Master thesis : Fast Service Chaining

Auteur : Iurman, Justin
Promoteur(s) : Mathy, Laurent
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité spécialisée en "computer systems and networks"
Année académique : 2016-2017
URI/URL : http://hdl.handle.net/2268.2/3208

Speeding up Snort with DPDK for NFV cooperation

Justin Iurman

Supervisor: Prof. Laurent MATHY
Committee members: Prof G. LEDUC, Prof B. DONNET, T. BARBETTE

Graduation Studies conducted for obtaining the Master's degree in COMPUTER SCIENCES

University of Liège
Faculty of Applied Sciences
Academic year 2016-2017

Abstract

Today, the network traffic keeps growing again and again. Software middleboxes are crucial elements and can't become bottlenecks, at the risk of dropping network performances. They are either a firewall, a NAT, an intrusion detection system, a WAN optimizer, a load balancer, etc. That's why they must be efficient and choices made for their implementation are very important.

The objective of this work is to speed up Snort, which is an intrusion detection system, in a context of user level service chaining. Some improvements are studied and implemented in order to reach that goal. Thanks to those, Snort itself is also improved.

This paper describes the whole work, step by step. It begins with an introduction to define the context and to explain each protagonist. Then, next chapters are each dedicated to specific tests and measurements, in a view of comparing each result and improve performances. DPDK, a fast I/O framework developed by Intel, is introduced to speed up Snort. In a context of cooperation, FastClick and Snort exchange packets through DPDK rings and jobs repetitions are avoided thanks to a notion which is also introduced: metadata. Those metadata are used to pass information between multiple middleboxes. Other ideas are also studied. Finally, it ends with future possible improvements and a conclusion that discusses whether it is worthwhile to improve elements, in a view of a cooperation.

Acknowledgements

First of all, i would like to thank Professor Laurent Mathy. He gave me the opportunity to work on a very interesting subject in which i learned a lot. It was a challenging subject because it involved many concepts i didn't know well. But, though, i don't regret my choice.

Also, I would like to thank Tom Barbette for his help, his time and his precious councils during the whole work in which he is really involved for his PhD thesis.

To both of you, a big thank-you for your kindness. I really enjoyed working for you and with you.

Contents

1 Introduction

Nowadays, the amount of network traffic keeps growing again and again. It is mainly due to the evolution and expansion of the Internet, especially with the Internet of Things[1]. One of the constant big challenge in networking area is to design elements from that point of view. Each network device must be able to quickly handle the traffic, without congestion, in order to execute its tasks with fluidity. That's especially the case for a firewall, for instance. Those devices are a critical point to maintain security inside a network and they have to treat data as fast as possible. In fact, that's the case for all middleboxes in general, where the traffic goes through while being acquired, analyzed and/or modified and then forwarded (or dropped).

In that context, not only the speed is important, but a particular attention is also paid to flexibility, cost and scalability. Indeed, the flexibility of scaling up/down or evolve a service is not negligible. For instance, a virtual router is a software function that replicates (in software) functionalities of a hardware-based router, which has used a dedicated hardware device. The advantage is to lower hardware costs, while having hardware interoperability. From this example, a virtual router is a form of Network Functions Virtualization.

So, hardware middleboxes are fast because they are hardware devices. But if we need to change something, such as adding a functionality, then we need to replace the entire device. This is not acceptable, especially nowadays, when a lot of changes are constantly applied to devices. That's why the concept of NFV is interesting. Software middleboxes (NFV) bring advantages seen previously. However, it is well known that a software solution is slower than a hardware solution. The main goal of a software middlebox is to be the most efficient as possible, by improving each part of it. For instance, a fast I/O framework can be used.

[1](IoT) Everyday objects that are connected to the Internet

1.1 Network Functions Virtualization

NFV refers to the notion of middleboxes virtualization. Here is a more formal definition of NFV.

"Network Functions Virtualization (NFV) involves implementing ***network functions in software*** *that can run on a range of industry standard hardware, and that can be moved to, or instantiated in, various locations in the network as required, without the need to install new equipment".*[1]

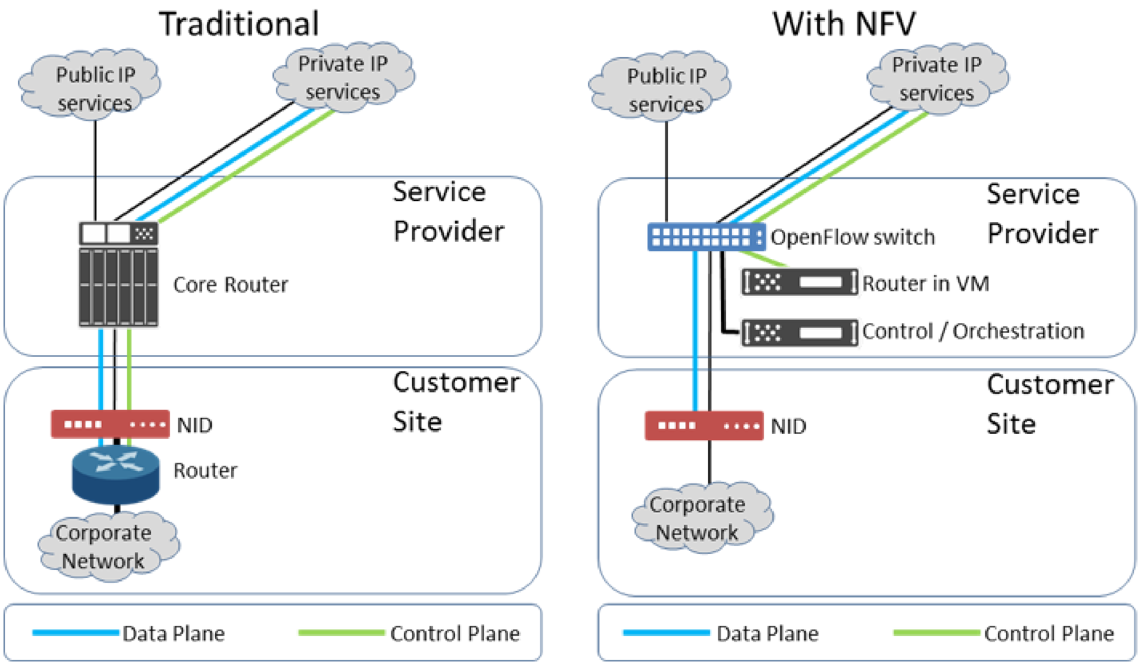


Figure 1: NFV benefits compared to traditional technique [2]

1.2 Fast Packet Processing

But still, as seen before, NFV does not solve speed/performance issue at all. There is still a need to improve everything, such as fast packet processing. Indeed, in Linux for instance, the Kernel Network Stack is **slow** because there is a lot of system calls to receive and send packets. Also, it involves copying data from kernel space to user space, which means context switching.

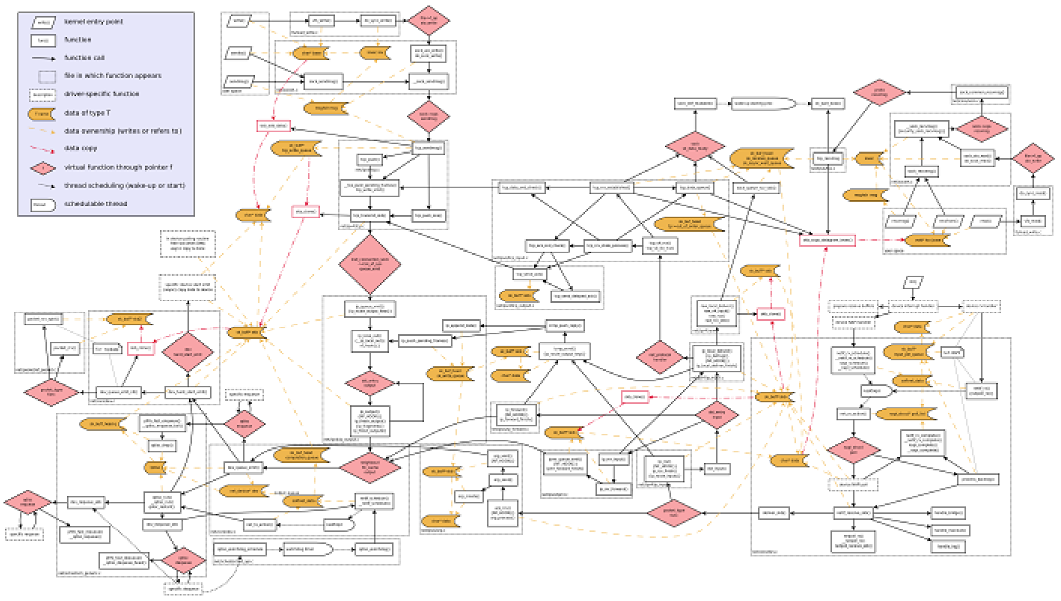


Figure 2: This figure, intentionally unreadable because too large, shows the complexity of the Linux Network Kernel Stack [3]

Recent years have seen a renewed interest for software network processing. However, as seen previously, a standard general-purpose kernel stack is too slow for linerate processing of multiple 10-Gbps interfaces. To address this issue, several userspace I/O frameworks have been proposed. In recent years, we have witnessed the emergence of high speed packet I/O frameworks, bringing unprecedented network performance to userspace. Those allow to bypass the kernel and obtain efficiently a batch of raw packets with a single syscall, while adding other capabilites of modern NIC[2], such as support for multi-queueing.[4]

In the context of this work, DPDK[3] is the chosen framework. Indeed, it is one of the most efficient. However, there exist some other good frameworks that differ on some points. Comparative tests and implementations have been realized here, in the University of Liege, and can be found as a paper (see reference [4]). This paper is important as it lays the foundations of this work.

[2]Network Interface Card
[3]Data Plane Development Kit

1.3 DPDK

DPDK
DATA PLANE DEVELOPMENT KIT

Data Plane Development Kit (DPDK), developed by Intel, is a set of libraries and drivers that address the requirements of the data plane, or forwarding engine, of network functions virtualization (NFV) for fast packet processing. The data plane is where the packet handling is done in network devices. The DPDK libraries use a run-to-completion scheduling model coupled with PMDs to eliminate the processing overhead of a scheduler and asynchronous interrupts. DPDK optimizes buffer management to eliminate data copies (zero copy). It also eliminates the use of locks for memory and workload management through optimized queue and ring management to achieve fast data plane performance. Portability is enabled via an environment abstraction layer (EAL) that hides the specifics of the underlying hardware and operating environment and provides a standardized programming interface to libraries. The DPDK is open source software for Linux and FreeBSD licensed under a BSD license. A project to maintain and enhance the software is coordinated through `http://dpdk.org`.[5]

In clear, DPDK enables faster development of high speed data packet networking applications, which means it allows to build applications that can process packets faster, thanks to the bypass of the kernel. Indeed, it uses a *fast path* instead of normal path of Network layers and context switching. Packets are delivered into user space directly (as raw packets). The difference between packet processing by the Linux kernel (see figure 3) and packet processing by DPDK (see figure 4) is shown below. A comparison between normal (slow) path and *fast path* is also provided (see figure 5).

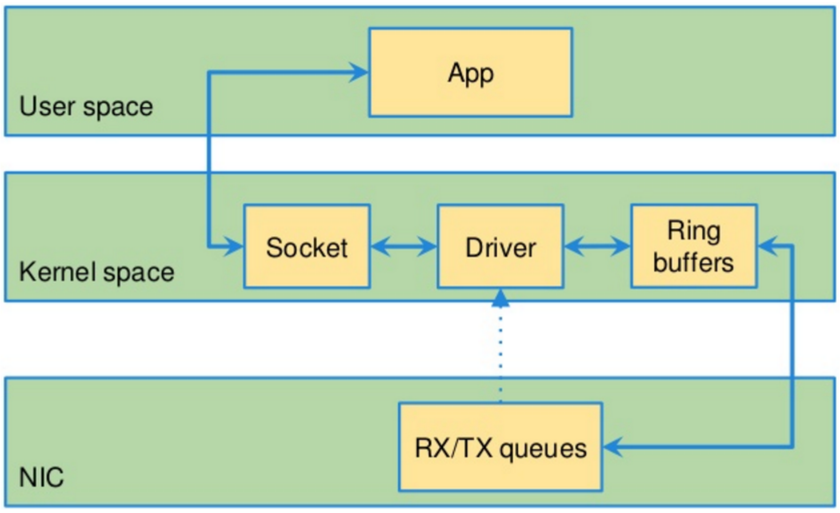


Figure 3: Packet processing in Linux [6]

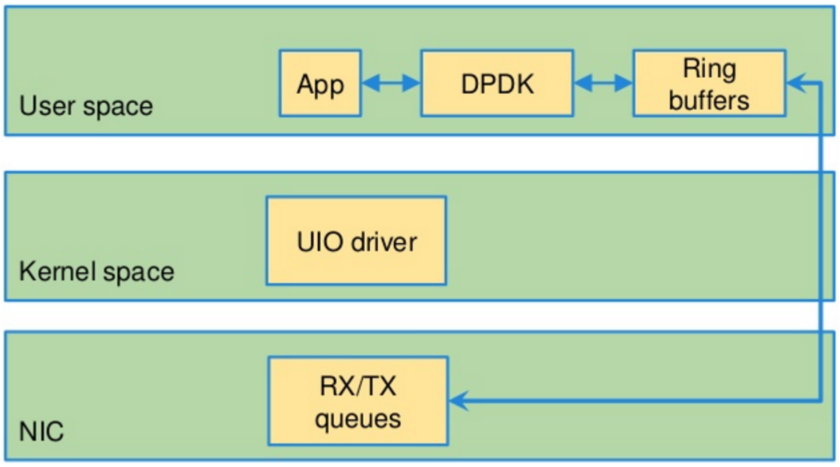


Figure 4: Packet processing with DPDK [6]

