April 9 2024) by Proofpoint Inc was used.

All tests were conducted using pcap `Thursday-WorkingHours.pcap`[4] from the Intrusion detection evaluation dataset (CIC-IDS2017)[5]. It contains traffic resembling real-world data with some common attacks. However, when trying to analyze this traffic, it has shown to give inconsistent results, which are caused by some high-rate parts of the pcap, e.g. average throughput was around 600 Mb/s, but it reached maximum of 1.7 Gb/s. It would be difficult to measure the real maximum throughput with Trex and would not give accurate results. Instead, the pcap file was normalized and 2 variants of the traffic were created, one with constant bitrate of 1 Mb/s and one with constant packet rate of 100 p/s. More information about the pcap files used can be found in Appendix A. The normalization of pcap files was done using python scripts and the `dpkt`[6] library.

So to sum up, each configuration was tested for all metadata variants and one variant with no metadata and all of this on two variants of traffic, one with constant bitrate and one with constant packet rate.

In each test, throughput was increased by a constant amount, until the packet drop rate exceeded 2 % when the test was stopped. For each step, real transfer speed was retrieved from Trex, that is why there are small deviations in the data. The drop rate was calculated from the DPDK Prefilter statistics as the ratio of the number of packet enqueue attempts to Suricata and the number of successfully enqueued packets to Suricata. In this way, the drop rate represents the ratio of packets that Suricata could not process because all packet queues were full. In each test run, meaning for each speed increment, both Suricata and DPDK Prefilter were started from scratch and the Trex NIC interface was restarted, this way the same conditions were ensured for each test run.

5.2.3 Constant bitrate tests

This set of tests used pcap normalized to bitrate of 1 Mb/s as a source of traffic. The packet rate peaked at 230 kp/s at throughput of 1 Gb/s and at 459 kp/s at 2 Gb/s.

In Figure 5.4 we can see the results of tests for configuration with 1 worker for each application. The throughput was gradually increased by 50 Mb/s. In this test scenario, all variants of metadata performed similarly to each other and started dropping packets in throughput over 1 Gb/s, then the drop rate remained under 2 % until 1.2 Gb/s, where it went over 4 %.

In Figure 5.5 are shown results of tests performed with the configuration of 1 DPDK Prefilter worker and 4 Suricata workers. The bitrate was increased by 100 Mb/s in each step. Similarly to the previous scenario, all variants performed fairly similar, but some differences can be seen here. The metadata variant with all the patterns joined in one database remained at 0 % drop rate up to 1.7 Gb/s, which is a small improvement. But better results can be seen for the variant with only payload metadata which had 0 % drop rate up to 1.8 Gb/s and even after that the drop rate was lower than for other variants.

[3] https://rules.emergingthreats.net/open/suricata-5.0/emerging-all.rules

[4] http://205.174.165.80/CICDataset/CIC-IDS-2017/Dataset/CIC-IDS-2017/PCAPs/Thursday-WorkingHours.pcap

