eXpress Data Path (XDP) Hands-On Introduction & Benchmarking

Henrique Vicente
Network Engineering Master's Degree
Universidade do Porto
up202005321@up.pt

Rogério Rocha
Network Engineering Master's Degree
Universidade do Porto
up201805123@up.pt

Telmo Ribeiro
Computer Science Master's Degree
Universidade do Porto
up201805124@up.pt

***Abstract*—This project aims to demonstrate the use of *XDP* in Linux distributions to accelerate packet processing and filtering while analyzing its performance. It is accompanied by examples of *XDP* use cases as motivation, reproducing some of its capabilities in an introductory and hands-on manner. This paper is part of a greater work composed by the *feasibility stage* and the *implementation stage*. The current iteration pertains to the *implementation stage* where the objectives, environment and the initial findings mentioned upon the first iteration are explored while delving into some *XDP* framework details. *eXpress Data Path (XDP)* is present in the Linux ecosystem since its 4.8 version. A technology that leverages *extended Berkeley Packet Filter (eBPF)* to execute a finer and faster control on packet processing. Traditional packet processing methods struggle to mitigate latency and offer suboptimal performance, *XDP* addresses these problems through processing in earlier layers on the network stack.**

***Index Terms*—eXpress Data Path, XDP, linux networking, packet processing, eBPF, extended Berkeley Packet Filter, benchmarks, cybersecutity, load balancing**

I. Introduction

Packet processing, filtering, and secure transmission are areas subject to continuous studies, both evolving through a multitude of efforts regarding network protocols and tools developed by engineers and scientists alike, pushing the underlying technologies forward.

The viability of such technologies depends on the continuous pursuit of improvements in areas such as load balancing, network analysis and network safety. Topics like firewalling and overall distributed denial-of-services (DDoS) mitigation are already benefiting from the aforementioned improvements and will be briefly mentioned during this development [1].

A partial solution regarding the aforementioned problems was provided through the development of the *eXpress Data Path (XDP)* framework, acting on the lower layers of the Linux network stack, enabling developers to install programs that process packets on the Linux kernel [2].

Extended Berkeley Packet Filter (eBPF) is a key concept in *XDP* development providing both an instruction set and an execution environment in the Linux kernel, where programs are modified in runtime, not requiring kernel recompilation [3].

Technologies developed over it, can often benefit from bypassing later processing by higher network layers, deciding on the earlier layers what the fate of a packet should be thus, accelerating the packet processing task.

Such is the case for the *XDP* framework, where a program is developed with the goal to express a set of behaviours which is translated into custom packet processing, implemented through the object recurring calls for every incoming network packet [3].

XDP can be implemented in three different modes: **(1) *SKB/Generic XDP*** - where programs are loaded as part of the ordinary network path but yet able to let packets bypass layer layers on the stack, generally allowing the least gain in performance from all modes but not requiring explicit support from specific hardware, **(2) *Native XDP*** - where the program is aided by the network drivers thus requiring support from the driver itself but resulting in measurable performance gains, and lastly **(3) *Offloaded XDP*** - where the program is loaded on the Network Interface Card (NIC), requiring support from the hardware itself but consequently able to bypass CPU cycles therefore, providing the greatest relative performance. *Offloaded XDP* unavoidably requires NIC support and as of our first revision only Intel, Mellanox, and Broadcom have NIC families extending this *XDP* capability.

XDP programs can implement intricate behaviours and surpass the few hundred lines of code however, they are unavoidably composed of primitive actions modeling the fate of each packet.

This primitive actions are ***XDP_PASS***- allowing the ordinary path through the network stack, ***XDP_DROP***- dropping the packet without trace point exceptions, ***XDP_ABORTED***- dropping the packet with trace point exceptions, ***XDP_TX***- transmitting the packet back into the transmitting NIC, and ***XDP_REDIRECT***- redirecting the packet to another NIC.

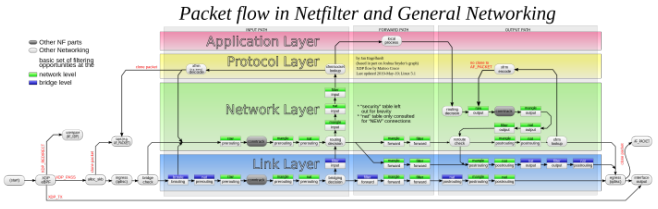

Fig. 1. *XDP* datapath in the Linux kernel

Figure 1 presents in greater detail the *XDP* packet flow in

the Linux kernel (*Offloaded XDP*).

