BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

OPTIMIZATION OF THE SURICATA IDS/IPS

OPTIMALIZACE IDS/IPS SYSTÉMU SURICATA

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. LUKÁŠ ŠIŠMIŠ**
AUTOR PRÁCE

SUPERVISOR **Ing. PAVOL KORČEK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Computer Systems (DCSY) Academic year 2020/2021

Master's Thesis Specification

Student: **Šišmiš Lukáš, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Application Development
Title: **Optimization of the Suricata IDS/IPS**
Category: Networking
Assignment:

1. Study the Intrusion Detection System (IDS) / Intrusion Prevention System (IPS) and focus on the open-source IDS/IPS software Suricata.
2. Analyze the performance parameters of the Suricata under simulated operation with selected rules.
3. Design the optimization of Suricata parameters in order to increase the performance (throughput) on the given hardware.
4. Implement the proposed optimizations.
5. Test the created implementation.
6. Discuss achieved results and the possibilities of further improvements.

Recommended literature:

- According to the instructions.

Requirements for the semestral defence:

- Points 1 to 3 of the assignment.

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

Supervisor: **Korček Pavol, Ing., Ph.D.**
Consultant: Kučera Jan, Ing., UPSY FIT VUT
Head of Department: Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work: November 1, 2020
Submission deadline: May 19, 2021
Approval date: February 4, 2021

Abstract

The recent rapid increase of network traffic bandwidth has sprung new challenges in securing the network. It is vital to keep monitoring the traffic to securely identify threats in the network. Systems like IDS (intrusion detection systems) alert us about events in the analyzed traffic. Suricata, as one of the available IDS, was chosen for this thesis. The ultimate goal of the thesis is to tune settings of AF_PACKET capture interface to reach the best performance possible and then suggest and implement an optimization for Suricata. Results of the AF_PACKET should be used as a baseline for comparison with future improvements. Optimization is based on implementing a new capture interface to Suricata that is based on Data Plane Development Kit (DPDK). DPDK helps to accelerate packet capture and this implies that it might improve the performance of Suricata. Results that compare AF_PACKET and DPDK performance are evaluated at the end of this master thesis.

Abstrakt

V dnešnom svete zrýchľujúcej sa sieťovej prevádzky je potrebné držať krok v jej monitorovaní. Dostatočný prehľad o dianí v sieti dokáže zabrániť rozličným útokom na ciele nachádzajúce sa v nej. S tým nám pomáhajú systémy IDS, ktoré upozorňujú na udalosti nájdené v analyzovanej prevádzke. Pre túto prácu bol vybraný systém Suricata. Cieľom práce je vyladiť nastavenia systému Suricata s rozhraním AF_PACKET pre optimálnu výkonnosť a následne navrhnúť a implementovať optimalizáciu Suricaty. Výsledky z meraní AF_PACKET majú slúžiť ako základ pre porovnanie s navrhnutým vylepšením. Navrhovaná optimalizácia implementuje nové rozhranie založené na projekte Data Plane Development Kit (DPDK). DPDK je schopné akcelerovať príjem paketov a preto sa predpokladá, že zvýši výkon Suricaty. Zhodnotenie výsledkov a porovnanie rozhraní AF_PACKET a DPDK je možné nájsť na konci diplomovej práce.

Keywords

Suricata, XDP, Hyperscan, Flow shunting, Bypass, AF_PACKET, PF_RING, DPDK, PCAP, Network monitoring, IDS, IPS, Network traffic detection, Suricata optimization, DPDK runmode

Kľúčové slová

Suricata, XDP, Hyperscan, Flow shunting, Bypass, AF_PACKET, PF_RING, DPDK, PCAP, Monitorovanie siete, IDS, IPS, Detekce síťového provozu, Optimalizace Suricaty, DPDK runmód

Reference

ŠIŠMIŠ, Lukáš. *Optimization of the Suricata IDS/IPS*. Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Pavol Korček, Ph.D.

Rozšírený abstrakt

Technologické výdobytky súčasnej doby nám otvárajú širokú škálu možností. Najmä oblasť IT sa posunula za posledné dekády ohromným spôsobom vpred. Počítače a technológie s nimi spojené napríklad umožňujú komunikovať ľuďom na diaľku, znižujú čas potrebný pre prenos informácie alebo zefektívňujú celkovú prácu ľudstva. Jednou z veľmi podstatných technológií, ktoré počítače na komunikáciu potrebujú, je sieť. V súčasnosti najpoužívanejšia sieť je Internet. Táto sieť existuje takmer od počiatku prvých experimentov prepojenia počítačov. Sieť sa priebežne vyvíja a v súčasnosti sa na ňu každým rokom pripája viac a viac zariadení. S ich zvyšujúcim počtom narastá aj objem a rýchlosť sieťovej prevádzky.

Ako bolo spomenuté na začiatku, nové technológie prinášajú mnoho možností. Avšak, ako sa tieto možnosti využijú závisí na úmysloch používateľov. Je preto možné, že niektorí pripojení používatelia sa rozhodnú páchať napríklad trestnú činnosť. Týmto spôsobom vznikajú určité hrozby, podobne, ako je tomu vo svete mimo informačných technológií. Políciu v tomto prípade môže substituovať administrátori jednotlivých podsietí. Na pomoc pri vykonávaní ich činnosti im slúžia rôzne nástroje. Pomocou nich sa snažia ustrážiť aktivity na sieti. Medzi tieto nástroje môže patriť napríklad firewall, antivírusové programy alebo softvér na overovania možnosti prístupu. Medzi tieto nástroje patria aj IDS/IPS systémy. Tieto programy umožňujú pozorovať prechádzajúcu sieťovú prevádzku, analyzovať ju voči vopred zadefinovaným pravidlám a v prípade podozrivého konania môžu upozorniť administrátora. IPS systémy môžu potom ešte naviac automaticky zabrániť vykonávaniu škodlivých aktivít.

Pomocou IDS/IPS nástrojov je možné odhaliť rôzne typy útokov. Môžu to byť napríklad distribuované útoky, prenos infikovaných súborov alebo phishing. Ako bolo spomenuté, na ich fungovanie je väčšinou potrebné analyzovať celú prechádzajúcu sieťovú prevádzku. Sieťová prevádzka je rozdelená do jednotlivých paketov. Tieto systémy musia každý prijatý paket dekódovať a pokúsiť sa nájsť zhodu s nejakým zadefinovaným pravidlom. S narastajúcou rýchlosťou a objemom dát to ale pre tieto systémy začína byť veľmi výpočetne náročná úloha.

Táto diplomová práca sa preto snaží navrhnúť a implementovať optimalizáciu pre systém Suricata. Je to jeden z dostupných IDS/IPS systémov. Prednosť systému Suricata spočíva najmä vo vysokej výkonnosti. Tá je, okrem iného, dosiahnutá kvalitnou multi-vláknovou architektúrou. To znamená, že môže analyzovať viacero paketov súčasne. Suricata a aj iné IDS/IPS systémy sa ale potýkajú s problémom vysokých výpočetných požiadavkov. Je snaha ich zmenšovať rôznymi optimalizáciami. V tomto prípade navrhnuté zlepšenie výkonnosti zahŕňa implementovanie a používanie nového rozhrania pre prijímanie paketov. Rozhranie už je implementované s použitím knižnice DPDK.

Na začiatku diplomovej práce je predstavenie systému Suricata. Kapitola začína predstavením jej architektúry. Následne sa pokračuje detekčnými pravidlami, ich zložením a spracovávaním. V ďalšej sekcii sa prezentujú 3 odlišné spôsoby fungovania vlákien v Suricate. To, aký spôsob sa zvolí, má vplyv na prepojenie modulov Suricaty. Potom v ďalšej rozsiahlejšej sekcii sú predstavené jednotlivé moduly slúžiace na príjem paketov. Moduly a ich teoretický princíp fungovania sú vyobrazené na diagramoch s textovým opisom. Pri subsekcii AF_PACKET sú rozobraté aj jeho možné vylepšenia v podobe eBPF filtrov a XDP. Predposledná sekcia v kapitole Suricaty rozoberá jej možné zlepšenie výkonnosti pomocou techniky bypass. To dovoľuje vynechať inšpekciu paketov pre vybrané sieťové toky. Do toku zvyčajne patria pakety, ktoré zdieľajú isté charakteristiky. Väčšinou je to definovaná ako pätica s rovnakou/opačnou zdrojovou/cieľovou IP adresou, s rovnakým/opačným portom a s rovnakým protokolom vrstvy L4 OSI modelu. Pri

spojení s XDP je možné tieto pakety vynechať už na úrovni ovládača sieťovej karty. Posledná sekcia prezentuje možné výstupy systému Suricata.

Na začiatku ďalšej kapitoly diplomovej práce sa prezentuje návrh architektúry, ktorá slúži na testovanie výkonnosti systému Suricata. Po tejto sekcii sú ďalej predstavené nástroje, ktoré sa používali pri ladení a testovaní systému. Z nich je najviac rozobratý nástroj ethtool. Na odosielanie sieťovej prevádzky slúži Replikátor. Je to nástroj na preposielanie sieťovej prevádzky s možnosťou amplifikácie až do 100 Gbps. Detaily ohľadom použitia tohto nástroja nájdeme v ďalšej sekcii. V tejto sekcii je popísaný aj testovací framework, ktorý bol navrhnutý a implementovaný pre zníženie časti manuálnej práce spojenej s testovaním Suricaty. Ten umožňoval zoradiť viacero meraní za sebou a vykonávať ich podľa vopred zadefinovaného scenára. Poslednou časťou sekcie je analýza zachytenej sieťovej prevádzky, ktorá bola používaná na testovanie. Tento súbor sa preposielal pomocou vyššie spomenutého replikátoru. Replikátor bol obsluhovaný pomocou testovacieho framework-u. Posledná sekcia kapitoly popisuje ladenie parametrov konfigurácie v systéme Suricata s rozhraním AF_PACKET. Najvyššiu dosiahnuteľnú výkonnosť takto zostaveného systému bolo potrebné získať pre následne porovnanie novej optimalizácie. Vytvorilo to základ pre ďalšie merania.

Následne, v ďalšej sekcii je ukázaný návrh zamýšľanej optimalizácie. V tejto sekcii je vysvetlené prečo sa ako nová optimalizácia zvolila implementácia DPDK rozhrania. Je tam popísaný samotný návrh riešenia. Ten je rozkreslený do viacerých diagramov, ktoré ilustrujú jednotlivé časti novej optimalizácie.

Posledná sekcia sa zaoberá implementáciou spolu s meraniami, ktoré boli vykonané. Táto sekcia postupne opisuje priebeh implementácie. Začína popisom zaradenia nového módu behu (rozhrania) do systému Suricata a jej procesu kompilácie. Ďalej opisuje počiatočnú inicializáciu a synchronizáciu samotného rozhrania a jeho vlákien. Najdôležitejšia časť rozhraní na príjem paketov v systéme Suricata je prijímací cyklus. Kód v tomto cykle je vykonávaný opakovane pričom počas behu sa snaží prevziať pakety zo sieťovej karty a poslať ich iným modulom k ďalšiemu spracovaniu. Synchronizácia vlákien sa počas ich behu v cykle nepoužíva kvôli vysokej réžií. Na konci sekcie sú popísané funkcie potrebné pre odosielanie paketov (režim IPS) a uvoľnenia paketu zo systému Suricata.

Samotný záver diplomovej práce je venovaný porovnaniu behov rozhraní DPDK a AF_PACKET v rôznych podmienkach. Najväčší dôraz je kladený na výkonnosť. Tá je založená na pomere prijatých a odoslaných paketov. Cieľom je aby počet prijatých paketov bol rovný odoslaným po čo najvyššiu rýchlosť. Testy prebiehali s rôznym počtom vlákien – s 1, 8 a 16 vláknami. Z výsledkov vyplýva, že nové DPDK rozhranie prekonáva výsledky rozhrania AF_PACKET vo všetkých testoch. Ukazuje sa, že výkonnosť Suricaty sa mierne zvýšila. To značí, že nová optimalizácia pomohla uvoľniť výpočetné prostriedky iným modulom. Zároveň miera zvýšenia výkonnosti naznačuje, že rozhranie pre príjem paketov neobmedzuje Suricatu v behu. Oveľa vyššia výkonnosť príjmu paketov v režime bez pravidiel implikuje, že detekčný modul s vysokým počtom pravidiel najviac vyťažuje výpočetné prostriedky systému Suricata. V poslednej časti sa ďalej disktuje pokračovanie v práci so systémom Suricata.

Optimization of the Suricata IDS/IPS

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Pavol Korček Ph. D. The supplementary information was provided by Mr. Ing. Jan Kučera. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Lukáš Šišmiš
May 18, 2021

Acknowledgements

I would like to express my deepest appreciation to my consultant Ing. Jan Kučera for his professional supervision and constructive feedback. I would also like to extend my sincere thanks to Ing. Pavol Korček Ph. D. for the support that he has provided me for the duration of writing this thesis.

Contents

1 Introduction **2**

2 Suricata **4**
2.1 Architecture 5
2.2 Detection rules 6
2.3 Runmodes 10
2.4 Capture options and running modes 12
2.5 Bypass and flow shunting 21
2.6 Outputs 21

3 Setting up the environment **23**
3.1 Network architecture 23
3.2 Configuration tool – ethtool 24
3.3 Testing framework 30
3.4 Performance tuning of AF_PACKET 35

4 Design of the proposed optimization **44**
4.1 Motivation 45
4.2 Analysis of the current implementation 45
4.3 Idea proposal 46

5 Implementation and benchmarks **52**
5.1 Implementation 52
5.2 Benchmarks 61

6 Conclusion and future work **70**

Bibliography **72**

A NIC settings **74**

B PCAP analysis **75**

Chapter 1

Introduction

In computer history, the ARPANET [6] is recorded as the first wide-area packet-switching network with the first connected users in 1969. As time progressed, the network grew and by 1988 it was estimated that about 60000 computers were connected. Users, mostly consisted of scientists, were excited about the new era in the computer world. However, they did not know what awaits them in November that year. Morris worm [11] was released and by combining several vulnerabilities in computer systems it was able to infect thousands of Berkeley Unix systems leading to a dysfunctioning the majority of them and partitioning the network for few days. It is recognized as the first computer threat distributed over the network and gaining attention all over the world.

The computer networks continued to grow, gaining more users and enabling faster transfer rates together with an increasing number of threats as well. People had started to focus on securing their devices and networks more and more over time. Administrators learned new ways to mitigate attacks and systems such as firewalls, antiviruses or intrusion detection systems came to light. The first concept of intrusion detection system was defined in the 1980s and was further researched and developed later that decade. Between 1984 and 1986 the first concept of such software was developed. [1]

„An intrusion detection system (IDS) is a device or software application that monitors a network or systems for malicious activity or policy violations. Any intrusion activity or violation is typically reported either to an administrator or collected centrally using a security information and event management (SIEM) system. A SIEM system combines outputs from multiple sources and uses alarm filtering techniques to distinguish malicious activity from false alarms.“ [7]

IDS can be installed at various points of a network, however, this thesis is focused on a network intrusion detection system (NIDS) rather than a host-based intrusion detection system (HIDS). The most common use case of NIDS is to monitor all traffic between the outer and inner network and raise alerts about potential threats. This is possible thanks to the analysis of packets passing through and a library of attacks or ruleset of the NIDS.

However, with rapidly increasing bandwidth worldwide, all systems and devices connected to the core networks must comply strict security and performance requirements. Otherwise, such systems can be viewed as bottlenecks. As a result, they can cause packet drops, increase in latency or may be unable to detect threats in real-time as expected. These factors are great motivation for network security research teams around the world to make sure intrusion prevention systems are not only performant enough to keep pace with an increase in network bandwidth but also to distinguish threats at a very high rate while not introducing severe latency.

This master thesis first attempts to introduce the reader to IDS/IPS project Suricata [10]. It explores its settings and also goes over tools that can be used to tune deployed Suricata instances. In the thesis, they are used to tune AF_PACKET running mode on given hardware. The performance results are later used as a baseline for future work.
The goal of the thesis is to propose and implement optimization of Suricata. Results of performed analysis have led me to implement a new running mode that uses a DPDK library. This should reduce the load of the capture interface and leave more resources to other Suricata modules. At the end of the master thesis, DPDK is compared to the performance of AF_PACKET and the results are evaluated.

Chapter 2

Suricata

Suricata [10] is a community-run, open-source project acting as an intrusion detection system (IDS) or an inline intrusion prevention system (IPS). It also has the ability to perform network security monitoring (NSM) and offline packet capture (pcap) processing. It performs deep packet inspection using pattern matching. It is mainly maintained and developed by the non-profit organization OISF and the Suricata source code is licensed under version 2 of the GNU General Public License. The first beta version was released in December of 2009.

Suricata provides a long list of configurable options ranging from the definition of network architecture over logging preferences to settings of packet capture strategies and runmodes. To understand the consequences of each option it is desired to, if not fully, at least partially understand concepts Suricata is built upon. With this knowledge, it is possible to achieve higher throughput and better performance thanks to appropriately chosen settings. Awareness of constraints in the design of Suricata and packet capture mechanisms, in general, can help to quickly identify possible bottlenecks in various configurations. As an overview and brief intro, this chapter provides diagrams and a more in-depth description of the internal structure of Suricata. You can find different stages of packet processing in Suricata below. Basic Suricata architecture is followed by deeper dive into separate sections of the configuration file. Configuration is stored in a *suricata.yaml* file with many options coupled with helpful comments.

Suricata can normally be set up in three modes:

- Host-based IDS (HIDS) (Figure 2.1) – runs on a single machine placed on the edge of the network, monitors the critical resources, and is mainly used for testing. It does not come across the traffic of other network devices as it sees only flows incoming/outgoing of the device it is installed on.

- Network IDS (NIDS) (Figure 2.2) – tap on a strategic network link, analyzes traffic and raises an alert if a rule is matched, packets are never intercepted. It is usually placed on the boundary of the local network (LAN) and the Internet (WAN). NIDS never blocks traffic and only notifies the network administrator about suspicious events that match the specified rules. Responsibility of at the attack mitigation is left on the network administrator. NIDS is further mentioned as IDS only.

- Inline IPS (Figure 2.3) – provides the possibility to monitor and block certain types of traffic if rules are matched. It works on an automated basis and requires no assistance from the network administrator once set up. Similarly to NIDS, it is situated in the

same place of the network. However, it additionally not only detects but also blocks matching network traffic. In some scenarios, a high rate of false-positive matches results in significantly a worse user experience.

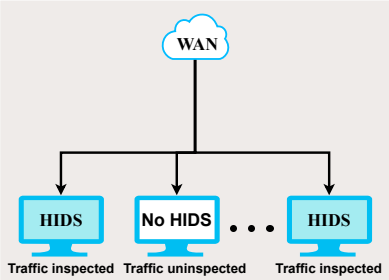

Figure 2.1: Host based intrusion detection system (HIDS).

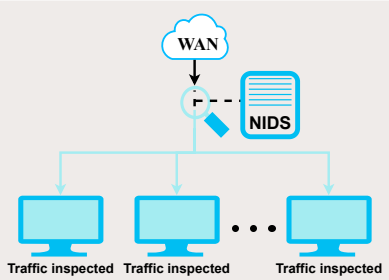

Figure 2.2: Network intrusion detection system (NIDS).

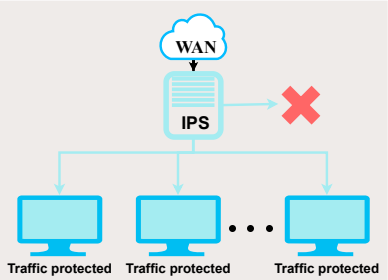

Figure 2.3: Inline intrusion prevention system (IPS).

2.1 Architecture

To introduce Suricata, high-level overview is presented in the following section along with Figure 2.4 depicting the main thread-modules of Suricata's architecture. Thread-module is an abstract name of multiple similar and connected functions aggregated into one unit. Thread-modules also separate functionality and responsibilities of individual thread-modules. [2]

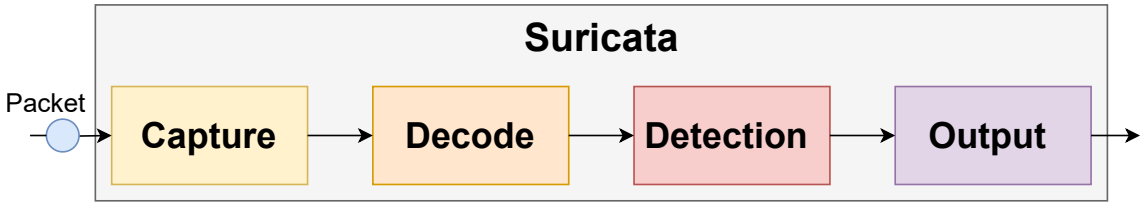

Figure 2.4: Suricata architecture of thread-modules.

As seen in Figure 2.4, Suricata comprises 4 main thread-modules:

- Capture – module receives packets from NIC, passes them to the next layer
- Decode – module decodes each incoming packet, reads data and saves decoded information into an internal representation of the packet. The decoding process starts from the bottom layers and continues to the top layers (of OSI model). The module also handles stream tracking and TCP reassembly. New decoding functions can be added at the end of the processing pipeline.
- Detect – module matches internal representation of packets against pre-defined rules. Module is parallelized into smaller modules processing one packet simultaneously.
- Output – module passes verdict (result) of the detection module to the configured outputs and appropriately adjust Suricata statistics. The module handles all events and alerts.

2.2 Detection rules

Detection rules (or signatures) are fundamental building blocks of intrusion detection/prevention systems. Each rule define behavior of network traffic. To operate, they require knowledge of the network architecture that they are supposed to protect. In Subsection 2.2.1, network definitions are described in more detail with examples of usage. Afterwards, variables defined here are used to modify and specify behavior of each rule. Rule composition and rule matching algorithms are then described in Subsection 2.2.2.

2.2.1 Network definitions

For Suricata to know which resources it needs to protect, it is required to specify lists of either subnets or exact IP addresses in its configuration file and lists of ports. As discussed at the start of Section 2.2, it is also possible to define user-defined variables to use in rules definition. Table 2.1 provides a syntax overview that can be used to define network hosts. It starts with definitions of the exact IP address and IP range via prefix. Additionally, these can be combined with basic operators as grouping ("[.., ..]") and negation ("!"). Listing 2.1 is an example of network definition extracted from *suricata.yaml* where variable `HOME_NET` uses grouping operation to define a group of three subnetworks defined by prefix and `EXTERNAL_NET` uses negation operator to match every IP address but the ones defined in `HOME_NET`.

Example	Meaning
192.168.0.21	Exact definition of an IP address
192.168.0.0/16	IP addresses ranging from 192.168.0.0 to 192.168.255.255
[192.168.0.0/24, 192.168.1.100]	IP addresses ranging from 192.168.0.0 to 192.168.0.255 and the exact IP address 192.168.1.100
[10.24.0.0/24, !10.24.0.100]	All IP addresses ranging from 10.24.0.0 to 10.24.0.255 but excluding the exact IP address 10.24.0.100
!$HOME_NET	All IP addresses except those included in HOME_NET variable
any	Arbitrary IP address

Table 2.1: Examples of defining network hosts.

```
vars:
  # more specific is better for alert accuracy and performance
  address-groups:
    HOME_NET: "[192.168.0.0/16,10.0.0.0/8,172.16.0.0/12]"
    EXTERNAL_NET: "!$HOME_NET"
    HTTP_SERVERS: "$HOME_NET"
    DNS_SERVERS: "$HOME_NET"
    ...
```

Listing 2.1: An example of network hosts definition cut from *suricata.yaml*.

2.2.2 Rules

Rule (or signature) is a common way for IDS/IPS to detect a threat in the network traffic passing through the system. Although it resembles basic pattern matching, it also uses additional fields to match network specific properties.

To form a threat into a rule, it is needed to extract certain features or properties of the threat that are shared even if the threat would target another device. Set of these features can then be written down and form a rule, which can then be appended to the set of existing rules (to ruleset). If the rule is correct, the next time the same or similar threat happens, system administrators are immediately notified in case of IDS or the attack is mitigated in case of IPS. However, attackers would have a huge advantage if each system administrator had to manually create their own rules for rulesets after each attack. Therefore, a community around the rulesets was formed to gather and manage rules received from system administrators around the globe. One of the most notable rulesets is Emerging Threats[1] from Proofpoint, which offers PRO and OPEN version. In PRO version they offer daily updates of the latest malware-detecting rules. OPEN version is often sufficient as it contains more than 20000 relevant rules.

Suricata Update[2] is a command line tool to manage and update installed rulesets. It automatizes the whole process of update, so it is hopefully less error-prone. Suricata rule is in Figure 2.6 for better comprehension of the structure and importance of it. Suricata-Update reads the configuration files and then finds and downloads the latest version of Emerging Threats Open ruleset for the version of Suricata installed on the system. Afterwards, it updates the ruleset according to the loaded configuration.

As can be seen in Figure 2.5, Suricata rule structure consists of 3 logical parts – ACTION, HEADER and RULE OPTIONS. Each part of the rule is described in the list below.

ACTION HEADER OPTIONS

Figure 2.5: Suricata rule extracted from the ruleset.

ACTION determines what happens when the rule matches. It can be considered as a result of the rule matching process. Values are:

- pass – allow a packet to pass,
- drop – drop packet and generate an alert to the configured output files (only works in IPS/inline mode), the communication is timed out as the packet is silently dropped,
- reject[src,dst,both] – send RST/ICMP unreach error to the sender/receiver/both sides of the communication,
- alert – generate an alert to the configured output files.

[1] https://rules.emergingthreats.net/
[2] https://suricata.readthedocs.io/en/latest/

HEADER part is specifying the traffic to match in broader granularity. Flows matching the header are further inspected with rule options. Header consists of:

- protocol – specifies the protocol to match – possible values are from all sorts of different layers of The Open Systems Interconnection (ISO/OSI) model[3] e.g., tcp (L4), udp (L4), icmp (L3), ip – stands for „any“ (L3), http (L7), ftp (L7), tls (L7). Availability of protocols can be manually configured in the *suricata.yaml* file.
- source and destination – provides high versatility in specifying the origin and target of the traffic. Exact IP addresses (e.g., 132.153.32.25) and IP ranges (e.g., 10.0.2.0/24) can be combined with operators of negation (`"!"`) and grouping (`"[.., ..]"`). It is also possible to use user defined variables from the *suricata.yaml* (e.g., $HOME_NET) and a keyword „any“ to match everything. Examples of defining IP addresses for source and destination are in Table 2.1.
- ports – specifies the ports to match, as with source and destination, Suricata provides a very flexible way to input ports with options to use exact ports, port ranges (`":"`), lists (`"[.., ..]"`) and exceptions (`"!"`). Examples of defining ports are similar as in Table 2.1 except port range is defined with colon (e.g. `" 8080:8088 "`) and not prefix.
- direction – specifies which way the rule has to match. Most signatures use one way direction operator (->), which specifies the source on the left and the destination on the right. It is also possible to use bidirectional operator (<>) to match rule both ways.