[5] https://www.unb.ca/cic/datasets/ids-2017.html

[6] https://dpkt.readthedocs.io/en/latest/

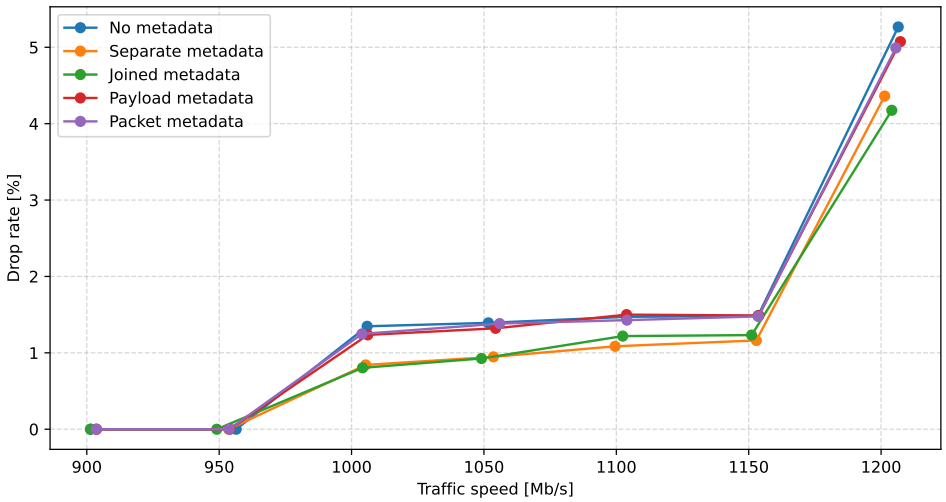


Figure 5.4: Drop rate with increasing bitrate, 1 DPDK Prefilter worker, 1 Suricata worker

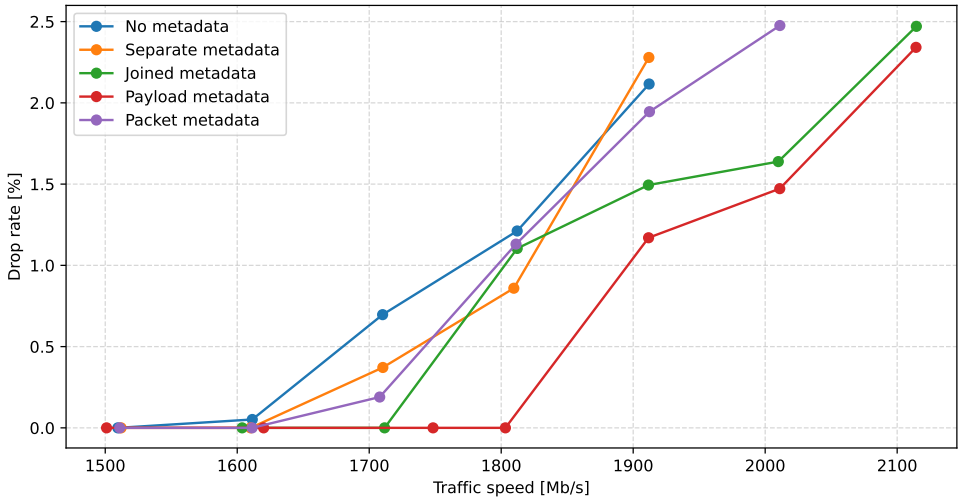


Figure 5.5: Drop rate with increasing bitrate, 1 DPDK Prefilter worker, 4 Suricata workers

In Table 5.1 is the comparison of results from the previous 2 test scenarios containing the maximum measured speed at which no drop rate was measured for each metadata variant and the comparison with the no metadata variant. The configuration with 1 Suricata worker performed similarly for all metadata variants with insignificant deviations. In the second configuration, with 4 Suricata workers, we can see 13.38 % improvement for variant with joined metadata and 19.40 % improvement for the payload metadata variant.

The reason why the configuration with 1 Suricata thread did not show any improvement could be that the performance gain is not high enough to enable Suricata to process a greater number of smaller packets. The most critical parts of the test are sections of the pcap with high packet rates. Because Suricata analyzes packets one by one, to process more packets

with one thread the analysis of one packet has to be finished before the arrival of another. There could be a situation in which the packets are processed slightly faster, but not fast enough before the next packet arrives. However, with more threads, the packet processing is better distributed among the threads and even though one thread does not process the packet fast enough, there is a better chance some other thread will be available thanks to the performance gain.

	1 Suricata worker		4 Suricata workers	
	Speed [Mb/s]	Improvement [%]	Speed [Mb/s]	Improvement [%]
No metadata	956	—	1510	—
Separate metadata	954	-0.21	1610	6.62
Joined metadata	949	-0.73	1712	13.38
Payload metadata	954	-0.21	1803	19.40
Packet metadata	953	-0.31	1611	6.69

Table 5.1: Maximum measured throughput with no drops using pcap normalized to constant bitrate

5.2.4 Constant packet rate tests

This set of tests used pcap normalized to a packet rate of 100 p/s as a source of traffic. The maximum bitrate was 1.67 Gb/s at throughput of 150kp/s and 2.28 Gb/s at 200kp/s.