With the packet being processed by the *eBPF* object, ***XDP_DROP*** will trivially cause the packet to not be further processed and thus, the primitive is missing from the previous figure. ***XDP_ABORTED*** behaves in a similar fashion. ***XDP_PASS*** will hand the packet to the Linux network stack which will behave almost accordingly to the standard procedure. ***XDP_TX*** can avoid kernel processing and be redirected to the transmitting interface. ***XDP_REDIRECT*** behaves in a similar manner to ***XDP_TX*** but the packet is handed to the target interface (not necessarily the one who transmitted it).

At the local view, *XDP* allows a finer and faster control over packet processing. The macroscopic view amounts to networks less prone to attacks and with minor delays when fully exploiting its capabilities. The latter can be explained by the performance gain on packet processing, which contributes in reducing the instances where the aforementioned step is the network bottleneck.

This paper is built upon the *feasibility stage* where the environment setup was asserted and the *feasibility test* was performed.

Initially, we offer an overview on the *XDP* framework while providing some background on the concept. Then it follows with the environment used to program and test the *XDP* toolset. Latter, we develop and analyze some programs pertaining to simple and yet incisive use cases. Finally, we provide and discuss the results of our own benchmarks.

II. Objectives

This section is reserved to enumerating and describe the project objectives. These are related to the *feasibility stage*, the project's initial milestones, and the *implementation stage*, the current interaction.

Feasibility Objectives:

- Environment - achieve an environment, through tools and components described in the following section, where the development of software, its execution, and its benchmarking can be performed.
- Feasibility - a feasibility proof validating both the capability of the ecosystem to provide results and laying the ground for the *implementation stage*.

Implementation Objectives:

- Hands-On *XDP*- development and analysis of *XDP* programs, tackling concrete scenarios regarding packet processing and filtering such as, filtering packets by their IPv4/IPv6 header fields and dropping packets from exploitable protocols.
- Benchmarking - development and benchmark of *XDP* primitive actions, this constitutes the main goal on the *implementation stage*.
- *XDP* Experience - section devoted to explaining the advantages of using *XDP* while regarding the rough edges.

III. Environment

The environment section concerns the tools, components, and overall resources employed to achieve the results that follow.

A. Operative System

XDP is supported by the Linux kernel, since version 4.8, and currently supported by Windows through agreements with NIC manufacturers.

The *implementation stage* described in the following sections was achieved in Ubuntu 24.04 LTS. Ubuntu supports *XDP* and is a major Linux distribution accommodated by the standard set up process developed by the *Red Hat* team [4].

B. Libraries

The *implementation stage* was implemented with recourse to the following three major libraries:

- **xdp-tools** - the set of essentials used to fully utilize the *XDP* framework [5].
- **libxdp** - the standard library to operate within the *XDP* environment [5].
- **libbpf** - the standard library to operate within the *eBPF* environment [6].

C. Building/Dumping

In order to build and optimize *eBPF* objects, a combination of GNU Compiler Collection (gcc) and Clang was leveraged. The object was dumped and analyzed with Low-Level Virtual Machine (LLVM).

D. Network Debugging Tools

Network debugging tools and overall Unix commands were essential to load/unload and test the execution of *eBPF* objects. Among others, **ping -4** and **ping -6** were used to send IPv4 and IPv6 packets in order to quickly examine crucial behaviour.

E. Virtual Ethernet Interfaces

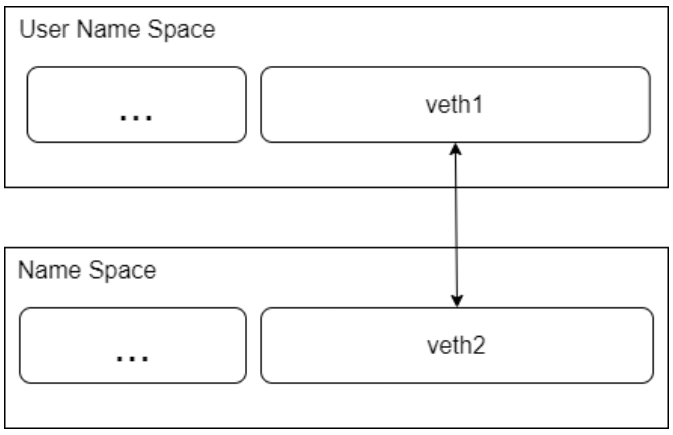

Fig. 2. Simple Network Topology

Virtual Ethernet Interfaces (veths) were used in order to create a simple network topology (figure 2) to test *XDP*

programs. The majority of network drivers do not support *Native XDP* and through the usage of veths, this was no longer a concern as veths inherently support the *Native XDP* mode.