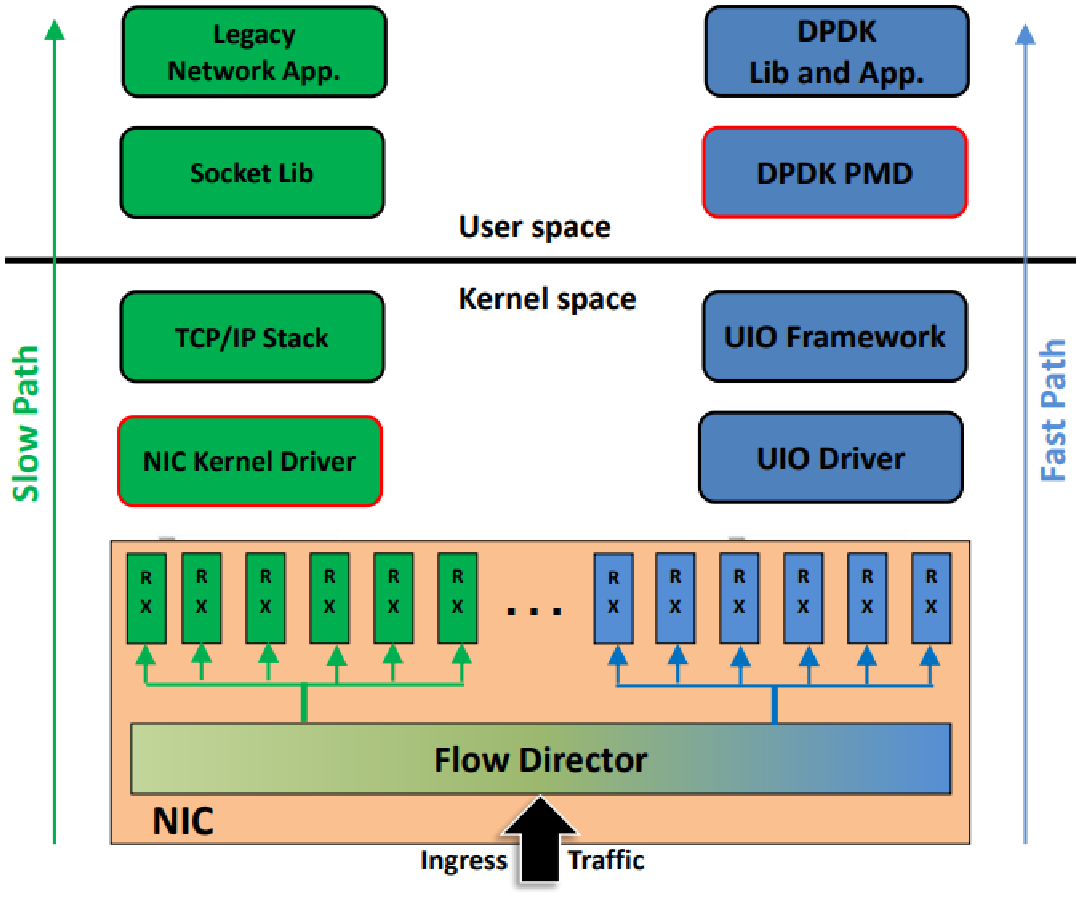


Figure 5: Slow path vs Fast path [7]

1.3.1 Components

DPDK components enable this fastpath technology for packet processing. Here is a short description for each of them below and a global overview (see figure 6).[8]

- **Environment Abstraction Layer** (EAL) is responsible for gaining access to low level resources such as hardware and memory space. It provides a generic interface that hides the environment specifics from the applications and libraries.
- **Memory Manager** is responsible for allocating pools of objects in memory. A pool is created in huge page[4] memory space and uses a ring to store free objects. It also provides an alignment helper to ensure that objects are padded to spread them equally on all DRAM channels.
- **Buffer Manager** reduces by a significant amount the time the operating system spends allocating and de-allocating buffers using advanced techniques such as Bulk Allocation, Buffer Chains, Per Core Buffer Caches etc. The Intel DPDK pre-allocates fixed size buffers which are stored in memory pools.
- **Queue Manager** implements safe lockless queues, instead of using spinlocks, that allow different software components to process packets, while avoiding unnecessary wait times.
- **Packet Flow Classification** provides an efficient mechanism to produce a hash based on tuple information so that packets may be placed into flows quickly for processing, thus greatly improving throughput.
- **Poll Mode Driver** (PMD) are designed to work without interrupt based signaling mechanisms. So, instead of using interrupts and wasting CPU attention, it uses polling and does not disturb the CPU.

[4]Huge Pages can go to 1 GB which means more TLB hits and less time spent in page replacement from local disk

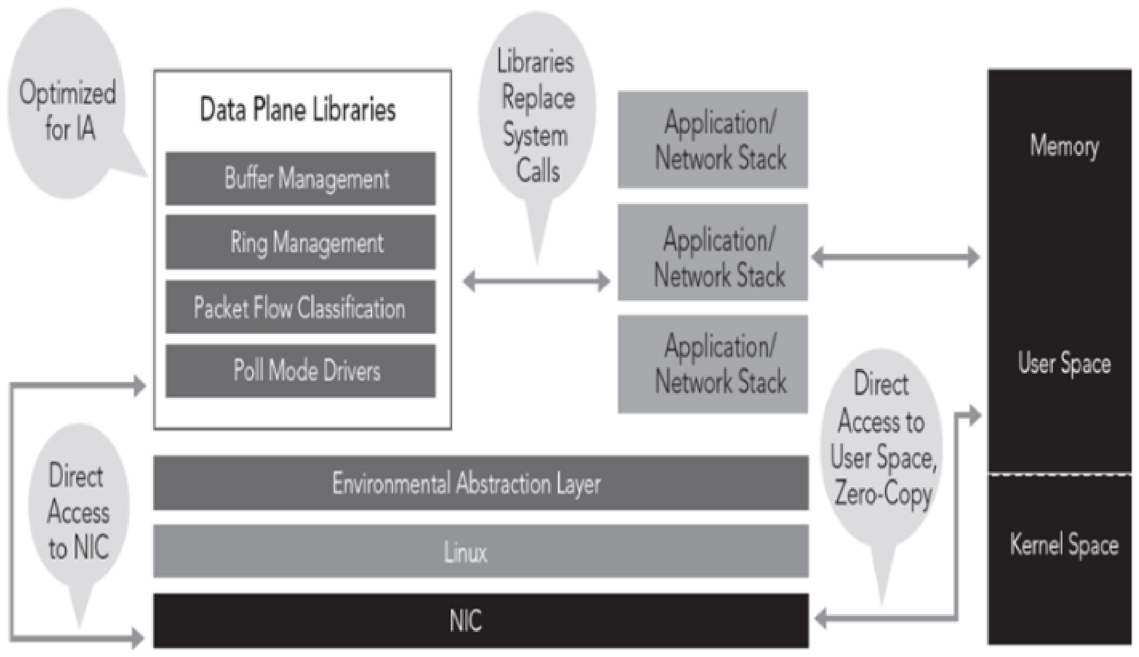

Optimized for IA
Data Plane Libraries
Buffer Management
Ring Management
Packet Flow Classification
Poll Mode Drivers
Libraries Replace System Calls
Application/ Network Stack
Application/ Network Stack
Application/ Network Stack
Memory
User Space
Kernel Space
Direct Access to User Space, Zero-Copy
Direct Access to NIC
Environmental Abstraction Layer
Linux
NIC

Figure 6: DPDK environment [8]

One could conclude that DPDK is nice because it's fast. However, we still need to pass packets to other applications and avoid doing the same job each time.

1.4 Context of this work

As previously said in section 1, sensitive middleboxes need to be fast and efficient. For instance a NAT or a firewall for performance, security, etc. In the context of this work, the main middlebox is an intrusion detection/prevention system called Snort. The more Snort runs fastly, the more it can handle network packets without missing one, the fastest it will detect/react to something suspicious without any latency and congestion.

Having an intrusion system like Snort running as fast as possible is not an easy thing. Of course, it is already thought and designed to be efficient by its developers. Nevertheless, with the arrival of new constraints but also new technologies, some new solutions and improvements could be investigated to speed it up.

This work first focuses on speeding up Snort alone. Then, it focuses on speeding up Snort in the context of a communication between NFVs, which are FastClick and Snort. If two NFVs communicating are fast and efficient, it does not mean that the communication is optimal too. Indeed, if one NFV is doing a job that the other has already done, it's a pity to waste time for that and it's not as fast as it could be. That's where the focus is put: improving the communication between NFVs by avoiding job repetition.

1.4.1 Goals

The first goal is to speed up Snort alone. This can be done by speeding up its packet processing. Since DPDK is said everywhere to be really fast and efficient, it will be used to achieve this goal.

The second goal and main objective of this work is to focus on the communication between two NFVs (FastClick and Snort) and try to speed it up. This can be done by firstly still using DPDK and secondly by passing metadata between the upfront software (FastClick) and Snort to prevent the latter from re-doing the work again. Since we need to have a sort-of shared memory between the two, the DPDK Ring API will be used to allow both sides to communicate through rings.

The ultimate goal is to "mimic" (import) Snort's detection engine into FastClick. But this solution will be proved as a bad idea in the conclusion and it will be explained why.

2 Snort

Snort[5] was created by Martin Roesch in 1998, in response to demand for a commercial version of the popular technology. It was developed by Sourcefire, which has been acquired by Cisco since 2013.

It is an open source network intrusion detection system (NIDS) and network intrusion prevention system (NIPS), capable of performing real-time traffic analysis and packet logging on IP networks. It can perform protocol analysis, content searching/matching, and can be used to detect a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, OS fingerprinting attempts, and much more.

In this paper, anything related to Snort is about the 2.9.9.0 version. Even if most of Snort does not change, there could be some changes from one version to another, especially with the brand new Snort 3. This section is fully based on "*Snort Users Manual 2.9.9*" (see reference [9]).

2.1 Modes

Snort can be configured to run in three different modes:

- **Sniffer mode** which simply reads the packets and displays them in a continuous stream on the screen (like tcpdump).
- **Packet Logger mode** which logs the packets to the disk (useful for network traffic debugging, for instance).
- **NIDS/NIPS mode** which performs detection/prevention on network traffic. This is the most complex and configurable mode.

2.2 NIDS vs NIPS

Snort can run either as a **passive** or **inline** device.

[5] `https://www.snort.org`

Rule Option	**Inline Mode**	**Passive Mode**
reject	Drop + Response	Alert + Response
react	Block and send notice	Block and send notice
normalize	Normalize packet	Don't normalize
replace	Replace content	Don't replace
respond	Close session	Close session

Table 1: Snort - Inline vs Passive mode comparison

2.2.1 NIDS

When Snort is in **Passive** mode, it acts as a **NIDS**, which is the detection mode. Drop rules are not loaded. It only examines network traffic to detect malicious packets and alerts on these. See figure 7 below.

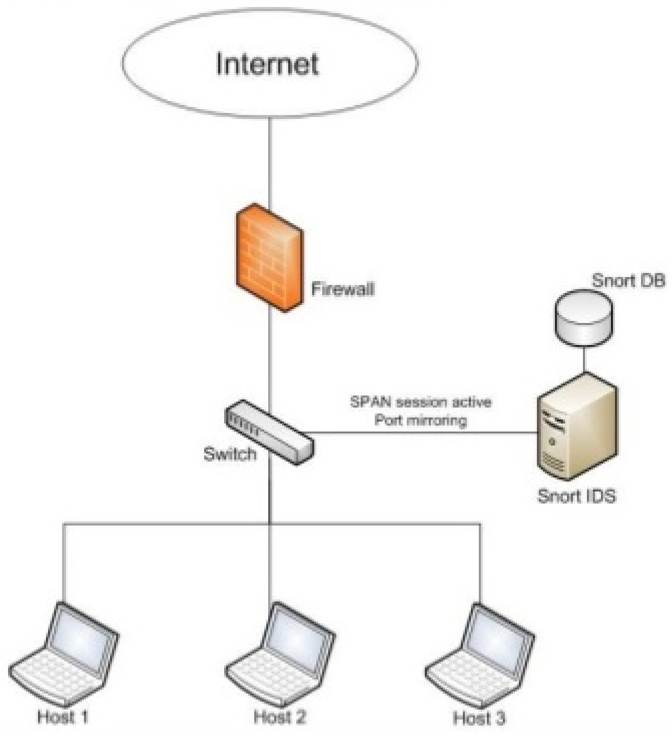


Figure 7: Snort running as a NIDS [10]

2.2.2 NIPS

When Snort is in **Inline** mode, it acts as a **NIPS**, which is the prevention mode. Drop rules are loaded and allowed to trigger. It means that Snort is connected between 2 or more network segments, each connected to a separate NIC. This will allow Snort to not only monitor but also block traffic flowing between these network segments when a signature is triggered. This blocking can create issues with False Positive alerts, and the inline operation can slow down traffic as well. See figure 8 below.

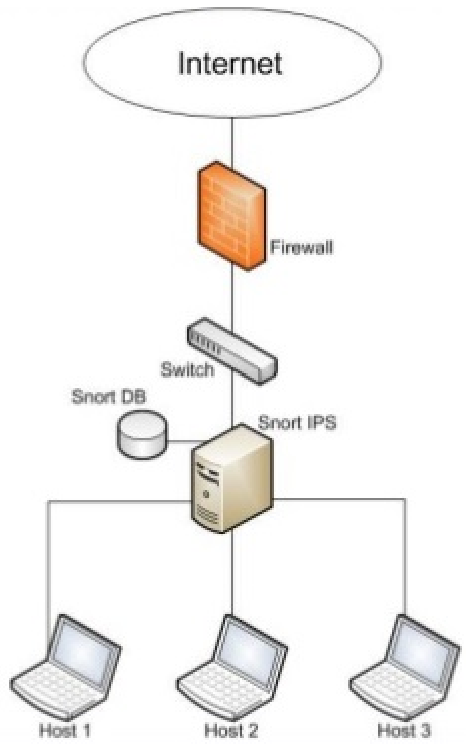


Figure 8: Snort running as a NIPS [10]

2.2.3 Comparison

NIDS	**NIPS**
Detection mode only	Active traffic control
Traffic replication required	Live traffic required
Decoupling detection and reaction functionalities	Detection and reaction support
Good assistant for network administration	No administrator assistance needed
Usually used for testing rules	Requires strict configuration
Passive	**Inline**

Table 2: Snort - NIDS vs NIPS comparison

2.3 Packet Processing Overview

Snort is a signature-based IDS/IPS, which means that, as it receives network packets, it reassembles and normalizes the content so that a set of rules can be evaluated to detect the presence of any significant conditions that merit further action. A rough processing flow is as follows (see figure 9 below):

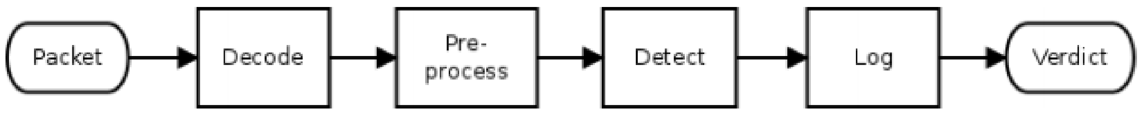


Figure 9: Packet Processing chain in Snort [9]

The different steps are:

1. **Packet** is where the acquisition of network packets takes place.
2. **Decode** each packet to determine the basic network characteristics such as source and destination addresses and ports. A typical packet might have ethernet containing IP containing TCP containing HTTP (e.g. *eth:ip:tcp:http*). The various encapsulating protocols are examined for sanity and anomalies as the packet is decoded. This is essentially a stateless effort.
3. **Preprocess** each decoded packet using accumulated state to determine the purpose and content of the innermost message. This step may involve reordering and reassembling IP fragments and TCP segments to produce the original application protocol data unit (PDU). Such PDUs are analyzed and normalized as needed to support further processing.
4. **Detect** is a two step process. For efficiency, most rules contain a specific content pattern that can be searched for such that if no match is found no further processing is necessary. Upon start up, the rules are compiled into pattern groups such that a single parallel search can be done for all patterns in the group. If any match is found, the full rule is examined according to the specifics of the signature.
5. **Log** is where Snort saves any pertinent information resulting from the earlier steps. More generally, this is where other actions can be taken as well such as blocking the packet.