Figure 5.6 shows the results of the tests using a configuration with 1 worker thread for each Suricata and DPDK Prefilter where the speed was increased by 5 kp/s in each step. The variants with no metadata, separate metadata, and packet metadata performed fairly similarly to each other, where the maximum speed with no drops was measured around 130 kp/s. However, we can see some difference for joined metadata and payload metadata, which improved to 145 kp/s with no packet drops.

In Figure 5.7 are shown results for the second configuration with 1 DPDK Prefilter worker and 4 Suricata workers with speed steps of 10 kp/s. We can see that the trend stays the same and that the joined metadata and payload metadata outperformed other types at speeds over 165 kp/s with no drop rate, compared to 140 kp/s for the other variants. Even then the drop rate stays under 1 % up to 240 kp/s after which it increases significantly.

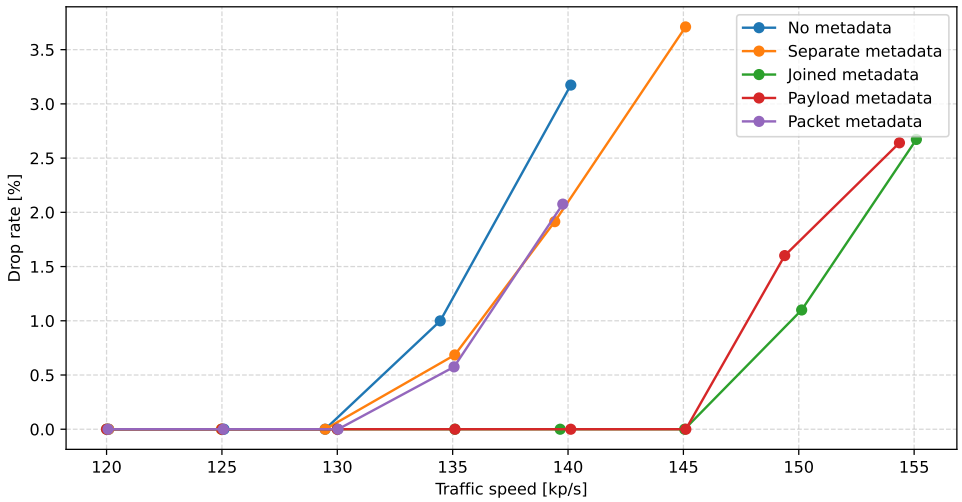


Figure 5.6: Drop rate with increasing packet rate, 1 DPDK Prefilter worker, 1 Suricata worker

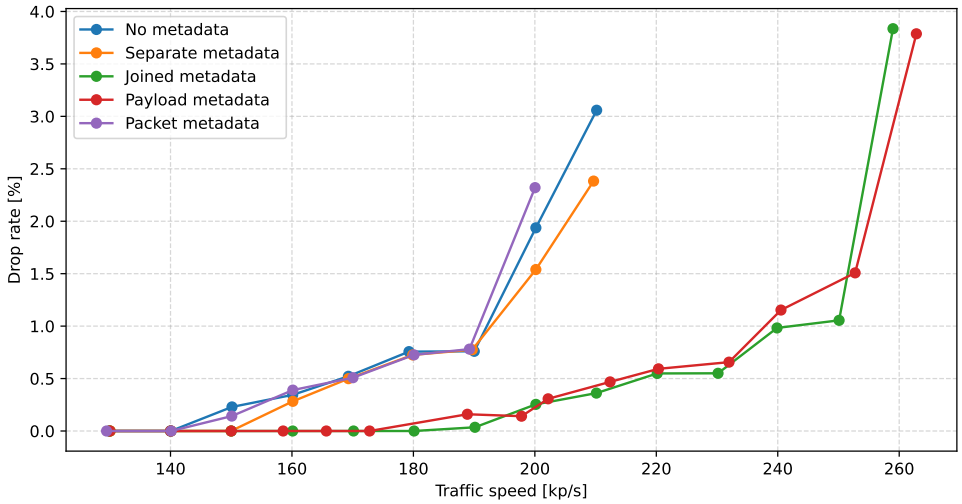


Figure 5.7: Drop rate with increasing packet rate, 1 DPDK Prefilter worker, 4 Suricata worker