The commands used in the creation of this structure can be found in a bash script present at the appendix.

IV. *XDP* Basics

This section is devoted to exploring the *XDP* syntax and workflow through a simple example. As such, to improve the readability and clarity from the previous paper we will be using an introductory style approach [7]. Readers familiar with the basics of *XDP* could skip this section entirely while those wanting to (re-)establish the foundations are invited to try replicating the examples.

A. *XDP Methodology*

The Introduction regarded the implementation of *XDP* through three different modes: **(1)** *SKB/Generic XDP*, **(2)** *Native XDP*, and **(3)** *Offloaded XDP*.

The studied scenario is performed over *SKB/Generic XDP* so the reader can easily replicate it without the need for specific hardware.

Virtual Ethernet Interfaces (veths) mentioned in the Environment section could be used to test *Native XDP*.

If the hardware is not an issue, the process remains the same except for the loading step.

The **step by step** approach to development within the framework is as follows:

1) Program Development;
2) Object Building;
3) Object Loading;
4) Object Analysis;
5) Object Unloading;

B. *Program Development*

Aiming to improve the previous iteration, we will drop the *feasibility stage* test case for a simpler example. The *Feasibility Test* and many other programs can still be found in the appendix.

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

SEC("xdp_drop")
int xdp_drop_prog(struct xdp_md* ctx) {
    // drop all packets
    return XDP_DROP;
}
char _license[] SEC("license") = "GPL";
```

Listing 1. *C* program that drops all packets.

SEC is used to place the compiled object fragments into different *Executable and Linkable Format (ELF)* sections.

The function **xdp_drop_prog** accepts a parameter of type **struct xdp_md***. This struct allows the access to the packet delimiters but it is not fully explored in the current scenario.

XDP_DROP is returned in every packet indiscriminately, essentially dropping all possible packets passing through the loaded interface.

Finally, the last line regards to the license associated with the program. Some *eBPF* helpers are accessible only by *General Public License (GPL)* [7].

C. *Object Building*

```
$ clang -O2 -g -Wall -target bpf -c xdp_drop.c -
o xdp_drop.o
```

The previous command compiles the **xdp_drop** object using **-O2** option to specify the optimization level.

To get debugging and warning data **-g** and **-Wall** were used.

D. *Object Loading*

```
$ sudo xdp-loader load -m skb -s xdp_drop veth1
xdp_drop.o
```

The **xdp_loader** from xdp-tools is being used to **load** the object in *SKB/Generic XDP* into **veth1**.

The ip command can be used to load *XDP* programs in *SKB/Generic XDP* if employed in a similar fashion to its use in the *feasibility stage*.

E. *Object Analysis*

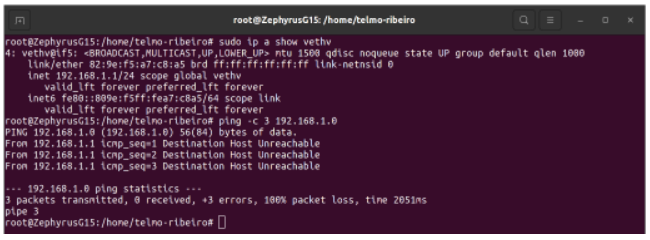

Fig. 3. ping output from the source interface

Figure 3 displays the ping command output from the source interface.

Notice the output **Destination Host Unreachable** when pinging the interface loaded with xdp_drop. This matches the expected scenario when the loaded object is dropping all the packets. Since the Internet Control Message Protocol (ICMP) request packets are dropped before being fully processed by the Linux network stack on the destination interface, then the matching reply will not be sent.

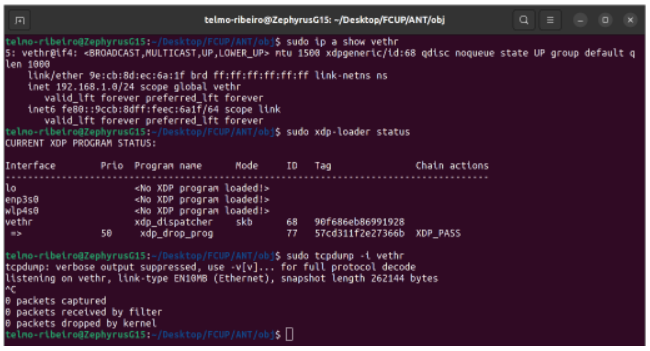

Fig. 4. xdp-loader status and tcpdump output from the target interface

Figure 4 displays the xdp-loader status and tcpdump output from the target interface.

Notice there are no ICMP requests being captured before being dropped. This matches the expected behaviour from *XDP*. Since the packets are being dropped at such an early layer on the linux network stack, tcpdump is not able to flag them. This could constitute a problem for debugging *XDP* programs if not for the tool explained in the next section.

F. *Object Unloading*