OPTIONS of rule specify additional information of the rule or extra requirements for the rule to match. Each rule needs to have signature ID (sid). If multiple rules share the same sid, revision (rev) option is also compulsory for Suricata to pick the latest rule. Options are enclosed by parenthesis and each rule is separated by semi-colon. Individual rule can be specified in two ways:

- key-value pair with syntax: <keyword>: <settings>;
  Example:
  ```
  msg:"ET ATTACK_RESPONSE Hostile FTP Server Banner
  (Bot Server)"; flow:established,from_server;
  ```
- key only with syntax: <keyword>;
  Example:
  ```
  nocase; dns.query;
  ```

Rule in Figure 2.6 is an example of information written in previous paragraphs put together. Action of the rule is to alert the administrator. Header of the rule defines the connection must be over TCP with any source IP address and with source port 21. Destination IP address must matches one of IP addresses defined in the variable HOME_NET (following the syntax from Table 2.1 of network definitions). Destination port does not matter for this rule. Rule option:

- msg – is shown to the network administrator in the defined outputs (as shown in Section 2.6),

[3] https://www.iso.org/ics/35.100/x/

- flow – signifies the connection must be established,
- content – packet must contain "220 Bot Server (Win32)",
- nocase – content search is case insensitive,
- classtype – used to categorize the rule as "trojan-activity",
- sid – signature's (rule's) ID is 2002811,
- rev – it is the fifth revision of the rule.

```
alert tcp any 21 -> $HOME_NET any (msg:"ET ATTACK_RESPONSE Hostile
FTP Server Banner (Bot Server)"; flow:established; content:"220 Bot Server (Win32)";
nocase; classtype:trojan-activity; sid:2002811; rev:5;)
```

Figure 2.6: Suricata rule extracted from the ruleset.

2.2.3 Rules processing

Rules are loaded in the order of which they are in the rulesets. However, they are then sorted according to the priorities and by default are evaluated in the order they have been listed in the upper paragraph. Action-order can be adjusted in *suricata.yaml* configuration file. Suricata rules are also compatible with Snort rules and on top that it accepts L7 protocols as mentioned in protocol part of rule format description. Upon loading, Suricata's detect engine groups similar rules to efficiently manage operating memory and performance. When ICMP packet arrives, no IP rules can be applied and thus all of them are skipped. Rules with common properties are usually placed in the same group as displayed in Figure 2.7. On the highest layer of the rule-groups tree lies the protocol, then direction of the flow, source and destination address and the further it is, the more specific the group is.

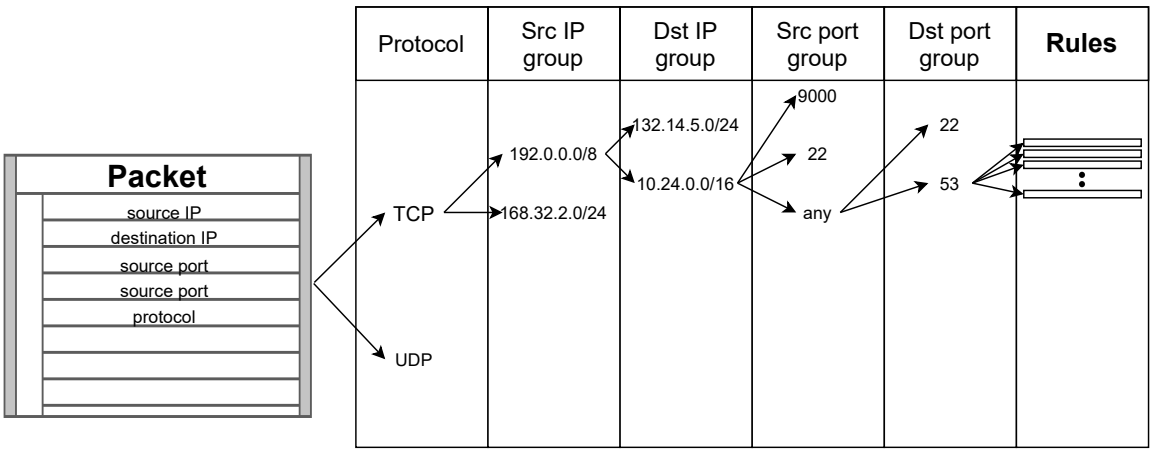

Figure 2.7: Rule groups.

However, with an increasing number of groups, Suricata demands more memory to store them. The quantity of groups determines the balance between performance and memory.

Each group is described by additional information in a multi-pattern matching (mpm) context stored in a group head. It is possible to manage memory requirements and limits in the settings file.

MPM groups are matched in the detection engine with MPM algorithm. In Suricata there are 4 alternative algorithms:

- ac – Aho-Corasick, enabled by default,
- ac-bs – Aho-Corasick, reduced memory implementation,
- ac-ks – Aho-Corasick, "Ken Steele" variant, recommended if Hyperscan is not available,
- hs – Hyperscan, required to add when Suricata is built. Requires x86 processors (32 or 64 bit architecture) and Streaming SIMD Extensions 3 at minimum. However, on most modern processors it can be installed without any problems.

Hyperscan[4] [15] is an open source project licensed under the BSD License and led by Intel. The main language is C/C++. It is a high-performance mpm library, follows the common syntax of regular expressions used in libpcre library but has an own API. It uses hybrid automata techniques to perform parallel matching of thousands regular expressions on data streams. As mentioned before, Suricata prefers to use Hyperscan in the detection engine to match regular expression in the passing network traffic.

2.3 Runmodes

To have a look at available runmodes in Suricata, it is first required to have deeper understanding of the core building blocks of Suricata – threads, thread-modules and queues. Suricata is a multi-threaded system. Therefore, it allows multiple threads to be active at the same time. A thread is a running process in an operating system, consisting of thread-modules and connected with other threads with queues. Thread-module is explained in the first paragraph of section 2.1. It is used to express a functionality part of Suricata. In Figure 2.4 there are 4 thread-modules:

- Capture,
- Decode,
- Detect,
- Output.

Queues are used to pass packets between the threads. As mentioned before, Suricata can run in one or multiple threads. Each thread can consist of one or more thread-modules. If the thread consists of multiple thread-modules, only one thread-module can run at a time. Each thread-module can only work with one packet at the time. However, Suricata is usually composed of a collection of threads assigned to different CPU cores. Each core can work individually, and, for that reason, Suricata engine can process multiple packets simultaneously. The way the architecture of these building blocks is configured by is called the `Runmode`. Suricata contains several pre-defined runmodes.

[4] https://www.hyperscan.io/

Runmode single as displayed in Figure 2.8 contains all thread-modules from capture to output in a single thread. It is mainly used in a process of initial configuration or debugging. The number of management threads is not limited and is configured in *suricata.yaml* file. Network traffic does not have to be load balanced since all traffic is forwarded to a single thread.

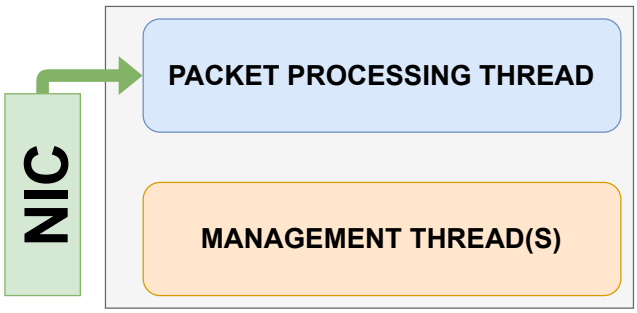

Figure 2.8: Runmode Single.

Runmode autofp is mainly used with inline IPS setups and while processing PCAP files. It is possible to note that in Figure 2.8 the whole packet processing is done in one thread while in Figures 2.9 and 2.10 the process is divided into flow worker threads (FWTs) and packet capture threads (PCTs). PCTs capture and decode the packets, then the packets are forwarded to FWTs to finish the packet processing. In autofp runmode number of packet capture threads is less than the number of flow worker threads. Network traffic is distributed between PCTs and they are then used as additional load balancers for FWTs. If there is a single PCT as in Figure 2.9, load is solely distributed in internals of Suricata. In setups similar to the Figure 2.10 packets are load-balanced by both the NIC and Suricata.

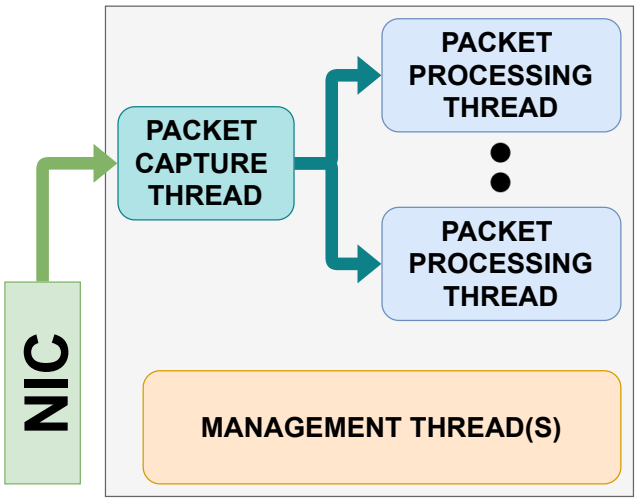

Figure 2.9: Single capture thread in autofp runmode.

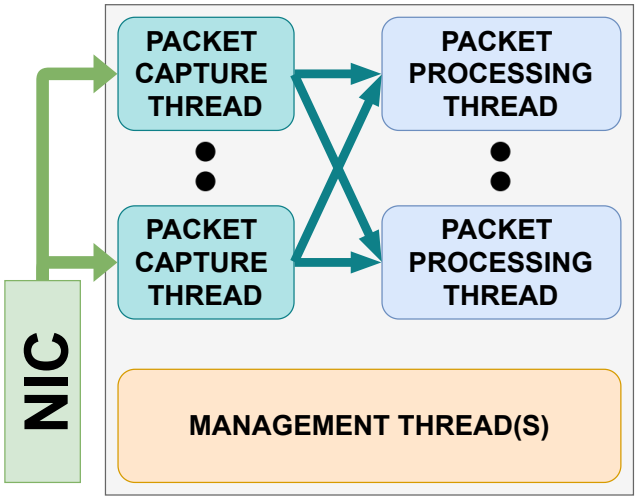

Figure 2.10: Multiple capture threads in autofp runmode.

In runmode workers, the whole process of packet inspection is performed in one individual thread for each packet. As it is possible to see in Figures 2.8, 2.9, 2.10 and 2.11, the only difference is a different amount of full packet processing pipeline threads and the fact that the NIC distributes the network traffic over the various threads. In most cases, workers runmode results in the best performance compared to other runmodes.

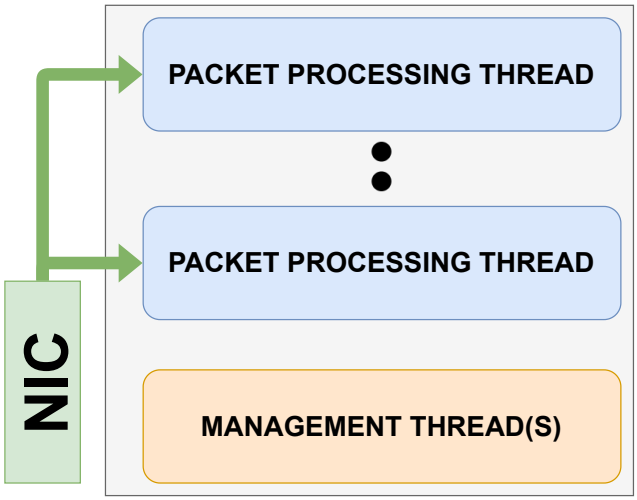

Figure 2.11: Runmode Workers.

2.4 Capture options and running modes

Sockets in Linux work as an API to access network interfaces to send or receive packets. Over the years of development, Suricata developers and other contributors have implemented numerous packet capture interfaces into running modes and usually named

after the implemented interface. The most used modes are AF_PACKET, PF_RING, NETMAP and PCAP. Additionally, these interfaces can be individually accelerated using optimization techniques discussed below in the individual sections of packet capture options.

Figures 2.13, 2.14 and 2.19 share similar behavior. It is possible to notice, especially in the upper part of the diagrams, the relationship of components – *Network driver*, *Buffer* and *Deferred reception*. It is explained in more detail in Figure 2.12. The component Deferred reception means that received packets from the NIC are put into the Buffer shown on the right side and the interrupt is sent to the CPU. When the CPU acknowledges the interrupt, it is scheduled with high priority. On handling the interrupt, the CPU must stop the activity that it is executing and switch context to packet processing. Once the context is switched, it allocates the socket kernel buffer through function `skb_allocate()` and copies there the data from the Buffer. References of all packets received between the interrupts can then be sent to the user application.

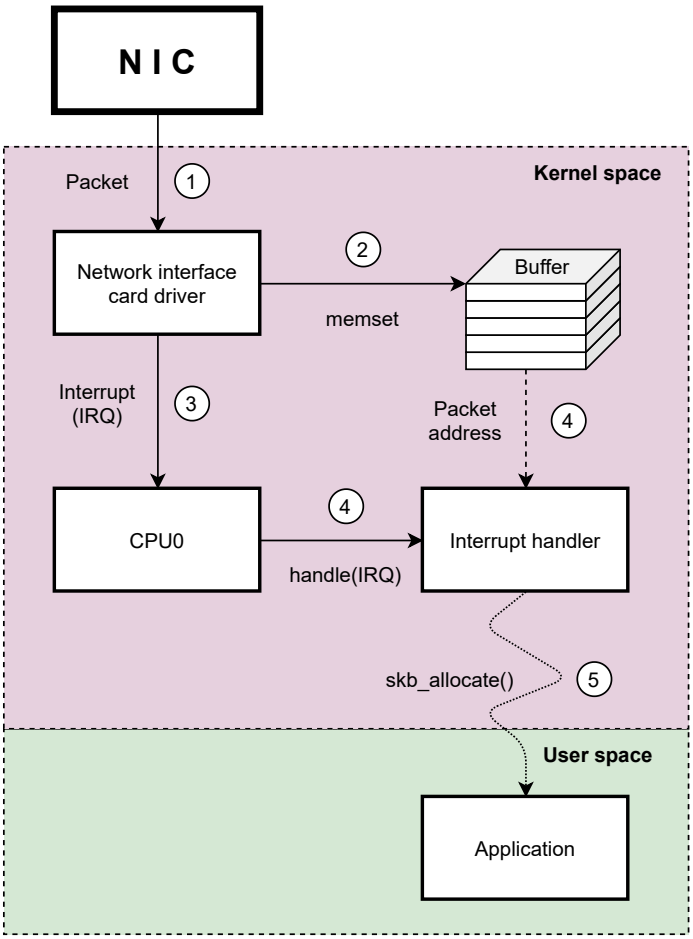

Figure 2.12: Packet processing and interrupt handling in the CPU.

2.4.1 AF_PACKET

The AF_PACKET socket was implemented in the Linux kernel since the version 2.2[5] and therefore it is commonly used in Linux network programming. It works in cooperation with the kernel, from where incoming packets are copied to the user space as soon as they are received from the physical layers of the card. Similarly, outward packets are copied to the kernel space just before they are sent to the NIC. The Figure 2.13 can help to understand the way AF_PACKET works. Starting from the top, packets gathered by the network interface are sent to the kernel. If the interface is bound to socket, the packet is cloned and sent to the user space, where the application is running. The original packet continues to be processed in the kernel. Some performance loss can be expected due to packet cloning.

In the Linux kernel version 2.4 "mmap" functionality was added to the AF_PACKET interface. It provides a configurable ring buffer mapped in user space which can be used to receive or send packets. As can be seen in Figure 2.14, packets are stored in a structure accessible from both physical (kernel space) and virtual (user space) layers.

Being able to reach packets stored in the ring buffer reduces the number of system calls needed to deliver a packet to the application in the user space and the number of required packet copies.

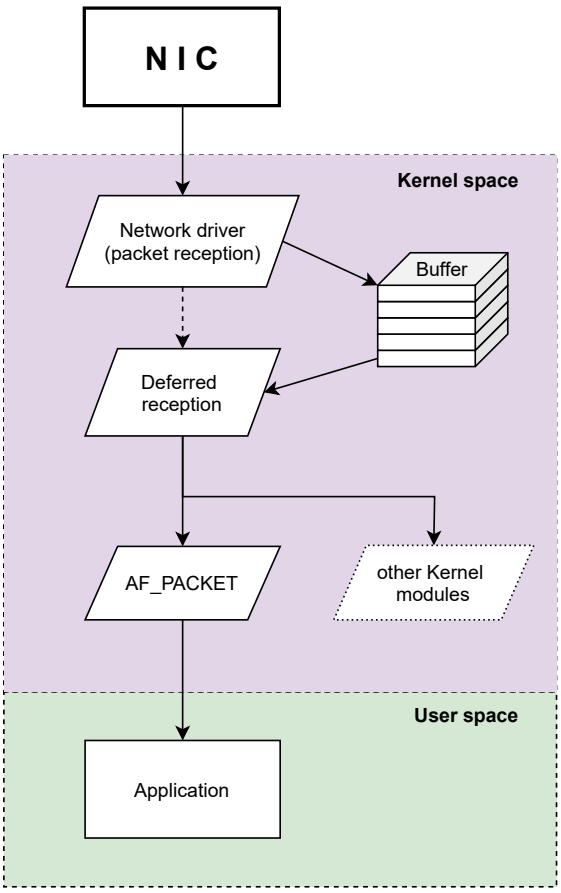

Figure 2.13: Packet flow in AF_PACKET capture interface.

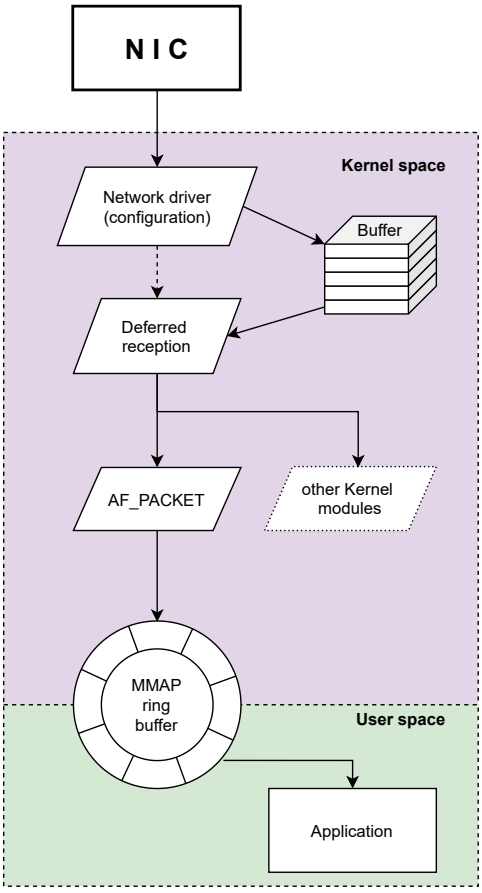

Figure 2.14: Packet flow in mmap version of AF_PACKET capture interface.

[5] https://manpages.ubuntu.com/manpages/precise/man7/packet.7.html

2.4.2 eBPF and XDP

(**C**lassic) **B**erkley **P**acket **F**ilter (cBPF) is a long-known technology implemented in Unix-like operating systems. It acts like a filtering mechanism that only allows to receive packets that comply with conditions specified in a user-set program (filter). These programs are run in a BPF 32-bit virtual machine that resides in the kernel. Before packet is accepted or rejected, programs in the virtual machine perform arithmetic operations on packet's data and test the results against set constraints. Classic BPF programs typically allow high-level text rule describing the pattern to match from which it is converted to machine code by assembler.

Extended **B**erkley **P**acket **F**ilter (**eBPF**) [8] enhances capabilities of the original BPF virtual machines. Since the Linux kernel version 3.18, virtual machine for BPF programs have increased the number of registers to 10 and doubled the size of the registers to 64-bits. Nowadays, all BPF programs are translated into an eBPF bytecode in the kernel before program execution. To compile eBPF program into machine code, it is required to use special compilers such as *clang*[6]. eBPF programs can be written in a subset of C, which is then compiled by one of eBPF compilers. This allows writing more complex filtering mechanism to more accurately inspect passing traffic. eBPF brought not only improved virtual machine but also new helper functions that ease programming, eBPF verifier to prevent any downtime with incorrectly written eBPF programs and eBPF maps which serves as an intermediate storage between the eBPF program and an userspace application and to keep state between invocations of eBPF program. There are different kinds of maps such as Hash, Array, Per-CPU Hash/Array and others.

Express **D**ata **P**ath (**XDP**) [8] builds on top of the eBPF mechanism to allow high performance networking data path. XDP is placed very early in packet processing pipeline – even before allocating socket kernel buffer. For this reason it allows eBPF program to make a decision in the beginning in the packet processing pipeline and thus save computational resources for packets that actually needs to be analyzed by a user-space application. Memory allocation can be very expensive operation. Similarly to eBPF, it operates on raw packets. As depicted in Figures 2.15, 2.16, 2.17 XDP can work in 3 modes within the packet processing pipeline:

- **hardware** – as depicted in Figure 2.15, eBPF program is loaded to a programmable partition of the NIC. It allows to save CPU resources for other tasks. Currently supported by Netronome only. [4]
- **driver** – program is loaded and executed within the NIC's driver but before the socket kernel buffer is allocated for the packet. This method is more prevalent among NICs producers. It is shown in Figure 2.16.
- **soft / generic** – illustrated in Figure 2.17, generic XDP implementation that should work on any Linux kernel with version 4.8 or higher. Although this mode is the most universal, it also comes with the highest performance penalty since the eBPF program is executed after the skb allocation.

[6] https://clang.llvm.org/

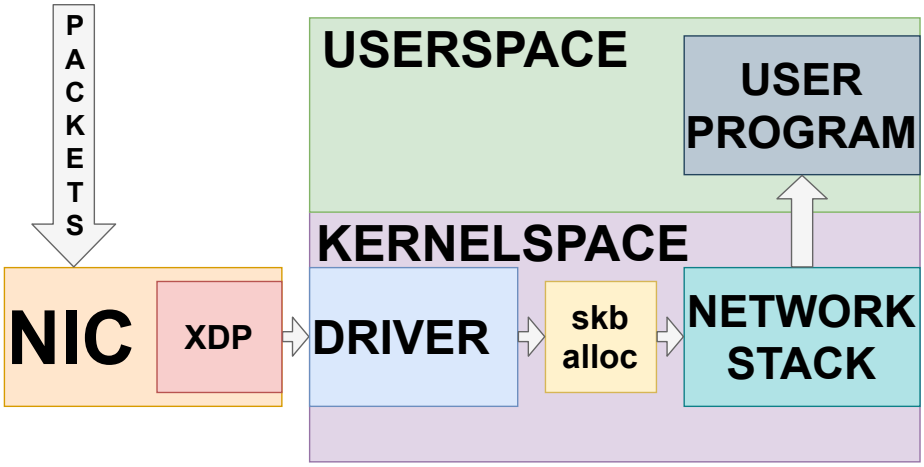

Figure 2.15: XDP Hardware mode.

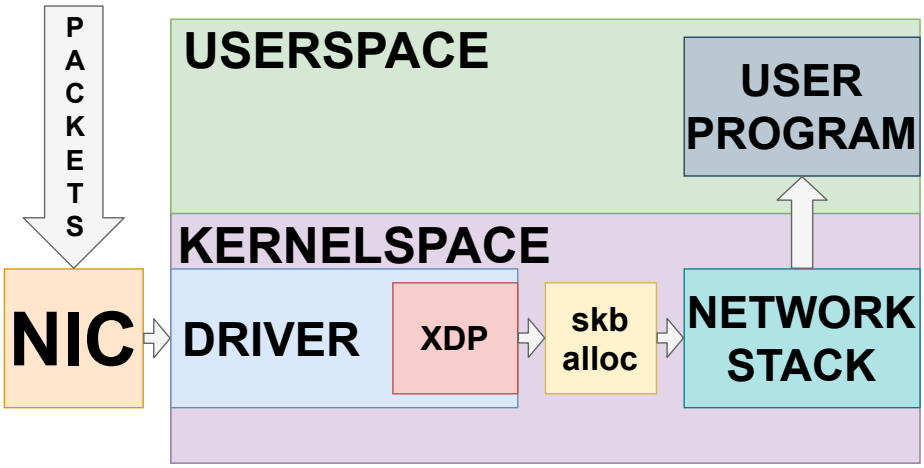

Figure 2.16: XDP Driver mode.

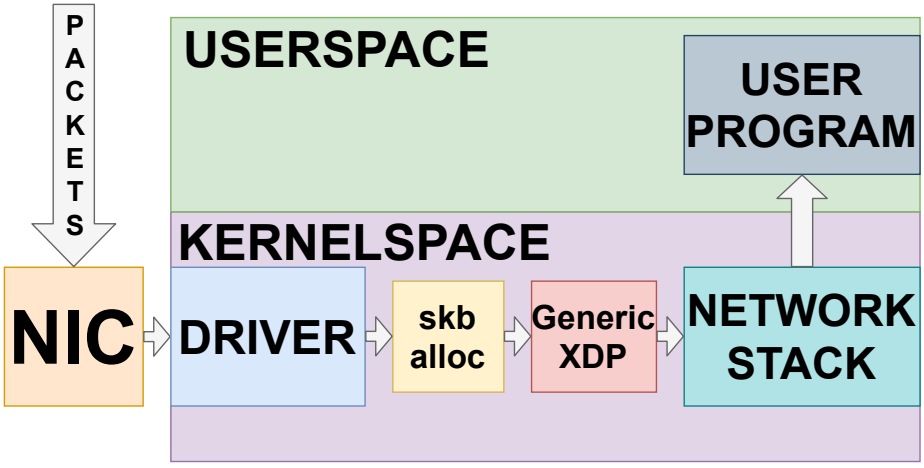

Figure 2.17: XDP Soft/Generic mode.

eBPF program is invoked once it receives a packet. It immediately starts processing it and after completion it returns an action code that determines fate of the packet. Possible outcomes are:

- XDP_PASS – packet continues to the network stack,
- XDP_DROP – packet is never to be seen again,
- XDP_ABORTED – program throws an error, packet is dropped,
- XDP_TX – transmit to the NIC,
- XDP_REDIRECT – redirect packet to a different NIC or CPU.

Suricata support of eBPF and XDP

In Suricata [5] these mechanisms are implemented in the most used interface – AF_PACKET – to extend possibilities of configuration. This can aid Suricata in improving performance even more. Suricata has some XDP and eBPF filters included directly in the source code to help with a smoother start. If enabled, it can be compiled and installed along Suricata. Traffic passing through Suricata can be cut through with an eBPF filter to ignore known and safe connections (e.g. video streams from popular streaming platforms or scheduled backups of data). To drop packets even earlier, XDP filter can be used as well. As shown in Figure 2.18, Suricata (user program) can dynamically talk to the filtering program and adjust properties of the ignored traffic via eBPF hash table (eBPF map). The hash table acts as a flow table that can tell kernel program whether the packet should or should not be processed. As mentioned before, since Suricata can dynamically adjust properties of ignored traffic, flow table can not only be set before start of Suricata but also during its execution. Flow table can also modified if Suricata hits a rule with a special keyword (**bypass**) in the rule's options section for the given flow. Flow is then added from the user program (Suricata) to the hash table (eBPF map). Incoming packets of the given flow are then ignored when the filter extracts flow information from the packets and compares them to values stored in the eBPF map. It is described in more detail in Section 2.5.

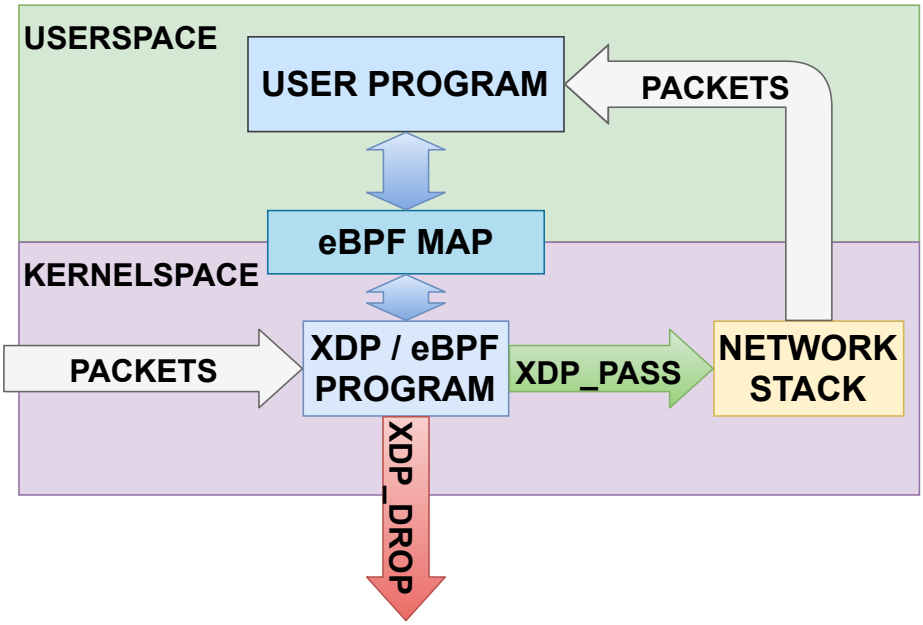

Figure 2.18: Usage of eBPF maps within XDP.

