eXpress Data Path (XDP) - Feasibility Findings

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*—The project aims to demonstrate the use of *XDP* in Linux to accelerate packet processing and filtering while analyzing its performance. The project is divided in two parts, the *feasibility stage* and the *implementation stage*. The current iteration pertains to the first stage where the objectives, environment and the initial findings are stated. *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, cybersecurity, benchmarks**

I. Introduction

The current stage of the modern world we live in, could only be attained due to the progress in data treatment and transmission, both evolving through a multitude of efforts regarding networks, protocols, and tools developed by engineers and scientists alike, pushing the underlying technologies forward. The viability of such technologies depends on concrete implementations to tackle firewalling[1], for instance, where distributed denial-of-services (DDoS) attacks are relatively straightforward to preform yet their countering is still subject of much concern, and the continuous pursuit of improvements in areas as load balancing and network analysis.

A partial solution regarding the aforementioned problems was provided through the recent developments on the *eXpress Data Path (XDP)* framework, described as the lowest layer of the Linux network stack, enabling developers to install programs that process packets into the Linux kernel [2].

Extended Berkeley Packet Filter (eBPF) programs modify the kernel operation in runtime, not requiring kernel recompilation [2]. Technologies developed over it, essentially, can bypass 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 can be translated into custom packet processing, implemented through its recurring calls for every incoming network packet. *XDP* can be implemented through three different modes: **(1)** Generic *XDP* - where programs are loaded as part of the ordinary network path, allowing the least gain in performance from all modes but not requiring explicit support from specific hardware, **(2)** Native *XDP* - where the program is loaded by the Network Interface Card (NIC) thus requiring support from the driver but resulting in measurable performance gain, and lastly **(3)** Offloaded *XDP* - where the program is loaded on the NIC, requiring support from the interface itself, but consequently bypassing the CPU altogether, therefore, providing greater performance. *XDP* programs are attained by incorporating primitive actions which in turn, will express desirable properties, those 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 to the NIC, and *XDP_REDIRECT* - redirecting the packet to another NIC or socket.

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. 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 lays the ground for benchmarks over primitive *XDP* capabilities. Initially, offering an overview on the *XDP* framework while providing some background on the concept. Then it follows with the tools and environment used to achieve the findings and the accompanying methodology and results from the feasibility test. Latter, the future work explicits the roadmap and the build up over this results including a few early reflections.

II. Objectives

This section is reserved to enumerate and describe the project objectives. Those objectives are related to the *feasibility stage*, this paper's current iteration, and the *implementation stage*, regarding the project's final milestones.

Feasibility Objectives:

- Environment - achieve an environment, through tools and components described in following sections, where the development of software, its execution, and its benchmarking can be preformed.
- Feasibility - pertains to the current iteration's main goal, via a feasibility proof validating both the capability of the ecosystem to provide results and laying the ground for the *implementation stage*.
- Reflections - discussion on the first findings and major milestones.

Implementation Objectives:

- Use Cases - development and analysis of *XDP* programs, tackling concrete scenarios regarding packet processing and filtering such as, filtering packets by their IPv4/IPv6 header fields, packet transmission on specific ports and dropping packets from exploitable protocols.
- Primitive Actions - development and analysis of *XDP* primitive actions, this constitutes the main goal on the *implementation stage*.
- Benchmarks - measure the #packets/time ratio for each of the primitive actions.
- Results - analysis on how the results obtained relate to the theoretical limits between the different *XDP* modes.

These goals are explored in greater detail on the following sections.

III. Environment

The environment section concerns to the tools, components, and overall resources employed to accomplish the *feasibility objectives* and set up the *implementation stage*.

Operative System:

While *XDP* is supported by the Linux kernel since the 4.8 version, and currently supported by Windows through agreements with NIC manufacturers, the *Feasibility Test* described in the following section was achieved in Ubuntu 22.04.4 LTS. Ubuntu not only supports *XDP*, but it also is one of the major Linux distributions to be accommodated by the standard set up process developed by the *Red Hat* team.

***extended Berkeley Packet Filter*:**

eBPF is a key concept in *XDP* development, being the focus of many tools applied, providing both an instruction set and an execution environment in the Linux kernel. Used to modify the packet processing framework, code is written in *restricted C* and then compiled into *eBPF* instructions, executed by the CPU on the Generic *XDP*, but it can also be ran by programmable devices such as SmartNICs [2].