The first three will be detailed deeper because they are important for the rest of this work.

2.4 DAQs (Packet Acquisition)

Snort 2.9 introduces the DAQ, or Data Acquisition library, for packet I/O. The DAQ replaces direct calls to libpcap functions with an abstraction layer that facilitates operation on a variety of hardware and software interfaces **without requiring changes** to Snort. It allows new flexibility for Snort by separating the network capture functions out into external, loadable modules (DAQs). Thanks to that, developers can now easily create their own modules. DAQ also integrates inline intrusion prevention capability that was previously only available with add-on patches. The DAQ type, mode, variable and directory may be specified either via the command line (when running Snort) or in the configuration file.

2.4.1 Pcap

Pcap (for Packet Capure) is the default DAQ using the pcap API (libpcap library on Unix-like systems). Snort can run in the classic sniffer mode similar to tcpdump. It can record packets to log files or it can run in IDS mode as a daemon.

This DAQ can't be run in inline mode because it is not intended.

2.4.2 Afpacket

This DAQ is available only on Linux. Afpacket works similar to the memory mapped pcap DAQ but no external library is required. It provides better performance.

Afpacket inline is the equivalent to pcap inline (which does not exist), because it is similar and can run in inline mode.

2.4.3 Ipq

IPQ is the old way to process iptables packets. It offers the same functionality provided by the snort_inline patch in earlier versions of Snort. It leverages the queue target in netfilter to move packets from the kernel to the user space application (Snort) for evaluation.

This module has been included with DAQ to support those with legacy configurations who used the snort_inline patch in previous Snort versions. The new module to integrate netfilter and Snort is called NFQ.

2.4.4 Nfq

NFQ is the new and improved way to process iptables packets.

2.4.5 Ipfw

IPFW leverages the divert sockets functionality supported in BSD systems (FreeBSD, OpenBSD).

Divert sockets is similar to the netfilter queue target because it allows the pf or ipfw firewalls running in the kernel to pass traffic off to a user space application for evaluation before inserting it back into the kernel to continue normal processing.

2.4.6 Dump

The DAQ Dump allows to test the various inline mode features available in Snort 2.9 like injection and normalization.

2.4.7 Netmap

Netmap is a framework for very high speed packet I/O. It is available on both FreeBSD and Linux with varying amounts of preparatory setup required.

Inline operation performs Layer 2 forwarding with no MAC filtering, akin to the Af-packet module's behavior. All packets received on one interface in an inline pair will be forwarded out the other interface, unless dropped by the reader and vice versa.

2.5 Decoder

As soon as packets have been gathered, Snort must now decode the specific protocol elements for each packet. The decoder is actually a series of decoders that each decode specific protocol elements. It works up the network stack, starting with lower level Data Link protocols, decoding each protocol as it moves up. After the decoder for Data Link layer, there is a decoder for the Network layer and then for the Transport layer.

As packets move through the various protocol decoders, a data structure is filled up with decoded packet data. As soon as packet data is stored in a data structure, it is ready to be analyzed by the preprocessors and the detection engine.

A non-exhaustive and shortened list of decoders (grouped by network layer) is show in table 3 below.

Network Layer	**Decoder**
Data Link	Ethernet Decoder
Network	IP Decoder
Transport	ICMP Decoder UDP Decoder TCP Decoder

Table 3: Snort - Non-exhaustive list of decoders

2.6 Preprocessors

Preprocessors were introduced in version 1.5 of Snort. They allow the functionality of Snort to be extended by allowing users and programmers to drop modular plugins into Snort fairly easily. Preprocessor code is run before the detection engine is called, but after the packet has been decoded. The packet can be modified or analyzed in an out-of-band manner using this mechanism.

Snort preprocessors fall into two categories. They can be used to either examine packets for suspicious activity or modify packets so that the detection engine can properly interpret them. A number of attacks cannot be detected by signature matching via the detection engine, so "examine" preprocessors step up to the plate and detect suspicious activity. These types of preprocessors are indispensable in discovering non-signature-based attacks. The other preprocessors are responsible for normalizing traffic so that the detection engine can accurately match signatures. These preprocessors defeat attacks that attempt to evade Snort's detection engine by manipulating traffic patterns.

Additionally, Snort cycles packets through every preprocessor to discover attacks that require more than one preprocessor to detect them. If Snort simply stopped checking for the suspicious attributes of a packet after it had set off an alert via a preprocessor, attackers could use this deficiency to hide traffic from Snort. Suppose a black hat intentionally encoded a malicious remote exploit attack in a manner that would set off a low priority alert from a preprocessor. If processing is assumed to be finished at this point and the packet is no longer cycled through the preprocessors, the remote exploit attack would register only an encoding alert. The remote exploit would go unnoticed by Snort, obscuring the true nature of the traffic.

2.6.1 frag3

The frag3 preprocessor is a target-based IP defragmentation module for Snort. It is designed with the following goals:

1. Fast execution with less complex data management.
2. Target-based host modeling anti-evasion techniques

2.6.2 Session

The Session preprocessor is a global stream session management module for Snort. It is derived from the session management functions that were part of the Stream5 preprocessor.

Session API It provides an API to enable the creation and management of the session control block for a flow and the management of data and state that may be associated with that flow by service and application preprocessors. It is used by TCP, UDP, IP, ICMP, for instance.

2.6.3 Stream

The Stream preprocessor is a target-based TCP reassembly module for Snort. It is capable of tracking sessions for both TCP and UDP.

Transport Protocols TCP sessions are identified via the classic TCP "connection". UDP sessions are established as the result of a series of UDP packets from two end points via the same set of ports. ICMP messages are tracked for the purposes of checking for unreachable and service unavailable messages, which effectively terminate a TCP or UDP session.

Stream API Stream supports the modified Stream API that is now focused on functions specific to reassembly and protocol aware flushing operations. Session management functions have been moved to the Session API. The remaining API functions enable other protocol normalizers/preprocessors to dynamically configure reassembly behavior as required by the application layer protocol.

2.6.4 and others...

There are still a lot of preprocessors, too many to be detailed each one by one. Here is a non-exhaustive list of other preprocessors:

- sfPortscan
- HTTP Inspect
- SMTP Preprocessor
- POP Preprocessor
- IMAP Preprocessor
- FTP/Telnet Preprocessor
- SSH
- DNS
- SSL/TLS
- ARP Spoof Preprocessor
- Sensitive Data Preprocessor
- Normalizer
- SIP Preprocessor

3 Snort I/O Performances

The first thing to do is to measure Snort performances when it runs "by default". Indeed, it is interesting to know those basic values to compare them later and see if it has improved something or, at least, if there is still a room for improvement.

Running Snort by default means running Snort with its default DAQ, which is pcap. It is more interesting to run it in inline mode (as a NIPS) instead of passive mode (as a NIDS) because a forwarding of packets is involved. However, as said in previous chapter, the pcap DAQ can't run in inline mode. Fortunately, there is a similar DAQ called afpacket that can run in inline mode.

All tests in this paper will mainly involve Snort in inline mode (***without any drop rules***). Each passive test is obviously faster than its inline equivalent, due mainly to the fact that there is no forwarding, but less interesting than an inline scenario for this work. Passive mode could have been used all along to measure performances instead of inline mode, as long as comparable things are compared. However, as said, the choice of working with the inline mode has been made. Nevertheless, a passive test can also be measured to compare with its inline equivalent.

For all tests in this paper, the network traffic is generated and sent by a real device (device A) to another real device (device B), while Snort is running and listening on device B. This way, the generated traffic is a real network traffic coming from real interfaces. Packets forwarded by Snort are sent back to device A to measure the throughput.

3.1 Packet Acquisition with Afpacket

3.1.1 Configuration

For this use case, Snort runs in inline mode with its "default" DAQ which is afpacket. Inline mode requires at least a pair of interfaces (e.g. *eth0:eth1*). The first interface is where packets are acquired (and also for backward packets) while the second interface is where packets are forwarded. Figure 10 below represents this test.

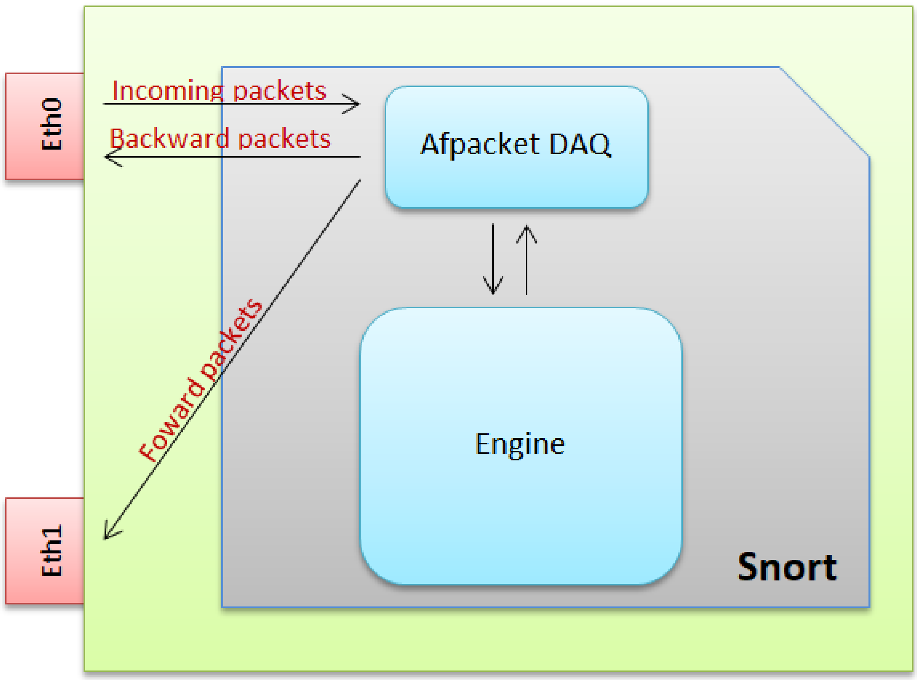


Figure 10: Snort running inline with Afpacket

The network traffic is coming from *eth0*. Snort acquires packets thanks to its afpacket DAQ and applies its internal packet processing chain to each packet. At the end of the process, the DAQ knows the verdict taken by the engine for each packet: pass packet, block packet, etc. If the packet is not blocked, it is forwarded. Sometimes, Snort can send "back" some packets through the backward path, such as injected packets that are generated and sent by Snort itself (e.g. TCP resets).

A quick reminder about Afpacket: it works similar to the memory mapped pcap DAQ but no external library is required. It provides better performance than pcap. However, the assumption is that it is going to be slow because it uses the kernel network stack.

3.1.2 Measurement

The traffic generator (device A) sends constantly at a rate of approximately 7,4 G/s. It measures the throughput on the interface where forwarded packets arrive (from Snort). With *one* UDP flow at this rate, the measured throughput is **51 M/s**. Note that the throughput is much higher (**731 M/s**) in passive mode, which confirms what was said about inline versus passive mode. The same test needs to be done for TCP with another generator because it only generates SYN packets and it does not represent real TCP connections.

As expected, it is quite slow in inline mode. Maybe much slower than expected. Just to make sure it's not wrong and as a comparison, we'll try to acquire packets with NFQ DAQ module, which is an improved way to process *iptables* packets.

3.2 Packet Acquisition with NFQ

3.2.1 Configuration

This use case is almost similar to the previous one. Snort runs in inline mode with NFQ, which does not require a pair of interfaces this time. Instead, we need to add an *iptables* rule to make traffic go through a specific *NFQUEUE* target: *iptables -A FORWARD -j NFQUEUE –queue-num 1* where *1* is the queue number. Then, the queue number must be given to Snort, thanks to NFQ DAQ variables, in order to notify NFQ where the traffic is. With some routing configuration, we allow forwarding just like in the previous use case. Note that *ip_forward* must be enabled. Figure 11 below represents this test.