```
$ sudo xdp-loader unload -a veth1
```

The **xdp_loader** is used with the flag **-a** to **unload** all the objects from **veth1**.

The ip command can also be used to unload *XDP* programs previously loaded in *SKB/Generic XDP* if employed in a similar fashion to its use in the *feasibility stage*.

V. *XDP* Hands-On

This section aims at motivating the reader to get familiar with *XDP* capabilities.

It was hinted during the Introduction that *XDP* can be applied in problems such as load-balancing and firewalling. The remaining section, is directed to illustrate a simple yet practical *XDP* scenario.

A. *A Simple DDoS Mitigation*

A common form of DDoS, named the ping flood attack, relies on sending more pings than the target machine can successfully process [8]. Internet Control Message Protocol (ICMP/ICMPv6) is the underlying protocol used on ping and we can avoid suffering extensive damage against it in a myriad ways, including firewalls. This next example implements a possible *XDP* based solution that drops both ICMP and ICMPv6 packets, a step to mitigate the aforementioned scenario.

B. *Program Developed*

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/ipv6.h>
#include <arpa/inet.h>

SEC("xdp_ddos_mitigation")
int xdp_ddos_mitigation_prog(struct xdp_md* ctx) {
    // get packet crucial delimeters:
    void* data_end = (void*)(long)ctx->data_end;
    void* data     = (void*)(long)ctx->data;
    // drop packet without eth header:
    if (data + sizeof(struct ethhdr) > data_end) {
        return XDP_DROP;
    }
    struct ethhdr* eth = data;
    // protocol = IPV4:
    if (eth->h_proto == htons(ETH_P_IP)) {
        // drop packet without ipv4 header:
        if (data + sizeof(struct ethhdr) + sizeof(
    struct iphdr) > data_end) {
            return XDP_DROP;
        }
        struct iphdr* iph = data + sizeof(struct
    ethhdr);
        // protocol = ICMP:
        if (iph->protocol == IPPROTO_ICMP) {
            // drop packet
            return XDP_DROP;
        }
        // pass IPV4 packet
        return XDP_PASS;
    }
    // protocol = IPV6:
    if (eth->h_proto == htons(ETH_P_IPV6)) {
        // drop packet without ipv6 header:
        if (data + sizeof(struct ethhdr) + sizeof(
    struct ipv6hdr) > data_end) {
            return XDP_DROP;
        }
        struct ipv6hdr* ipv6h = data + sizeof(struct
     ethhdr);
        // protocol = ICMPv6
        if (ipv6h->nexthdr == IPPROTO_ICMPV6) {
            // drop packet
            return XDP_DROP;
        }
        // pass IPV6 packet
        return XDP_PASS;
    }
    // pass default packet
    return XDP_PASS;
}
```

Listing 2. *C* program that drops all ICMP packets

The actions performed after all checks fail is **XDP_PASS** capturing the default behaviour.

If a packet is missing its header or it is not accessible (sanity conditions) they are dropped.

Finally, if the protocol is either ICMP or ICMPv6 then the action performed is **XDP_DROP**.

Trivially, this program behaves accordingly to the previous stipulation.

C. *Object Analysis*

After building and dumping it, the generated object was then loaded in *Native XDP* into **vethr**, a virtual ethernet interface residing in the user namespace.

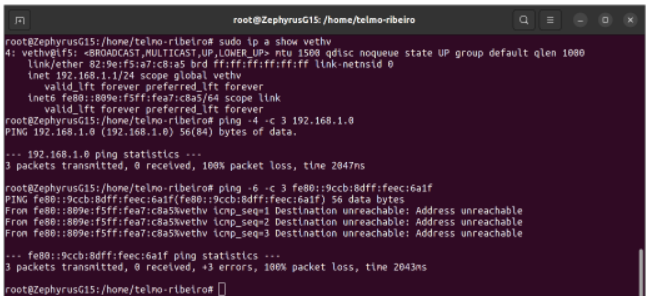

Fig. 5. ping output from source interface

Figure 5 displays the ping command output from the source interface.

Notice the output is either **Destination Unreachable: Address Unreachable** or just plain timeouts when pinging the interface loaded with **xdp_ddos_mitigation**. This matches the

expected scenario when the loaded object is dropping all the ICMP/ICMPv6 packets.

Since ICMP/ICMPv6 request packets are dropped before being fully processed by the linux network stack on the destination interface, then the matching reply will not be sent as it was the case for the initial example.