2.4.3 PF_RING™

PF_RING™ is a type of network socket and a kernel module developed by company ntop[7]. It enables high-speed packet capture and analysis. Similarly to mmap version of AF_PACKET, module creates a ring buffer shared between the user space and the kernel, as can be noted in Figure 2.19. It is a free open source project, and it is available since Linux 2.6.32. Kernel module is licensed under the GNU GPLv2 license, LGPLv2.1 for the user-space PF_RING™ library. Thanks to the modular architecture of PF_RING™ it is possible to add additional closed-source, paid ntop modules such as Zero Copy, FPGA or other modules. Modules can further increase performance.

Internally it uses Linux New API (NAPI) to poll packets from the NIC, copy them to the circular buffer from where the application (in the user space) can retrieve all packets.

2.4.4 PCAP

Running mode PCAP reads the specified pcap files and replaying them to Suricata for offline analysis. It allows to test rules or configuration on real-life traffic in situations where deployment in live networks is not possible.

2.4.5 DPDK

Data Plane Development Kit (DPDK) [12] is a framework to enable high-speed packet processing. It is an open source project, licensed under the Open Source BSD License. It was originally developed by Intel, in 2017 it was joined to Linux Foundation. Packet processing workloads are accelerated on a wide range of device architectures like x86, ARM or PowerPC. Currently, DPDK only supports Linux and FreeBSD operating systems. It is

[7] https://www.ntop.org/

important to note that the framework needs to have access to advanced parts of the NIC. These parts are accessible via the device driver. Therefore, some NICs require a custom driver to be installed in order to work with the DPDK library. However, majority of NIC manufacturers like Intel, Mellanox or Napatech support the DPDK on their cards. DPDK might not be supported especially on lower-end NICs.

Officially, Suricata does not support DPDK, however, there has been discussions about supporting the framework and also an effort to implement it[8]. Unfortunately, at the time of writing this thesis, it was still not publicly available.

Lack of DPDK support in Suricata was motivation for bachelor thesis of Igor Mjasojedov led by Jan Kučera. Thesis analyses different intrusion detection systems, out of which Suricata is considered as the best candidate for further development. There are also described possible options to add a support for DPDK with final performance tests at the end. It was implemented following the common architecture of DPDK applications [9].

In general, DPDK enables packet processing in user space, similarly to previous packet capture running modes. By bypassing the kernel, more resources can be concentrated on the user space application processing the packets. The DPDK architecture can be seen in Figure 2.20. Packets are transferred directly from the NIC to the user application while kernel space is only used to store NIC's driver and to configure the NIC.

As can be seen in Figure 2.20, from the large part DPDK architecture bypasses kernel and uses the kernel loaded NIC's drivers only for the configuration. The transfer of packets from the NIC to the application is managed by the Poll Mode Driver (PMD) which provides a direct connection between these two parts. However, it must be supported by the network interface. Other capture interfaces use CPU interrupts. This is shown in Figures 2.13, 2.14, 2.19 as *Deferred reception*. Polling method has direct access to the Buffer and repeatedly queries the NIC for new data. This method is more performance oriented as no forced context switching is required and packets are transferred straight to the application. To ensure high portability of DPDK applications Environment Abstraction Layer (EAL) is present in the DPDK API. EAL provides a unifying layer between the application and the NIC to prevent tight dependencies on the architecture the application was developed upon. The DPDK framework therefore allows zero-copy transfer of packets from the NIC to the application, while providing support for cross-platform and architecture independent usage.

[8] https://github.com/vipinpv85/DPDK_SURICATA-4_1_1

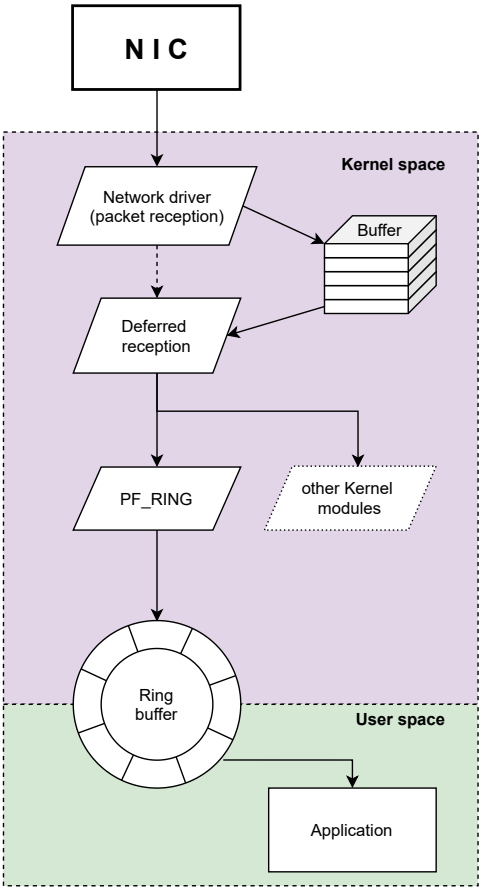

Figure 2.19: Packet flow in PF_RING capture interface.

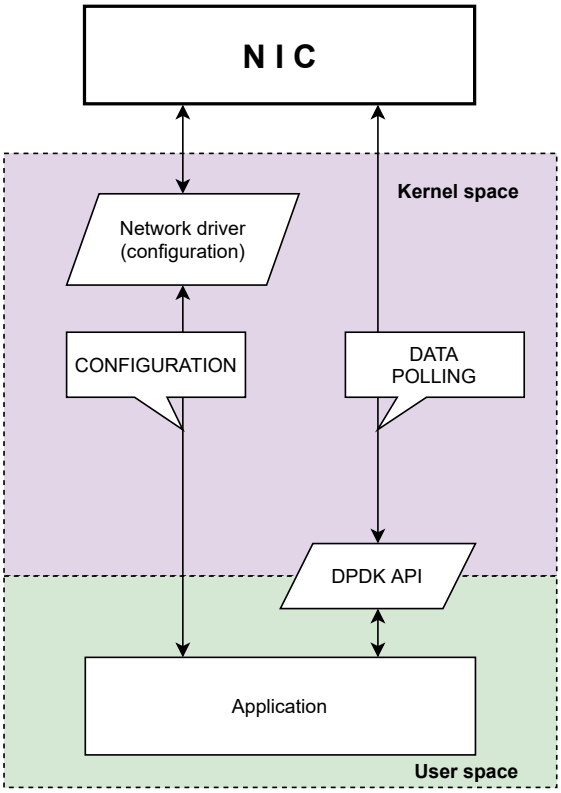

Figure 2.20: Packet flow in DPDK capture interface.

2.5 Bypass and flow shunting

Traffic bypass is one of the Suricata features to lower the delay and enable higher packet processing throughput. The idea behind bypass is fairly simple and straightforward. Divide network traffic into flows and, using rules or other techniques, mark flows that can be bypassed. The next packets of these flows are not inspected anymore and we drop/redirect them as fast as we can. In Suricata, bypassing a packet can be done on multiple levels. Performance benefits of bypass can vary depending on where packet inspection stops. The sooner the packet processing of the given flows is stopped, the more the enhancement is efficient. Bypass can be executed internally in Suricata (local bypass) or in a capture interface as mentioned, for example, in Subsection 2.4.2. Since local bypass is implemented directly in Suricata, it is capture interface agnostic and therefore it can be always used. On the other side, capture interface bypass must be supported directly in capture interface implementation by providing functions that should execute on bypass. From currently implemented capture interfaces there are only 2 that supports it – AF_PACKET and NFQ. AF_PACKET makes use of key-value storage of eBPF hash table to tell the filter program in the kernel to drop or redirect packets. Kernel capture bypass is more effective because it saves not only Suricata from reading the packet, decoding it and rules classification but it also frees kernel from processing the packet in the network stack (allocating skb, interrupt handling and more). In case of XDP hook in the hardware (NIC) bypass function can ever more boost performance.

Flow shunting is a technique that classifies traffic into flows and then it uses bypass to flag flows that meet a certain condition. Usually it is activated for long flows once a certain (preset) number of packets is reached. It is not common for attacks and threats to send pre-packets (packets before the attack) and they usually occur on the very beginning. This makes long flows less interesting for Suricata and they can be considered as safer compared to short aggressive flows. Alternatively, it can be used for safe domains (e.g. products of Google, Amazon). More details can be found in reference [3, 14].

2.6 Outputs

In *suricata.yaml* you can also set which properties and features of the analyzed traffic network are logged and sent to the outputs. Suricata can send data off to:

- eve.json – stores information about analyzed data in JSON format. As it contains many details, it is not suitable for manual inspection but rather it can be accessed by certain tools for a better analysis. From this file an administrator can identify sources of the attack and mitigate the threat. The file can be fed to the ELK (Elastic search, Logstrash, Kibana) stack and Evebox to conveniently view alerts in a web browser.
- stats.log – holds the basics statistics, which are exported in fixed intervals and appended at the end of file. It can serve well for a quick manual inspection. The results of this file are later used in analysis of Suricata performance. Stats can be logged at different levels of granularity – from per thread to cumulative results.
- fast.log – a line based alerts log.
- Syslog – a line based alert log sent to Syslog.

As is shown in Listing 2.2 outputs can be customized independently in the Suricata's configuration file with options to include extra fields into the output.

```
default-log-dir: yes
stats:
  enabled: yes
  interval: 8

# Configure the type of alert (and other) logging you would like.
outputs:
  - fast:
      enabled: yes
      filename: fast.log
      append: yes # after Suricata starts do not create new file

  # Extensible Event Format (nicknamed EVE) event log in JSON format
  - eve-log:
      enabled: yes
      ...
```

Listing 2.2: An extract of Suricata output configuration from *suricata.yaml*.

Chapter 3

Setting up the environment

Sometimes it is hard to configure Suricata well. But focusing solely on Suricata most likely yields in unsatisfactory performance results. Tuning is a complex process that includes an examination of other parts of the system and even the other network devices and many other factors to consider. Therefore, in this chapter, common bottlenecks of packet capture and packet processing are described in more detail with their possible explanations.

3.1 Network architecture

As Suricata tuning is a complicated process of adjusting tightly coupled hardware and software requirements and capabilities, it is important to define an architecture of network, where Suricata is planned to be deployed. As the tuned settings are not generic, focus on the designed architecture is essential. Components need to be wisely selected as building the infrastructure can be costly. Incorrectly chosen components (e.g., due to some incompatibilities) can lead to unwanted bottlenecks, which add extra expenses to fix.

A testbed is an implementation of the designed architecture. It serves especially as a platform for experiments where easily repeatable, transparent and accurate tests are possible. It can disprove or confirm theories and ideas. The term in various industries can have different shapes. In networking it is usually a mesh of different devices needed for test purposes. It is common practice that in software engineering testbeds are not connected to the live network. It attempts to prevent any damage to production services and to shield the testbed from potential intruders.

Testbed shown in Figure 3.1 is used for measurements in this thesis. It connects two machines named `Pinot` and `Claret`.

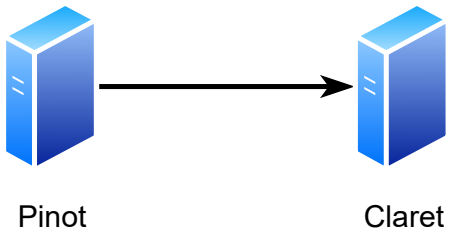

Figure 3.1: Architecture of the testbed used for measurements.

Pinot:

- OS: Scientific Linux 7.3 (kernel version 3.10.0)
- CPU: 6 cores Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00 GHz
- RAM: 64 GB
- NIC: internally made replicator rated for 100 Gbps
- Role: replay PCAP file of captured traffic with destination MAC addresses changed to `claret`'s NIC MAC address. PCAP size is approximately 300 GB. Replicator does not respect relative timestamps in the PCAP and transmission speed can be set arbitrarily.

Claret:

- OS: CentOS 8.1 (kernel version 4.18)
- Suricata: version 6.0.3-dev
- CPU: 10 cores Intel® Xeon® Silver 4114 @ 2.20 GHz
- RAM: 64 GB
- NIC: 2 port Mellanox ConnectX-5 MT27800 card rated for 100 Gbps
- Role: receive packets on the NIC, let Suricata inspect them

3.2 Configuration tool – ethtool

Ethtool[1] is a small utility supported by many vendors of NICs. It attempts to create a unifying interface to query and configure settings of NICs. It is not always true because not all options are implemented by the manufacturer of the NIC or not all options are supported or configurable by the NIC itself.

NIC's settings are initialized by the network driver with default settings from the manufacturer. Altering some parts of the configuration can lead to an increase in throughput or latency. However, it is important to clearly decide on priorities (throughput vs latency) before the configuration process as reaching a goal in one (e.g., throughput) usually decreases results of the other (e.g., latency). In this thesis, the goal was to achieve balanced results, with a possibility to focus more on the throughput.

Ethtool groups similar settings into individual switches. Lower letter switches (e.g., -g) query data from the NIC and capital letter switches (e.g., -G) configure the NIC. When adjusting settings, a set of options are commonly found after the specified interface. List of few frequently used switches, which are described below:

- `-g` | `-G` – ring buffer settings,
- `-a` | `-A` – pause frames settings,
- `-c` | `-C` – interrupt coalescing settings,

[1] https://linux.die.net/man/8/ethtool

- `-k` | `-K` – offload settings,
- `-n` | `-N` – receive network flow classification,
- `-l` | `-L` – NIC queues count settings,
- `-x` | `-X` – receive flow hash settings,
- `-S` – statistics.

3.2.1 Ring buffer size

Ring buffer is an important component in receiving packets. It is a buffer where the NIC stores all packets as they are received. Overwriting of existing data happens when the application is not able to drain the packets from the ring buffer quickly enough. In AF_PACKET packet capture interface, the application must communicate with the driver as the ring buffer is shared only between the driver and the NIC. The driver sends the packets to the higher layer of kernels and then to the application which holds the relevant socket. Ring buffer is shown in Figures 2.13, 2.14 and 2.19 as a component named Buffer. Enlarging the buffer size requires more operating memory because it is allocated in the kernel-space. Command bellow sets size of the ring buffer to maximum.

```
# ethtool -G eth4 rx 8192
```

3.2.2 Pause frames

When the receiver's NIC is getting overwhelmed by the incoming packets and the buffer is about to overflow, it can send a pause frame to let the sender know to stop transmitting packets for a specified period of time. The switch can then buffer some packets that are later sent. Historically, there were attempts in adopting this technique by manufacturers, however, it was never widely used and additionally, it caused problems. Command bellow disables receive (rx) and transmit (tx) pause frames.

```
# ethtool -A eth4 rx off tx off
```

3.2.3 Interrupt coalescing

In networking, interrupt coalescing is a technique that holds interrupts for a specified amount of time and then they are aggregated into one. It aims to reduce interrupt load by not firing them too frequently (by default, each received packet generates one interrupt). Potentially, this can have throughput benefits as the CPU handles batch of packets at once. However, it can lead to an increase in latency. The command bellow sets time for received interrupt coalescing to 125 milliseconds. It also disables adaptive moderation to ensure replicable results. Using the option rx-frames (example: `rx-frames 100`) it is also possible to configure the NIC to send interrupt either after timeout or packet count is reached.

```
# ethtool -C eth4 adaptive-rx off adaptive-tx off rx-usecs 125
```

3.2.4 Offloading

In modern network cards, there are several offloading features. It means that part of the network traffic is not processed by the CPU but rather it is directly processed on the chip of the NIC. It can then reduce load on the CPU.
For instance, General Receive Offload (GRO) can reduce processing overheads of small packets. Network card reassembles short packets into larger and then sends them to the application. As a result, CPU processes fewer packets. There are various kinds of offloading libraries, however, network traffic offloading usually means packet manipulation or modification. Raw packets, in the form as they are transferred over the network, are very important for intrusion detection/prevetion systems as rules (signatures) may not match the offloaded (and thus modified) traffic. For this reason, it is better to turn off every possible offload the network card may have.

```
for i in rx tx tso ufo gso gro lro tx nocache copy sg txvlan rxvlan; do
    ethtool -K eth4 $i off;
done
```

Options `rxhash` and `ntuple` allow classification of incoming network flows upon the results of the hash function. It is covered in more detail in Sections 3.2.5 and 3.2.6 but it is advised to enable them.

```
ethtool -K eth4 rxhash on ntuple on
```

3.2.5 Receive side scaling

To fully comprehend for what receive flow hash is used, it is first required to understand basic concepts of interrupt affinity and receive side scaling (RSS). Both techniques are commonly combined to maximize throughput of NICs.

Interrupt affinity

Interrupt (IRQ) is sent from a device (hardware level) to a CPU as a form of a request for some action to execute. In terms of NICs, it is usually meant to poll packets from the NIC's ring buffer. Packet acquisition through interrupt handling is shown in Figures 2.13 or 2.19 as deferred reception because the NIC inserts packets into the ring buffer and notifies the CPU by an interrupt. The described process is illustrated in Figure 2.12. Additionally, in the same diagram (2.12), component CPU0 is representing core number zero of the CPU. The default behavior is the same between single and multi-core systems. As a result, core 0 can be easily overloaded under high packet load and that leads to packet drops.

The term interrupt affinity means binding an application or a device to a specified set of CPU cores. This means the interrupt is not handled only by core 0 but it is distributed among all specified CPU cores. As a consequence, latency improvements and higher network throughput can be expected. Bash script `/usr/sbin/set_irq_affinity.sh` is included in the installation of Mellanox driver. After executing the script with a specified interface, it balances the load of interrupts among all cores. It is also possible to use other Bash scripts (`set_irq_affinity_bynode.sh` or `set_irq_affinity_cpulist.sh`) to distribute IRQs among the selected set of CPU cores only.

Receive side scaling

Knowledge from previous paragraphs can help to better understand basics behind receive side scaling (RSS). It relies on multiple hardware queues to which it can distribute incoming packets. Contemporary NICs support multiple receive and transmit descriptor queues (multi-queues). Sometimes queues are not differentiated and are labeled as combined. It is generally advised to assign one queue to one physical CPU core as opposed to assigning more queues to one CPU or using hyper-threaded logical cores. Switches "-l|-L" are used to view settings and manage the number of enabled queues. An example of command to configure the NIC to use 20 combined queues is:

```
# ethtool -L eth4 combined 20
```

When RSS is enabled, it might use XOR or Toeplitz hash algorithms to appropriately balance packets into the queues. Toeplitz hash algorithm allows specification of Random Secret Key (RSK) with a length of 40 bytes to determine how the packets are distributed. It serves as a seed that gets mixed with input values inside the hash algorithm. Input values are packet features aligned into n-tuples and described in more detail in Subsection 3.2.6. The algorithm provides load-balancing statistically. That means hashes of packets with different properties result in different queues bound to specific cores. The more random the traffic is (different IPs, protocol ports), the more uniform the distribution is.

It is sufficient for generic applications like web applications or streaming applications concerned about the transferred data only. However, for IDS/IPS to monitor and analyze passing network traffic, both directions of communication is required. Flow-based rules depend on packets from both inward and outward directions. As it is demonstrated in Figure 3.2, using the default hash algorithm, packets of different directions end up in different queues because hash of incoming 5-tuple is different from outgoing 5-tuple. 5-tuple[2] is a set of five different values usually consisting of source IP address, source port, destination IP address, destination port and transport protocol. It can be optionally set to n-tuple configured according to 3.2.6. This would lead to huge number of cache misses as it would be vital for detect threads to fetch data out of caches of different cores to get the most recent data of network flows.

To tackle this problem, use the key from Listing 3.1. It is an absolutely symmetric key for RSS. It puts packets of the same flow in the equivalent queues [16]. Toeplitz hash algorithm with symmetric hashing is demonstrated in Figure 3.3 as packets from both directions are put into the same queue. As stated before, one CPU core is bound to one queue and because of that, cache of each CPU core is occupied by packets of the same flow. As a result, cache misses are minimized since cores are not required to transfer data out of caches of different CPU cores.

```
0x6d5a  0x6d5a  0x6d5a  0x6d5a
0x6d5a  0x6d5a  0x6d5a  0x6d5a
0x6d5a  0x6d5a  0x6d5a  0x6d5a
0x6d5a  0x6d5a  0x6d5a  0x6d5a
0x6d5a  0x6d5a  0x6d5a  0x6d5a
```

Listing 3.1: Symmetric receive side scaling key.

Commands in Listing 3.2 set the RSS function to Toeplitz and its hash key to the previously mentioned key. After applying latency and cache misses should be lowered.

[2] https://www.ietf.org/rfc/rfc6146.txt

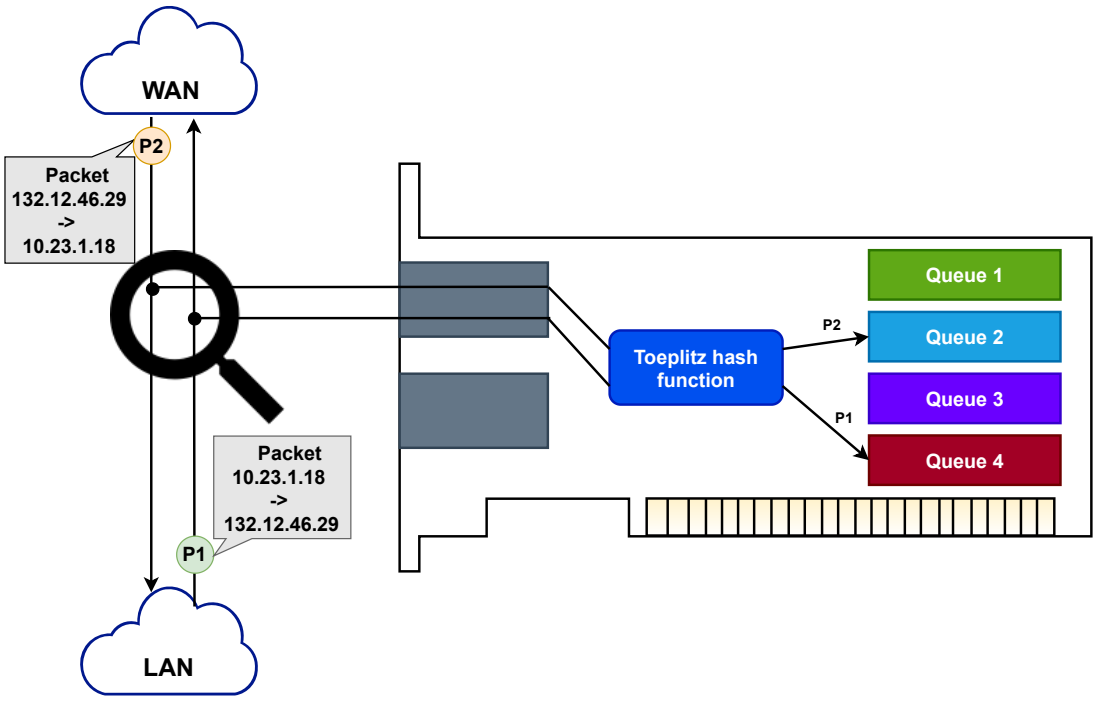

Figure 3.2: Receive side scaling with asymmetric hashing.

```
# echo toeplitz > /sys/class/net/eth4/settings/hfunc
# ethtool -X eth4 hkey \
    6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:\
    6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A equal 20
```

Listing 3.2: Setting a symmetric receive side scaling key on an interface.

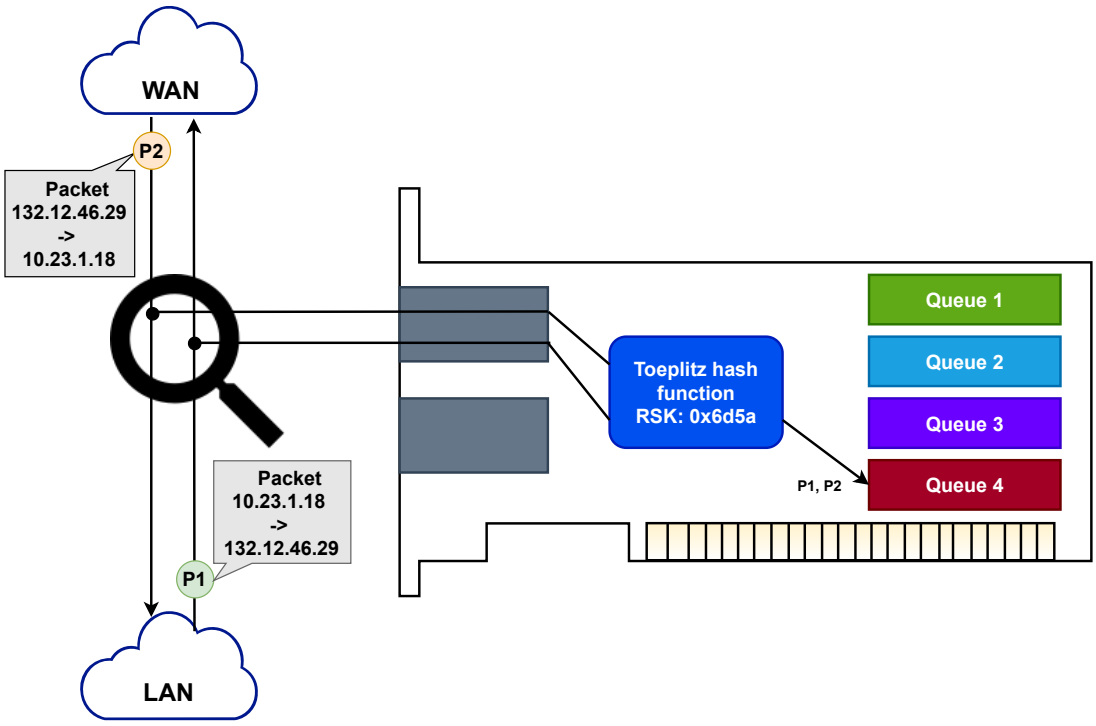

Figure 3.3: Receive side scaling with symmetric hashing.

3.2.6 Receive network flow classification

As mentioned before, Toeplitz hash function uses certain parameters of the packet formed into `n-tuple` where `n` specifies the number of parameters that are used as input parameters to the hash function. These parameters can be set using
`ethtool -N <iface> rx-flow-hash <flow type> <options>`
command with flow types (e.g., ip4, tcp4, udp4, ip6, tcp6, udp6) and options:

- m – Layer 2 destination (MAC) address,
- v – Layer 2 VLAN tag,
- t – Layer 3 protocol,
- s – Layer 3 IP source address,
- d – Layer 3 IP destination address,
- f – Layer 4 bytes 0 and 1 (source port in TCP and UDP),
- n – Layer 4 bytes 2 and 3 (destination port in TCP and UDP),
- r – Discarding packets of the flow type.