Libraries:

The *Feasibility Test* was implemented with recourse to a long set of libraries. However, the focus set needed to successfully build, execute, and analyze *XDP* programs is the following:

- *bpftools* - a collection of command-line tools used to interact with the Linux kernel, they provide multiple functionalities to process *eBPF* objects and was mainly used to test its execution.
- *xdp-tools* - a set of utilities used with *XDP* programs to better interact with the Linux system. It was mainly used to load/unload simultaneous *eBPF* object to/from the interface.
- *libbpf* - as a focal point of the *eBPF* environment it allows, for instance, the use of macros such as *SEC*, explained in greater detail upon the next section.

Build/Dump:

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

Network Debugging Tools:

Network debugging tools and overall Unix commands were essential to, for instance, load/unload and test the *eBPF* object's execution. Among others, *ping -4* and *ping -6* were used to send IPv4 and IPv6 packets, a behaviour crucial to examine the proposed use case.

Network Interface Cards:

Native *XDP* and Offloaded *XDP* benefit from greater performance, however, they require explicit NIC support. Intel, Mellanox, and Broadcom have NIC families extending *XDP* capabilities.

After surveying which NICs support Native *XDP*, the lack of availability made itself evident. Although still up for consideration, in order to circumvent this major drawback and establish comparisons, the use of previous results regarding the throughput of such modes is advocated.

IV. Feasibility

This sections is concerned with the objectives, methodology, and results from the *Feasibility Test (FT)*.

The *FT* covers two objectives: **(1)** validate the environment described in the previous section, allowing the *implementation stage* to be carried out over it, and **(2)** render the possibility to develop a *XDP* program and test its execution.

The Introduction regarded the implementation of *XDP* through three different modes: **(1)** Generic *XDP*, **(2)** Native *XDP*, and **(3)** Offloaded *XDP*. The *FT* is focused on Generic *XDP*, since it allows the accomplishment of the aforementioned objectives without the need for supporting hardware. The developed program models the drop of all packets except for IPv6.

Methodology (step by step):

- Develop the program;
- Build the object;
- Load the object;
- Experiment the environment

***XDP* program:**

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

SEC("xdp_pass_ipv6")
int xdp_pass_ipv6_prog(struct xdp_md* ctx) {
    void* data_end = (void*)(long)ctx->data_end;
    void* data     = (void*)(long)ctx->data;
    if (data + sizeof(struct ethhdr) > data_end) {
        return XDP_DROP;
    }
    struct ethhdr* eth = data;
    if (eth->h_proto == htons(ETH_P_IPV6)) {
        return XDP_PASS;
    }
    return XDP_DROP;
}

char _license[] SEC("license") = "GPL";
```

Listing 1. *C* program to drop all packets but IPv6.

SEC enables the placement of compiled object fragments into different *Executable and Linkable Format (ELF)* sections. The function *xdp_pass_ipv6* accepts a parameter of type *struct xdp_md**. This struct allows the initialization of pointers referencing crucial header delimiters, being used to assert the access of legal fields within the header. If the protocol used is IPv6, the packet is passed, through the return of *XDP_PASS*. Otherwise, the packets are dropped while returning *XDP_DROP*. Finally, the last line regards to the license associated with the program, that being *General Public License (GPL)*[3].

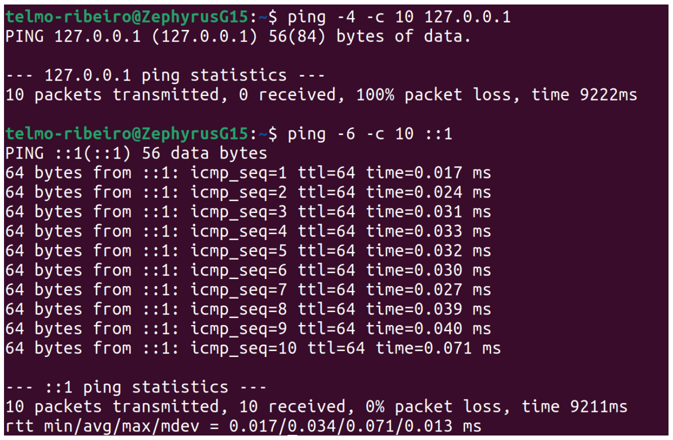
Loading:

```
$ sudo ip link set interface xdpgeneric obj
    xdp_pass_ipv6.o sec xdp_pass_ipv6
```

The *ip* command is expressive enough to load the object in question, however, *xdp_loader* from *xdp-tools*, offers greater capabilities and will be use in advanced scenarios.

Experimentation:

Commands such as *xdp-loader status*, *bpftool prog show*, *ip link show*, *ping*, among other network debugging tools where used to experiment on *XDP*.