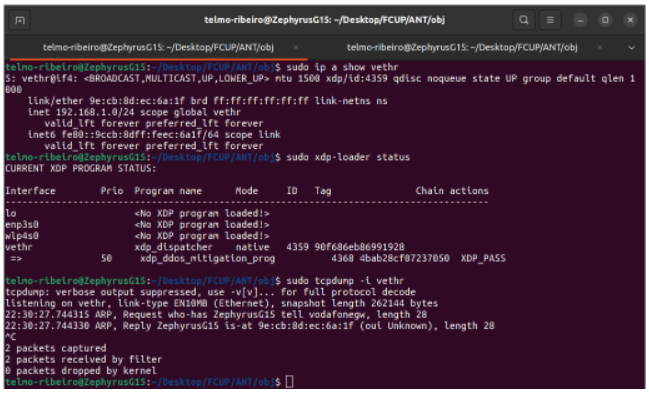

Fig. 6. xdp-loader status and tcpdump output from target interface

Figure 6 displays the xdp-loader status and tcpdump output from the target interface.

As it was the case with the previous scenario, no ICMP/ICMPv6 requests are getting captured before being dropped and it happens for the same reason, tcpdump does not operate at such an early layer.

The next snippet conveys the solution to capturing packets pre/pos modification by *XDP* programs.

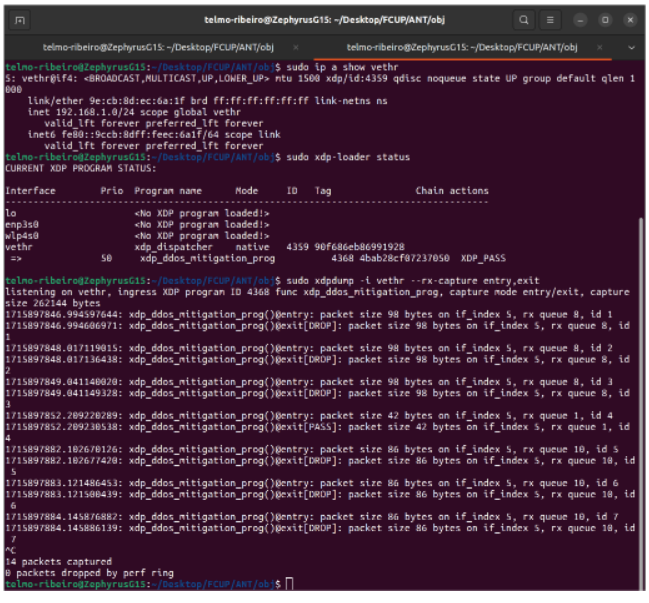

Fig. 7. tcpdump output from target machine

Figure 7 display the xdp-loader status and xdpdump output from the target interface.

In contrast to **tcpdump**, **xdpdump** from xdp-tools can capture all the packets incoming and outgoing from an interface, that includes packets dropped within it.

This solution is a step from *XDP* towards meaningful debug.

VI. Benchmarks

This section probes *XDP* actual speeds.

The Environment section listed the requirements to successfully replicate the scenarios contained in this iteration. Details such as specific hardware components were purposely left unmentioned since, ideally, they would not impact the general ability to develop on the *XDP* framework. This segment expands on benchmarking however, which is heavily impacted by the machine being tested in. Different bottlenecks will lead to different results and resources such as the number of cores, threads, capacity on the PCIe bus, type of NIC used, among others [2] will play a crucial role regarding the results obtained. As such, the laptop and some major components used to perform the following experimentation, are now disclosed in the table I.

TABLE I
Hardware Specification

Hardware	Specifics
Laptop	2021 ROG Zephyrus G15 GA503
CPU	AMD Ryzen™ 9 5900HS (8-core/16-thread/20MB cache)
NIC	MT7921 802.11ax PCIe
RAM	16GB DDR4 on board

A. *Methodology*

During the benchmarking tests, the two major resources were: **(1) xdp-bench** and **(2) xdp-trafficgen**, both tools provided by the xdp-tools package. They allow for an autonomous benchmarking regarding the different *XDP* modes and primitives and an equally autonomous packet generation tool to stress test any given interface.

```
$ sudo xdp-trafficgen udp [options] <interface>
```

This tool uses the *XDP* kernel subsystem to generate packets and transmit them trough a given interface. The packets are dynamically generated and transmitted in the kernel, achieving a very high performance in the common order of millions of packets per second per core.

common flags used:

- **-t** - this flag enables the specification of how many threads are going to be used for packet generation.

```
$ sudo xdp-bench <command> [options] <interface>
```

The command above allows the use of the **xdp-bench** utility. This tool enables the benchmarking of each *XDP* primitive on the three different modes (when supported by the hardware).

common flags used:

- **-m** - this flag allows the selection of the *XDP* mode used on loading;

- **-e** - this flag starts **xdp-bench** in ”extended” output mode, providing more information about the loaded object and statistics.