Commonly used are combinations of options `"sd"` and `"sdfn"`. Example of the command might look like:

```
# ethtool -N eth4 rx-flow-hash tcp6 sdfn
```

3.2.7 Statistics

Counters of acquired or dropped packets can be seen on several places in Linux. Tools like `ifconfig` are not always accurate as they are mainly designed as log tools and may not always have access to correct memory chunks. On the other hand, as ethtool is more tied to network drivers, results should be more favorable. It has direct access to NIC's counters and offers a wide variety of stats from packet/bytes counters for each separate queue to cumulative results of the whole NIC. It can be used to verify the number of packets received by the NIC. However, it is incorrect to verify count of dropped packets.

For measurements connected with dropped and received packets, a term *buffer overflow* (BO) was defined. As shown in Equation 3.1, BO is calculated as percentage of packets received (P_{rx}) divided by packets transmitted (P_{tx}).

Packets received (P_{rx}) is the difference of components acquired from the ethtool stats counter "rx_packets". Equation 3.2 shows that minuend is the number of received packets after all packets were transmitted and the subtrahend is the number of received packets before the transmission. For easier calculation a little script in Listing 3.3 is used. It is expected to run this script before any measurements, afterwards it should be run after each separate measurement.

Packets transmitted (P_{tx}) is the number of packets transmitted by the specified device/s and it is expected the number is either gathered from the statistics of the transmitting NIC or gathered from the number of packets contained in the PCAP file. Calculation is therefore very similar to Equation 3.2. It is expected that measurements run in an isolated environment where packets of sources different from the specified transmit device/s are not present.

$$BO[\%] = \frac{100 * P_{rx}}{P_{tx}} \tag{3.1}$$

$$P_{rx} = rx\_stop - rx\_start \tag{3.2}$$

```
expr $(ethtool -S eth4 | grep rx_packets: | grep -o "[0-9]*") - $ETHLAST;
ETHLAST=$(ethtool -S eth4 | grep rx_packets: | grep -o "[0-9]*");
```

Listing 3.3: Bash script for getting P_{rx} after separate measurements.

3.3 Testing framework

To reach the final configuration, many options need to be explored. Manual testing can be rather labor-intensive and repetitive work for researchers. It also opens a possibility to introduce a human error in measurements. To eliminate these problems, automated testing is viewed as a better approach. Section introduces the proposed architecture of the testing with the design of the testing framework.

For tests to be replicable, it is essential to replay the same traffic. For this reason, tuning process can not be just tapped into the live network. For consistent results, a large PCAP file is recorded from the live traffic and that is used in all tests. This ensures that configurations are measured under the usual traffic load that goes through the tapped interface.

3.3.1 Architecture

High-level architecture in Figure 3.4 shows how the control station connects to the servers and executes measurements. Connections to the machines are made over Secure Shell (SSH) and Secure Copy Protocol (SCP). Commands to control the servers are executed over SSH and files are transferred over SCP. Control station can be any computer that has access to these servers and has a Bash shell.

Internal framework structure in Figure 3.4 shows modular design with a configuration file. Certain modules (e.g., transmit module) can be replaced or reconfigured in case the hardware architecture is altered. This enables flexibility and removes dependencies in testing. Individual components are implemented in Bash scripting language.

Sequence diagram in Figure 3.5 shows a proposed testing scenario. Control station starts the Suricata instance on Claret and then after Suricata warm-up time it connects to Pinot, reconfigures and starts the transmission of packets with additional parameters as transmission speed and a specified PCAP file. Upon finish, Pinot transmit log is transferred to the control station and similarly, control station stops Suricata on Claret and pulls statistics. The downloaded files are parsed and data are stored in a file of comma-separated values (CSV). The results can later be processed with graph imaging tool to visualize and merge several measurements together into one graph.

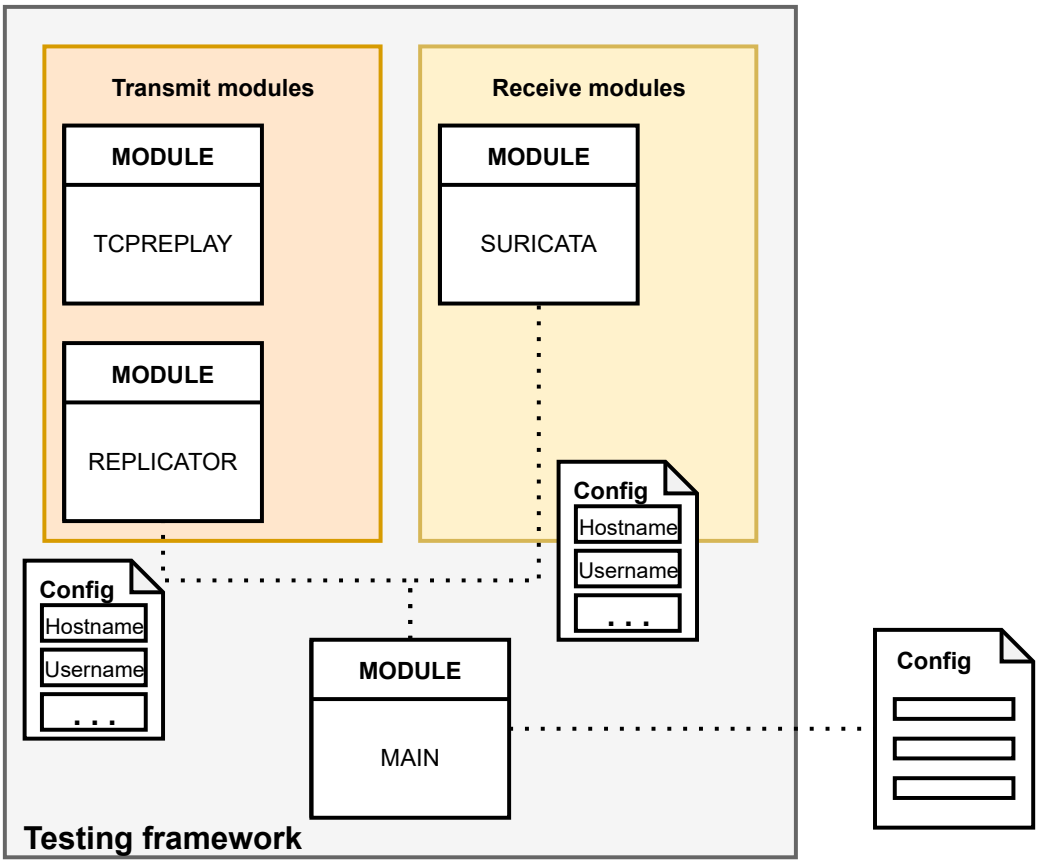

Figure 3.4: Internal structure of the testing framework.

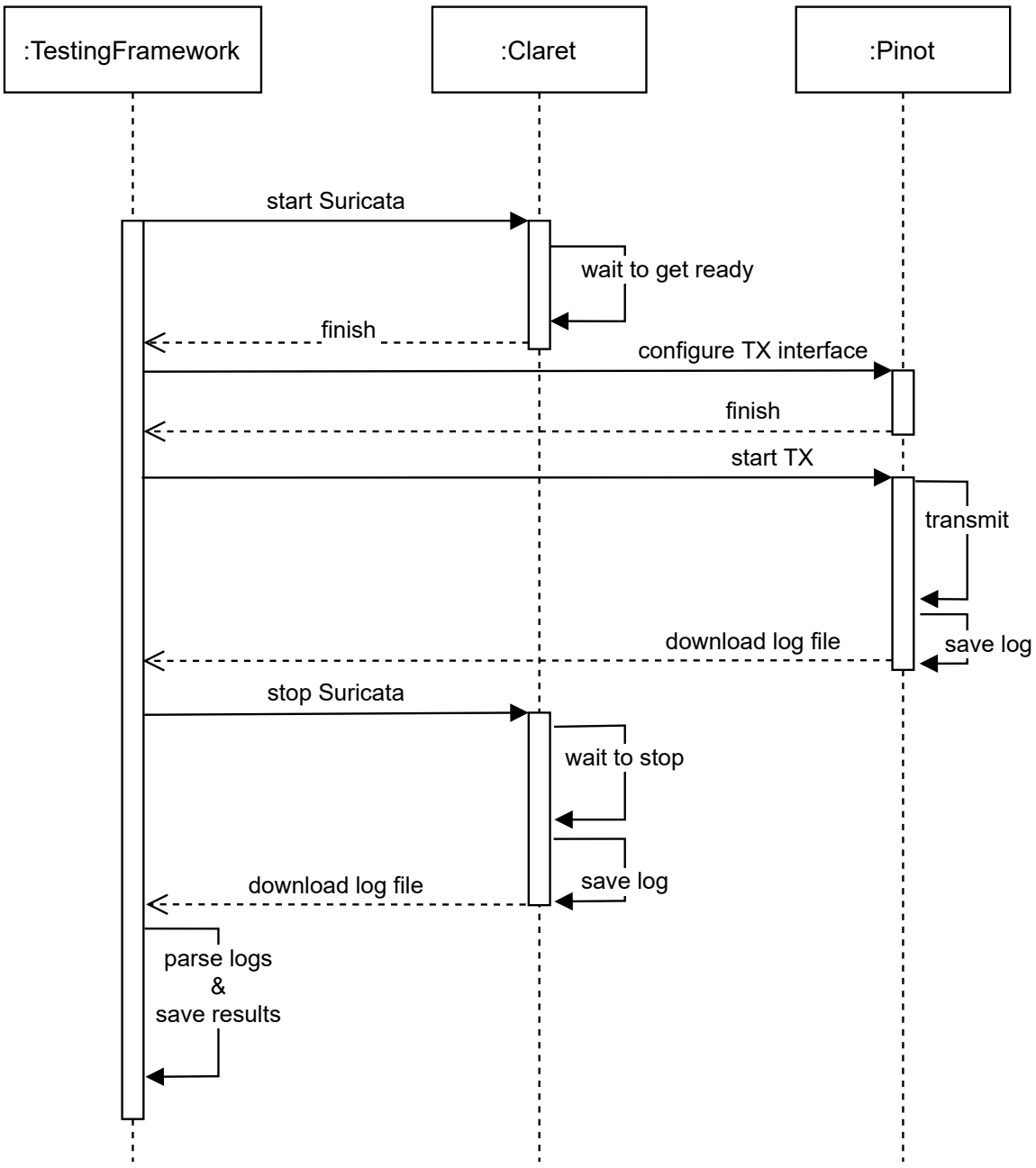

Figure 3.5: Sequence diagram of the usual testing scenario.

3.3.2 Replicator

Replicator, as mentioned in the earlier paragraphs, is a custom made FPGA network card. It resides on Pinot, from where packets are transmitted to Claret as shown in Figure 3.1. Broad variety of options for packet transmission is one of the main advantage of the replicator. Out of the most highlighted ones are ability to:

- replay the same PCAP file over and over in a loop,
- set number of loops (infinite is also an option),
- set the transmission speed,
- set the number of replications (up to 10),
- modify the individual replications.

The last two points are making the replicator very special as they allow to amplify the bandwidth of transmitted traffic up to 10 times where a single replication can transmit up to 10 Gbps. As noted, individual replications can be modified. For instance, that includes change in source or destination MAC, IPv4 or IPv6 address. The basic principle is shown in Figure 3.6 where incoming packet is processed and individual replications change the value of packet properties. This not only results in higher bandwidth but it also creates more network flows. This behavior better simulates conditions of heavy loaded network.

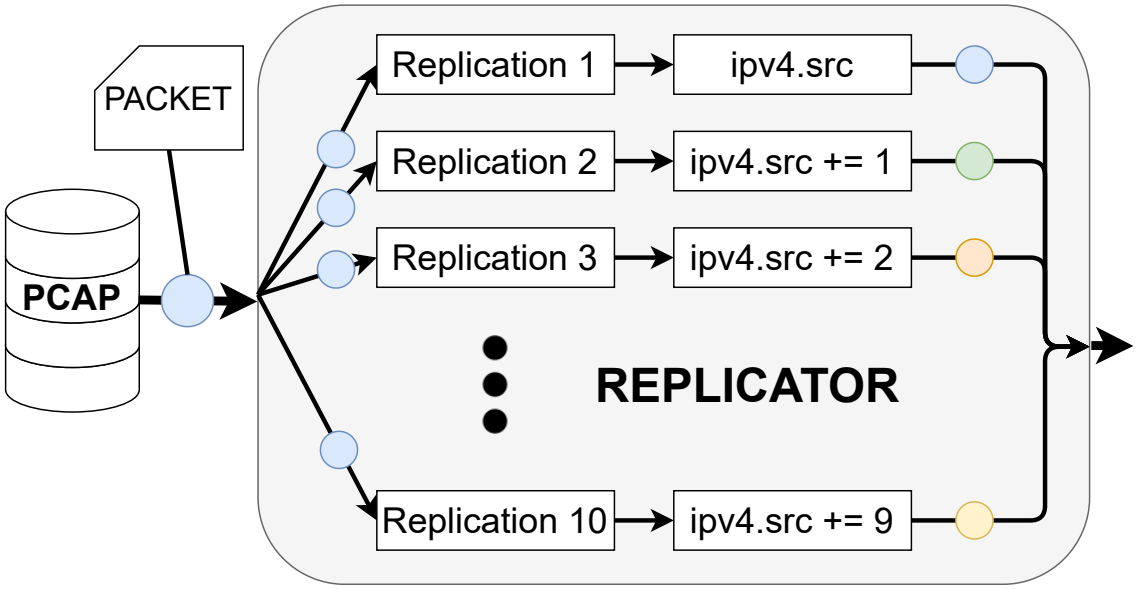

Figure 3.6: Replicator – transmission of a multiplied packet.

3.3.3 PCAP

For repeatable experiments it is essential to have a stable environment and an ability to make changes independently step by step. In the proposed architecture of the testbed (Figure 3.1) there are 2 components – replicator on Pinot and Suricata on Claret. Change in any of these can affect results of measurements. However, it is unlikely that change

occurs on Pinot as this machine is rated for sufficient throughput of 100 Gbps. Thus, the only point where change is expected and anticipated is Claret.

Except stable connection between devices, it is also crucial to have easily repeatable traffic. This problem can be easily solved by capturing live traffic on a selected network and then storing it into PCAP file. In comparison to the traffic created artificially (i.e. using Scapy[3]), it has a benefit of being more authentic. PCAP files are data files that contain individual packets captured from the live network. These files can be used for later analysis or, as in this case, for controlled retransmission.

Listing 3.4 shows preliminary analysis of the PCAP file used in all later measurements. Packets were captured from a live network and therefore the PCAP file consists of mixed traffic. However, the traffic should not contain too many malicious activities since it is an internal network. As described in Subsection 3.3.2, replicator is able to set arbitrary transmission speed. Therefore, data bit rate of the PCAP causes no problem for retransmission of the PCAP. The data size and amount of packets captured in the PCAP ensure there are enough data to replay.

To have a better picture of what is included in the PCAP, it is better to make an analysis of possible threats captured in the file. This helps us evaluate how successful Suricata is in dealing with these threats. Analysis provides a set of individually generated alerts that are grouped together to get their sum. Listing B.1 shows all alerts generated by Suricata after PCAP was transmitted from Pinot from one replication. All rules that are used in later measurements were enabled. Retransmission was done with one replication and was fully successful with all packets received on Claret. From the generated alerts, it can be noted that the traffic is not malicious, most of the alerts are connected with invalid packets of TCP connection.

```
File encapsulation:  Ethernet
Packet size limit:   file hdr: 65535 bytes
Number of packets:   509 M
File size:           393 GB
Data size:           385 GB
Capture duration:    1166 seconds
Data byte rate:      330 MBps
Data bit rate:       2 642 Mbps
Average packet size: 756,18 bytes
Average packet rate: 436 kpackets/sec
```

Listing 3.4: PCAP analysis by `capinfos`.

For PCAP analysis, it is suitable to use offline Suricata, as shown in Listing 3.5 to prevent any problems with packet transmission. In offline mode, Suricata only reads the PCAP file and raises alerts if a rule is matched. Configured outputs are used to store the information from analysis. The command on the bottom in Listing 3.5 extracts alert messages from *fast.log* file, sorts them and groups them. The analysis report can be used later to compare with Suricata outputs.

```
suricata -r ~/pcap/testdata.pcap
cut -c 48- fast.log | grep -Eo ".*}" | grep -Eo ".*\[\*\*\]" | sort | uniq -c
```

Listing 3.5: PCAP analysis by Suricata.

However, since replicator uses replications in a way as explained in 3.3.2, it creates new traffic flows by adjusting certain properties of packets. To see how Suricata reacts to this

[3] https://scapy.net/

kind of traffic, further analysis was done by using 4 replications. Packets were transmitted at speed 900 Mbps per replication. Purpose of the second analysis was to test theory that the number of alerts generated by 4 transmitting replications is equal to the quadrupled number of alerts generated by 1 replication. With 4 replications enabled, one replication sends the original PCAP and the others send altered PCAP (changing bit in source IP address). The Listing B.2 shows results of the second analysis.

From the analysis, it is possible to obtain that count of alerts have totally disproved the theory. Therefore an increase in the number of replications does not dramatically increase count of generated alerts. In some cases (e.g. `ET POLICY` rule) the number of alerts have quadrupled. However, majority of generated alerts do not follow this pattern. This assumption was therefore dismissed. The results of this comparative analysis does not affect future measurements. Suricata can still be under a heavy load of incoming packets. Suricata's goal is to inspect all incoming packets.

It is also possible to see that majority of alerts are from Suricata rulesets, two alerts are from Emerging Threats rulesets.

3.4 Performance tuning of AF_PACKET

Implementing any kind of optimization without a possibility to compare with already existing solutions would not tell much. As discussed in sections before, it is crucial to have a solid and stable testing environment to move forward any settings in current solutions. Similarly, it is necessary to evaluate newly proposed design step by step whether implementation is on the right track and is competent enough compared to other existing solutions.

This section present steps taken to achieve the best results that were captured with the current hardware. To decide what configuration suits the best the designed network architecture, it is vital to iteratively perform various tests of the deployed Suricata instance. As stated before, tuning the Suricata is a complex process and has no generic settings for the top performance. The process consists of an evaluation of results gathered from tests of Suricata loaded under various configurations. Group of steps are then coupled with measurements that are displayed on the graphs. Individual graphs can serve as a baseline results for a proposed optimization.

3.4.1 Performance measuring tools

Configuring Suricata correctly is usually a lengthy process. Knowing its architecture helps but since it is a very complex piece of software, it is often hard to understand. Even then it does not guarantee fast configuration process. Another (*„machine learning like“*) approach would be to take Suricata almost as a black box that takes initial configuration file, starts processing packets and then according to the feedback, adjust individual settings of the configuration file and repeat. It is obvious this approach would not be feasible in human version. The usual way of configuration is somewhere in the middle – little bit of architectural knowledge combined with trial-and-error approach with focus onto certain areas of the configuration file. Certain settings can be generally applied for better performance but some must be tuned per environment where Suricata is deployed (e.g. hardware Suricata is running on or popular type of traffic that occurs on the network). Getting feedback in form of received/dropped packets would still leave us blind

in many areas. Therefore, this subsection presents performance measuring/tuning tools to gather valuable information and uncover details of these areas.

Performance Counters for Linux (perf)

Perf[4] is a popular performance analysis tool among Linux developers to see at what places spends the running program the most time. It can detect possible bottlenecks or places that can be improved. It has been initially released in 2009 and it is still widely used today. To report actual performance usage, it is based on event counting statistical methods. It has various sub-commands such as:

- **stat** – measures events statistics of one specific command/program,
- **top** – dynamically updated list of top used (hot) functions (Figure 3.7),
- **record** – record and save sampling data of the program to a file (to be processed later),
- **report** – reads a file of sampling data and generates different kinds of reports.

Figure 3.7 shows usage of sub-command top (`perf top`). On the top, in the blue-colored frame, we can see details about event sampling. The second line of the output describes of what individual rows in the list is composed of. Below, the most used functions are listed in descending order according to its overhead percentage. Color of overhead percentage also gives a clue to how serious the usage is. Starting from the bottom – no to low usage is white, low to medium usage is green and medium to high usage is red. Symbol hints what part of the program uses the most of the CPU time.

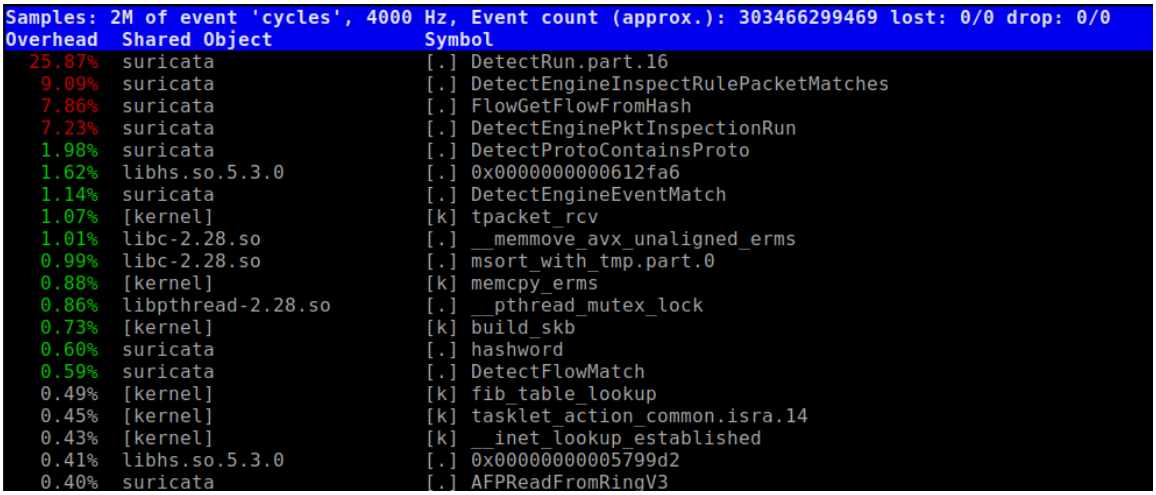

Figure 3.7: Screenshot of running `perf top`.

htop

Htop[5] (shown in Figure 3.8) as an interactive performance monitor/process manager is the *Swiss army knife* of Linux administrators for diagnosing problems of the server. It is

[4] https://man7.org/linux/man-pages/man1/perf.1.html
[5] https://man7.org/linux/man-pages/man1/htop.1.html

well known tool among Linux users. As seen in Figure 3.8 the first part of the output represents a load of individual processor's cores between the squared brackets. In this case, there are 40 cores (physical and logical cores are combined) with a significant load on the odd numbers since Suricata is set to run only on NUMA node 0. Representation of core numbers with respect to NUMA nodes location differs between the operating systems. Load in the squared brackets is divided in two colors – red and green. Red color denotes load of kernel related tasks (e.g. interrupt handling) and green represents load from user space applications (e.g. packet inspection in Suricata).
How much memory is left can be viewed between the squared brackets at lines denoted as `Mem` and `Swp`. Label `Mem` represents a load of Random Access Memory (RAM) whereas `Swp` stands for additional Swap memory.
The green-colored line denotes descriptions of individual columns. The list of rows described individual processes that are run on the machine with respective usage of resources.

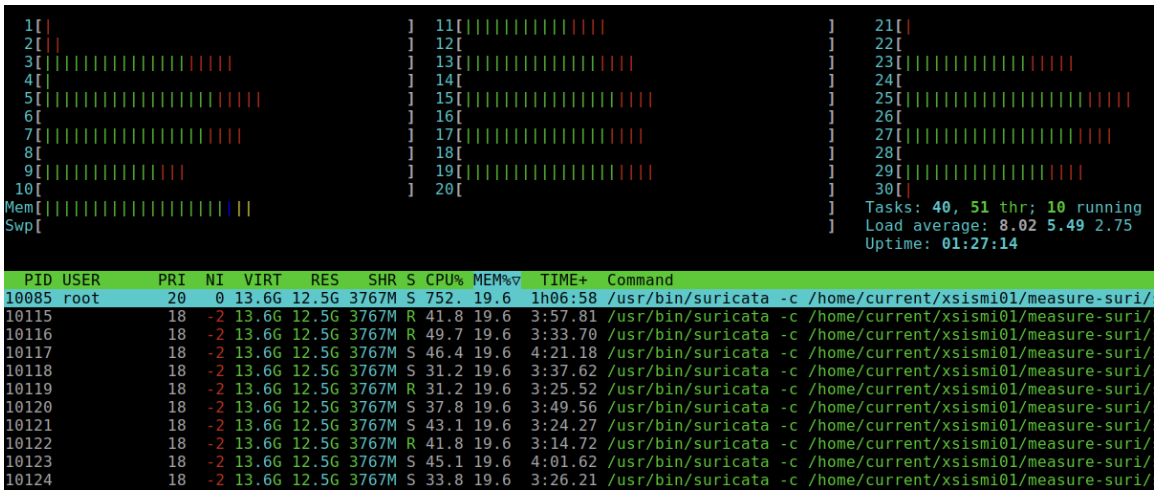

Figure 3.8: Example of running `htop`.

Intel Performance Counter Monitor (pcm)

Similarly to previously mentioned tools, Intel PCM[6] is a performance monitor that provides deeper insight into how individual components of Intel platform are busy. In the first section of Figure 3.9, individual cores of available processors are displayed along with respective statistics to each core. When performance tuning is a goal, it is useful to look at columns IPC and L3HIT. Under the core list there are averaged statistics/counters per processor socket (NUMA node).

IPC stands for Instruction per Cycle and it is equal to the average number of executed instructions per CPU cycle. If IPC counter equals to 4, it means the core is 100% utilized. Because of Hyperthreading[7], on 1 physical core resides 2 logical cores. As a result their IPC counters are summed when calculating actual IPC on physical core.

[6] https://software.intel.com/content/www/us/en/develop/articles/intel-performance-counter-monitor.html

[7] https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

L3HIT is a ratio between successful and failed lookups in L3 cache. Cache in the CPU design serves as a temporary storage of data that CPU might use later. There are different levels of cache implemented. Ranging from L1 up to L3 cache, each level provides different levels of access times (the lowest is the fastest) and different sizes (the highest is the biggest). Data that does not fit into these caches are pushed into operating memory. Fetching the data from operating memory requires huge amount of time compared to Lx cache of the processor. For this reason, it is in the best intentions to keep majority of the data in cache. However, because of the great amount of traffic and limited size of Lx caches it is not possible to have 100% hit rate. It is indeed favored to be as close as possible.

Under the core list we can find distribution of cores C-states[8]. Cores in Intel architecture have different power and performance strategies. These strategies are called C-states and go from C0 (the most performant, the least power saving) through C1 up to C6 (the least performant, the most power saving). As a limited number of cores can run in C-state C0, other cores can run at C-state C1. In Figure 3.9 it is possible to notice that half of the cores run in C0/C1 (cores of the socket where Suricata is running) and the other half is in power saving mode (the other NUMA node).

The next section shows statistics of Intel Ultra Path Interconnect[9] (UPI) which is a successor of Intel Quick Path Interconnect (QPI). There are two sockets displayed in Figure 3.9 with 2 UPI links with close to no data transfers between the sockets. Bandwidth of respective links in absolute and relative numbers is displayed in the relevant cells.

The very last section shows counters for the main operating memory controller and individual processor sockets. READ/WRITE columns denote the amount of gigabytes (GB) that are transferred from or to the main memory controller. Column LOCAL shows ratio of memory request to local memory controller.