Table 5.2 contains a summary of tests with a constant packet rate. Unlike the scenarios with constant bitrate, here the differences between the metadata variants are more apparent, with both payload and joined metadata having improved by 12.40 % in configuration with 1 Suricata worker. The payload metadata achieved a 22.86 % improvement in the second configuration and even better performance was measured for the joined metadata with an improvement of 28.57 %. In both configurations, joined metadata and payload metadata performed the best with the others not having much impact on the performance.

	1 Suricata worker		4 Suricata workers	
	Speed [kp/s]	Improvement [%]	Speed [kp/s]	Improvement [%]
No metadata	129	—	140	—
Separate metadata	129	0.00	150	7.14
Joined metadata	145	12.40	180	28.57
Payload metadata	145	12.40	172	22.86
Packet metadata	130	0.77	140	0.00

Table 5.2: Maximum measured throughput with no drops using pcap normalized to constant packet rate

5.2.5 Results summary

We can see that the best-performing metadata variants are the joined metadata and the payload metadata. This is because the payload metadata variant performs scanning on the same buffer as in Suricata, where the buffer is the whole packet. Opposed to the packet metadata variant, which scans the whole packet in the DPDK Prefilter, but Suricata can scan only part of the packet, which leads to more false positives in the DPDK Prefilter, meaning that Suricata has to still inspect the packet in most cases.

The results have shown that the implemented changes perform better at constant packet rates, rather than constant bitrates. This is because the traffic is processed packet by packet and it is faster to process one large packet rather than many smaller ones, where the overhead is much greater. The pattern-matching has to be performed separately for each packet, which is costly in terms of resources. The results have also shown that the system can better withstand peaks in bitrate if the packet rate stays the same than with increased packet rates with constant bitrate.

Overall, the improvement was greater with more Suricata workers, most likely because of better load distribution between threads. The performance gain in all threads sums up and creates space for more packets to be processed. The new component is not always beneficial, as the results for 1 worker show. The one thread does not provide enough flexibility for the improvements to be that noticeable.

Chapter 6

Conclusion

The goal of the theoretical part of this thesis was to study and analyze how pattern-matching works in Suricata. The main objective was to design a new component that adds detection metadata to packets to potentially improve the performance of Suricata. The new component was implemented into the DPDK Prefilter to simulate specialized hardware.

To design the detection metadata, I needed to study Suricata, DPDK library, DPDK Prefilter, and Hyperscan, which I have chosen as a pattern-matching engine. I have described the core functionality of Suricata, its building blocks called detection modules, different runmodes and configurations. I have also described how rules are written and used to detect potential attacks and other malicious traffic. More attention was paid to the detection module, pattern management, and prefiltering. Further, I have described DPDK Prefilter, its role and communication with Suricata. The primary sources of information were documentation and source code analysis, used mainly for the detection module and the DPDK Prefilter. I also have described Hyperscan, its different modes and capabilities, and the scanning process.

Using the gained knowledge, I designed a component that moves part of pattern-matching from Suricata to DPDK Prefilter, where it approximately matches patterns and adds metadata about the detection to the packet. Suricata then uses this metadata to skip pattern-matching if no patterns were found in the prefilter, in other cases it checks the packet as it would normally. The last part of the thesis focused on measuring the impact of implemented changes on Suricata.

The performance tests were conducted using the CIC2017 IDS evaluation dataset and the Trex traffic generator. Multiple variants of the detection metadata were tested to find the best configuration. All variants of metadata were tested with two configurations of Suricata and two different types of traffic – one with constant bitrate and one with constant packet rate. Traffic throughput was increased by a fixed amount in each test run until the packet drop rate exceeded 2 %. The primary metric of improvement was the maximum throughput of the application with no packet drops.