TX only flags:

- **-p** - this flag alongside with the argument **swap-macs** is used to check the correct functionality of **TX**, swapping the source and destination mac addresses in order to observe incoming traffic on the interface that firstly sent it.

```
$ sudo xdp-bench redirect [options] <interface-
in> <interface-out>
```

Another special case is the **REDIRECT** command, which naturally requires the specification of ingress and egress interfaces.

B. *DROP Results*

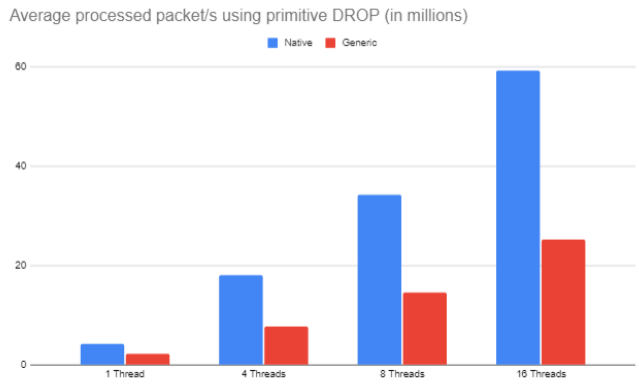

Fig. 8. DROP results

Figure 8 exhibits the behaviour of **DROP** between *SKB/-Generic XDP* and *Native XDP* for 1, 4, 8, and 16 threads respectively.

The x-axis exponential scale in regards to the number of threads, is a compromise that weighted in favor of an approach to the standard CPU architectures. The number of packets processed are growing almost linearly with the number of threads utilized and the only reason it appears to be growing exponentially in the graph is because the x-axis is in exponential scale itself.

Our findings place **DROP** as the primitive achieving the most performance which matches the expected scenario, since the *eBPF* object call can rapidly determine that the packet should not be processed any further. *Native XDP* performing considerably better that *SKB/Generic XDP* is an expected recurring trend since, as mentioned before, *Native XDP* processing happens through the network drivers in an earlier instance of the linux network stack in comparison with *SKB/-Generic XDP*, still better than the standard procedure, but being processed by subsystems of the kernel itself.

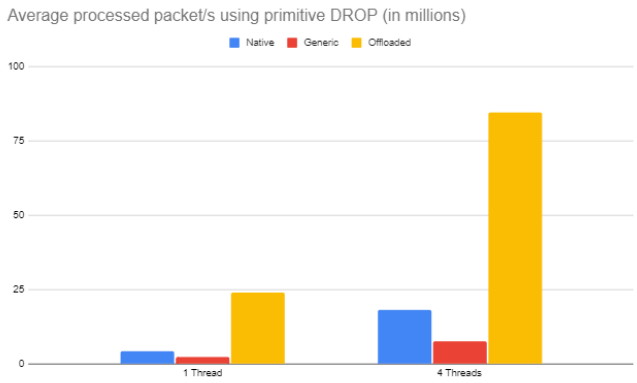

Fig. 9. DROP vs DROP Offloaded Results

C. *DROP Results - bigger picture*

Figure 9 compares the previous results for **DROP** and *Offloaded XDP* for 1 and 4 threads respectively.

It is important to stress this is an **unfair comparison** since the results from *Offloaded XDP* were obtained in a different machine during previous studies [2] (lack of supporting hardware). Instead of focusing in the percentage increase, here the focus is the order of magnitude a **Mellanox ConnextX-5** can accomplish when all the other bottlenecks are actively being mitigated.

D. *PASS Results*

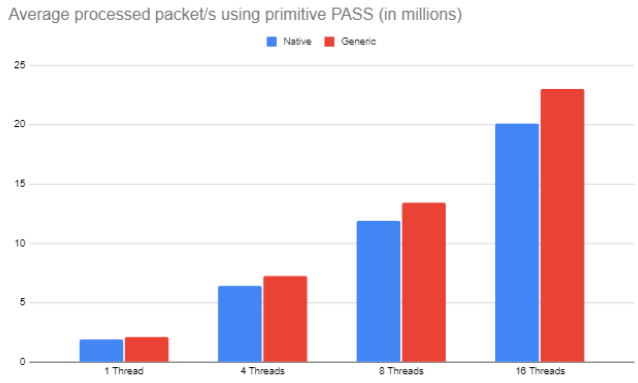

Fig. 10. Pass Results

Figure 10 exhibits the behaviour of **PASS** between *SKB/-Generic XDP* and *Native XDP* for 1, 4, 8, and 16 threads respectively.