[8] https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/energy-analysis-metrics-reference/c-state.html

[9] https://en.wikipedia.org/wiki/Intel_Ultra_Path_Interconnect

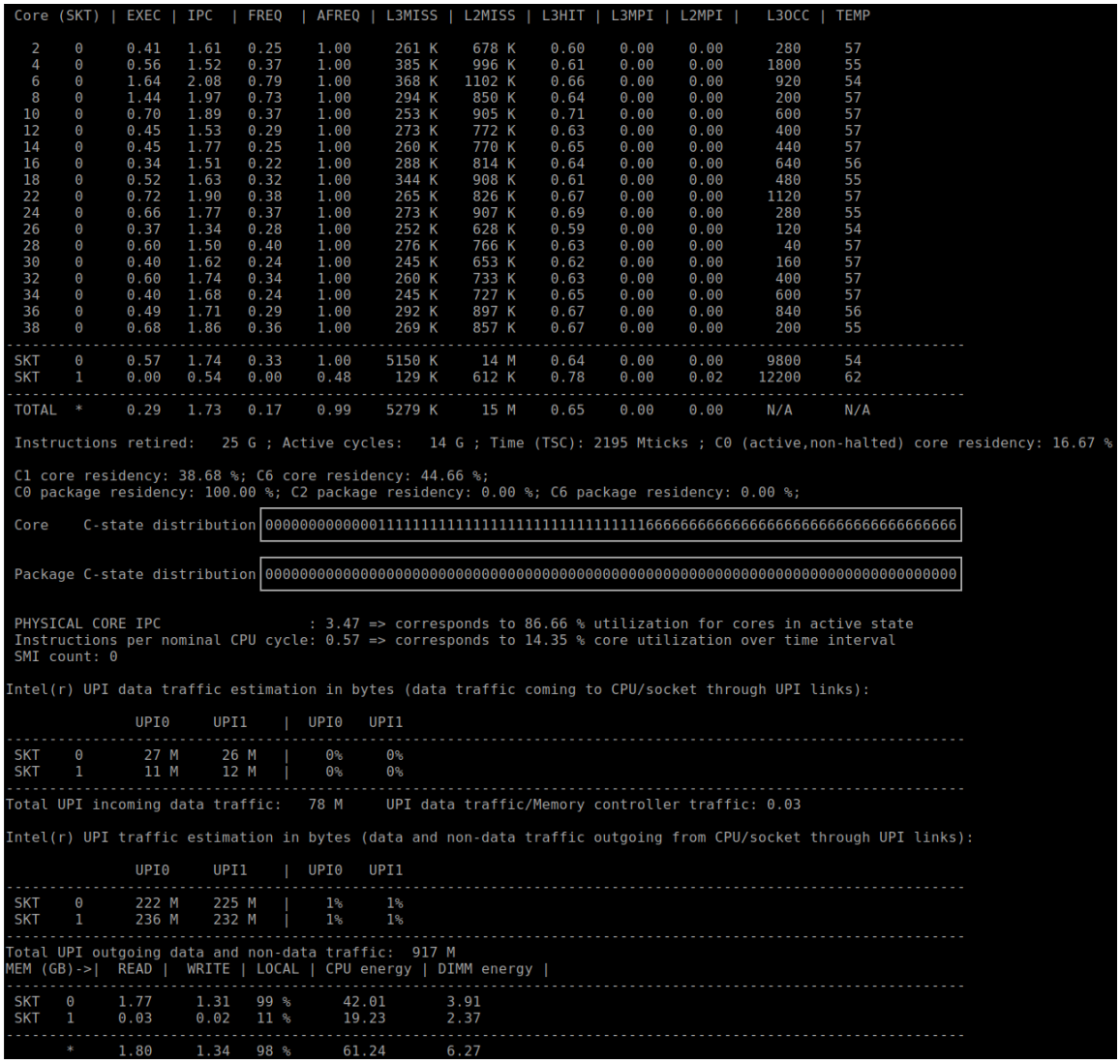

Figure 3.9: Screenshot of `pcm`.

3.4.2 Pre-baseline measurements

In the beginning, setting a goal that is not too hard to grasp was crucial. As mentioned earlier, it is important to gradually move forward. The goal was to measure buffer overflow of Suricata with AF_PACKET capture interface and **with no rules applied**. For the later measurements, the results might be considered as potential top performance for Suricata (although it certainly is not taken as a fact) as enabling rules degrade performance of Suricata. However, the main point was to test functionality and reliability of the testing framework architecture. It also proved that PCAPs acquired for the replicated tests and other limitations of hardware/software cause no problems.

At this stage, progress was made in iterative steps, starting with successful packet transmission and reception. Even with no rules enabled, Suricata handles and decodes all incoming packets. In these tests, only the detection module was not tested. It continued with picking the seemingly best NIC configuration according to `ethtool` statistics mentioned in Subsection 3.2.7. After many experiments, configuration that can be found in Listing A.1 was used. This configuration remained through all measurements of AF_PACKET runmode. Section 3.2 explains individual settings in more detail. All measurements were performed with the testing framework mentioned in Section 3.3. As rules were disabled, some already implemented optimization techniques such as Hyperscan or XDP were not yet applicable as the performance enhancement is linked to more efficient ruleset usage. During this process, version of Suricata together with a version of operating system got fixed to values mentioned in Section 3.1.

AF_PACKET results of the tuned configuration can be seen in Figure 3.10. In the figure, it is possible to observe that packet reception is consistently reliable up until 22 Gbps with buffer overflow from 0 to 1 percent. All three variants of measurements received 100% packets in sub-20 Gbps receive speeds. After that, the number of dropped packets starts to increase. It is possible to notice that 8 replications (8 times more flows than in the original PCAP file) puts more stress on Suricata than 4 replications at the same transfer rates. As the goal of this test was to ensure all machines are set up correctly, there was enough data to stop the measurements. It was certain Suricata handles traffic very well up until 22 Gbps. It was possible to enable rules and start measurements.

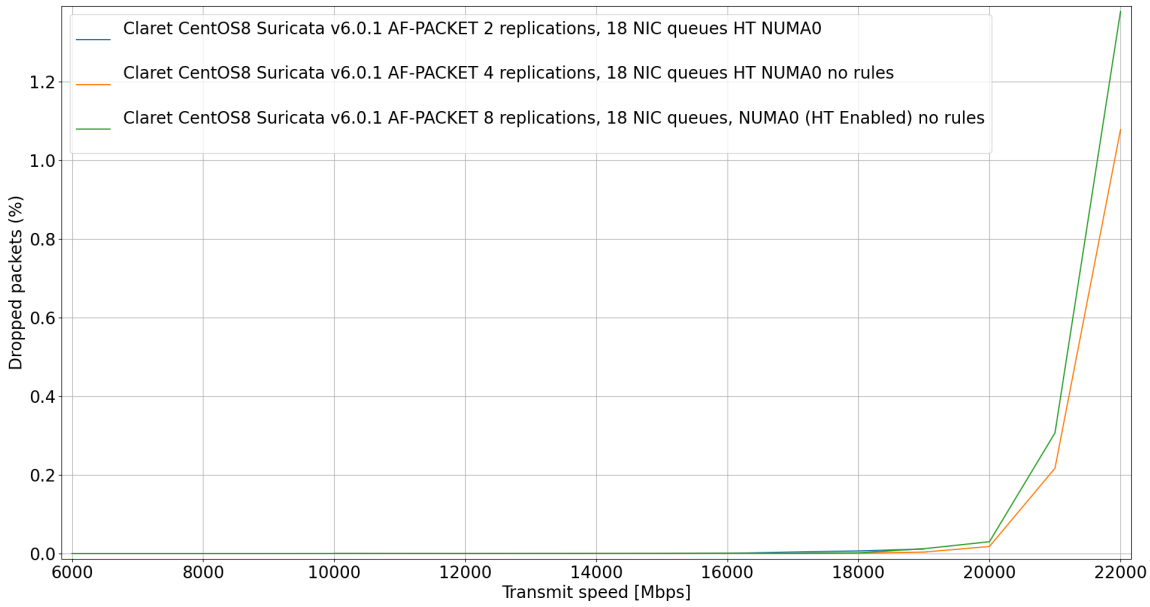

Figure 3.10: Optimized AF_PACKET capture interface with no rules enabled.

3.4.3 Baseline measurements

After the initial measurements and tests, it was possible to set a new goal – **measure Suricata buffer overflow with rules enabled**. During this stage it was important to establish a set of rules (ruleset) and use it throughout all measurements. Performance could vary due to the always changing set of Emerging Threats rules. This could further cause imprecise measurements because of a different number of rules enabled. Chosen ruleset stored in *suricata.rules* was added to the source control system of the testing framework to be easily accessible for future installations.

As described in the previous subsection, the configuration file was updated in iterative steps according to the feedback in the form of dropped packets. With the help of profiling tools described in Subsection 3.4.1 it was possible to reach great results even on the available hardware described in Figure 3.1. Figure 3.11 represents Suricata performance with traffic sent from 4 replications. NUMA node 0 had 18 cores allocated for Suricata worker threads and 2 cores for management threads. NIC was configured with commands of Listing A.1 – 18 combined queues, 10ms interrupt coalescing, enabled RSS. Suricata runs in `workers` mode and AF_PACKET is set to `cluster_qm` mode. Cluster_qm mode binds RSS queues with individual CPU cores in one-to-one relationship. Packets from 1 queue always lands on the 1 bound CPU core. Therefore, RSS also acts as a load balancer. 1 core inspects packets from both directions because of the special symmetric RSS hash key (Listing 3.1). From the output of PCM (Subsection 3.4.1) displayed in Figure 3.9 it was possible to observe IPC (Instruction Per Cycle) hovering around 3.3 instructions per cycle (topping at 3.5 IPC). This means Suricata is greatly tuned and even surpasses IPC of 2.7 from Suricata Extreme Performance Tuning (SEPTun) [13]. L3HIT rate was around 65% which is still very good result at this amount of traffic. Inter-processor communication (UPI) was almost zero and in 99% of memory requests local NUMA memory controller was able to provide the requested data.

Hyperscan was also enabled to speedup rule matching. eBPF filter program was also

available. If needed, it was hooked to the kernel via XDP. In case flow shunting is enabled, it stops packet inspection after 20 packets of the given flow. Suricata uses special flow rules to count number of packets of the given flow. When the number of packets reaches the limit (20), a bypass rule is activated. The flow is then bypassed at the driver level.

Graph in Figure 3.11 displays three different measurements. Blue line is tuned Suricata with previously mentioned settings and finely tuned suricata.yaml configuration file. Orange line had additionally enabled XDP bypass as described in Subsections 2.4.2 and 2.5. Green line had enabled both XDP bypass and flow shunting – also described in Subsection 2.5. Classic Suricata (blue line) handles traffic up to 6 Gbps. XDP located in the driver helps to push performance even further to almost 7.5 Gbps. Emerging Threats rules do not use bypass very often. Dropping packets had the same characteristics in both versions. On the other hand, aggressive flow shunting method proves very high packet throughput – reaching up to 18.5 Gbps. Higher receive speeds results in a slight increase in packet drop.

Graph in Figure 3.12 proves the same packet processing behavior of Suricata happens even if Suricata is under a load of different number of replications. It can handle very well even high amounts of flows. These results were achieved after careful tuning of Suricata over time. Number of different tuning paths were explored. These results are reference results and new capture interfaces or other Suricata enhancements can be compared with these performance graphs. Alternatively, it is also possible to reuse the configuration files for later measurements.

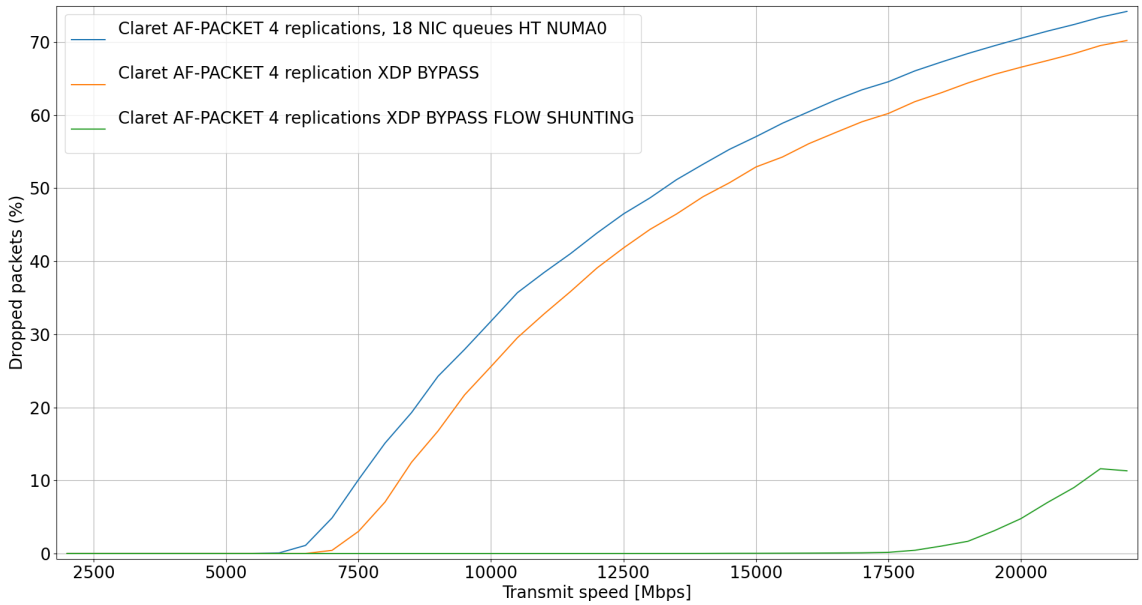

Figure 3.11: AF_PACKET capture interface with rules enabled.

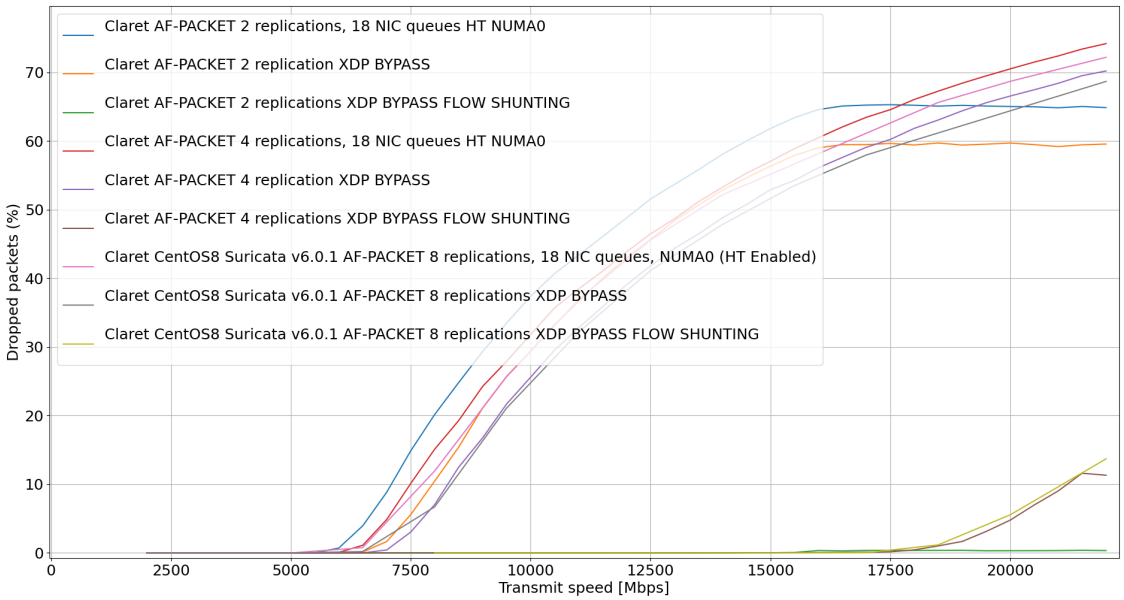

Figure 3.12: AF_PACKET capture interface with rules enabled with tested under different number of replications.

Chapter 4

Design of the proposed optimization

To figure out how to improve system's performance, it is crucial to use a set of suitable tools for such tasks. Some of these tools were presented in Subsection 3.4.1. In each application, there are always multiple places to optimize. When using such tools, it is possible to pinpoint the biggest bottlenecks of the application. After choosing a single bottleneck to focus on, investigation of the problem follows. This process results either in elimination of the bottleneck or inability to overcome the selected bottleneck. However, the former option happens much more often. In most cases, a quick ad hoc solution is to add more resources. This might be sometimes impossible and, among other things, it implies higher operation costs of such systems.

The problem of finding the root cause of the bottleneck and implementing a solution to eliminate it can look often more challenging upfront but it can easily pay off in the future. It requires deep understanding of not only the application but also the stack the application is running on and technology that it uses. The whole optimization process can be longer than, e.g. increasing available resources. As a result, the upfront costs can be higher. Removing the root cause of the bottleneck can pay off in the long run. It might imply that it lowers service expenses and speeds up processing/response time.

As a result, the former approach is better suited for short fixes or onetime operations such as data migration. The direct mitigation of the bottleneck can therefore be more worthy in the long running applications. An example can be Suricata. Instances of Suricata deployed in production operate continuously and uninterrupted for 24/7.

Therefore, individual modules of Suricata should be highly optimized and use as much resources as possible to provide the best performance. Suricata already tries to do its part well by optimizing individual modules. For example, Suricata uses a high-level of parallelism thanks to being a multi-threaded program – this was more elaborated in Section 2.3. Another popular IDS/IPS application – Snort[1] uses for detection a single thread. It is therefore limited by the single-core performance of the CPU (to scale up to 100 Gbps it would require 250 cores [17]). In detection module, Suricata additionally uses an open-source, high-performance, regular expression-matching library from Intel described in Subsection 2.2.3. The library improves detection in Suricata instances with large rulesets thanks to SIMD instructions. Suricata also uses various optimized capture

[1] https://www.snort.org/

interfaces either implemented by either Suricata developers or other companies. These capture interfaces were more described in Section 2.4.

4.1 Motivation

Previous chapters introduce topics such as Suricata architecture or various capture interfaces. It is first required to fully comprehend all of them, to be able to successfully move towards any optimization of Suricata. As Suricata is open-source and public, there have been efforts to speed certain modules of Suricata. These attempts were not always successful as some of them might not have yielded the best results or have been abandoned over time. DPDK capture interface or GPU (CUDA) acceleration are examples of such failed efforts.

After tuning of settings of Suricata with AF_PACKET capture interface, I performed an analysis of Suricata performance with tools mentioned in Subsection 3.4.1. High load of kernel tasks (red-colored ticks) shown in Figure 4.1 has sparked my attention and I decided to inspect it a little further. These kernel calls are attributed to interrupt handling. Interrupts originate in AF_PACKET capture interface as a way to signal kernel to handle packets from the NIC. This can result in a load as high as 20% of the total application load. Among other things, this analysis has helped to set a proposal for Suricata optimization for this master thesis. I will attempt to lower the load of capture interface by implementing a new one – based on the DPDK framework.

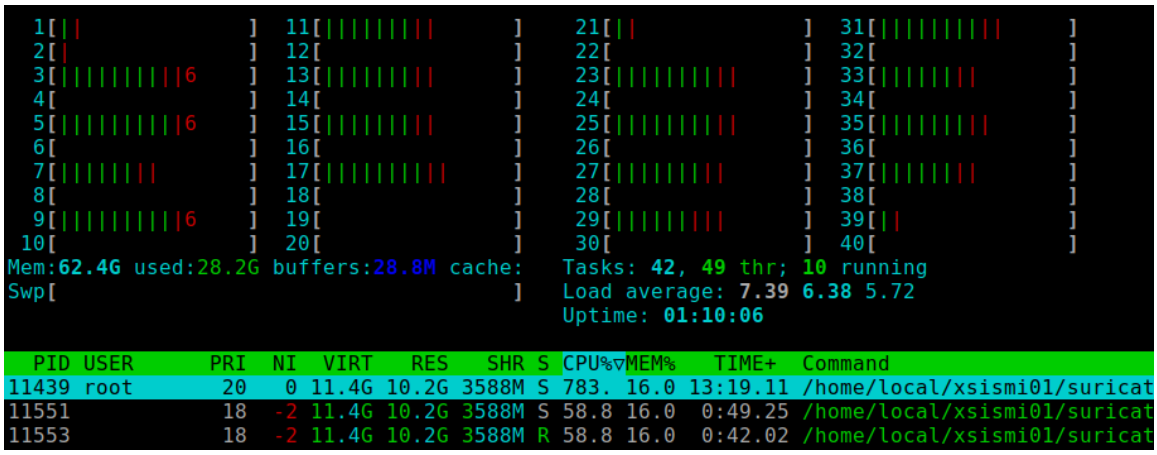

Figure 4.1: Profiling Suricata in htop (AF_PACKET capture interface enabled).

4.2 Analysis of the current implementation

As mentioned earlier, there have been attempts to implement DPDK interface into Suricata. It was a rejected pull request[2] and a bachelor thesis of a student[9].

The author of the previously mentioned pull request is maintaining the implementation in a new separate branch[3]. However, this implementation is based on Suricata 4.1.4 and

[2] https://github.com/OISF/suricata/pull/4902
[3] https://github.com/vipinpv85/DPDK_SURICATA-4_1_1

uses DPDK of version 19.11.3. Additionally, lack of documentation contributed to closing the pull request. It is in my best effort to follow Suricata design and code guidelines for a successful integration. With thorough documentation it might be used by a broader audience.

The bachelor thesis successfully proved the possibility of implementation of DPDK into the Suricata and Suricata integration within the DDoS Protector (ex DCPro)[4] system. However, it lacks certain attributes of performance-oriented and maintainable capture interface. With results of the mentioned thesis, I was unable to select capture interface other than DPDK. Although the implementation provides a way to configure Environment Abstraction Layer (EAL), it is not possible to adjust parameters of other things related to DPDK such as packet mempool size.

Current implementation also uses packet copy, which, if avoided, can bring further performance benefits. It also follows old architecture of IPS with a verdict module. This is explained in more detail in the next section. The new architecture should leave responsibility of releasing and transmitting a packet to Suricata release function.

The listed problems, among other things, increased the motivation for the new implementation of DPDK capture interface.

4.3 Idea proposal

As mentioned in the introduction of this chapter, to successfully design an optimization for Suricata, deep understanding of topics presented in the previous chapters are required. Knowledge of Suricata architecture presented in Section 2.1 is crucial to implement a new capture interface. Additionally, insight acquired from Subsection 2.4.5 can aid reader in comprehending the presented proposal.

Suricata architecture, as shown in Figure 2.4 consist of 4 main thread modules. These are described in more detail in Section 2.1. Figure 2.4 presents a high-level overview that not always reflects the actual implementation of the individual modules.

Listing 4.1 is an extract from the main configuration file that contains the **threading** part of the configuration. From there, it is possible to notice 4 main CPU sets of threads. Names of the thread groups expose how the Suricata is implemented internally. Individual thread groups serve for specific purposes. Figure 4.2 illustrates more in-depth architecture of Suricata. Individual parts are also explained in Suricata documentation[10].

```
threading:
  set-cpu-affinity: yes
  cpu-affinity:
    - management-cpu-set:
        cpu: [ 0, 2 ]
    - receive-cpu-set:
        cpu: [ 4, 6 ]
    - worker-cpu-set:
        cpu: [ 8, 10, 12, 14 ]
    - verdict-cpu-set:
        cpu: [ 4, 6 ]
```

Listing 4.1: Threading part of suricata.yaml configuration file.

[4] https://www.liberouter.org/technologies/ddos-protector/

Incoming data in Figure 4.2 are first passed to the receive CPU set. This thread group handles data reception and decodes it to the format Suricata can understand. Packets decoded from this group serve as an input for the next thread group. They are put in packet pools to be processed by worker threads. Worker CPU set handles, for example, reassembling TCP streams, defragmentation, detection or logging. Packets leaving the worker thread can either be handed over to the verdict CPU set (in case of IPS) or freed from Suricata memory (in case of IDS). Verdict threads receives packet and according to the information in the packet structure it executes the desired action. These verdict threads can either drop packet or transmit it via a configured NIC.

Management CPU set takes mainly care of flow tables that are also used in the detection process. These threads access memory of all threads and thus are not directly incorporated in the processing pipeline.

Some capture interface like AF_PACKET simplifies the module architecture by using a callback function. This function is called by Suricata when a packet is being released from internal packet pools. The capture interface can implement not only release of the packet's data from the interface but also a transmit functionality to support IPS mode. Using this method, **Verdict** module can be removed from the pipeline shown in Figure 4.2.

The DPDK capture interface is designed to follow the presented architecture. It focuses on workers runmode, which is more described in Section 2.3. It essentially means that all modules are bundled together in one thread. This implies that one module is directly connected to the next module without switching between threads. Suricata then consists of many worker threads that avoid inter-thread communication.
Before individual modules are created, DPDK application needs to have environment abstraction layer initialized. In DPDK framework, this is handled by a main lcore. Sequence diagram in Figure 4.3 shows the process of DPDK initialization. It is expected that this lcore is separated from Suricata completely or that it is assigned to the management thread set. As Suricata starts, the main lcore initializes EAL. Packet processing/Worker threads are created later in the process of Suricata initialization. During spawning of these threads, separate NICs are configured with the loaded interface configuration. DPDK packet memory pools are created individually by each thread. These are filled with packets received by the NIC. Load is distributed to separate packet mempools by receive side scaling (RSS) of the NIC.

Decision tree diagram in Figure 4.4 presents how packets are received and forwarded from the NIC to Suricata. It consists of 1 main loop that detects when Suricata is stopped. Inside the main loop, pointers to individual packet `Mbufs` are from the NIC in an array. The size of `Mbuf` array is variable and depends on the size of packet burst received by the NIC. In the receive module, each packet in the `Mbuf` array is assigned to an internal packet structure returned from Suricata's packet queue. Certain parts of initialization of this packet structure are shared among all capture interfaces. DPDK capture interface then also sets a pointer to the individual `Mbuf` (stored in the DPDK memory pool) and a pointer to a packet release function.

Packet release function must be implemented in the receive module. Proposed idea of the function is illustrated by a diagram in Figure 4.5. In case of IPS mode, this function handles the packet according to the action set by Suricata. If the packet is flagged by **drop** action, packet is not forwarded further and is only released. In other cases, the packet is transmitted on the configured interface.
Regardless of the selected mode (IDS/IPS), the packet release function then returns `Mbuf` to the DPDK packet memory pool and also returns the packet structure to the Suricata

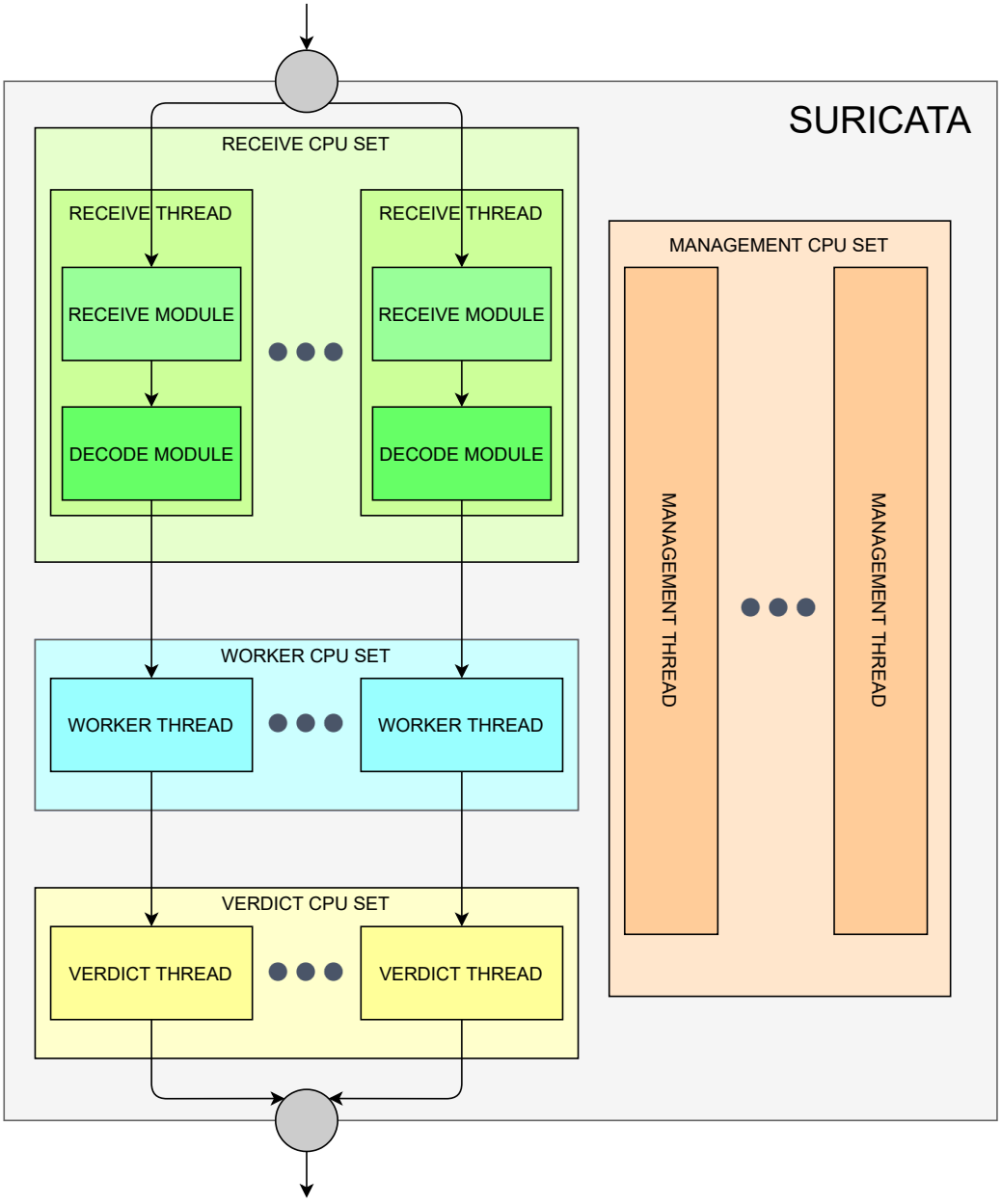

Figure 4.2: Suricata architecture through thread sets.