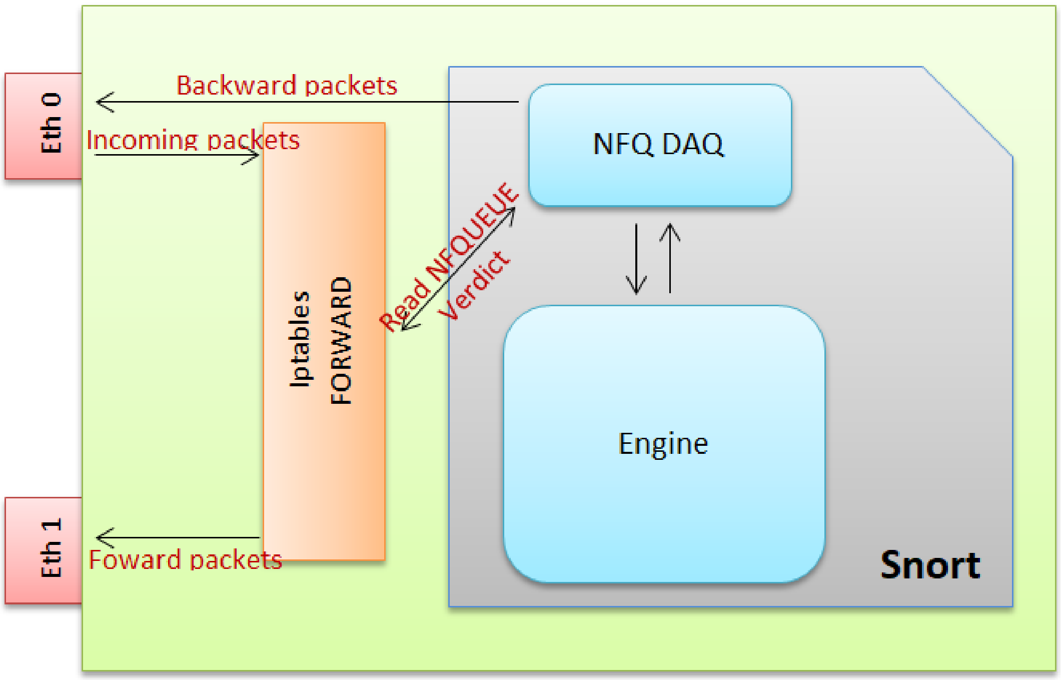


Figure 11: Snort running inline with NFQ

The network traffic is coming from *eth0*. Snort acquires packets thanks to its NFQ DAQ, which reads the NFQUEUE where packets are, and applies its internal packet processing chain to each packet. At the end of the process, the DAQ knows the verdict taken by the engine for each packet: pass packet, block packet, etc. This verdict is used to issue a pass/drop verdict to the nfq. If a packet is not blocked, it is forwarded through the forward path. Sometimes, Snort can send "back" some packets through the backward path, such as injected packets that are generated and sent by Snort itself (e.g. TCP resets).

3.2.2 Measurement

Another related test has been made to measure the kernel forwarding alone, in order to know the maximum throughput possible for these tests. The result is a throughput of **792 M/s**. So, no matter what, we won't have a throughput greater than this one.

The traffic generator (device A) sends constantly at a rate of approximately 7,4 G/s. It measures the throughput on the interface where forwarded packets arrive (from Snort). With *one* UDP flow at this rate, the measured throughput is **210 M/s**. Note that the throughput is exactly the same in passive mode. The reason is that the kernel forwarding happens no matter what, because in passive mode each packets are marked as "pass" for nfq. So, the inline cost is still here in passive mode. In our case, it is not possible to measure a real passive mode for nfq. The same test needs to be done for TCP with another generator because it only generates SYN packets and it does not represent real TCP connections.

As expected, it is four times faster than afpacket. We were rather expecting this kind of value. In fact, this demonstrates that afpacket is not that appropriate for inline mode (and so for forwarding packets), while NFQ is dedicated to inline mode. This ascertainment can be proved by checking overheads while running the inline afpacket test. The observed result is an overhead in *sendTo* function, which confirms that the forwarding slows down the system. Even by modifying a lot of options, afpacket keeps being slow. Those options are for instance increasing the buffer size or disabling gro/lro according to Snort documentation: "*Some network cards have features which can affect Snort. Two of these features are named "Large Receive Offload" (lro) and "Generic Receive Offload" (gro). With these features enabled, the network card performs packet reassembly before they're processed by the kernel. By default, Snort will truncate packets larger than the default snaplen of 1518 bytes. In addition, LRO and GRO may cause issues with Stream target-based reassembly. We recommend that you turn off LRO and GRO*".[11]

It is now time to perform the same measurement but this time with DPDK, which should speed up Snort a lot.

3.3 Packet Acquisition with DPDK

One thing is sure: Snort running inline with Afpacket is slow. It is a bit better with NFQ but we can do much more. A huge improvement would be to use DPDK in the DAQ as it should definitely increase performances. In order to use DPDK to acquire packets, a new DAQ module had to be developed. Fortunately, several can be found already on the

Internet[6,7].

The chosen one has been studied and checked before anything. It looks like the other DAQ modules since the developer inspired from the global logic of existing modules. Specific parts are those using DPDK, for instance by calling some DPDK functions. Here is a non-exhaustive list with most important functions:

- **rte_eth_dev_configure** Configure an Ethernet device. This function must be invoked first before any other function in the Ethernet API. This function can also be re-invoked when a device is in the stopped state.
- **rte_eth_rx_queue_setup** Allocate and set up a receive queue for an Ethernet device.
- **rte_eth_tx_queue_setup** Allocate and set up a transmit queue for an Ethernet device.
- **rte_eth_dev_start** Start an Ethernet device. The device start step is the last one and consists of setting the configured offload features and in starting the transmit and the receive units of the device. On success, all basic functions exported by the Ethernet API (link status, receive/transmit, and so on) can be invoked.
- **rte_eth_dev_stop** Stop an Ethernet device. The device can be restarted with a call to rte_eth_dev_start.
- **rte_eth_rx_burst** Retrieve a burst of input packets from a receive queue of an Ethernet device.
- **rte_eth_tx_burst** Send a burst of output packets on a transmit queue of an Ethernet device.

An Ethernet device is designated by an integer (greater than 0) named the device port identifier. The term *interface* is replaced by a notion of *port* in DPDK context. To be used with DPDK, an interface must be bound, which means being a network device using DPDK-compatible driver instead of a network device using kernel driver.

[6] `https://github.com/btw616/daq-dpdk` (now offline, currently used)

[7] `https://github.com/NachtZ/daq_dpdk/` (online and almost similar to previous, multithread support, not used)

3.3.1 Configuration

For this use case, Snort runs in inline mode with DPDK DAQ. Inline mode requires at least a pair of devices (e.g. *dpdk0:dpdk1*, for ports 0 and 1). Port 0 is where packets are acquired (and also for backward packets) while port 1 is where packets are forwarded. Figure 12 below represents this test.

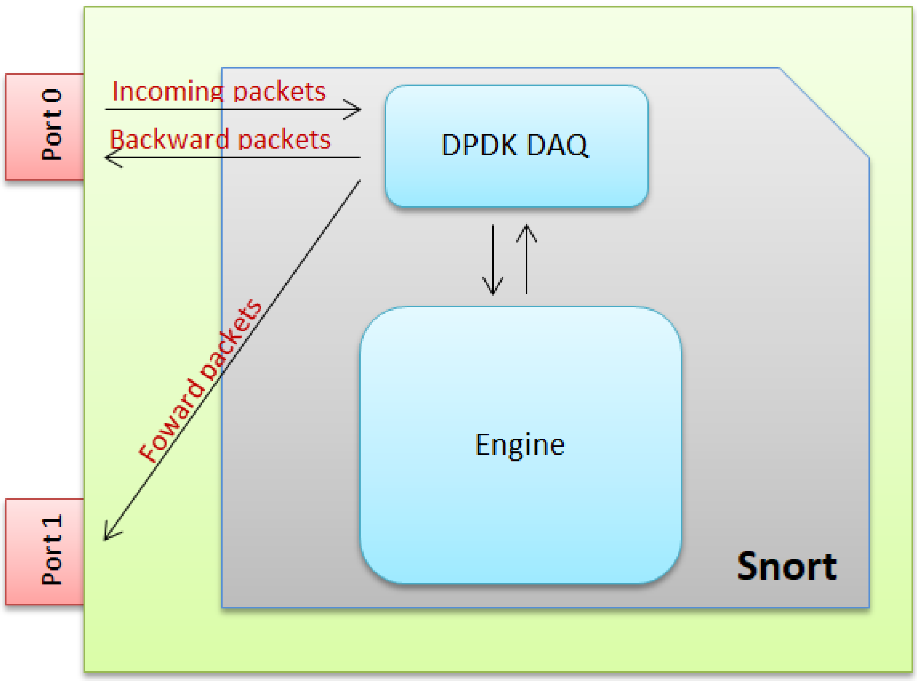


Figure 12: Snort running inline with DPDK

The network traffic is coming from port 0 (*dpdk0*). Snort acquires packets thanks to the DPDK DAQ and applies its internal packet processing chain to each packet. At the end of the process, the DAQ knows the verdict taken by the engine for each packet: pass packet, block packet, etc. If the packet is not blocked, it is forwarded. Sometimes, Snort can send "back" some packets through the backward path, such as injected packets that are generated and sent by Snort itself (e.g. TCP resets).

A quick reminder about DPDK: it is fast because of its properties (see section 1.3). The assumption is that it is going to be faster than Afpacket and NFQ.

3.3.2 Measurement

The traffic generator (device A) sends constantly at a rate of approximately 7,4 G/s. It measures the throughput on the interface where forwarded packets arrive (from Snort). With *one* UDP flow at this rate, the measured throughput is **1,21 G/s** in inline mode.

The same test needs to be done for TCP with another generator because it only generates SYN packets and it does not represent real TCP connections.

As expected, it is faster than Afpacket and NFQ. Let's try to understand why and justify it in next section.

3.4 Results summary

Results of each use case are shown in table 4 below.

Use case	Passive	Inline
Snort (Afpacket)	731 M/s	51 M/s
Snort (NFQ)	/	210 M/s
Snort (DPDK)	-	**1,21 G/s**

Table 4: Measurements result - Snort alone with one UDP flow

Since inline Afpacket is not performing as expected, we will base our comparison on NFQ. Passive measures are only provided to compare Afpacket and NFQ. One can see that the percentage increase is 476.19% (around 6 times better) from NFQ to DPDK. One of the major reasons DPDK is faster is because it provides poll-mode drivers. Basically, rather than the normal NIC driver which uses OS interrupts to deliver packets, the DPDK drivers poll the NIC (no interrupts overhead). This means it trades CPU cycles for latency: one finds out about packet arrival earlier than with interrupts, at the cost of asking. By bypassing the Linux kernel, its overhead is bypassed too, which means no system calls, no context switching on blocking I/O, no data copying from kernel to user space, no interrupt handling in kernel, etc. For more details on why DPDK is faster, please refer to section 1.3.

3.5 Conclusion

DPDK is really nice due to its speed performances. It is definitely faster than Afpacket or NFQ and this confirms that improvements are efficient and useful.

We could stop here and directly use Snort (alone) with DPDK. It would be efficient. The only problem is that any kernel services are lost: no firewall, no NAT, etc. DPDK alone is efficient, but nothing else can be done. There is a need for something else working in front of Snort-DPDK. The next section investigates the communication between processes, by adding an upfront software to Snort and measuring the performances.

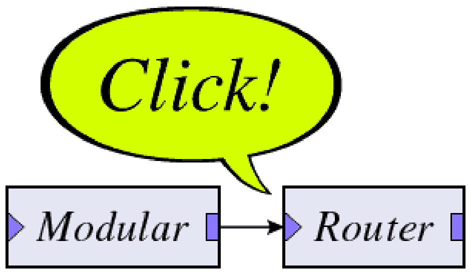
4 Inter-process Communication

"In computer science, inter-process communication (IPC) refers specifically to the mechanisms an operating system provides to allow the processes to manage shared data".[12]

One of the approach is to use, for instance, a **shared memory**: *"Multiple processes are given access to the same block of memory which creates a shared buffer for the processes to communicate with each other"*.[12]

As seen in previous section, DPDK alone is not enough, although it is very efficient. We need *something* in front of it to run a firewall, a NAT and/or a load balancer, for instance, whatever as long as it's realistic. In this work, FastClick will act as the upfront software and communicate with Snort. The idea is to have a lot of things running on FastClick, such as a NAT, a firewall, or anything else.

4.1 Click



Click Modular Router is a flexible, modular software architecture for creating routers. Click routers are built from fine-grained components; this supports fine-grained extensions throughout the forwarding path. The components are packet processing modules called elements. The basic element interface is narrow, consisting mostly of functions for initialization and packet handoff, but elements can extend it to support other functions (such as reporting queue lengths). To build a router configuration, the user chooses a collection of elements and connects them into a directed graph. The graph's edges, which are called connections, represent possible paths for packet handoff. To extend a configuration, the user can write new elements or compose existing elements in new ways, much as UNIX allows one to build complex applications directly or by composing simpler ones using pipes. Several aspects of the Click architecture were directly inspired by properties of routers. First, packet handoff along a connection may be initiated by either the source end (push processing) or the destination end (pull processing). This cleanly models most router packet flow patterns, and pull processing makes it possible to write composable packet schedulers. Second, the flow-based router context mechanism lets an element automatically locate other elements on which it depends; it is based on the observation that relevant elements are often connected by the flow of packets. These features make individual elements more powerful and configurations easier to write. Click was implemented on general-purpose PC hardware as an extension to the Linux kernel.[13]

4.1.1 FastClick

FastClick is an extended version of the Click Modular Router featuring an improved Netmap support and a new DPDK support. It is developed at the University of Liege and is the result of an ANCS paper available at `http://hdl.handle.net/2268/181954`.[14]

This new version implements several improvements in order to increase performances. Here is a list of main improvements[4] related to DPDK and provided by FastClick:

- **I/O Batching**: *Both DPDK and Netmap can receive and send batches of packets, without having to do a system call after each packet read or written to the device.*
- **Zero copy**: *DPDK directly provides a swapped buffer, as such it can directly be given to the transmit ring instead of copying its content.*
- **Compute batching**: *With compute batching, batches of packets are passed between Click elements instead of only one packet at a time, and the element's job is done on all the packets before transmitting the batch further. Linked lists are used to represent batches, for which support is already inside Click, and accommodate better with splitting and variable size batches. Elements which can take advantage of batching inherit from the class "BatchElement" (which inherit from the common ancestor of all elements "Element"). Thereby, packet batching is fully backward compatible with Click elements.*

And others... See related paper[4] for more information about all tests and improvements.

4.2 Packet Acquisition with Afpacket

4.2.1 Configuration

For this use case, it is globally the same principle as for section 3.1. The only big difference is that FastClick is now in-between interfaces and Snort, since we need a third-party. Snort won't be directly listening to incoming interface anymore.

Instead, FastClick will setup a virtual interface (a tap device) and listen on incoming interface. Incoming traffic will be forwarded to virtual interface. The virtual interface is created thanks to the Click element called **KernelTap**.

"KernelTap reads Ethernet packets from and writes Ethernet packets to a /dev/tun or /dev/tap* device. This allows a user-level Click to hand packets to the virtual Ethernet device. KernelTap will also transfer packets from the virtual Ethernet device. KernelTap allocates a /dev/tun* or /dev/tap* device (this might fail) and runs ifconfig to set the interface's local (i.e., kernel) address to ADDR and the netmask to MASK. If a nonzero*

GATEWAY IP address (which must be on the same network as the tun) is specified, then KernelTap tries to set up a default route through that host" [15].