The test results have shown that some of the designed metadata variants improved the performance, however, it is dependent on the characteristics of the incoming traffic. On traffic with constant bitrate and configuration with 4 Suricata workers, there were two variants of the metadata that performed better than others: payload metadata and joined metadata. The variant with payload metadata improved throughput by more than 19 % from 1.5 GB to 1.8 GB. The joined metadata variant registered an improvement of more than 13 % at a maximum throughput of 1.7 GB. More significant differences were measured for traffic with constant packet rate, where the same two metadata variants showed a gain

in throughput. The payload metadata improved maximum throughput by almost 23 % from 140 kp/s to 172 kp/s. The joined metadata variant improved even more – by 28.57 % with throughput up to 180 kp/s.

The results open the door for other work as well. I decided to combine all prefilter engine patterns for scanning in the DPDK Prefilter. For payload metadata, this seems to be a good solution. However, for packet metadata, it would be worth trying to split them into more groups, which could lead to better improvement.

Another experiment that includes the packet metadata could be modifying the patterns based on their prefiltering functions to achieve the same functionality in Hyperscan. However, there is a chance that it would not be possible because it depends on the compatibility of operations performed in the prefiltering functions and Hyperscan capabilities. Another solution would be to move all the packet prefilter engines to the DPDK Prefilter and run them the same way as Suricata.

Another use of the results would be to implement the detection metadata in hardware, either as hardware offloading or designing a specialized pattern-matching component with the use of the detection metadata.

Bibliography

[1] Alicherry, M., Muthuprasanna, M. and Kumar, V. High Speed Pattern Matching for Network IDS/IPS. In: *Proceedings of the 2006 IEEE International Conference on Network Protocols*. 2006, p. 187–196. DOI: 10.1109/ICNP.2006.320212.

[2] Amazon Web Services. *Amazon EC2 with Suricata on AWS* [online]. [cit. 2024-01-28]. Available at: https://aws.amazon.com/solutions/implementations/amazon-ec2-suricata/.

[3] Black Hat. Portrait of an Imminent Cyberthreat. *2017 Black Hat Attendee Survey*. July 2017. Available at: https://www.blackhat.com/docs/us-17/2017-Black-Hat-Attendee-Survey.pdf.

[4] Borg, A., Chen, J. and Jouppi, N. A Simulation Based Study of TLB Performance. In: *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*. 1992, p. 114–123 [cit. 2024-04-21]. DOI: 10.1109/ISCA.1992.753309.

[5] Check Point Software Technologies Ltd. *Intrusion Detection System (IDS) Vs Intrusion Prevention System (IPS)* [online]. [cit. 2024-01-14]. Available at: https://www.checkpoint.com/cyber-hub/network-security/what-is-an-intrusion-detection-system-ids/ids-vs-ips.

[6] Cisco Systems. *TRex Documentation* [online]. [cit. 2024-04-29]. Available at: https://trex-tgn.cisco.com/trex/doc/.

[7] Day, D. and Burns, B. A performance analysis of snort and suricata network intrusion detection and prevention engines. In: *Fifth international conference on digital society, Gosier, Guadeloupe*. 2011, p. 187–192.

[8] Intel Corporation. *Programer's Guide* [online]. [cit. 2024-04-19]. Available at: https://doc.dpdk.org/guides/prog_guide/index.html.

[9] Intel Corporation. Accelerating Suricata Throughput Performance Using Hyperscan Pattern-Matching Software. 2017. Available at: https://www.intel.com.tw/content/dam/www/public/us/en/documents/solution-briefs/hyperscan-scalability-solution-brief.pdf.

[10] Intel Corporation. *Hyperscan 5.4 Developer's Reference Guide* [online]. February 2023 [cit. 2024-01-29]. Available at: https://intel.github.io/hyperscan/dev-reference/.

[11] Kurose, J. F. and Ross, K. W. *Computer Networking: A Top-Down Approach*. 8th ed. Boston, MA: Pearson, 2020. ISBN 978-9356061316.