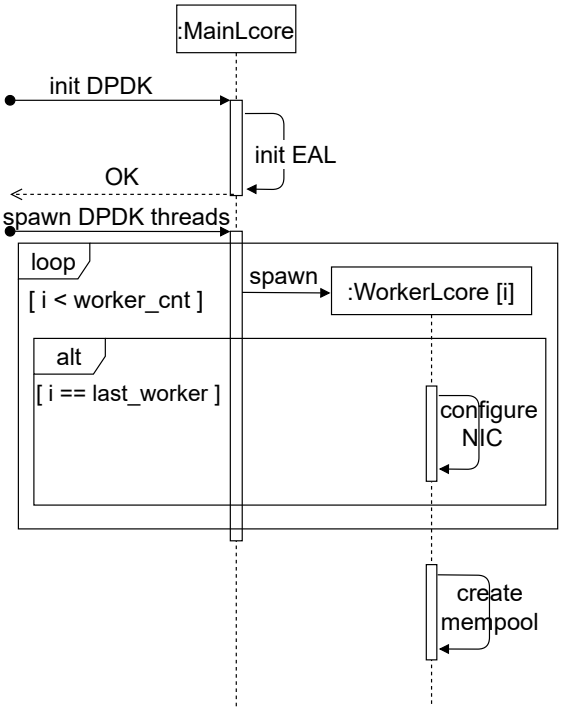

Figure 4.3: Initialization of DPDK capture interface.

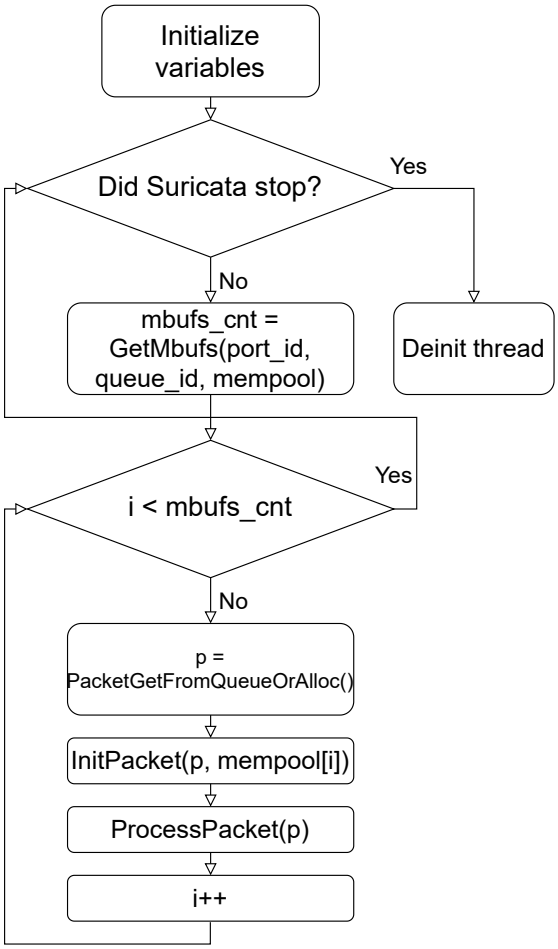

Figure 4.4: DPDK receive loop function.

internal packet queue. This allows DPDK to assign `Mbufs` to incoming packets, and it also allows Suricata to be able to handle new packets acquired by the DPDK receive function.

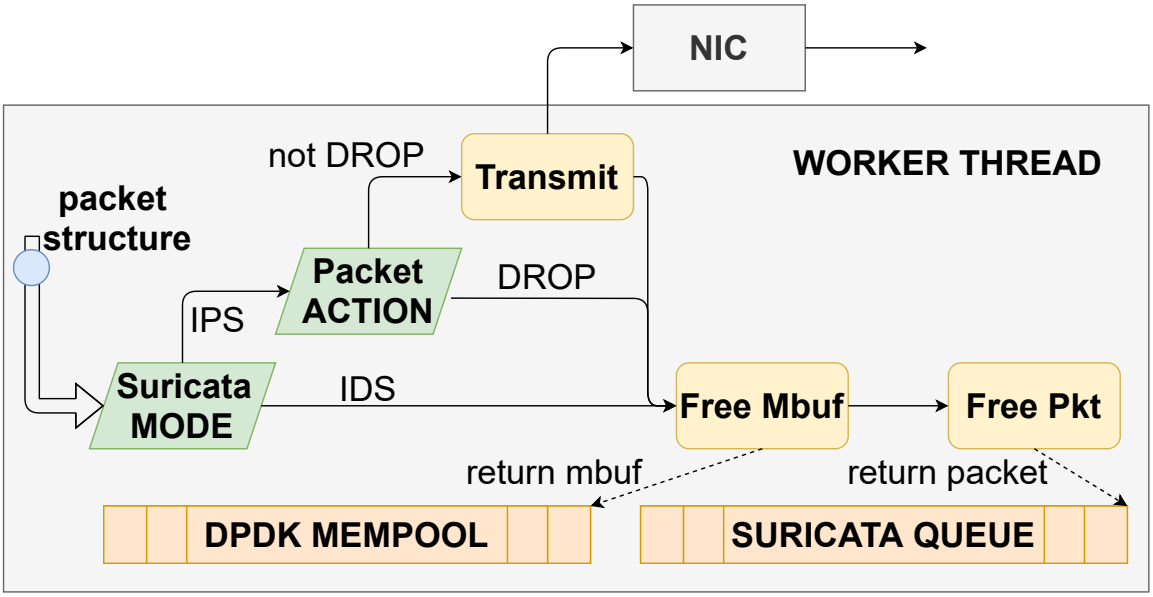

Figure 4.5: DPDK release function.

The diagram in Figure 4.6 presents connection of individual modules and components from the bigger picture. A packet received by the NIC is assigned to individual queue. Packets from this queue are fetched in burst and assigned to individual `Mbufs` stored in a DPDK memory pool. Array of `Mbufs` are then passed to `ReceiveLoop` function. The purpose of the function is to iterate over the array and for each received `Mbuf` request a packet structure from Suricata queue. When the packet structure is obtained, its attributes are initialized. It also stores a pointer to the `Mbuf` and thus also to packet data. After all attributes are set, packet structure is left to be processed by Suricata. Meanwhile `ReceiveLoop` function continues in the same process as described. Once Suricata executes detection on the packet, it calls packet's release function. If Suricata runs in an IPS mode and packet's determined action is not a drop action, the packet can be transmitted to the configured interface. The release function always releases `Mbuf` that packet structure was pointing to and the packet structure itself.

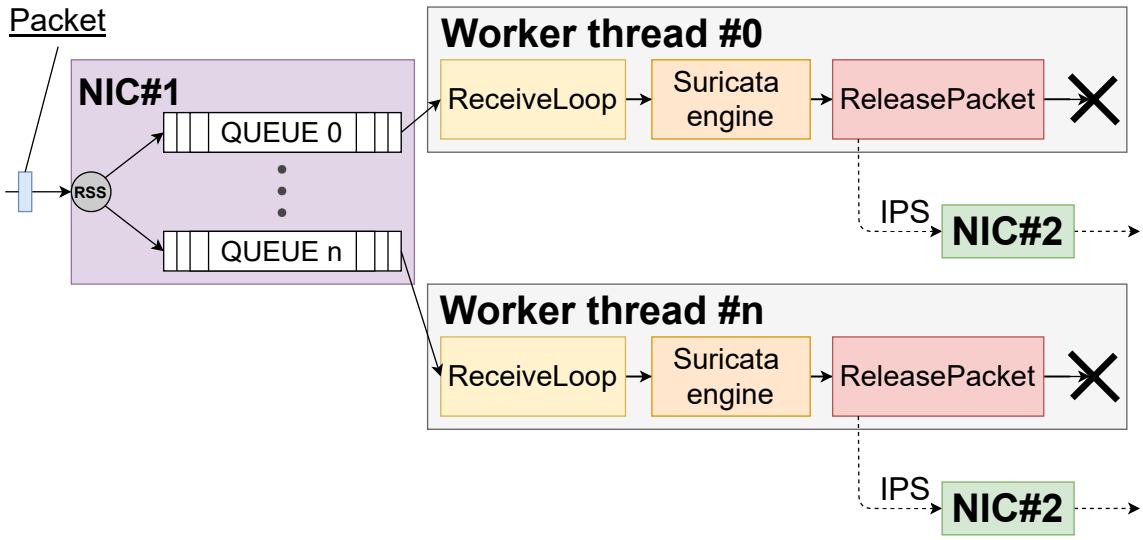

Figure 4.6: Packet lifecycle in Suricata.

Chapter 5

Implementation and benchmarks

Chapter 4 proposed a design how integration of DPDK capture interface could look like. Knowledge gained in the previous chapter serves as building blocks for further development. This chapter develops on the obtained insight in this topic and describes the process of implementation of DPDK capture interface into Suricata. Many references to previous sections prove the importance of presented information.

Section 5.1 describes a series of actions that had to be gradually implemented and combined into a working capture interface. It starts with including files to Suricata build system, then goes over to runmode registration and finally outline a process of implementing the core of Suricata capture interface. It might even serve as guide to implement a new packet capture interface.

Section 5.2 focuses primarily on benchmarking the newly implemented DPDK runnnig mode. It presents a list of tests that measures performance of DPDK and AF_PACKET and then compares the results. The individual measurements are always evaluated and accompanied with a brief summary. The measurements mostly focus on performance, where the main indicator is the rate of lost packets.

5.1 Implementation

5.1.1 Capture interface registration

Adding source files

Suricata code base is divided in different folders. Folders store files that share some common characteristics. They either combine source files that use the same programming language (e.g. `python/`, `rust/`) or serve one designated purpose (e.g. `doc/`, `qa/`). All Suricata C source files are stored a single folder named `src/`. Further hierarchy is achieved by source filenames prefixes (e.g. `detect-*` or `stream-*`). Even though structure of Suricata code base is mostly flat, it causes no problems in navigation between source files.

Suricata capture interfaces are implemented in the C programming language. The source files need to be therefore stored in `src/` folder. To actually support a new capture interface it is required to add two source files associated with their header files. The file that will implement the actual capture process have a prefix `source-`. The other file with a prefix `runmode-` introduces the runmode to Suricata. To support DPDK, it was necessary to create 4 files:

- src/runmode-dpdk.{c,h},
- src/source-dpdk.{c,h}.

Suricata would not be able to see these files unless they are added to its build system. Makefile is the primary build tool for Suricata. However, these Makefiles are generated from templates. Files named like `Makefile.am` usually act as a template for future Makefiles. These templates are generated by GNU Automake[1] tool. The Automake tool generates `Makefile.in` files from the `Makefile.am` files. Afterwards from these files the final `Makefile` files are generated with tool GNU Autoconf[2].

Suricata uses GNU Autoconf to produce configure scripts for build, instal or package procedures. A template for creating a configure file is named `configure.ac` and is located in the root folder of Suricata code base. The file consists of code written in Bourne shell and specific Autoconf macros that are evaluated during the creation of the configure script. Configure script accepts various parameters to produce the required `Makefile`. These parameters determine how Suricata is built and compiled. But except CFLAGS (build parameters), configure script also accepts parameters of individual capture interfaces. According to these parameters, individual runmodes can be included in/excluded from Suricata binary. Additional features as extra debug information or verbose stats can be also enabled with these parameters.

Code in Listing 5.1 adds a support for DPDK argument in the `configure` script. The first argument in the macro `AC_ARG_ENABLE` creates a new feature (`--enable-dpdk`) in the `configure` script, the second argument includes a hint to show when help is called. The last argument assigns the default value to the variable. According to the assigned value in the variable, the `configure` script can also check if all dependencies for the project are present on the system. In case of DPDK it verifies that the DPDK library is available.

```
AC_ARG_ENABLE(dpdk,
              AS_HELP_STRING([--enable-dpdk],
                             [Enable DPDK support [default=no]]),,
              [enable_dpdk=no])
```

Listing 5.1: Adding a DPDK parameter to the configure script.

As previously mentioned, `Makefile.am` act as a template to generate the final `Makefile`. Therefore, to include the newly created DPDK source files to the compilation process, it is required to add them to the list of Suricata sources in `Makefile.am`. Source files in `Makefile.am` are sorted in ascending alphabetical order. This also has an effect that individual modules are grouped together. The convention can be seen in Listing 5.2 where new source files are included to already existing list of other source files.

```
suricata_SOURCES =
...
 runmode-dpdk.c runmode-dpdk.h \
...
 source-dpdk.c source-dpdk.h \
```

Listing 5.2: Adding DPDK source files to the Makefile.am.

The whole process of generating the files and running the `configure` script also defines constants that serve as directives. They are set according to default values and

[1] https://www.gnu.org/software/automake/
[2] https://www.gnu.org/software/autoconf/

parameters passed to the `configure` script. If `--enable-dpdk` parameter is used during the configuration process, `HAVE_DPDK` constant is set. These constants determine what parts of source code are available compiled into the binary file. This method excludes the code that is not used from the source files (e.g. disabled capture interface) but also leaves an option to include it in the future. The example can be:

```
#ifdef HAVE_DPDK
// code that is executed when DPDK is enabled
#else
// code that is executed when DPDK is disabled
#endif /* HAVE_DPDK */
```

Registering the capture interface

Once the new source files are added to the built system, it is possible to move forward. `TmModule` is a structure in Suricata that contains attributes required to be initialized in order to register a new module. Suricata holds an array of these structures and according to this array it is able to run individual modules/capture interfaces. The array has a fixed size equal to the number of implemented modules. Each module initializes the `TmModule` structure on the assigned index number, similarly as in Listing 5.3. For DPDK, the array contains space for two structures. One structure is for the receive module and can be accessed by `TMM_RECEIVEDPDK` (Listing 5.3) and the other structure is for the decode module and can be accessed by `TMM_DECODEDPDK`.

```
void TmModuleReceiveDPDKRegister(void)
{
    tmm_modules[TMM_RECEIVEDPDK].name = "ReceiveDPDK";
    tmm_modules[TMM_RECEIVEDPDK].ThreadInit = ReceiveDPDKThreadInit;
    tmm_modules[TMM_RECEIVEDPDK].Func = NULL;
    tmm_modules[TMM_RECEIVEDPDK].PktAcqLoop = ReceiveDPDKLoop;
    tmm_modules[TMM_RECEIVEDPDK].PktAcqBreakLoop = NULL;
    tmm_modules[TMM_RECEIVEDPDK].ThreadExitPrintStats = ReceiveDPDKThreadExitStat;
    tmm_modules[TMM_RECEIVEDPDK].ThreadDeinit = ReceiveDPDKThreadDeinit;
    tmm_modules[TMM_RECEIVEDPDK].cap_flags = SC_CAP_NET_RAW;
    tmm_modules[TMM_RECEIVEDPDK].flags = TM_FLAG_RECEIVE_TM;
}
```

Listing 5.3: Initialization of TmModule structure for the DPDK receive module.

From Listing 5.3, it is possible to observe the module mainly consists of name and a set of functions. The purpose of the module in Suricata is determined by the assigned flag (`TM_FLAG_RECEIVE_TM`). Functions can be assigned to the structure's attributes depending on the purpose of the module. In case of a receive module, the main functions are:

- Thread Init – initializes thread during the startup,
- Packet Acquisition Loop – the main capture method that receives packets and forwards them to Suricata,
- Thread Exit Print Stats – as Suricata stops, function is responsible to dump the counters for the last time,
- Thread Deinit – frees memory of allocated objects or frees resources in general.

Some functions of the list could have been already spotted in Section 4.3. In the design phase I had in mind the architecture of the Suricata modules. These functions are described in more elaborate way in the following sections.

However, the `TmModule` structure would be never initialized if a function call of `TmModuleReceiveDPDKRegister` would not be initiated from the Suricata core. It is therefore required to also append the function call at the end of setup functions in `src/suricata.c`.

Runmode implementation

Runmodes as described in Section 2.3 offer a variety of options how individual modules are connected and how they inter-operate. As discussed in Section 4.3, this **thesis is focused on workers runmode**. But functionality of runmode single can also be acheived by only enabling one worker thread. Source files with prefix `runmode-` can therefore suggest that they contain source code related to registering and enabling the runmode. Each capture interface must have this part implemented as initial configuration is often different from interface to interface. In `runmode-*` source files capture interfaces are able to tell Suricata which runmodes are supported. They also load configuration for individual runmodes. In Listing 5.4 DPDK runmode registers workers running mode. The third argument is a helper text and the last argument is a function `RunModeIdsDpdkWorkers` that is called when DPDK workers runmode is selected.

```
RunModeRegisterNewRunMode(RUNMODE_DPDK, "workers",
        "Workers DPDK mode, each thread does all"
        " tasks from acquisition to logging",
        RunModeIdsDpdkWorkers);
```

Listing 5.4: DPDK runmode registration.

The name of the function `RunModeIdsDpdkWorkers` follows naming conventions of Suricata. IDS in the middle of the name might suggest it is only suitable for detection only. It is not true thanks to the architecture presented in Section 4.3. Workers runmode is fully functional also in IPS mode. Packets are transmitted during release of their resources of Suricata. It will be described in more detail in the following sections.

The main purpose of the function is to load the DPDK configuration and initialize environment for the DPDK application. Once ready, it spawns all worker threads with configurations of separate network interfaces. Following Section 5.1.2 describes how configuration is loaded and DPDK initialized.

5.1.2 Configuration and DPDK initialization

All runmodes need to be setup in a certain way. Configuration allows us to adjust behavior of these runmodes. The settings are usually stored in the main configuration file – `suricata.yaml`. Suricata has already functions that are able to retrieve values of this configuration file. Suricata capture interfaces implement functions for parsing configuration of individual interfaces from a list of interfaces.

But before the list of interfaces is loaded, it is required to first initialize DPDK environment abstraction layer (EAL). To support DPDK parameters in Suricata, I added a new section to the `suricata.yaml` file called **`dpdk:`**. This section should contain settings that are globally applied to the DPDK application. EAL parameters are stored in a subsection **`eal-params:`**. The subsection can consist of arbitrary EAL parameters that

can change the way EAL is configured. Code inside `RunModeIdsDpdkWorkers` function loads subsection `dpdk.eal-params:` and crafts an array of arguments. Both short and long versions of argument names are supported. The array is required for the main DPDK initialization function – **`rte_eal_init`**. The function usually accepts command line arguments that are passed to the main function of application. The crafted array simulates the usual contents of these arguments. Important arguments to mention can be:

- **-l/--lcores** – specifies what cores will be available to the DPDK application,
- **-a** – specifies what NIC will be available according to Peripheral Component Interconnect Express[3] (PCIe) address. It can be used multiple times.

After EAL is initialized, Suricata continues to load configuration of interfaces. In section `dpdk-ifaces:` (Listing 5.5) of `suricata.yaml` file it is possible to specify settings of individual interfaces. For convenience, it is also possible to leave some values of settings to `auto`. For example, in case of `threads:` setting, value *auto* assigns all available DPDK lcores to the interface. Name of each interface must be specified with PCIe address of a NIC that is allowed to be used within DPDK. Settings from Listing 5.5 are loaded to internal DPDK structure of interface configuration (`DPDKIfaceConfig`). This configuration is passed to DPDK threads on spawn. From Listing 5.5 it is possible to observe that implementation support various kinds of settings, from the basic settings as enabling promiscuous mode (`promisc:`) to more advanced as enabling offload (`checksum-checks-offload:`). It also allows complete control over the size of DPDK memory pool and the number of transmit or receive descriptors.

```
dpdk-ifaces:
  - interface: 0000:3b:00.0
    threads: auto
    rx-queues: auto
    tx-queues: auto
    promisc: true
    multicast: true
    checksum-checks: true # if Suricata should validate checksums
    checksum-checks-offload: true # if possible, offload checksum validation
    mtu: 3000
    mempool-size: 65535
    mempool-cache-size: 250
    rx-descriptors: 1024
    tx-descriptors: 1024
    # IPS mode for Suricata works in 3 modes - none, tap, ips
    # - none: disables IPS functionality (does not further forward packets)
    # - tap: forwards all packets and generates alerts (omits DROP action)
    # - ips: the same as tap mode but it also drops packets that are flagged by
    rules to be dropped
    copy-mode: ips
    copy-iface: 0000:3b:00.1
```

Listing 5.5: The default configuration of DPDK interfaces section.

[3] https://www.intel.com/content/www/us/en/io/pci-express/pci-express-architecture-general.html

5.1.3 Worker threads de/initialization

Thread initialization

Once configuration is loaded into `DPDKIfaceConfig` structure, Suricata then continues to spawn worker threads. The number of threads for each interface is determined by the `threads:` option in configuration of the interface. Suricata uses **pthread** library for multithreading. However, because of DPDK's special nature, it is required to use a different approach to spawn a thread. In DPDK, it is possible to use function `rte_eal_remote_launch`. During creation of new threads Suricata also creates 2 per-thread variables. Initialization of (`ThreadVars`) structure is shared among all capture interfaces and runmodes. Structure `DPDKThreadVars` groups all attributes that are capture mode specific (e.g. a pointer to DPDK packet mempool). Structure `DPDKIfaceConfig` is referenced from the thread that spawned worker threads. In general, Suricata avoids sharing variables among multiple threads. This is the common and correct approach to multi-threaded programming as multi-thread synchronization brings a huge performance penalty. Figure 5.1 visualizes the creation of thread variables (outlined rounded rectangles) and referencing the interface configuration (dashed shapes). Worker threads keep reference to `DPDKIfaceConfig` only during the initialization phase.

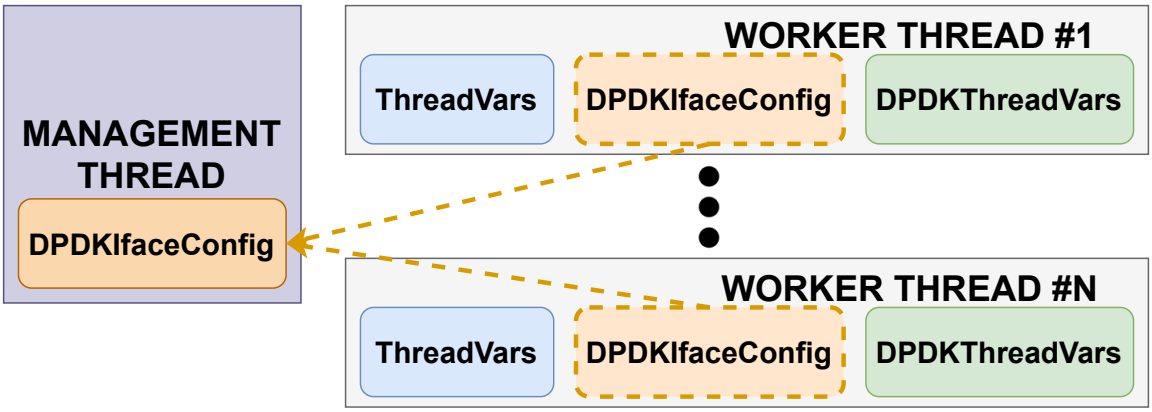

Figure 5.1: Thread initialization in Suricata.

`DPDKIfaceConfig` contains an atomic counter which is incremented by each spawned thread. Thanks to the counter, threads are synchronized. They are effectively waiting for the last thread to reset the counter and join them. Meanwhile, the last thread is responsible for NIC configuration according to values of `DPDKIfaceConfig` structure. It starts with configuration of basic settings such as multicast or promiscuous mode.

It attempts to enable some offloads like checksum validation or RSS. Receive side scaling is automatically enabled if more than 1 worker thread is used. It uses symmetric hash key described in 3.1 to forward packets of the same to the same receive queues.
When Suricata runs in IPS mode, it is also beneficial to enable `DEV_TX_OFFLOAD_MBUF_FAST_FREE` and `DEV_TX_OFFLOAD_MT_LOCKFREE` offloads to speed up transmission rates.

As the last thread joins the waiting threads, each thread creates an individual DPDK mempool according to settings stored in `DPDKIfaceConfig` structure. Reference to this

mempool is stored in `DPDKThreadVars` structure. Each thread also sets up its receive and transmit queues of the selected port.

Before the start of the NIC, each thread is added to a list of peers. A peer in the list is a worker thread. The list contains every running worker thread and comes to its full potential when Suricata runs in IPS mode. Each worker must have a peer to send packets to. The `copy-*` options in Listing 5.5 defines behavior of IPS. PCIe address in `copy-iface:` specifies a NIC where received packets will be forwarded after Suricata detection. Possible modes of IPS (`copy-mode:`) are also described in Listing 5.5. If a free peer with a given NIC is not found in the list, the worker is only appended to the end of the list (and paired later). After the list is completed, links between peers are verified. If a worker is left without any peers to bind, Suricata fails to start. This technique is used to bind all receive queues of one NIC together with all transmit queues of the NIC. Figure 5.2 illustrates the result of a successful peering. The whole process can look as follows.

Workers of NIC#1 are spawned first, and they are appended to the peers list. Since thread scheduling algorithms are not deterministic, workers are appended to the list in random order. Workers of NIC#1 then create receive and transmit queues and assign them to their `DPDKThreadVars` structure. The process repeat for worker threads of NIC#2. However, in this case workers are not only appended to the list but also try to look for a free worker (of NIC#1) willing to peer. After peering, workers have access to each others transmit queues. Peers list operates with workers to ensure only valid transmit queues are used. In Figure 5.2 the same colored workers are peered and pointers to queues are stored in their `DPDKThreadVars` structure. Dotted arrows are potential paths of packets after peering if Suricata runs in IPS mode.

To provide a more specific example, it is possible to assume there are 4 receive/transmit queues per interface. Packet received by NIC#1 is put into RX queue number 1. Worker#1 reads the queue and processes all packets through its pipeline of modules. After that, Worker#1 is ready to release the packet. If the packet is not evaluated to be dropped, Worker#1 adds the packet to transmit queue number 1 of NIC#2.