Snort will now listen to this new virtual incoming interface. The same idea is used for the outgoing interface. FastClick and Snort can now communicate and this use case is equivalent to the one in section 3.1, with the addition of an upfront software (FastClick) in front of Snort. Figure 13 below represents this test.

Note: FastClick uses DPDK to read/write from/to (real) interfaces.

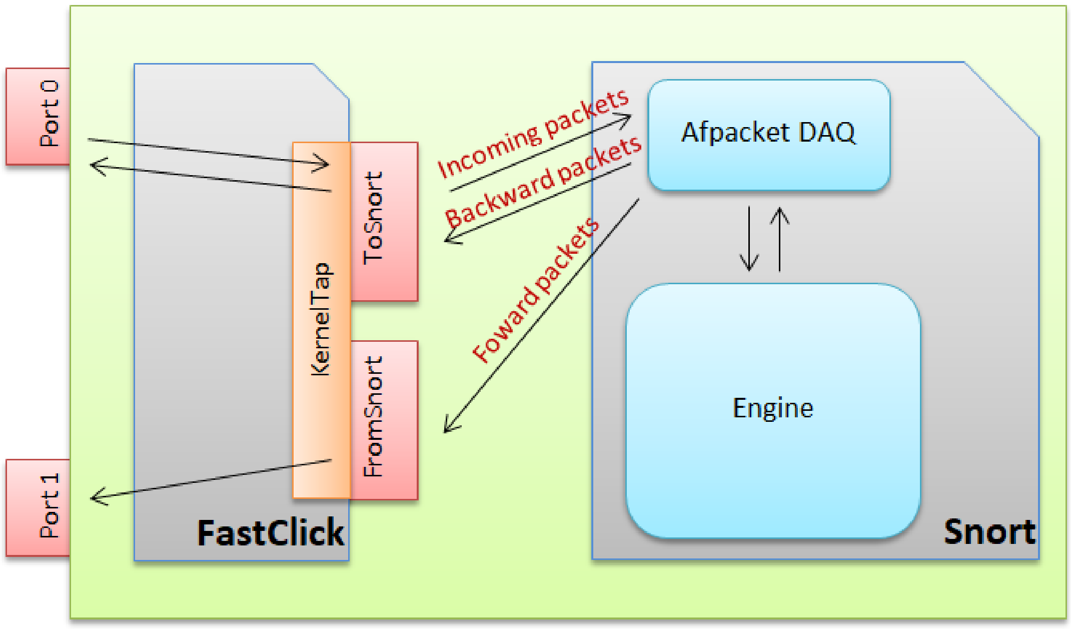


Figure 13: FastClick with Snort running inline with Afpacket

The network traffic is coming from *port 0* (DPDK). FastClick acquires packets thanks to DPDK and forward them to *ToSnort* (virtual interface).

Snort acquires packets from *ToSnort* (virtual interface) thanks to its afpacket DAQ and applies its internal packet processing chain to each packet. At the end of the process, the DAQ knows the verdict taken by the engine for each packet: pass packet, block packet, etc. If the packet is not blocked, it is forwarded to *FromSnort* (virtual interface). Sometimes, Snort can send "back" some packets through the backward path, such as injected packets that are generated and sent by Snort itself (e.g. TCP resets).

FastClick then forwards packets from *FromSnort* to *port 1*, thanks to DPDK. Backward packets (from *ToSnort*), if any, are forwarded to *port 0*, also with DPDK.

4.2.2 Measurement

The traffic generator (device A) sends constantly at a rate of approximately 7,4 G/s. It measures the throughput on the interface where forwarded packets arrive (from device B with FastClick and Snort). With *one* UDP flow at this rate, the measured throughput is **0,33 G/s** in inline mode. The same test needs to be done for TCP with another generator because it only generates SYN packets and it does not represent real TCP connections.

This time, the throughput seems more correct than for Snort alone with afpacket. It can easily be explained by the fact that FastClick uses DPDK, which is faster and avoids problems encountered in the test with afpacket alone. And even if it uses *KernelTap*, it is not excessively heavy, which explains that the throughput is a bit higher than inline NFQ.

Actually, we don't need the kernel at all. That's a good news because it's slow and sometimes unpredictable (see afpacket alone). Instead, DPDK can be used again. And more specifically, we could use a DPDK ring as communication between FastClick and Snort, in order to replace the need of the kernel.

4.3 Packet Acquisition with DPDK Ring

Since a ring is a shared memory, some people may not understand why a classic shared memory (*shmem*) is not used instead. The answer is easy: if so, packets would still need to be copied in the shared memory. Indeed, with DPDK, there's no need to copy packets because they are already there. Hence, it's faster to use a DPDK ring, in a DPDK context.

In a DPDK context means that the upfront software (FastClick) already uses DPDK to acquire and forward packets. Thereby, FastClick simply needs to store packets in a DPDK ring for Snort, and no copy is involved. Moreover, FastClick is responsible for allocating resources (pool, rings) because it runs as a DPDK primary process. Snort (its DAQ), on the other hand, runs as a DPDK secondary process and only needs to find resources that already exist in order to acquire them (e.g. get a ring by its ID, get a pool by its ID). That's quite easy.

The new DPDK Ring DAQ module has been implemented that way. It is based on the same global logic of DPDK DAQ, without all its now-useless calls to configuration functions. Previously, a lot of functions were called to configure the NIC, queues, etc. Now, we don't need them anymore. Indeed, only the *link* (ring) between FastClick and Snort needs to be "configured" (actually, this is the primary process that creates/configures it; Snort only acquires an existing resource).

Here is a **complete** list of *Ring* functions:

- **rte_ring_lookup** Search a ring from its name.

- **rte_ring_dequeue_burst** Dequeue multiple objects from a ring up to a maximum number. This function calls the multi-consumers or the single-consumer version, depending on the default behavior that was specified at ring creation time.
- **rte_ring_enqueue_burst** Enqueue several objects on a ring. This function calls the multi-producer or the single-producer version depending on the default behavior that was specified at ring creation time.

Notice how small the list is. It shows how easy and lighter the new DPDK Ring DAQ module is compared to its DPDK equivalent.

4.3.1 Configuration

For this use case, FastClick with DPDK is still used as an upfront software in front of Snort, which is running in inline mode with new DPDK Ring DAQ. FastClick needs to run first because it allocates resources needed by Snort. It creates three rings: one for sending packets to Snort, one for packets sent by Snort through the backward path, and one for packets sent by Snort through the forward path. Snort acquires packets by reading in the first ring and then writes packets to the other two rings. Figure 14 below represents this test.

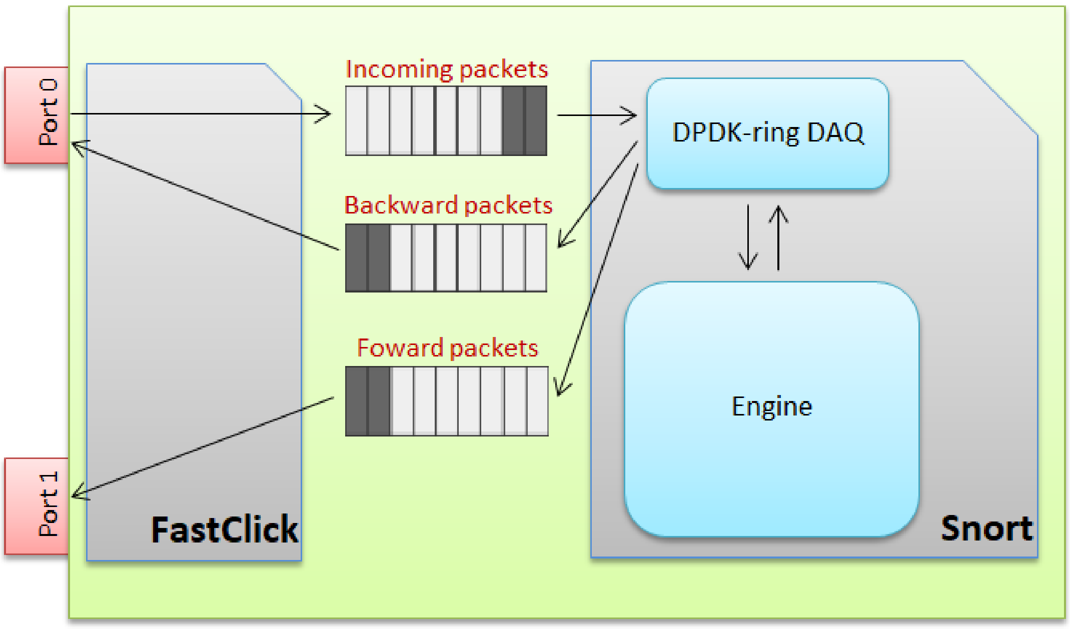


Figure 14: FastClick with Snort running inline with DPDK ring

The network traffic is coming from *port 0* (DPDK). FastClick acquires packets thanks to

DPDK and forward them to the ring dedicated to incoming packets.

Snort acquires packets by reading the ring thanks to its DPDK-Ring DAQ and applies its internal packet processing chain to each packet. At the end of the process, the DAQ knows the verdict taken by the engine for each packet: pass packet, block packet, etc. If the packet is not blocked, it is forwarded to the ring dedicated to the forward path. Sometimes, Snort can send "back" some packets through the backward path, such as injected packets that are generated and sent by Snort itself (e.g. TCP resets).

FastClick then forwards packets, read from the ring dedicated to forward packets, to *port 1*, thanks to DPDK. Backward packets (read from the ring dedicated to backward packets), if any, are forwarded to *port 0*, also with DPDK.

A quick assumption is that DPDK-Ring is going to be much more efficient than Afpacket.

4.3.2 Measurement

The traffic generator (device A) sends constantly at a rate of approximately 7,4 G/s. It measures the throughput on the interface where forwarded packets arrive (from device B with FastClick and Snort). With *one* UDP flow at this rate, the measured throughput is **0,99 G/s** in inline mode. The same test needs to be done for TCP with another generator because it only generates SYN packets and it does not represent real TCP connections.

As expected, it is faster than FastClick and Snort running inline with Afpacket. Let's try to understand why and justify it in next section.

4.4 Results summary

Results of each use case are shown in table 5 below.

Use case	**inline**
FastClick, Snort (Afpacket)	0,33 G/s
FastClick, Snort (DPDK ring)	**0,99 G/s**

Table 5: Measurements result - FastClick and Snort with one UDP flow

As expected, FastClick/Snort(DPDK Ring) is faster than FastClick/Snort(Afpacket). For more details, go back to section 1.3 about DPDK: same reason than the comparison between Snort-alone (Afpacket) versus Snort-alone (DPDK). Snort is now almost able to handle 1 G/s. There's a percentage increase of 200%, which is 3 times better.

Also, one can easily see that Snort alone running DPDK DAQ is faster than FastClick with Snort running DPDK-Ring DAQ. This could raise some doubts in people's minds, asking themselves why. The explanation is simple: with Snort alone, we do less things. Indeed, there's no FastClick running. Now, someone could protest again and ask why we don't stop here and use Snort alone with DPDK. This is the good timing to remind people about one important aspect of this work that will remove any persistent doubts. First, as said previously, Snort alone with DPDK does not allow to have a firewall, a NAT or whatever in front of it. Moreover, the goal was not to speed up Snort itself but to speed up Snort for a communication between NFVs, optimizing the communication itself.

4.5 Conclusion

FastClick in front of Snort running DPDK-Ring shows a good improvement. But we're not totally satisfied yet. We know that Snort is doing some jobs that the upfront software (FastClick) already did. In a NFV cooperation context, it is a waste of time and we could improve it by sort-of synchronizing them. What if FastClick had a way to notify Snort not to do a job already done ? What if, even better, FastClick had also a way to send some information to Snort in order to save some time ? This idea is investigated in next section and tries to improve the cooperation between two NFVs.

5 NFV Cooperation

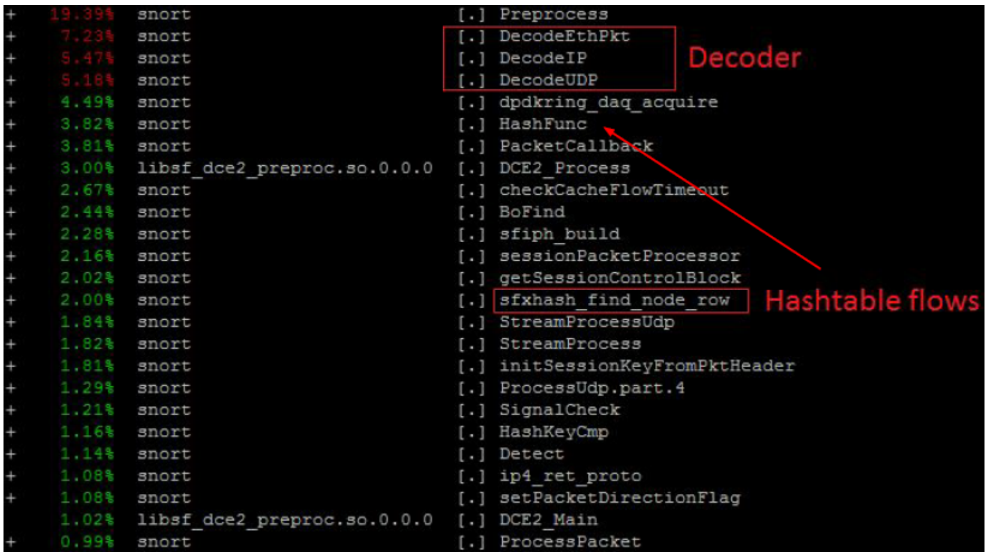
Since we need a third-party, an upfront software in front of Snort, we also need to speed up the communication between the two. The main goal of this section focuses on finding ways to improve the communication, and so the cooperation, between two NFVs.

We already know that, sometimes, they work on same jobs during a communication, which is a stupid waste of time. As said previously, it's nice to improve the efficiency of Snort itself but there is still something to improve in a cooperation context. It would be nice to find a solution to avoid those jobs repetition. For that, one could simply search for redundancies in Snort compared to what the upfront software already does. One could also directly check for overheads in Snort, by profiling it when running in some interesting situations. In fact, we will first check for overheads and then see if they are already included in a job performed by the upfront software (FastClick). This way, we will find some critical parts in Snort that could be improved in order to improve the cooperation between NFVs. Note that this will inevitably improve Snort's velocity too (because less jobs to perform).