[12] Langdale, G. *Hyperscan: How We Match Regular Expressions* [online]. [cit. 2024-01-29]. Available at: https://www.intel.com/content/www/us/en/collections/libraries/hyperscan/regular-expression-match.html.

[13] Mogul, J. C. and Ramakrishnan, K. K. Eliminating Receive Livelock in an Interrupt-Driven Kernel. In: *Proceedings of the USENIX 1996 Annual Technical Conference*. San Diego, California: [b.n.], January 1996.

[14] OISF. *Suricata User Guide* [online]. [cit. 2024-01-17]. Available at: https://docs.suricata.io/en/suricata-7.0.2/.

[15] Peng, Z., Wang, Y., Hu, L. and Jing, Z. An Efficient Adaptive Architecture for Multi-pattern Matching. *Journal of Computer*. 2018, vol. 29. DOI: 10.3966/199115992018102905010.

[16] Red Hat, Inc. *Performance Tuning Guide* [online]. February 2023 [cit. 2024-01-29]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/index.

[17] Sharifi, A., Zad, F., Farokhmanesh, F., Noorollahi, A. and Sharif, J. An Overview of Intrusion Detection and Prevention Systems (IDPS) and Security Issues. *IOSR Journal of Computer Engineering*. january 2014, vol. 16, p. 47–52. DOI: 10.9790/0661-16114752.

[18] Shchapaniak, A. *Using Metadata to Optimize the Suricata IDS/IPS*. Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology.

[19] Vailshery, L. S. *Internet of Things (IoT) and non-IoT active device connections worldwide from 2010 to 2025* [online]. [cit. 2024-01-14]. Available at: https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/.

[20] 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/.

[21] Wang, X. *Introduction to Hyperscan* [online]. September 2017 [cit. 2024-01-29]. Available at: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-hyperscan.html.

[22] Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G. et al. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, February 2019, p. 631–648. ISBN 978-1-931971-49-2. Available at: https://www.usenix.org/conference/nsdi19/presentation/wang-xiang.

[23] Wong, K., Dillabaugh, C., Seddigh, N. and Nandy, B. Enhancing Suricata intrusion detection system for cyber security in SCADA networks. In: *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*. 2017. DOI: 10.1109/CCECE.2017.7946818.

Appendix A

Testing pcap files

Information about used pcap files for testing, generated with `capinfos`[1] utility.

```
File type: Wireshark/tcpdump/... - pcap
File encapsulation: Ethernet
Number of packets: 9,316 k
File size: 8,085 MB
Data size: 7,935 MB
Data byte rate: 274 kBps
Data bit rate: 2,193 kbps
Average packet size: 851.81 bytes
Average packet rate: 319 packets/s
Interface #0 info:
                    Encapsulation = Ethernet (1 - ether)
                    Number of packets = 9316593
```

Listing A.1: Original pcap file

```
File type: Wireshark/tcpdump/... - pcap
File encapsulation: Ethernet
Number of packets: 9,316 k
File size: 8,085 MB
Data size: 7,935 MB
Data byte rate: 125 kBps
Data bit rate: 1,000 kbps
Average packet size: 851.81 bytes
Average packet rate: 146 packets/s
Interface #0 info:
                    Encapsulation = Ethernet (1 - ether)
                    Number of packets = 9316593
```

Listing A.2: Testing pcap file normalized to 1 Mb/s

[1] https://www.wireshark.org/docs/man-pages/capinfos.html

```
File type: Wireshark/tcpdump/... - pcap
File encapsulation: Ethernet
Number of packets: 9,316 k
File size: 8,085 MB
Data size: 7,935 MB
Data byte rate: 85 kBps
Data bit rate: 681 kbps
Average packet size: 851.81 bytes
Average packet rate: 100 packets/s
Interface #0 info:
                    Encapsulation = Ethernet (1 - ether)
                    Number of packets = 9316593
```

Listing A.3: Testing pcap file normalized to 100p/s