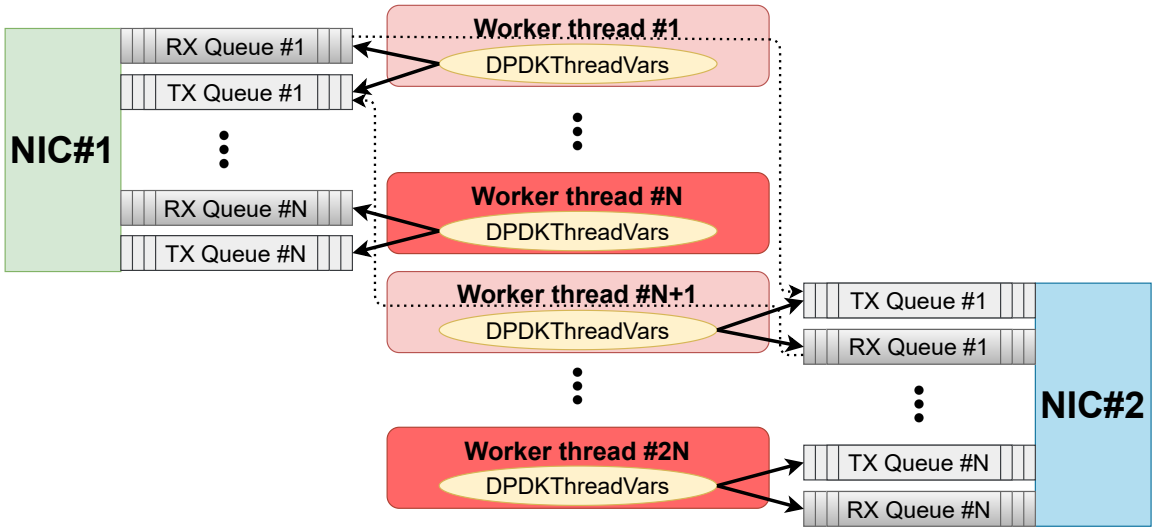

Figure 5.2: Peers list in Suricata.

As previously mentioned, worker threads try to minimize the inter-thread communication. Up to this point, it might seem they heavily rely on communication via mutexes and synchronization controls. However, this is not true after initialization phase. After all worker threads are added to the peers list, the NIC is started and threads enter to a continuous capture (receive) loop.

Thread deinitialization

After Suricata stops, worker threads go to deinit phase. This allows threads to release all allocated resources. DPDK requires no special deinitialization actions. It first frees DPDK packet memory pool created in the initialization phase and then frees `DPDKThreadVars` structure.

5.1.4 Receive loop

Receive loop is the core function of the whole capture interface. It is repeatedly executed until interrupted by, for example Suricata exit signal. It must be performance oriented, as inability to capture all packets can lead to missed security events. With this in mind, I was able to design and implement the function. The design could have been seen already in Figure 4.4.

Actual implementation is not far from the proposed design. It only made it more specific. Implementation of this part also makes use of some functions from DPDK library. As shown in Figure 4.4, the functions start with initialization of variables. Execution then enters the main loop. Then receive module fetches a list of received `Mbufs` to thread's `DPDKThreadVars` structure. This operation is accomplished by `rte_eth_rx_burst` function. A place to retrieve packets from is determined by given port id and queue id. Combination of these two identification numbers is unique for every worker thread. Execution continues if some packets are received, otherwise it jumps to the beginning of the main loop.

Receive module loops over each received `Mbuf` structure from the array. The module needs to obtain a packet structure (`Packet`) from the internal Suricata queue, since the modules following the receive module use generic packet structure. The same function as in Figure 4.4 is used to receive the packet structure. `PacketGetFromQueueOrAlloc` attempts to retrieve a free packet from the queue and if it fails, it will allocate a new one.

The packet structure is then initialized with various information, among others:

- packet release function,
- Suricata mode (IPS/IDS),
- internal checksum validation,
- pointer to the `Mbuf` structure in memory,
- worker peer,
- packet data.

Packet data are set directly from the `Mbuf` structure. Since `Mbuf` is a large structure, it might not always be easy to know where the actual packet data are and how long they are. DPDK has macros like `rte_pktmbuf_mtod` or `rte_pktmbuf_pkt_len` to solve the problem. Macro `rte_pktmbuf_mtod` returns actual packet data section of the passed `Mbuf` structure. Macro `rte_pktmbuf_pkt_len` returns length of the packet data section.

Afterwards, packet is only sent for Suricata processing and execution continues in either next `Mbuf` structure processing or statistics update followed by a jump to the start of the main loop.

5.1.5 Packet transmission and release

After Suricata finishes processing the packet, it is no longer needed. It can be therefore released from the system. In general, Suricata has 2 approaches – either via a Verdict module or via a packet release function. As discussed in Section 4.3, DPDK is designed to not use Verdict modules. Instead, a release function is assigned to each packet. Function's only required parameter is the packet structure. Since the packet structure comes out of the Suricata engine, it contains results of the detection. Implementation of the DPDK release function very closely corresponds with a function design proposed in Figure 4.5. The details are also described in Section 4.3. Function `rte_eth_tx_burst` is used if a packet is supposed to be transmitted. The function takes the same arguments as `rte_eth_rx_burst` – port and queue identification number and an array of `Mbuf` structures. However, in this case, port and queue ID is fetched from the peer. Also, array only contains one element – the `Mbuf` structure.

The packet structure can be released as soon as the packet is sent. If Suricata does not run in IPS mode, the packet is released immediately. But before packet structure can be released, it is required to free the `Mbuf` structure. DPDK function `rte_pktmbuf_free` takes `Mbuf` pointer and frees space for incoming packets. If `Mbuf` structure is not freed, DPDK would not be able to capture more packets because of depleted `Mbuf` pointers. After `Mbuf` is freed, it is possible to call function `PacketFreeOrRelease` with the packet structure as the only argument. The function frees the packet structure in case it is only allocated. If the packet structure was obtained from the Suricata queue, it is released. The packet structure can then hold information about a newly incoming packet.

5.1.6 Statistics

It is important to have an overview of how the deployed Suricata instance performs. Monitoring count of received/dropped packets can provide basic insights of Suricata's health. For example, increase in dropped packets can indicate some problem that should be more investigated. Statistics in Suricata are collected at all times. They are accessible in real time from various places. For instance, they can be accessed in output files like `eve.log`/`stats.log` or over Unix socket.

In DPDK, statistics are updated continuously in a function `DPDKDumpCounters`. The function fetches statistics of the configured NIC and then updates individual thread counters. Statistics are fetched on queue-basis. By default, DPDK has statistic information for only 16 queues. This can be configured by adjusting `RTE_ETHDEV_QUEUE_STAT_CNTRS` during build of the DPDK framework. If number of queues is higher than `RTE_ETHDEV_QUEUE_STAT_CNTRS`, Suricata warns the user during startup but continues in execution.

Suricata has special receive module function to gather statistics and print them out at the end. DPDK receive module required no special handling in this case. Therefore, in `ReceiveDPDKThreadExitStats` it only calls `DPDKDumpCounters` and prints out the statistics.

5.2 Benchmarks

As the name of this section suggests, the section presents achieved results of DPDK runmode implementation to Suricata. It also compares the results with AF_PACKET – the most commonly used implementation of the receive module. When measuring performance of systems, it is crucial to have a stable environment. The testbed used for benchmarks is presented in Section 3.1. There have been problems like incomplete data transfers or tedious measurement process (Section 3.3). These problems have been resolved prior/during performance tuning of AF_PACKET running mode.

The test cases were executed under the same conditions as in the case of AF_PACKET. That included over 20000 rules from ET Open ruleset as presented in Subsection 2.2.2 and the same PCAP file as analyzed in Subsection 3.3.3. Flexible architecture of the testbed allowed to switch between individual running modes effortlessly.

Graph consists of two axis, horizontal axis displays transmission speed in megabits per second and vertical axis shows buffer overflow in percentages. As in previous measurements, the main performance indicator is buffer overflow (BO) – the count of received packets divided by the count of transmitted packets. Section 3.2.7 describes the exact process of BO calculation.

Tests are executed mainly in two variants – with disabled or enabled rules. Testing performance without rules has proven to be a very efficient strategy. It provides a capability to see how much rules affect the performance of Suricata. It can also show the top possible performance of Suricata. On the other hand, Suricata would not be helpful if it would detect nothing. For that reason, it is beneficial to see the actual, close to real-life performance of Suricata.

Benchmarks start with 1 worker thread as the simplest testing scenario. To provide more insight how Suricata scales, later tests are using more worker threads. The second benchmark uses 8 worker threads and the third uses 16 workers. Between individual benchmark runs no settings are changed, except thread section of the configuration file.

5.2.1 Benchmark with 1 worker thread

Testing Suricata performance with 1 core can provide early insights to benefits and problems of the new capture interface. Developing a single threaded programs is easier and helps to exclude problems like thread synchronization or concurrency control. It therefore simplifies debugging certain bugs. On the other hand, some bugs may only appear in concurrent environments.

The very first milestone in the implementation was to measure results of Suricata packet capture. This means that rules were not applied. As discussed in Subsection 3.4.2 that Suricata only receives and discards packets. Suricata actually does full packet decoding and creates/modifies records in flow tables. Only the detection part is skipped. This further eliminates possible problem-causing parts. The test is heavily focused on receive performance of the individual capture interfaces.

Figure 5.3 presents results of the measurements. It is possible to see DPDK capture interface (orange line) over-performs AF_PACKET (blue line). Amount of received packets by DPDK (8 Gbps) doubles the amount of received packets by AF_PACKET (<4

Gbps) under 5% buffer overflow threshold. This proves the implementation of DPDK brings benefits of faster and more reliable capture method.

Graph in Figure 5.4 presents results of a Suricata benchmark that also applied rules. Full capture to detection pipeline is, again, placed in 1 thread. Compared to 5.3, it is possible to see a decrease in overall Suricata throughput to about 600-750 Mbps. This proves how the detection part with large amount of rules is performance heavy. However, DPDK implementation proves to be more efficient even in this test. It continuously increases Suricata throughput by additional 100 Mbps compared to AF_PACKET implementation. Zero loss packet detection of DPDK goes up to 700 Mbps.

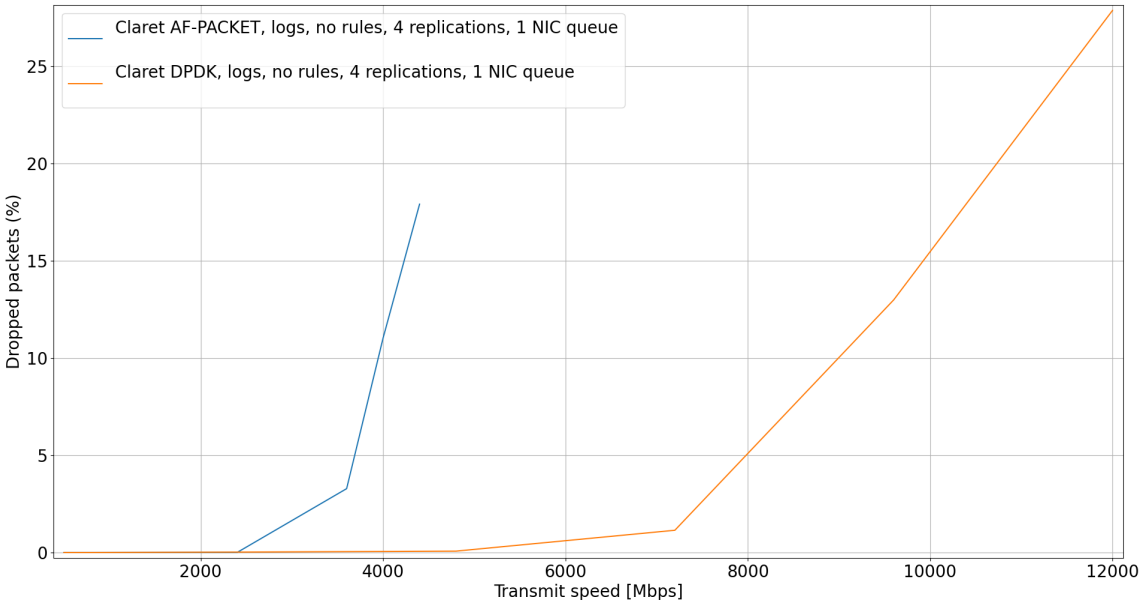

Figure 5.3: 1 queue and disabled rules.

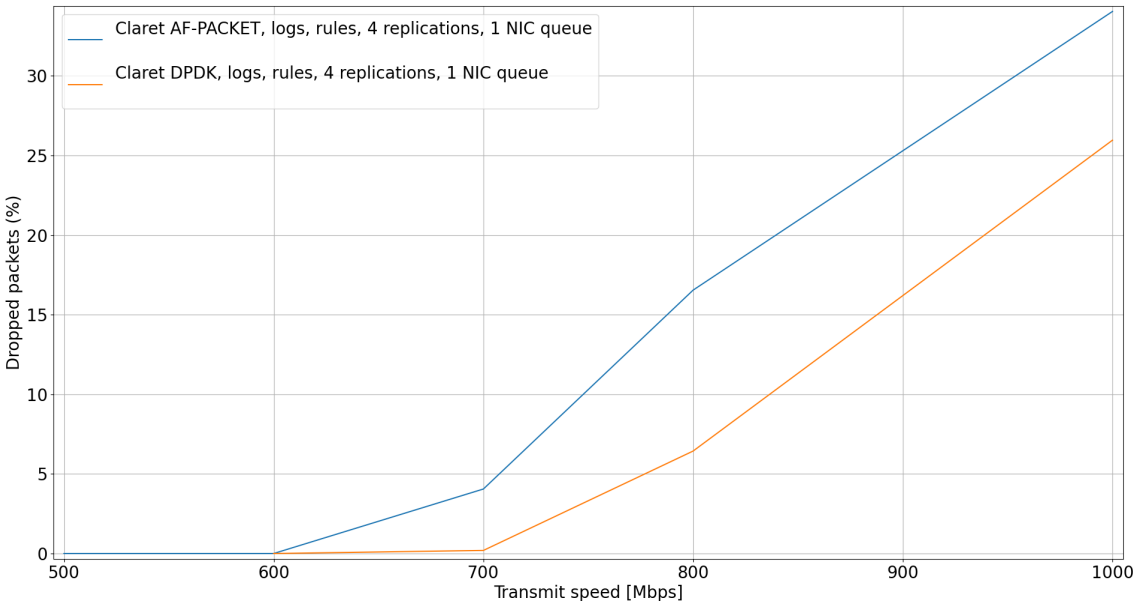

Figure 5.4: 1 queue and enabled rules.

5.2.2 Benchmark with 8 worker threads

The next set of measurements are set to 8 workers. All threads are assigned to physical cores on the same NUMA node as the NIC. Graphs resulting from these measurements are in Figures 5.5 and 5.6.

In a benchmark without any rules applied (Figure 5.5), AF_PACKET drop rate steeply increases after input rate surpassesmusi 20 Gbps. On the other hand, DPDK performance have packet drop very close to zero until 28 Gbps. From that moment, buffer overflow steadily increases to 2%. Compared to Figure 5.4, DPDK runmode does not scale proportionately to the number of threads/cores. However, in this case we can see a large bump in the overall performance – from zero-loss throughput of 7 Gbps with 1 worker thread to 28 Gbps with 8 workers. In case of AF_PACKET capture interface, the throughput scales in proportion to the number threads. From Figure 5.3 it is possible to see that 1 worker is able to receive traffic at rate of around 2 Gbps without any packet loss whereas in Figure 5.5 8 workers can receive traffic up to 16 Gbps.

Results of measurements of 8 workers with enabled rules can be seen in Figure 5.6. It shares the same behavior as portrayed in Figure 5.4 in a sense of seriously decreased performance compared to no-rule versions. However, it can be expected due to huge amount of rules. Figure 5.6 that performance of both capture interfaces is rather similar. Differences are more prevalent at higher transfer rates (6+ Gbps) where DPDK is superior to AF_PACKET. However, at this speed, drop rate of packets reaches 10+ percents in both runmodes. In practical terms, this means that Suricata can fail to detect important security events. When Figures 5.4 and 5.6 are compared, the increase in count of threads goes hand in hand with a proportional increase of Suricata throughput. On average, each added thread increases throughput of about 550 Mbps.

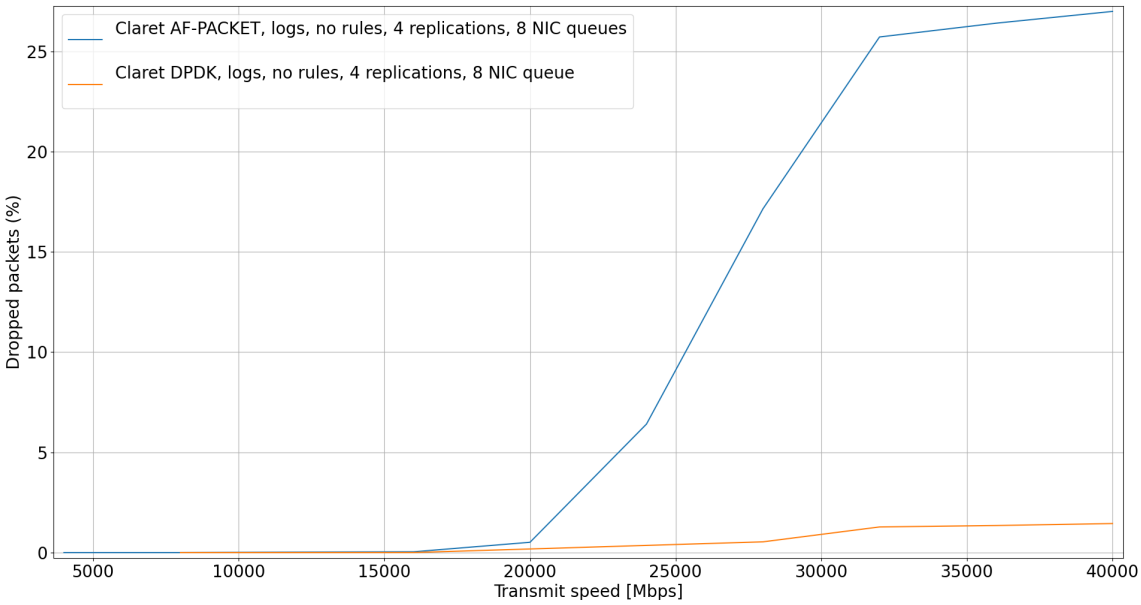

Figure 5.5: 8 queues and disabled rules.

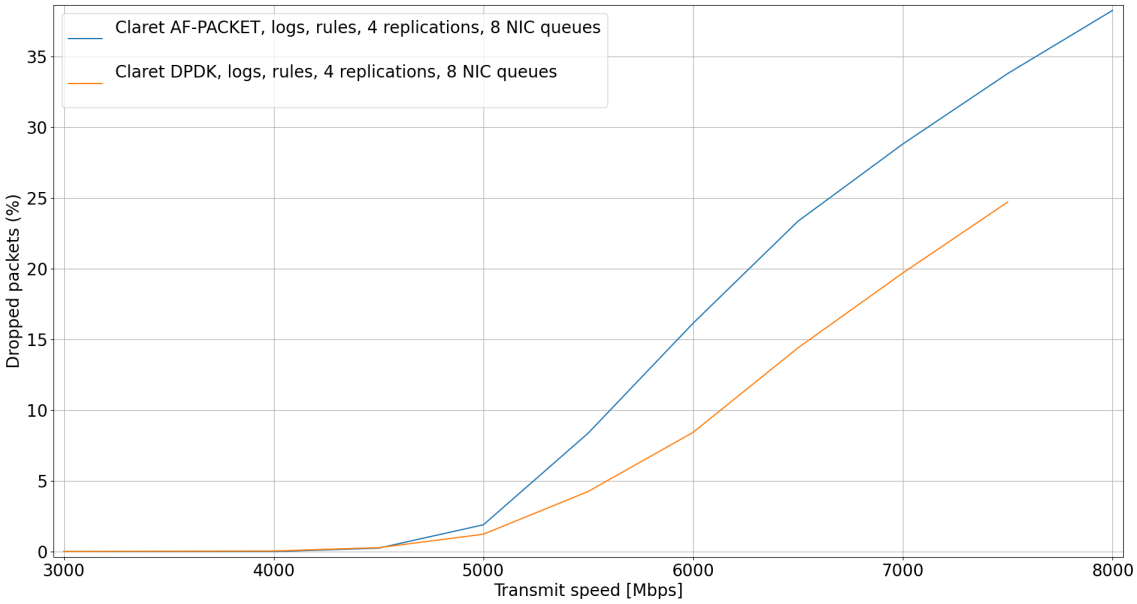

Figure 5.6: 8 queues and enabled rules.

5.2.3 Benchmark with 16 worker threads

The last set of tests were executed on 16 worker threads. Workers were spread over 8 physical and 8 hyperthreaded cores. Because of the limited number of physical cores, Hyperthreading was enabled. Having more cores (even hyperthreaded) brings more benefits. While worker on one physical core waits for the packet or some other operation, the worker on the same but hyperthreaded core can continue in the inspection. Hyperthreading is explained in Subsection 3.4.1. The number of DPDK receive descriptors must have been increased to 32768 because of performance drops.

Figure 5.7 shows, again, an increase in received packets. The more threads equal to better performance. But performance gain is not as noticeable as in the previous case. The bigger difference between performance of 1 and 8 workers is attributed to the higher utilization of physical cores. In contrast, performance gained by increasing 8 to 16 workers is less remarkable because the added cores are hyperthreaded. However, this step costs no more in terms of CPU occupancy as 16 worker variant uses 8 logical cores that are paired with 8 physical cores.

In DPDK, 16 queues allow better data reception and thus handle speed of 40 Gbps with practically zero packet loss. Further inspection was not needed as the goal is to mostly focus on rule-based version. Improvement in the performance can be also seen in AF_PACKET implementation – from 16 Gbps to around 22 Gbps.

The results of the most performant version of Suricata with rules are featured in Figure 5.8. DPDK implementation outperforms AF_PACKET by additional 500 Mbps in zero/close to zero loss transmission speeds. The difference is even more apparent at higher transfer rates. However, the drop of packets is again 10+%.
Figure 5.9 presents a graph with a different metric. X axis is the same as in previous graphs but Y axis represents average number of generated alerts per second. Even though presented tests are executed for the same amount of time, dividing count of alerts by transmit duration (doing average) can be beneficial in tests of different duration. From the graph, it is possible to observe almost the same behavior of AF_PACKET and DPDK. It is especially striking between 4 Gbps and 6 Gbps. Graph in Figure 5.8 proves that within this range, both capture interfaces are able to receive almost all packets. With the given hardware and configuration, after 6 Gbps Suricata can not process packets fast enough and dropped packets start to increase. Within transmit rate from 6 to 8 Gbps, Suricata drops up to 20% of packets (in AF_PACKET running mode). Interestingly enough, within this range in Figure 5.9, graph's line shows the steepest increase in generated alerts. This might be caused by coincidence and transmission speeds of over 6 Gbps are able to reach PCAP section which triggers the most rules with alert actions. Figure 5.9 also shows that AF_PACKET runmode generates more alerts in range from 6 to 7.5 Gbps. Results of this metric varied over several tests. In general, all measurements followed the pattern. However, results of individual runmodes fluctuated in a certain range. Therefore, the rate of generated alerts is rather similar in both cases.

This 16 worker setup was analyzed with different performance tools. These tools, as explained in Subsection 3.4.1, can provide deeper insight into performance analysis of Suricata. It is possible to compare outputs of individual implementations to get more detailed information about Suricata performance. Listings 5.6 and 5.7 show output of top 20 processes extracted from the output of `perf top` command. In both cases, the load is spread among functions with similar responsibilities. Also, detect module takes up the most resources. From the output, it is also possible to notice that it is important to focus

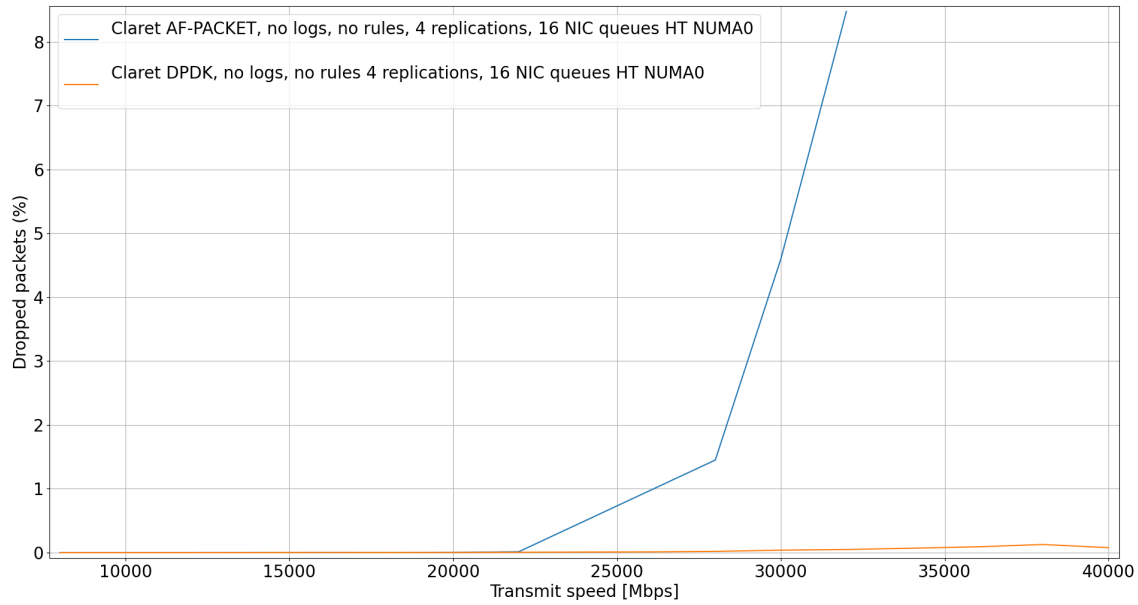

Figure 5.7: 16 queues and disabled rules.

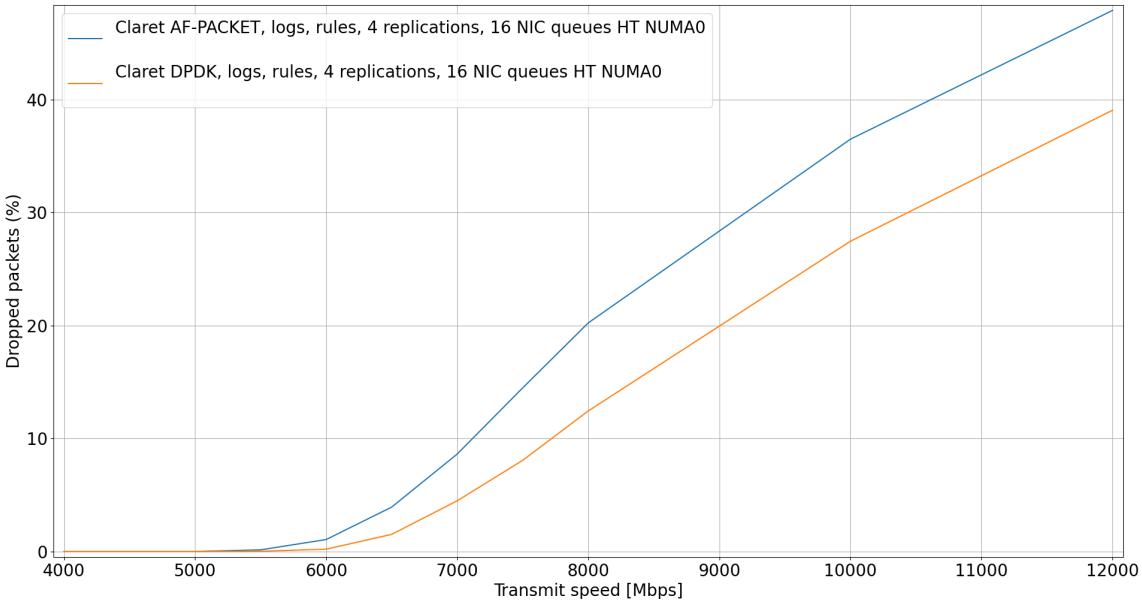

Figure 5.8: 16 queues and enabled rules.