5.1 Snort monitoring

There are several possibilities to monitor Snort while running. One of them is provided by Snort itself and is activable in the configuration. Thanks to that, it is for instance possible to check the overhead over all of its preprocessors. The only issue with this solution is that it outputs results at the end of the execution, which does not allow a live monitoring. A better choice is thus an easy tool called *perf top* which is a system profiling tool that generates and displays a performance counter profile in real time.[16] It's quite useful in order to see which functions are considered as overheads during execution. Since Snort is running on a specific core, we can isolate the monitoring by profiling that one specific core.

Figure 15 below shows a *perf top* output example of Snort running with DPDK-Ring DAQ on a specific core. It will allow us to see functions where most of the time is spent.



```
+   19.39%  snort                          [.] Preprocess
+    7.23%  snort                          [.] DecodeEthPkt
+    5.47%  snort                          [.] DecodeIP
+    5.18%  snort                          [.] DecodeUDP
+    4.49%  snort                          [.] dpdkring_daq_acquire
+    3.82%  snort                          [.] HashFunc
+    3.81%  snort                          [.] PacketCallback
+    3.00%  libsf_dce2_preproc.so.0.0.0    [.] DCE2_Process
+    2.67%  snort                          [.] checkCacheFlowTimeout
+    2.44%  snort                          [.] BoFind
+    2.28%  snort                          [.] sfiph_build
+    2.16%  snort                          [.] sessionPacketProcessor
+    2.02%  snort                          [.] getSessionControlBlock
+    2.00%  snort                          [.] sfxhash_find_node_row
+    1.84%  snort                          [.] StreamProcessUdp
+    1.82%  snort                          [.] StreamProcess
+    1.81%  snort                          [.] initSessionKeyFromPktHeader
+    1.29%  snort                          [.] ProcessUdp.part.4
+    1.21%  snort                          [.] SignalCheck
+    1.16%  snort                          [.] HashKeyCmp
+    1.14%  snort                          [.] Detect
+    1.08%  snort                          [.] ip4_ret_proto
+    1.08%  snort                          [.] setPacketDirectionFlag
     1.02%  libsf_dce2_preproc.so.0.0.0    [.] DCE2_Main
+    0.99%  snort                          [.] ProcessPacket
```

Figure 15: Snort running with DPDK-Ring DAQ on 1 core and receiving one UDP flow

Note that the received traffic is *one* UDP flow at a rate of *7,4 G/s*. On figure 15 above, one can directly see that the *Preprocess* function is where 20% of the time is spent. It is normal and predictable since Snort uses a lot of preprocessors. Moreover, we have already improved the acquisition part, while Snort is running without any drop rule. Thus, the bottleneck is located at the decoder/preprocessors level. A first idea to reduce preprocessors overhead is to disable some unneeded preprocessors in the configuration. But this alone won't be enough.

By investigating Snort preprocessors list, one will be able to locate which preprocessors are responsible of this overhead. The result incriminates the *Session* preprocessor, which especially involves those functions:

- **HashFunc**: *Function responsible for hashing a data tuple to obtain a key*
- **checkCacheFlowTimeout**: *Check for any flow that is timed out (with a view to removing it)*
- **getSessionControlBlock**: *Get a SCB that contains all data about a session*
- **sfxhash_find_node_row**: *Hashtable function to find a node by key*
- **HashKeyCmp**: *Function responsible for the comparison between two hash nodes*

But, this is not all. There are still some functions where the time spent is higher than others. They represent the *Decoder* part and are:

- **DecodeEthPkt**: *Decodes an Ethernet frame (checks the header, fills in internal structures, etc)*
- **DecodeIP**: *Decodes an IP frame (checks the header, fills in internal structures, etc)*
- **DecodeUDP**: *Decodes an UDP frame (checks the header, checks options, fills in internal structures, etc)*

A total of 4 tests were carried out in order to compare each possible case for UDP and TCP:

- *One* **UDP** flow
- *Many* **UDP** flows
- *One* **TCP** flow
- *Many* **TCP** flows

Results for *one UDP flow* is already known and showed on figure 15. The same test with *many UDP flows* only increases a bit the overhead on the *Session* preprocessor but does not change anything in the *Decoder*. It can easily be explained by the fact that each packet has to be decoded, no matter how many flows are involved, while many flows involve more allocations for new session structures. Also, regardless of the flows number, a packet is classified each time. The same two tests are applied to TCP and lead to almost the same observation, except that TCP-related functions are involved instead of UDP (a lot more functions because TCP is more complicated than UDP). Indeed, the logic does not change for TCP when headers are checked and packets are classified, but the overhead is a bit higher due to the complexity of TCP.

Now that we are aware of what causes an overhead, let's try to compare those jobs with what the upfront software (FastClick) does. Actually, FastClick is already able to check headers. Thereby, it becomes a useless job for Snort. Also, there are usually stateful elements in FastClick, such as a NAT, which keep a hash table of flows to manage flow classification. Thereby, again, this job also becomes useless for Snort. We will focus on a way to pass those data to Snort in order to be faster. But, right before, let's have a deeper look at those two jobs, what they are exactly and what they do.

5.2 Headers Check

For tests realized and seen above, one UDP flow (or TCP flow) was used. Basically, it is a UDP (or TCP) packet encapsulated in an IP packet, which is itself encapsulated in an Ethernet frame. Figure 16 below shows a detailed structure example for UDP.

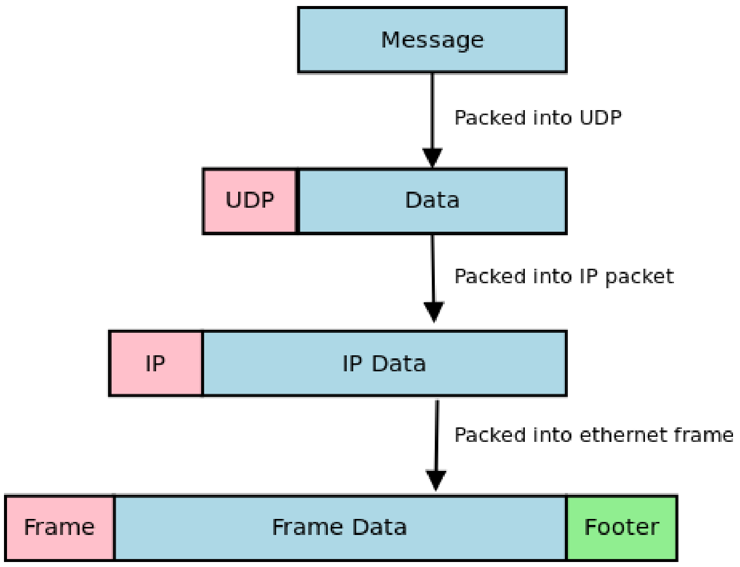


Figure 16: Structure of a UDP/IP packet (*eth:ip:udp*) [17]

Thanks to the figure 16 above, one can easily guess the equivalent example for TCP. It also shows that each type of packet has its own header containing necessary data. Thereby, there is a need to check each header because they could contain wrong/corrupted data (intentionally or introduced by error due to a bad communication channel). Globally, this is always the same routine, with specific requirements depending on the header type. The routine could be described as follows: check that the header length is correct, then check the value of each field if needed and finally, if required, check the *checksum* value and compare it to the computed *checksum* (they must match). Quick reminder: a check sum is basically a value that is computed from data packet to check its integrity. Figures 17, 18, 19 and 20 below show the header structure of several packet type.

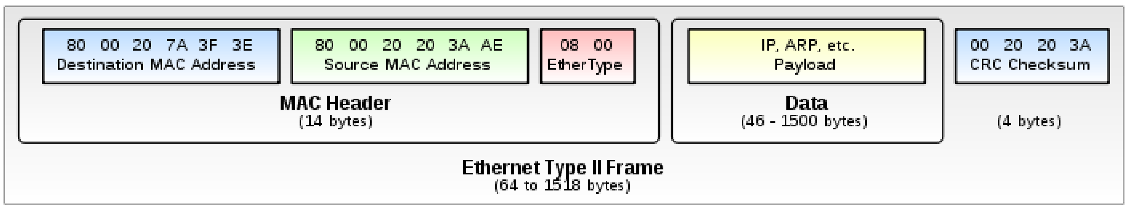


Figure 17: The most common Ethernet Frame format, type II [18]

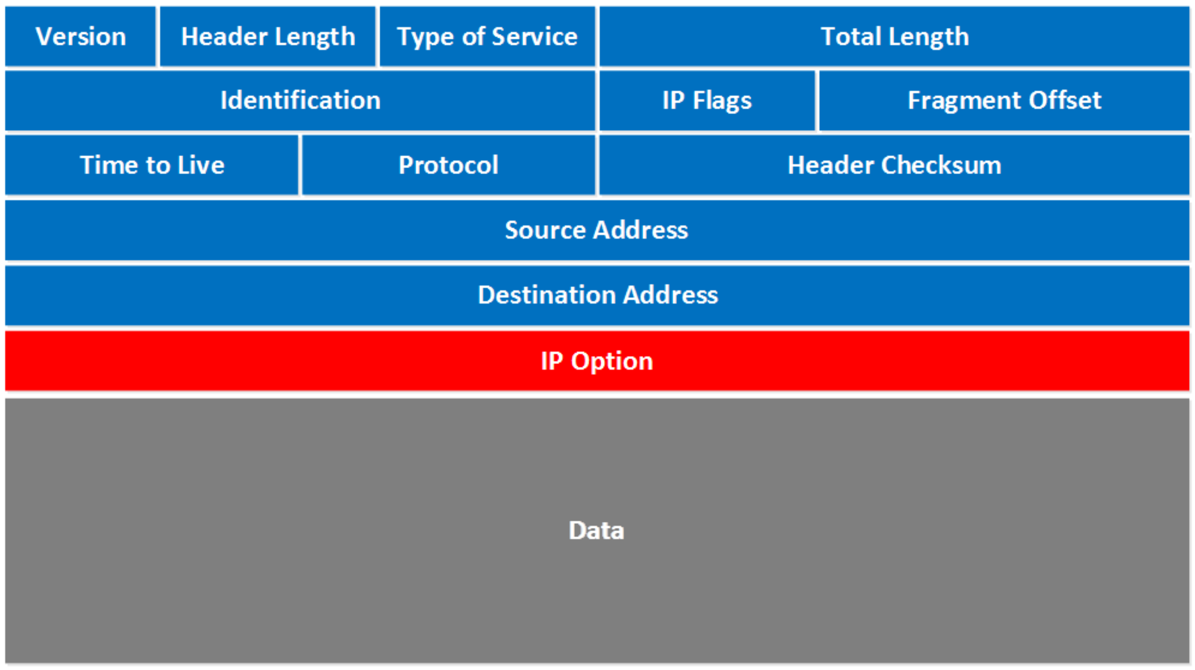


Figure 18: IPv4 header [19]

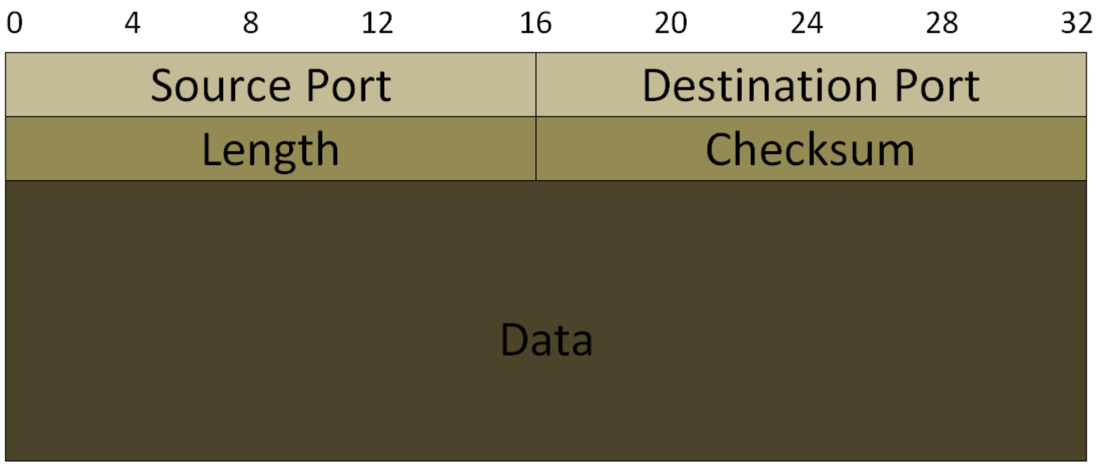


Figure 19: UDP header [20]

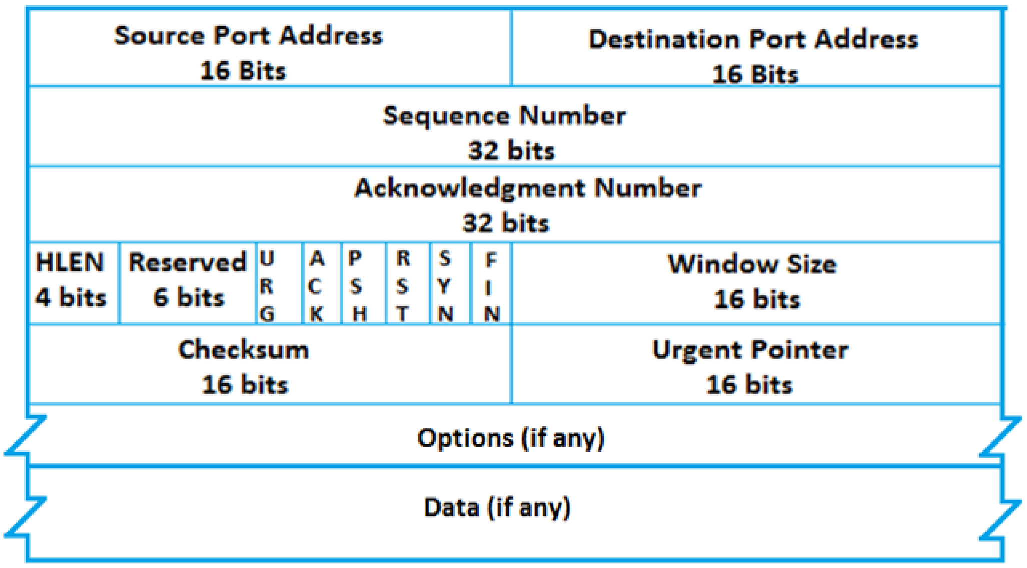


Figure 20: TCP header [21]

As a proof of concept for this work, we only focus on UDP (*eth:ip:udp*) and TCP (*eth:ip:tcp*). All headers check are avoided by Snort for these two when the upfront software (FastClick) says it so, in case it has already checked them. This means that those verification happen in FastClick, not in Snort anymore.