Notice that **PASS** performs considerably worse than **DROP**. After the *eBPF* object determines that a packet should be passed, it will be processed by the standard linux network stack so, this initial ”touch”, introduces an overhead to the performance already provided by the standard procedure, while **DROP** is able to continuously recycle frames.

PASS breaks the trend in which *Native XDP* performs better than *SKB/Generic XDP*. We advocate this could be explained

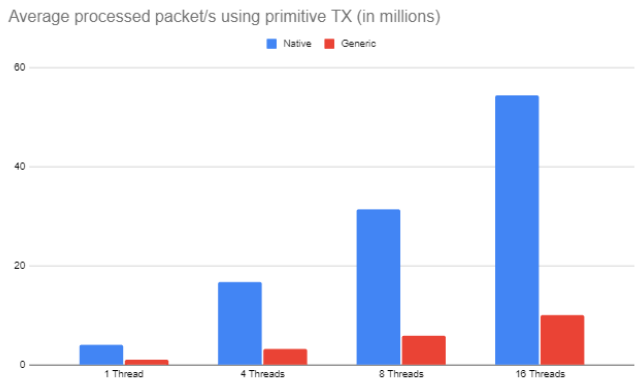

Fig. 11. TX Results

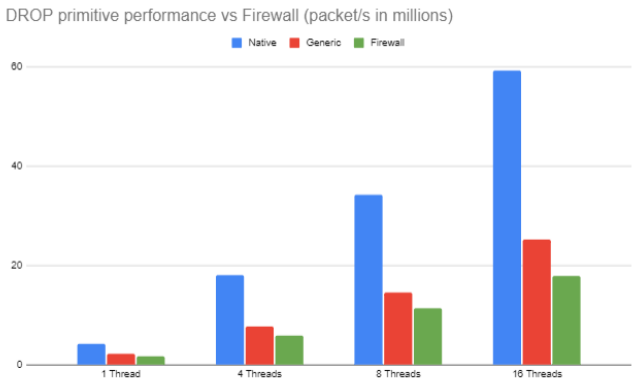

Fig. 13. iptables x DROP Results

by the network driver trying to execute the packet processing just to unavoidably pass it to the network stack.

E. TX Results

Figure 11 exhibits the behaviour of **TX** between *SKB/-Generic XDP* and *Native XDP* for 1, 4, 8, and 16 threads respectively.

TX displays the network driver, swapping the packet fields and returning it to the original interface, outperforming the kernel subsystems executing that same task.

F. REDIRECT Results

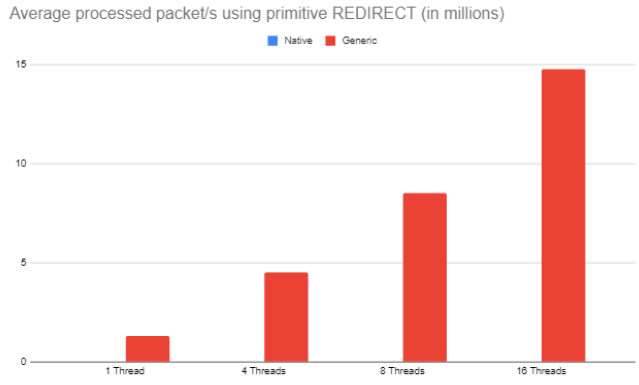

Fig. 12. Redirect Results

Figure 11 exhibits the behaviour of **REDIRECT** from *SKB/Generic XDP* for 1, 4, 8, and 16 threads respectively.

Contrary to all the other primitives, **xdp-bench** does not yet support *Native XDP* mode for this primitive.

G. iptables vs DROP Results

Figure 13 exhibits the behaviour of **DROP** from *SKB/-Generic XDP* and *Native XDP* and iptables for 1, 4, 8, and 16 threads respectively.

Here we compare the original **DROP** results against a **DROP** rule created through iptables, allowing for a possible baseline to be draw between *XDP* and standard firewall utilities.

H. Additional Notes

The recalling of figure 1 and the the underlying differences of *SKB/Generic XDP*, *Native XDP*, and *Offloaded XDP* explain the general trends happening through the results.

The two major tests where deviations could appear are **TX** and **REDIRECT**. Those primitives are not usually the target of benchmarks (specially in *SKB/Generic XDP* and *Native XDP*) and therefore, it was hard to draw a baseline for them.