```
telmo-ribeiro@ZephyrusG15:~$ ping -4 -c 10 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.

--- 127.0.0.1 ping statistics ---
10 packets transmitted, 0 received, 100% packet loss, time 9222ms

telmo-ribeiro@ZephyrusG15:~$ ping -6 -c 10 ::1
PING ::1(::1) 56 data bytes
64 bytes from ::1: icmp_seq=1 ttl=64 time=0.017 ms
64 bytes from ::1: icmp_seq=2 ttl=64 time=0.024 ms
64 bytes from ::1: icmp_seq=3 ttl=64 time=0.031 ms
64 bytes from ::1: icmp_seq=4 ttl=64 time=0.033 ms
64 bytes from ::1: icmp_seq=5 ttl=64 time=0.032 ms
64 bytes from ::1: icmp_seq=6 ttl=64 time=0.030 ms
64 bytes from ::1: icmp_seq=7 ttl=64 time=0.027 ms
64 bytes from ::1: icmp_seq=8 ttl=64 time=0.039 ms
64 bytes from ::1: icmp_seq=9 ttl=64 time=0.040 ms
64 bytes from ::1: icmp_seq=10 ttl=64 time=0.071 ms

--- ::1 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9211ms
rtt min/avg/max/mdev = 0.017/0.034/0.071/0.013 ms
```

Fig. 1. ping output for the loopback interface

Figure 1 displays ping command's output over ICMPv4 and ICMPv6 regarding the loopback interface loaded with the aforementioned object. Although this output alone does not constitute a proof of correctness on *xdp_pass_ipv6*, and thus came the need for multiple tests and debugging tools, it was a reliable indication on the program well behaviour.

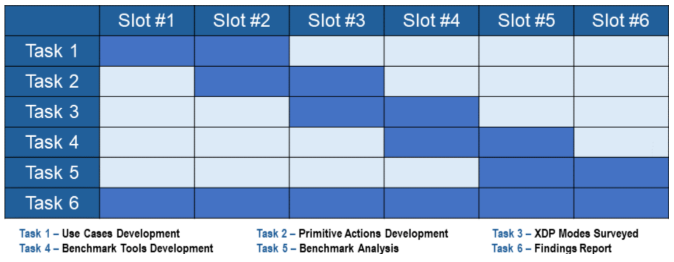
The tests performed on the interface, corroborated the *XDP* program correctness, therefore, deriving the objectives achievement for the *FT*.

V. Future Work

This final section is reserved for explaining the steps on the *implementation stage* and provide some final regards.

The *Feasibility Test* was fortuitous in exhibiting the environment and tools chosen, allow for the intended demonstration. Nevertheless, the *feasibility stage* only tackled two primitive actions. The other three: **(1)** *XDP_ABORTED*, **(2)** *XDP_TX*, and **(3)** *XDP_REDIRECT*, in addition to extra use cases worth exploring, will be the focus on the *implementation stage*.

Although benchmarking was not part of the current paper's proposal, it will be a central point in the next report. It will only regard the primitive cases and will be preformed through a mix of *XDP* specific structures[4] and software developed exclusively to benchmark them, making possible to count the packets processed and express it through time ratios.



	Slot #1	Slot #2	Slot #3	Slot #4	Slot #5	Slot #6
Task 1	🟦	🟦				
Task 2		🟦	🟦			
Task 3			🟦	🟦		
Task 4				🟦	🟦	
Task 5					🟦	🟦
Task 6	🟦	🟦	🟦	🟦	🟦	🟦

Task 1 – Use Cases Development
Task 2 – Primitive Actions Development
Task 3 – XDP Modes Surveyed
Task 4 – Benchmark Tools Development
Task 5 – Benchmark Analysis
Task 6 – Findings Report

Fig. 2. Gantt chart disclosing tasks per fixed time slots

The *Feasibility Test* yielded positive results, creating a safe foundation to expand upon the next stage. During this paper's development, challenges regarding the *XDP* modes were also clarified so new measures could be erected to mitigate them.

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] 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.

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

[4] 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.