5.3 Flows classification

A **flow** is a traffic stream of network packets that share the same identifiers. Typically, it's defined by traffic with same source/destination IP addresses, same source/destination ports and same protocol.

A quick reminder about the *Session* preprocessor of Snort: it provides an API to enable the creation and management of the session control block for a flow and the management of data and state that may be associated with that flow by service and application preprocessors. It is used by TCP, UDP, IP, ICMP, for instance.

Basically, a flow classification consists of determining to which flow a packet belongs to. It is done by hashing some packet IP-fields such as source/destination ports and addresses. The output hash value is the key identifying a flow. To retrieve an entry in the flow table (hash table), a search is performed based on the key. If the key is not stored in the flow table, it's a new flow and a new structure is allocated. Otherwise, a packet is considered as part of an existing flow. An overview of how it works is represented in figure 21 below.

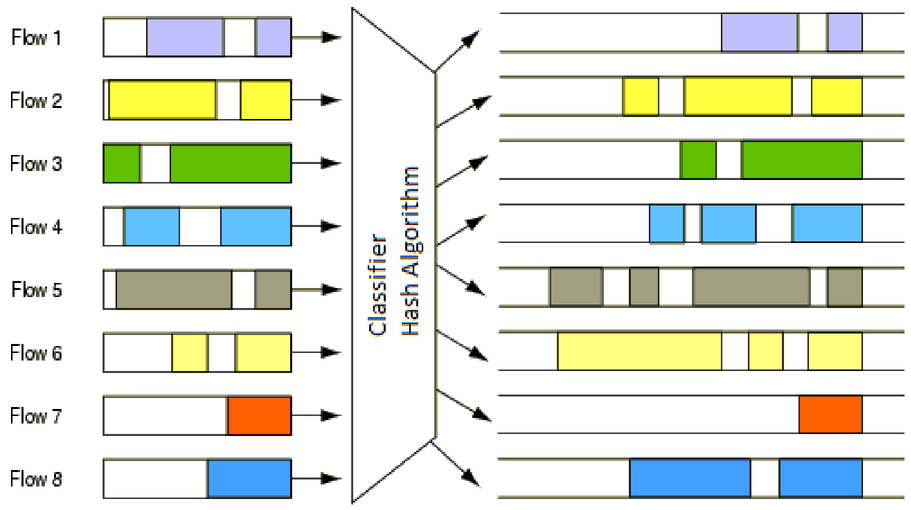


Figure 21: Flows classification

As said previously, FastClick has usually stateful elements, such as a NAT, which keep a hash table of flows. Snort does the same, which is a waste of time. By passing those flow IDs directly to Snort, it allows Snort to bypass costly hashes because the classification can be done instantaneously. Snort will use these IDs to classify packets in its own flow table, which will be faster.

5.4 Metadata

Now that useless jobs seen above have been deeper investigated, let's focus on the way to pass some data to Snort in order to avoid those time consuming repetitions. We need to find a way to tell Snort not to perform headers check when FastClick already does it. Moreover, we also need to give Snort a flow ID attached to each packet, because there's no need for Snort to have its own flow classification while FastClick already has it. This solution of passing some data from the upfront software (FastClick) to Snort is called metadata. Those metadata will contain needed information in order to make Snort act as expected.

5.4.1 FastClick implementation

In order to add this notion of metadata, we first need to modify a bit FastClick. When it acquires packets with DPDK, each packet is stored in a DPDK structure called *rte_mbuf*. See *Mbuf Library*[22] for more information. Figure 22 below shows an overview of this structure.

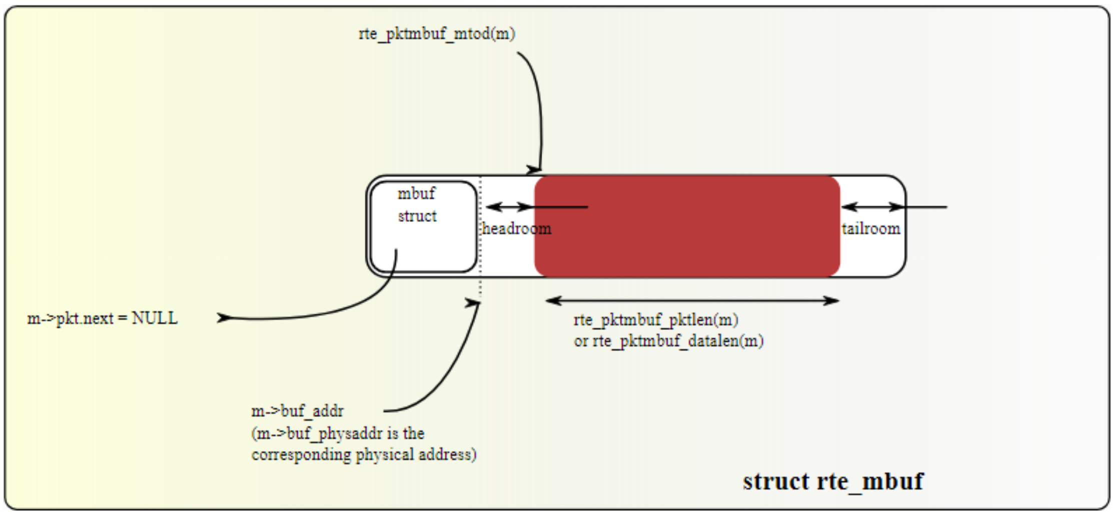
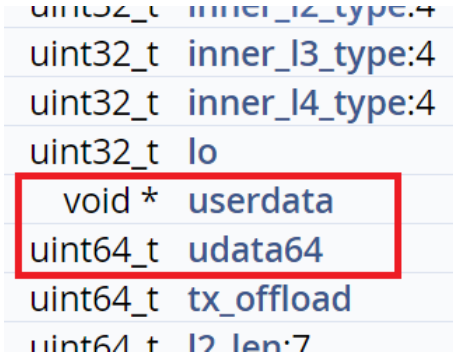


Figure 22: rte_mbuf structure - An mbuf with One Segment[22]

Before FastClick pushes packets to the DPDK ring, we need for each packet to add meta-data inside. This structure contains some unused/private fields reserved for user data. This is the case for two of them, which are *userdata* and *udata64*, as shown below on figure 23.



uint32_t	inner_l2_type:4
uint32_t	inner_l3_type:4
uint32_t	inner_l4_type:4
uint32_t	lo
void *	userdata
uint64_t	udata64
uint64_t	tx_offload
uint64_t	l2_len:7

Figure 23: Fields used to pass metadata[23]

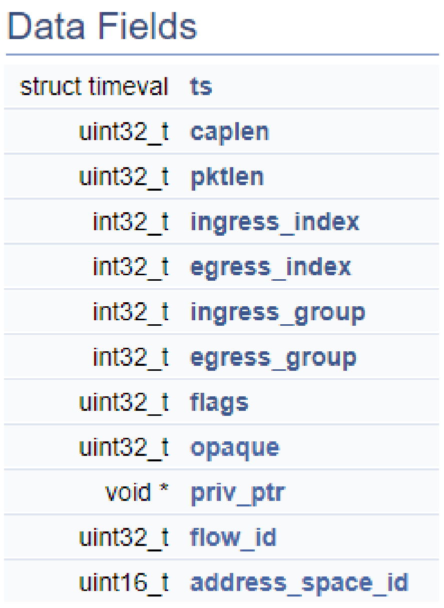
The choice to use *userdata* was made, which is a pointer. Actually, we will store the flow ID (a 32 bits integer) inside the pointer, as a virtual address. This way, the other side (Snort) receiving it will simply check if *userdata* is *NULL* or not. If *NOT NULL*, it means that Snort does not have to perform headers check because the upfront software already does it. Also, the flow ID is retrieved by casting the false address to a 32 bits integer. Another better solution could be to split those two data in distinct fields, such as using *userdata* pointer to tell Snort to skip headers check or not (value *NULL* or *NOT NULL*) and *udata64* to carry the flow ID (more appropriate). The initial solution was taken because *udata64* wasn't well documented and, as it was working as expected and didn't require more data, it was kept unchanged.

Note that there are also other possibilities if we needed to pass more data. For instance, the *rss_hash* field could be used to pass the flow ID, which would make *userdata* or *udata64* available for something else. Another possibility is to store extra data inside the *headroom* or inside the *tailroom* directly (see figure 22 above), which is not recommended but it should work as expected.

5.4.2 Snort implementation

Without a surprise, we also need to modify Snort to implement those metadata. Inside the DPDK Ring DAQ first, then inside Snort engine.

Inside the DAQ. When a packet is acquired, it is sent to the engine. In fact, what is sent is a structure (a sort-of header information) and raw packet data. We need to pass received metadata, from *rte_mbuf* (which is not sent because Snort does not need it), to the engine. This is done thanks to that header structure named *_daq_pkthdr*[24]. Figure 24 below shows its definition. Both fields *priv_ptr* and *flow_id* were chosen to carry metadata, because they are either reserved for that purpose or not used by Snort. In *priv_ptr*, which is a pointer, we simply store the value of *userdata* from *rte_mbuf*. In *flow_id*, which is a 32 bits integer, we store the value of *userdata* (after a cast). This way, the engine will be able to check two different things: if it needs to perform headers check (*priv_ptr* is *NULL* or not) and if a flow ID is provided (*flow_id* is greater than 0 or not).


Data Fields

struct timeval	ts
uint32_t	caplen
uint32_t	pktlen
int32_t	ingress_index
int32_t	egress_index
int32_t	ingress_group
int32_t	egress_group
uint32_t	flags
uint32_t	opaque
void *	priv_ptr
uint32_t	flow_id
uint16_t	address_space_id

Figure 24: Structure *_daq_pkthdr*[24]

Inside Snort engine. As said previously, the engine will check both *priv_ptr* and *flow_id* when needed. Mainly two files have been modified in order to take metadata into consideration.

- **src/decode.c** *This is the decoder, where all packets layers are decoded. Snort checks the value of priv_ptr and performs headers check or not, depending on the value (NULL or NOT NULL).*
- **src/preprocessors/spp_session.c** *This is the Session preprocessor, where packets are classified in flows. Snort performs a lookup directly on the value of flow_id or on the flow, depending on the value (greater than 0 or not).*

5.4.3 Configuration

Figure 14 (see section 4.3) shows the configuration for this test. Indeed, this is exactly the same test, except that this time FastClick passes metadata to Snort. Also, in order to simulate a stateful function inside of FastClick (such as a NAT), we will use a Click element named **AggregateIPFlows**: "*AggregateIPFlows monitors TCP and UDP flows, setting the aggregate annotation on every passing packet to a flow number, and the paint annotation to a direction indication. Non-TCP/UDP packets and short packets are emitted on output 1, or dropped if there is no output 1. AggregateIPFlows uses source and destination addresses and source and destination ports to distinguish flows. Reply packets get the same flow number, but a different paint annotation. Old flows die after a configurable timeout, after which new packets with the same addresses and ports get a new flow number. UDP, active TCP, and completed TCP flows have different timeouts. Flow numbers are assigned sequentially, starting from 1. Different flows get different numbers. Paint annotations are set to 0 or 1, depending on whether packets are on the forward or reverse subflow. (The first packet seen on each flow gets paint color 0; reply packets get paint color 1. ICMP errors get paints 2 and 3.) AggregateIPFlows can optionally apply aggregate annotations to ICMP errors.*"[25]. This way, FastClick will classify each packet into flows and will be able to pass the aggregate annotation (the flow ID) through metadata. Thereby, Snort flow table will be faster, thanks to lookups based directly on the flow id.

A quick reminder: *AggregateIPFlows* is only here as a simulation because, usually, FastClick already has such functions (a NAT, for instance) and so the job is already done. In such case, we could directly use the flow table of NATs to remember a unique ID like *AggregateIPFlows* does.

The assumption is that the "*with metadata*" version will inevitably be faster than the "*without metadata*" version.

5.4.4 Measurement

The traffic generator (device A) sends constantly at a rate of approximately 7,4 G/s. It measures the throughput on the interface where forwarded packets arrive (from device B with FastClick and Snort). With *one* UDP flow at this rate, the measured throughput is **1,14 G/s** in inline mode. The same test needs to be done for TCP with another generator because it only generates SYN packets and it does not represent real TCP connections.

Also, the same test needs to be done with many flows instead of only one flow. It was not possible but we figured it out why and found an alternative. In fact, the generator only generates different ports with same IP addresses. Usually, there are a few ports pairs but much more IP addresses pairs. Thereby, ports use a linked list, which makes *AggregateIPFlows* really slow. The bottleneck is not Snort anymore and thereby we can't trust this measurement because it does not reflect the reality. The alternative was to test it with traces from the University of Liege, which contain real network traffic. This way, we don't need *AggregateIPFlows* anymore and it is always interesting to observe how Snort behaves in real situations, with improvements. The result is unchanged and we can assume that it works with much more flows, as expected. Another alternative would be to force the generator to randomize IP addresses.

As expected, it is faster than the "*without metadata*" version, which is not a surprise since Snort is doing less useless things.