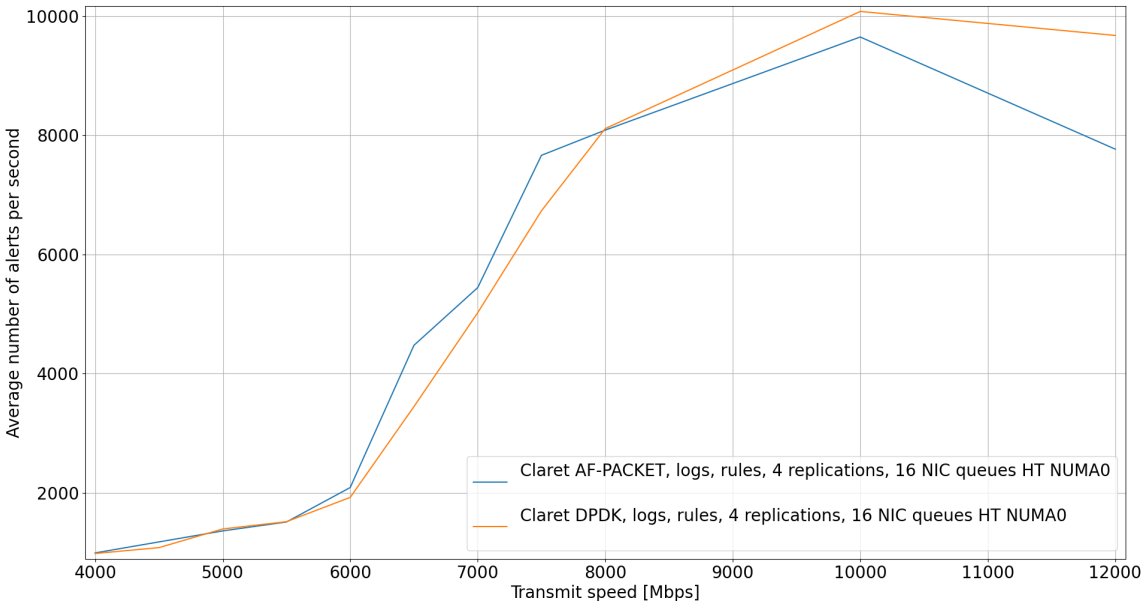

Figure 5.9: Generated alerts from 16 queues and enabled rules.

analysis not only around the application itself (e.g. suricata) but also include a load of other related components (e.g. libhs.so.5.3.0, [kernel]).

AF_PACKET functions in Listing 5.6 are usually under processes like suricata or [kernel]. In case of DPDK, processes like suricata with librte_pmd_mlx5.so.20.0.3 are used. The latter process is part of DPDK PMD driver and is used to repeatedly fetch packets from the NIC. It is interesting to observe that sum of capture interface's functions usage is in case DPDK (5.65%) higher than in AF_PACKET (4.57%). The listings may provide incomplete information (only top 20 processes) but further processes are only more lightweight than the presented ones. It is surprising since DPDK in overall performs better than AF_PACKET.

The two runmodes were also under analysis of `pcm` tool. The results were again quite similar. In certain aspects – like L3HIT was AF_PACKET (0.76%) better than DPDK (0.67%). However, in terms of instruction per cycle metrics, DPDK with 3.08 IPC per physical core was better compared to AF_PACKET with 2.98 IPC per physical core. Both running modes had UPI (QPI) equal or less than 1 percent.

```
29.07%  suricata            [.] DetectRun.part.16
10.02%  suricata            [.] DetectEngineInspectRulePacketMatches
 9.08%  suricata            [.] DetectEnginePktInspectionRun
 2.84%  suricata            [.] DetectProtoContainsProto
 1.97%  suricata            [.] FlowGetFlowFromHash
 1.48%  libhs.so.5.3.0      [.] 0x0000000000612fa6
 1.23%  suricata            [.] DetectEngineEventMatch
 1.14%  [kernel]            [k] tpacket_rcv
 1.03%  [kernel]            [k] memcpy_erms
 0.73%  [kernel]            [k] build_skb
 0.68%  libhs.so.5.3.0      [.] 0x0000000000612641
 0.65%  [kernel]            [k] tasklet_action_common.isra.14
 0.63%  [kernel]            [k] fib_table_lookup
 0.61%  libpthread-2.28.so  [.] __pthread_mutex_lock
 0.59%  suricata            [.] DetectFlowMatch
 0.41%  suricata            [.] SCHSMatchEvent
 0.40%  libhs.so.5.3.0      [.] 0x0000000000579866
 0.39%  [kernel]            [k] __netif_receive_skb_core
 0.38%  libhs.so.5.3.0      [.] 0x0000000000579933
 0.38%  suricata            [.] AFPReadFromRingV3
```

Listing 5.6: Top 20 processes of `perf top` when Suricata runs with AF_PACKET.

```
25.82%  suricata                  [.] DetectRun.part.16
 8.58%  suricata                  [.] DetectEngineInspectRulePacketMatches
 8.22%  suricata                  [.] DetectEnginePktInspectionRun
 2.55%  suricata                  [.] DetectProtoContainsProto
 2.37%  suricata                  [.] ReceiveDPDKLoop
 1.98%  suricata                  [.] FlowGetFlowFromHash
 1.21%  libhs.so.5.3.0            [.] 0x0000000000612fa6
 0.97%  suricata                  [.] DetectEngineEventMatch
 0.65%  suricata                  [.] rte_pktmbuf_free
 0.62%  librte_pmd_mlx5.so.20.0.3 [.] 0x00000000001bc6ac
 0.49%  librte_pmd_mlx5.so.20.0.3 [.] 0x00000000001bc6ce
 0.47%  suricata                  [.] DetectFlowMatch
 0.47%  libpthread-2.28.so        [.] __pthread_mutex_lock
 0.46%  librte_pmd_mlx5.so.20.0.3 [.] 0x00000000001bc2c9
 0.46%  librte_pmd_mlx5.so.20.0.3 [.] 0x00000000001bc72e
 0.43%  libhs.so.5.3.0            [.] 0x0000000000612641
 0.42%  librte_pmd_mlx5.so.20.0.3 [.] 0x00000000001bc0bb
 0.42%  librte_pmd_mlx5.so.20.0.3 [.] 0x00000000001bc404
 0.41%  librte_pmd_mlx5.so.20.0.3 [.] 0x00000000001bc2a6
 0.40%  suricata                  [.] DecodeEthernet
```

Listing 5.7: Top 20 processes of `perf top` when Suricata runs with DPDK.

5.2.4 Overall results evaulation

Several variants of benchmarks were executed to test Suricata performance. AF_PACKET and DPDK were the main tested capture interfaces. DPDK was able to consistently outperform AF_PACKET implementation by leaving more resources to other modules. Performance improvements were big in measurements without rules. A bottleneck appeared after applying huge amount of rules. This diminished DPDK's performance improvements. It led me to a conclusion that AF_PACKET is able to capture most of the packets and has a similar performance to DPDK because both capture interfaces are blocked by the detection module. As DPDK was more lightweight, it slightly helped to improve the performance.

Looking at 1 and 8 worker measurements with rules, it is possible to see that Suricata scales appropriately to the number of enabled cores. Each worker was on a separate physical core. Similar scaling does not apply to measurements of 8 and 16 workers. It is caused by using Hyperthreading for 16 workers. By creating 2 logical cores from 1 physical core it allows better utilization of CPU cores. It does not reach performance gains of 2 physical cores but it is better than 1 physical core only. It allowed to employ additional NIC queues into Suricata. In overall, Suricata scales well on physical cores. However, the same number of Hyperthreaded physical cores allows even higher CPU utilization than the *physical cores only* approach. This possibly suggest that implementing `autofp` may be also beneficial as DPDK is able to receive great amount of traffic just on a few cores. Therefore, some threads could be focused on packet reception and some work solely on packet inspection.

Chapter 6

Conclusion and future work

Goal of this master thesis was to come up with a possible optimization for Suricata IDS/IPS system. The goal could be divided in two parts. Prior to any optimization proposals, I needed a baseline. Therefore, optimizing AF_PACKET running mode was the first part. The second was to propose a Suricata optimization. According to my performance analysis and previous efforts of CESNET, I have chosen to implement DPDK capture interface.

Before I could even start with the master thesis, I had to get comfortable with many different things like Suricata, a network stack of Unix systems or performance and network analysis tools. Similarly, building a proper and functional testing environment was not less important. This included not only interconnecting individual computers but also, for example, implementing correct build and install scripts. The journey was accompanied by many challenges to solve. Out of those, I could mention incomplete data transfers due to a variety of reasons or Suricata compilation problems on old Linux kernel. As a result, I was able to gain huge amount knowledge in a short time.

After the testbed was setup and I had a basic overview of Suricata I could proceed to the initial phase of the thesis – performance experiments of Suricata AF_PACKET running mode. I realized very soon that repeatedly obtaining performance data is monotonous and very time-consuming process. This resulted in a need of a new testing framework. Because of very specific requirements of testbed's environment, it was not possible to use some generic test framework. I had designed and implemented modular testing framework. I chose Bash scripting language for the implementation. Output of the testing framework are CSV files. Handling raw data and interpreting them in spreadsheet application was again tedious process. For this reason, Python3 graph making script was designed and implemented in parallel to the development of the testing framework. As running individual tests in the test framework can still be a time-consuming process, I also added a scheduler to the testing framework. It allows to schedule long operations in an execution queue.

Previously mentioned things, although time-consuming at first, provided essential information through benchmarks and tests of Suricata. It then helped to steer the configuration and optimization process in the right direction. During the process, I was able to deeply dive into Suricata internals and configuration of Linux network stack. That included configuration of the NIC driver, BIOS and also adjustment in kernel settings. All these steps were required to explore possible options to acheive the top performance of Suricata. When I reached high throughput and good performance results for the given

hardware and could not move further, I stopped. With limited time, possible optimization techniques can be a very broad area to explore (and it is a very long process).

By finishing the optimization process, I have reached the first milestone of the thesis – achieve baseline results of AF_PACKET running mode. After that, I performed an analysis of the accomplished Suricata configuration. I have found out, capture interface can take as much as 20% of processing power. DPDK framework seemed like an adequate candidate for capture interface as in many regards it is considered to be one of the very efficient capture interfaces. Also previously failed efforts of DPDK integration to Suricata increased the motivation for this attempt. I proceeded to explore a source code of Suricata. That included not only capture interface source codes but also parts remotely connected with it. It was a lengthy process, but it helped to form a big picture of Suricata architecture. Lack of comprehensive developer documentation requires studying internal implementation of Suricata from the source code. But before any design proposals, I had to have good understanding of DPDK framework. For that reason, I continued to study architecture of DPDK applications and also tried to apply the knowledge to some application prototypes.

With knowledge of both Suricata and DPDK, I was able to progress to the implementation design. I followed common architectural practices applied in Suricata to be consistent with other capture interfaces. AF_PACKET architecture was chosen as the main capture interface to follow. After consulting the proposed design, I started implementation. The actual implementation sticks to the suggested design. I have developed the capture interface with my best intentions to be performance oriented and user friendly as well. It provides a wide variety of options in the configuration file. As can be observed from benchmarks, it consistently outperforms AF_PACKET capture interface. However, AF_PACKET still has a place in Suricata as DPDK is supported usually by modern NICs. Results of measurements also show that while DPDK helped to increase the overall performance of Suricata, the capture interface is not the main bottleneck. Detection module with large amount of rules is the heaviest part in Suricata pipeline. In general, the results of the implementation are exciting and future of DPDK capture interface is looking very promising.

Even though DPDK implementation outperformed AF_PACKET, I can still see a room for improvement. The performance can be further enhanced by implementing bypass functionality. This would skip the detection for certain flows of traffic. If implemented properly, the NIC itself can already decide the fate of packet flow. It can work similarly to XDP bypass. For this use-case `rte_flow` library of DPDK might be used.

Then, for the integration of Suricata into other applications, it would be possible to create some sort of API to connect Suricata with already existing applications. This can be possibly be acheived by running Suricata as a secondary DPDK process and use `rte_ring` for inter-application communication.

It was also possible to notice from the benchmarks that DPDK receives packets at higher rates than AF_PACKET. This also suggests an idea that implementing `autofp` can bring additional performance benefits. On contrary to `workers` runmode, small amount of cores could be dedicated to receive and decode modules while the majority of cores would receive packets from these cores and focus on packet inspection.

I believe there are many possible paths how this project can further continue. Suricata is always-evolving piece of software that also might come up with even better optimization ideas. To my best intentions, I will work in cooperation with CESNET on further improvements of DPDK running mode.

Bibliography

[1] Bruneau, G. *The History and Evolution of Intrusion Detection* [online]. SANS Institute, . 2001 [cit. 2020-11-02]. Available at: https://www.sans.org/reading-room/whitepapers/detection/history-evolution-intrusion-detection-344.

[2] Ghafir, I., Prenosil, V., Svoboda, J. and Hammoudeh, M. A Survey on Network Security Monitoring Systems. August 2016. DOI: 10.1109/W-FiCloud.2016.30. Available at: https://doi.org/10.1109/w-ficloud.2016.30.

[3] Gonzalez, J. M., Paxson, V. and Weaver, N. Shunting: A Hardware/Software Architecture for Flexible, High-Performance Network Intrusion Prevention. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2007, p. 139–149. CCS '07. DOI: 10.1145/1315245.1315264. ISBN 9781595937032. Available at: https://doi.org/10.1145/1315245.1315264.

[4] Kicinski, J. and Viljoen, N. EBPF Hardware Offload to SmartNICs: clsbpf and XDP. [online]. 2016, [cit. 2020-12-29]. Available at: https://netdevconf.info/1.2/papers/eBPF_HW_OFFLOAD.pdf.

[5] Leblond Éric and Manev, P. *Introduction to eBPF and XDP support in Suricata* [online]. 2020 [cit. 2020-12-30]. Available at: https://cdn2.hubspot.net/hubfs/6344338/Resources/Stamus_WP_Intro_to_eBPF_and_XDP_in_Suricata_Online.pdf.

[6] Lukasik, S. Why the Arpanet Was Built. *IEEE Annals of the History of Computing*. 2011, vol. 33, no. 3, p. 4–21. DOI: 10.1109/MAHC.2010.11.

[7] Martellini, M. and Malizia, A. *Cyber and Chemical, Biological, Radiological, Nuclear, Explosives Challenges: Threats and Counter Efforts*. 1st ed. Springer, 2017. ISBN 978-3-319-62108-1.

[8] Miano, S., Bertrone, M., Risso, F., Tumolo, M. and Bernal, M. V. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In: *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. 2018, p. 1–8 [cit. 2020-12-30]. DOI: 10.1109/HPSR.2018.8850758.

[9] Mjasojedov, I. *Systém pro ochranu před DoS útoky s využitím IDS*. Brno, CZ, 2020. Bachelor thesis. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis-file/23110/23110.pdf.

[10] OISF. *Suricata project* [online]. 2020 [cit. 2020-10-21]. Available at: https://suricata-ids.org/.

[11] Orman, H. The Morris worm: a fifteen-year perspective. *IEEE Security Privacy*. 2003, vol. 1, no. 5, p. 35–43. DOI: 10.1109/MSECP.2003.1236233.

[12] Projects, L. *Official site of Data Plane Development Kit project*. 2021. [Online; accessed 2021-01-06]. Available at: https://www.dpdk.org/.

[13] Purzynski, M. and Manev, P. *Suricata Extreme Performance Tuning* [online]. 2016 [cit. 2021-03-21]. Available at: https://github.com/pevma/SEPTun/blob/master/SEPTun.pdf.

[14] Technology, A. *Shunt Away Unwanted Suricata Traffic with Accolade Adapters* [online]. 2016 [cit. 2020-12-03]. Available at: https://accoladetechnology.com/wp-content/uploads/2018/07/Suricata-Flow-Bypass-Tech-Brief.pdf.

[15] 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.

[16] Woo, S. and Park, K. *Scalable TCP Session Monitoring with Symmetric Receive-side Scaling* [online]. 2012 [cit. 2020-12-21]. Available at: http://an.kaist.ac.kr/~shinae/paper/2012-srss.pdf.

[17] Zhao, Z., Sadok, H., Atre, N., Hoe, J. C., Sekar, V. et al. *Achieving 100Gbps Intrusion Prevention on a Single Server*. USENIX Association, november 2020 [cit. 2021-04-04]. Available at: https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng.

Appendix A

NIC settings

```
sudo systemctl stop irqbalance
sudo systemctl disable irqbalance
sudo service irqbalance stop
sudo chkconfig irqbalance off
sudo service firewalld stop
sudo systemctl disable firewalld
sudo service iptables stop
sudo sysctl -w net.ipv4.tcp_ecn=0
sudo sysctl -w net.ipv4.ip_forward=1
echo 0 | sudo tee cat /sys/bus/pci/devices/0000\:3b\:00.1/roce_enable
echo 0 | sudo tee /sys/bus/pci/devices/0000\:3b\:00.1/d3cold_allowed
sudo ip link set ens1f1 mtu 3000
sudo ip link set ens1f1 promisc on arp off up
echo 1 | sudo tee /proc/sys/net/ipv6/conf/ens1f1/disable_ipv6
for i in rx tx tso ufo gso gro lro tx nocache copy sg txvlan rxvlan \\
         rx-udp_tunnel-port-offload rx-vlan-filter; do
                    sudo /usr/local/sbin/ethtool -K ens1f1 $i off;
                echo $i
done
for proto in tcp4 udp4 ah4 esp4 sctp4 tcp6 udp6 ah6 esp6 sctp6; do
    sudo /sbin/ethtool -N ens1f1 rx-flow-hash $proto sdfn
done
sudo /usr/local/sbin/ethtool \\
        -C ens1f1 adaptive-rx off adaptive-tx off rx-usecs 10 rx-frames 64
sudo /usr/local/sbin/ethtool -G ens1f1 rx 8192
sudo /usr/local/sbin/ethtool -A ens1f1 rx off tx off
echo toeplitz | sudo tee /sys/class/net/ens1f1/settings/hfunc
sudo /usr/local/sbin/ethtool -K ens1f1 rxhash on ntuple on
sudo ethtool -L ens1f1 combined 18
sudo /usr/local/sbin/ethtool -X ens1f1 hkey \\
        6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:\\
        6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A equal 18
sudo /usr/sbin/set_irq_affinity_cpulist.sh
    "2,4,6,8,10,12,14,16,18,22,24,26,28,30,32,34,36,38" ens1f1
```

Listing A.1: Optimized settings for the mlx5 NIC

Appendix B

PCAP analysis

```
      1 ET HUNTING Suspicious TLS SNI Request for Possible COVID-19 Domain M2
     10 ET POLICY possible Xiaomi phone data leakage DNS
    430 SURICATA Applayer Detect protocol only one direction
    107 SURICATA Applayer Protocol detection skipped
 186180 SURICATA Applayer Wrong direction first Data
    101 SURICATA FRAG IPv4 Fragmentation overlap
      2 SURICATA HTTP Host header invalid
      4 SURICATA HTTP METHOD terminated by non-compliant character
      1 SURICATA HTTP Request abnormal Content-Encoding header
     91 SURICATA HTTP Request line incomplete
      1 SURICATA HTTP Request line with leading whitespace
      2 SURICATA HTTP Request unrecognized authorization method
      4 SURICATA HTTP Response invalid protocol
      2 SURICATA HTTP Response invalid status
      2 SURICATA HTTP URI terminated by non-compliant character
     32 SURICATA HTTP Unexpected Request body
    106 SURICATA HTTP gzip decompression failed
     25 SURICATA HTTP invalid response chunk len
      2 SURICATA HTTP invalid response field folding
      1 SURICATA HTTP request field missing colon
      1 SURICATA HTTP request header invalid
      2 SURICATA HTTP response field missing colon
      9 SURICATA HTTP response header invalid
      2 SURICATA HTTP too many warnings
  19194 SURICATA HTTP unable to match response to request
      2 SURICATA IKEv2 weak cryptographic parameters (Auth)
      2 SURICATA IKEv2 weak cryptographic parameters (Diffie-Hellman)
      2 SURICATA IKEv2 weak cryptographic parameters (PRF)
      2 SURICATA SMTP invalid pipelined sequence
      1 SURICATA SMTP no server welcome message
     53 SURICATA STREAM 3way handshake SYN resend different seq on SYN recv
    277 SURICATA STREAM 3way handshake SYNACK in wrong direction
     13 SURICATA STREAM 3way handshake SYNACK resend with different ack
    130 SURICATA STREAM 3way handshake SYNACK with wrong ack
      6 SURICATA STREAM 3way handshake excessive different SYN/ACKs
    484 SURICATA STREAM 3way handshake right seq wrong ack evasion
18809496 SURICATA STREAM 3way handshake wrong seq wrong ack
     27 SURICATA STREAM 4way handshake SYNACK with wrong ACK
    155 SURICATA STREAM 4way handshake SYNACK with wrong SYN
```

```
   8801 SURICATA STREAM CLOSEWAIT ACK out of window
  24362 SURICATA STREAM CLOSEWAIT FIN out of window
   4871 SURICATA STREAM CLOSEWAIT invalid ACK
    191 SURICATA STREAM ESTABLISHED SYN resend
    812 SURICATA STREAM ESTABLISHED SYN resend with different seq
    129 SURICATA STREAM ESTABLISHED SYNACK resend
    743 SURICATA STREAM ESTABLISHED SYNACK resend with different ACK
6593241 SURICATA STREAM ESTABLISHED invalid ack
16603512 SURICATA STREAM ESTABLISHED packet out of window
  18442 SURICATA STREAM FIN invalid ack
  22334 SURICATA STREAM FIN out of window
  39852 SURICATA STREAM FIN1 FIN with wrong seq
      8 SURICATA STREAM FIN1 ack with wrong seq
  16101 SURICATA STREAM FIN1 invalid ack
    450 SURICATA STREAM FIN2 FIN with wrong seq
  10919 SURICATA STREAM FIN2 invalid ack
      3 SURICATA STREAM Last ACK with wrong seq
6881283 SURICATA STREAM Packet with invalid ack
   2095 SURICATA STREAM Packet with invalid timestamp
 237709 SURICATA STREAM SHUTDOWN RST invalid ack
    177 SURICATA STREAM TIMEWAIT ACK with wrong seq
  35296 SURICATA STREAM bad window update
   6660 SURICATA STREAM excessive retransmissions
   1061 SURICATA STREAM reassembly overlap with different data
     63 SURICATA TCP invalid option length
      6 SURICATA TCP option invalid length
   2006 SURICATA TLS invalid handshake message
    679 SURICATA TLS invalid record type
   2685 SURICATA TLS invalid record/traffic
     74 SURICATA TLS too many records in packet
    114 SURICATA UDP packet too small
      3 SURICATA UDPv4 invalid checksum
```

Listing B.1: Generated alerts from PCAP file that was transmitted by 1 replication

```
      1 ET HUNTING Suspicious TLS SNI Request for Possible COVID-19 Domain M2
     40 ET POLICY possible Xiaomi phone data leakage DNS
    434 SURICATA Applayer Detect protocol only one direction
     58 SURICATA Applayer Protocol detection skipped
 188927 SURICATA Applayer Wrong direction first Data
    344 SURICATA FRAG IPv4 Fragmentation overlap
   1568 SURICATA FRAG IPv6 Fragmentation overlap
      2 SURICATA HTTP Host header invalid
      4 SURICATA HTTP METHOD terminated by non-compliant character
      1 SURICATA HTTP Request abnormal Content-Encoding header
     32 SURICATA HTTP Request line incomplete
      2 SURICATA HTTP Request unrecognized authorization method
      2 SURICATA HTTP Response invalid protocol
      2 SURICATA HTTP Response invalid status
      1 SURICATA HTTP URI terminated by non-compliant character
     31 SURICATA HTTP Unexpected Request body
    100 SURICATA HTTP gzip decompression failed
```

```
     22 SURICATA HTTP invalid response chunk len
      2 SURICATA HTTP invalid response field folding
      1 SURICATA HTTP request field missing colon
      2 SURICATA HTTP response field missing colon
      9 SURICATA HTTP response header invalid
      2 SURICATA HTTP too many warnings
  18516 SURICATA HTTP unable to match response to request
      2 SURICATA IKEv2 weak cryptographic parameters (Auth)
      2 SURICATA IKEv2 weak cryptographic parameters (Diffie-Hellman)
      2 SURICATA IKEv2 weak cryptographic parameters (PRF)
      2 SURICATA SMTP invalid pipelined sequence
      1 SURICATA SMTP no server welcome message
     33 SURICATA STREAM 3way handshake SYN resend different seq on SYN recv
    174 SURICATA STREAM 3way handshake SYNACK in wrong direction
     16 SURICATA STREAM 3way handshake SYNACK resend with different ack
    109 SURICATA STREAM 3way handshake SYNACK with wrong ack
      8 SURICATA STREAM 3way handshake right seq wrong ack evasion
16747892 SURICATA STREAM 3way handshake wrong seq wrong ack
     19 SURICATA STREAM 4way handshake SYNACK with wrong ACK
    287 SURICATA STREAM 4way handshake SYNACK with wrong SYN
   5356 SURICATA STREAM CLOSEWAIT ACK out of window
  12716 SURICATA STREAM CLOSEWAIT FIN out of window
   4099 SURICATA STREAM CLOSEWAIT invalid ACK
    194 SURICATA STREAM ESTABLISHED SYN resend
     69 SURICATA STREAM ESTABLISHED SYN resend with different seq
    113 SURICATA STREAM ESTABLISHED SYNACK resend
     38 SURICATA STREAM ESTABLISHED SYNACK resend with different ACK
5615369 SURICATA STREAM ESTABLISHED invalid ack
15599046 SURICATA STREAM ESTABLISHED packet out of window
  12621 SURICATA STREAM FIN invalid ack
  17269 SURICATA STREAM FIN out of window
  25883 SURICATA STREAM FIN1 FIN with wrong seq
      1 SURICATA STREAM FIN1 ack with wrong seq
   8704 SURICATA STREAM FIN1 invalid ack
      7 SURICATA STREAM FIN2 FIN with wrong seq
     91 SURICATA STREAM FIN2 invalid ack
     12 SURICATA STREAM Last ACK invalid ACK
6338431 SURICATA STREAM Packet with invalid ack
  26756 SURICATA STREAM Packet with invalid timestamp
 697535 SURICATA STREAM SHUTDOWN RST invalid ack
    226 SURICATA STREAM TIMEWAIT ACK with wrong seq
  19999 SURICATA STREAM bad window update
   4810 SURICATA STREAM excessive retransmissions
    354 SURICATA STREAM reassembly overlap with different data
    252 SURICATA TCP invalid option length
     24 SURICATA TCP option invalid length
   1642 SURICATA TLS invalid handshake message
    545 SURICATA TLS invalid record type
   2187 SURICATA TLS invalid record/traffic
    408 SURICATA UDP packet too small
    126 SURICATA UDPv4 invalid checksum
```

Listing B.2: Generated alerts from PCAP file that was transmitted by 4 replication