Although our experiments with the benchmarking tools and methodology do not indicate a possible flaw, the lack of third-party results to compare with can allow for deviations in our analysis.

VII. Final Regards and Future Work

This segment is reserved for final regards about the overall *XDP* experience and provide some pointers how this work could have been incremented.

A. The Bad

```
$ sudo ethtool -K <interface> gro on
```

The previous command enables Generic Receive Offload (GRO) in a given interface. GRO is a must when benchmarking *Native XDP*. In veths it is not enabled by default and may lead to **xdp-bench** not producing meaningful output or debugging pointers and, as such, the reader is advised to check it manually.

```
$ sudo ethtool -L <interface> rx <value> tx <
value>
```

This command changes the amount of channels dedicated to rx and tx queues. In veths the default values are 1 and as such it must be changed when using **xdp-bench** to benchmark multiple threads.

B. *The Ugly*

The lack of community support lead to many issues being solved directly with the developers. While benchmarking, problems made themselves evident, problems these that were not shared through documentation or forums.

Many times, the **xdp-loader**, an essential tool to load/check programs in interfaces, will fail to load *eBPF* objects, yet the standard ip command used in the *feasibility stage* will suffice.

In other instances, functions such as **inet_addr()** from the **arpa/inet.h** library will fail to be loaded within *XDP* programs. This function is widely used in many examples from the RedHat team, supporting the translation between a string and the address it represents (bytes) but in our own programs we had to come up with other solutions.

xdp-tools itself and the libraries it depends on, when downloaded from the RedHat team GitHub, may not be working properly even in stable versions and, as such, downloading from ubuntu-launchpad is more adequate[6][5].

C. *The Good*

The development team is active which allows issues to be addressed in a relatively short time (1-3 days).

In the *XDP* project GitHub page, a much needed tutorial and dependencies setup guide [4] can be found which makes the initial setup less troublesome.

Bypassing Linux network stack layers and being able to manipulate the kernel's functionality without recompiling it, makes *XDP* one of the best choices when looking for flexibility and high performance packet processing. *XDP* is being widely adopted, although slowly, enabling a better ecosystem to develop on since there exists more software allowing *Native XDP* (same cannot yet be said about the relation between *Offloaded XDP* and NICs).

Allowing the developer to create their own programs and quickly have them ran in specified interfaces makes the *XDP* custom packet processing/filtering very enticing, opening new routes to dynamically create firewall and load-balancing solutions.

The viability of this technology is intricately related with the capacity of the development team to mitigate/solve the problems mentioned in the support for the necessary tools and with the environment's ease of setup.

D. *Future Work*

With the time spent debugging this yet somewhat obscure solution, some steps could not be properly refined and other would be interesting to see in a new iteration of this work.

Because of the lack of hardware availability, *Offloaded XDP* could not be properly tested which undermines proper regards about the difference between the different modes, and there is a tendency by the RedHat team to focus on *Offloaded XDP* [2] when benchmarking which makes any result comparison with the community sub optimal.

Since **xdp-bench** does not support **REDIRECT** in *Native XDP* other stress test tools would need to be used to properly derive benchmarking results.

This papers takes an introductory stance to *XDP* program development, yet, future works regarding practical solutions to load-balancing and other *XDP* capabilities would be welcomed.

REFERENCES

[1] G. Bertin, "Xdp in practice: integrating xdp into our ddos mitigation pipeline," in *Technical Conference on Linux Networking, Netdev*, pp. 1–5, 2017.

[2] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pp. 54–66, 2018.

[3] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Computing Surveys (CSUR)*, pp. 1–36, 2020.

[4] X. Team, "xdp setup dependencies." https://github.com/xdp-project/xdp-tutorial/blob/master/setup_dependencies.org, 2024. [Accessed 16-05-2024].

[5] W. Grant, "xdp-tools package in ubuntu — launchpad.net." https://launchpad.net/ubuntu/+source/xdp-tools/1.4.2-1ubuntu4, 2024. [Accessed 16-05-2024].

[6] S. Langasek, "libbpf package in ubuntu — launchpad.net." https://launchpad.net/ubuntu/+source/libbpf/1.3.0-2build2, 2024. [Accessed 16-05-2024].

[7] H. Liu, "Get started with xdp," 2022.

[8] V. K. Yadav, M. C. Trivedi, and B. Mehtre, "Dda: an approach to handle ddos (ping flood) attack," in *Proceedings of International Conference on ICT for Sustainable Development: ICT4SD 2015 Volume 1*, pp. 11–23, 2016.