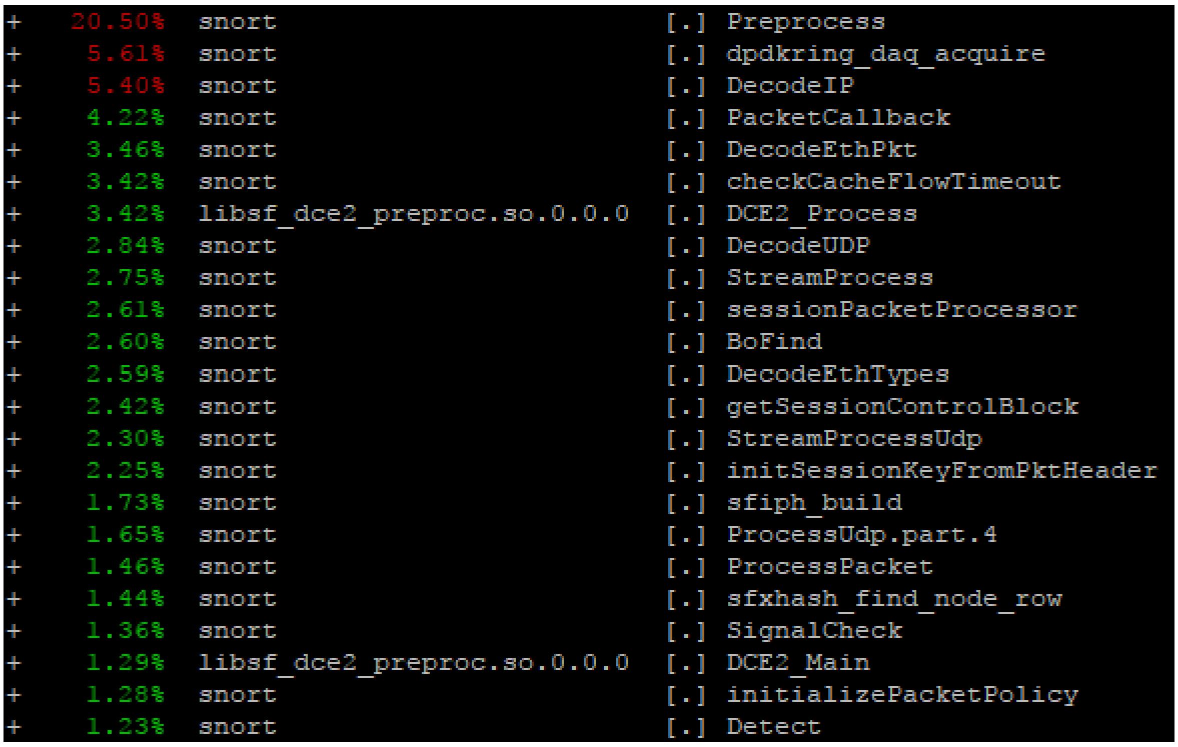
5.5 Results summary

A comparison of results between "*without metadata*" (see table 5) and "*with metadata*" versions is shown in table 6 and figure 25 below.

Use case	**inline**
FastClick, Snort (DPDK ring **without** metadata)	0,99 G/s
FastClick, Snort (DPDK ring **with** metadata)	**1,14 G/s**

Table 6: Measurements result - FastClick and Snort (DPDK Ring) with one UDP flow

As said previously, the metadata version is faster because it allows Snort to do less useless things. One can observe on table 6 that there's a percentage increase of 15%, which is not negligible. Note also that we almost reach the performance of Snort alone with DPDK (see table 4), which also means that we almost cancel the cost of an upfront stateful function (e.g. a NAT in FastClick). And this is a really, really good point !



```
+   20.50%  snort                          [.] Preprocess
+    5.61%  snort                          [.] dpdkring_daq_acquire
+    5.40%  snort                          [.] DecodeIP
+    4.22%  snort                          [.] PacketCallback
+    3.46%  snort                          [.] DecodeEthPkt
+    3.42%  snort                          [.] checkCacheFlowTimeout
+    3.42%  libsf_dce2_preproc.so.0.0.0    [.] DCE2_Process
+    2.84%  snort                          [.] DecodeUDP
+    2.75%  snort                          [.] StreamProcess
+    2.61%  snort                          [.] sessionPacketProcessor
+    2.60%  snort                          [.] BoFind
+    2.59%  snort                          [.] DecodeEthTypes
+    2.42%  snort                          [.] getSessionControlBlock
+    2.30%  snort                          [.] StreamProcessUdp
+    2.25%  snort                          [.] initSessionKeyFromPktHeader
+    1.73%  snort                          [.] sfiph_build
+    1.65%  snort                          [.] ProcessUdp.part.4
+    1.46%  snort                          [.] ProcessPacket
+    1.44%  snort                          [.] sfxhash_find_node_row
+    1.36%  snort                          [.] SignalCheck
+    1.29%  libsf_dce2_preproc.so.0.0.0    [.] DCE2_Main
+    1.28%  snort                          [.] initializePacketPolicy
+    1.23%  snort                          [.] Detect
```

Figure 25: Snort running with DPDK-Ring (metadata) on 1 core and receiving one UDP flow

Thanks to *perf top*, we can see metadata improvements on figure 25 above by comparing it to the equivalent test without metadata (see figure 15). Indeed, we clearly see that the overhead on functions related to the decoder has been decreased. This is the same for Session preprocessor functions *HashFunc* and *HashKeyCmp* which are directly impacted by improvements and don't even appear on the screen anymore. Note that the *Preprocess* function has still the same overhead, which could be explained by the fact that there are other preprocessors and most of the time is spent in that part (since there is no drop rule for the detector). Those overhead reductions match perfectly the throughput difference between the two tests.

5.6 Conclusion

FastClick passing metadata in front of Snort, running DPDK-Ring, shows a quite nice improvement. This time we could stop here because it shows a satisfactory performance. However, we always want more and metadata improvements open a door to other perspective of improvements. One of these new ideas is about the flow table in Snort. Now, thanks to metadata, the classification is faster but Snort still manages flows itself (flow deletion after a purge, a timeout, etc). It would be more interesting to have a unified system of flows shared between NFVs. For that, the upfront software (FastClick) could "pilot" Snort in order to delete its own flows. We could also extend this system to more than two NFVs. This idea is investigated in next section.

6 Unified State

What if NFVs didn't have to manage internal states themselves, but instead develop a system where an element would be responsible for maintaining an unified state, whatever that state is about, between everyone ? This is possible as long as NFVs share some similar states. It could be a nice solution to avoid management repetition of a state, since they're sometimes treating the same data and the same job is performed by more than one NFV.

For instance, in this work, the upfront software (FastClick) runs a stateful function, such as a NAT, and maintains a table of flows. Snort does the same. In previous section, a faster flows classification was implemented in Snort, thanks to metadata. But still, Snort manages itself its own flow table, such as flow deletion. Let's see what we could do to avoid those useless actions and to delegate the responsibility to the upfront software, with a view to having a unified and centralized state.

6.1 NFV Flow State

The upfront software (FastClick) would act as a controller for unifying flow state. A nice way to notify another NFV (e.g. Snort) to delete a flow would be through our new metadata system. We could add in the metadata a flow ID representing the flow that should be freed. A first improvement would be to allow sending multiple flow IDs at a time, which should be better because faster. The big problem with this solution is that we need to wait for a packet to be sent. Let's imagine the case when there is no packet. Well, it wouldn't be possible to send those metadata, which is not acceptable because it makes no sense to unify states if we can't notify directly when a flow has to be freed. This leads to another better idea, based on this one.

Actually, we need to be independent of packets to be sent in order to directly send flow state information. To do this, we could send meta packets built exclusively for that purpose. This is the best solution proposed here but a choice has still to be made: either FastClick sends meta packets to a dedicated ring or to the current ring for packets. The first option allows to exclusively treat meta packets in priority in a dedicated ring while the second option allows to spare space memory (no extra ring).

A totally different possibility without using metadata would be that the upfront software (FastClick) share its flow table and give only read access to other NFVs. This would be the best option to spare memory space because only one flow table would exist. But this is definitely not the most elegant (nor modular) solution.

And finally, perhaps some other ideas that we haven't thought of...

6.2 Conclusion

This solution is still at the theory step and has not been implemented yet. But this would certainly be a good point and give an improvement. Moreover, this would be a better system thanks to a unified and centralized state of flows. It would be more modular and could also lead to the discovery of other new improvements.

7 Future Improvements

The very first thing to do would be to use a decent TCP generator in order to measure all UDP equivalent measurements seen during this work. We know it works and performances are not dropping, thanks to a test realized with real traces. But, anyway, it would be more professional to do so, no matter what.

The next interesting thing to do would be implementing a unified flow state system, as seen in previous section, which could certainly bring a better performance and a cleaner system.

Another nice option to investigate would be to use the freshly new version of Snort, which is Snort 3 (aka *Snort++* or *Snort 3.0 Alpha 4*). Indeed, it is claimed to be more efficient than Snort 2. The new version comes with nice features, such as:

- Support multiple packet processing threads
- Make key components pluggable

Only those two, among others, would make the whole thing more faster and modular. More faster thanks to multithreading support. More modular because we wouldn't need to directly modify Snort's core anymore. Since each part is now pluggable, we should only add a new plugin to implement new things. Moreover, it is now developed with C++, which is object-oriented and a bit more high-level than C. As a consequence, the code looks more organized and more robust. It would be worth a try to compare both versions and check the difference between performances.

Finally, one could focus on new ideas to continue improving performances. After all, there's always something to improve when we deeply analyze it. Only the sky is the limit.

8 Conclusion

One of the main goal of this work was to improve the speed of Snort in a NFV cooperation context. We started by measuring performances of Snort alone, in its original version, with several available data acquisition modules. As expected, it was not that fast, although acceptable. A first observation showed that the packets acquisition could be improved. For that purpose, we introduced DPDK in this work, which is a fast I/O framework, used for fast packet processing. Then, we measured again the performances of Snort alone, this time using DPDK, and saw that it was way more efficient as it is claimed to be.

In a view of NFV cooperation, we needed to add an element that would make some jobs before Snort, such as a NAT, a firewall or whatever as long as it is realistic. To achieve that, we used FastClick (an improved framework based on Click Modular Router) which especially uses DPDK. In order to have a comparison, we reiterated the same tests with the original Snort but this time with FastClick running in front of it, acting as an upfront software. Again, observed performances showed that some improvements were possible and needed. Using DPDK was the solution, especially using DPDK rings between FastClick and Snort to exchange packets. Those rings serve as shared memories.

Trying to find new improvements, we found out that Snort was performing some jobs that were already done by FastClick. Since we were in a NFV cooperation context, it was clearly a waste of time. A nice improvement was to introduce the notion of metadata, used by FastClick to send needed information to Snort in order to speed up the cooperation and avoid the latter from doing useless jobs. Again, as each step, it showed a good performance improvement. At the end of this work, we developed a theoretical technique to have a unified state system between NFVs, and specifically a unified flow state, which should also improve performances.

All of these steps above were conclusive and gave satisfiable improvements. We can now affirm and conclude that it is worth having improved everything for a cooperation between NFVs because it almost reaches the same performance as Snort alone using DPDK, which is fast. So, it almost cancels the cost of an upfront software, which is a really good point.

Finally, a quick reminder about the very last and ultimate goal of this work that was to import ("mimic") the detection engine of Snort inside FastClick. We decided that it would be a bad idea to do so for two reasons. Firstly because we may be less efficient than Snort itself. Secondly, and mainly, because it would require a constant update depending on Snort's changes. It could be painful and laborious. As a conclusion, we will keep using FastClick in front of Snort with all improvements, which is a better compromise.

9 List of Figures

10 List of Tables

11 References

[1] ETSI, “Network Functions Virtualization.” http://www.etsi.org/images/files/ETSITechnologyLeaflets/NetworkFunctionsVirtualization.pdf, 2016. [Online].

[2] Moor Insights & Strategy, “The Battle of SDN vs. NFV.” http://www.moorinsightsstrategy.com/the-battle-of-sdn-vs-nfv/. [Online].

[3] OpenWrt Wiki, “Networking in the Linux Kernel.” https://wiki.openwrt.org/doc/networking/praxis. [Online].

[4] T. Barbette, C. Soldani, and L. Mathy, “Fast Userspace Packet Processing.” http://orbi.ulg.ac.be/bitstream/2268/181954/1/userspaceio.pdf, May 2015. [Online].

[5] QLogic, “Network Function Virtualization Using DPDK.” http://www.qlogic.com/Resources/Documents/WhitePapers/Adapters/WP-Network_Function_Virtualization_Using_Data_Plane_Dev_Kit.pdf. [Online].

[6] D. Haryachyy, “Understanding DPDK.” https://www.slideshare.net/garyachy/dpdk-44585840. [Online].

[7] J. Ronciak, J. Fastabend, D. Zhou, M. Chen, and C. Liang, “DPDK - What it is and where it’s going.” http://www.it-events.com/system/attachments/files/000/001/102/original/LinuxPiter-DPDK-2015.pdf?1448979544. [Online].

[8] A. Shrut, “DPDK for Layman.” https://www.linkedin.com/pulse/dpdk-layman-aayush-shrut. [Online].

[9] The Snort Project, “SNORT Users Manual 2.9.9.” https://www.snort.org/documents/snort-users-manual. [Online].

[10] S. Tino, “Snort IPS.” https://www.slideshare.net/SimoneTino/snort-ips. [Online].

[11] Snort, “Snort Manual - Packet Acquisition.” http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node7.html. [Online].

[12] Wikipedia, “Inter-process communication.” https://en.wikipedia.org/wiki/Inter-process_communication. [Online].

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek, “The Click Modular Router.” https://pdos.csail.mit.edu/papers/click:tocs00/paper.pdf. [Online].

[14] T. Barbette, “Fastclick.” https://github.com/tbarbette/fastclick. [Online].

[15] Click Modular Router, "KernelTap Element Documentation." http://read.cs.ucla.edu/click/elements/kerneltap. [Online].

[16] Linux die.net, "perf-top(1) - Linux man page." https://linux.die.net/man/1/perf-top. [Online].

[17] James Slocum, "UDP/IP Frame." http://jamesslocum.com/images/udp_ip_frame.png. [Online].

[18] Wikipedia, "Ethernet frame." https://en.wikipedia.org/wiki/Ethernet_frame. [Online].

[19] NetworkLessons.com, "IPv4 Packet Header." https://networklessons.com/cisco/ccnp-route/ipv4-packet-header/. [Online].

[20] Cisco Skills, "UDP Header." https://ciscoskills.net/2011/03/28/understanding-udp/udp-header/. [Online].

[21] NetworkLessons.com, "Lesson 2. Transmission Control Protocol (TCP)." https://networkinglessons.wordpress.com/page/2/. [Online].

[22] DPDK, "Mbuf Library." http://dpdk.org/doc/guides/prog_guide/mbuf_lib.html. [Online].

[23] DPDK, "rte_mbuf Struct Reference." http://dpdk.org/doc/api/structrte__mbuf.html. [Online].

[24] Fossies Dox, "daq 2.0.6 _daq_pkthdr Struct Reference." https://fossies.org/dox/daq-2.0.6/struct__daq__pkthdr.html. [Online].

[25] Click Modular Router, "AggregateIPFlows Element Documentation." http://www.read.cs.ucla.edu/click/elements/aggregateipflows. [Online].