Jacobs University Bremen

Campus Ring 1
28759 Bremen, Germany

An Evaluation of the eXpress Data Path

by

Milen Vitanov

Bachelor Thesis in Computer Science

Prof. Dr. Jürgen Schönwälder
Bachelor Thesis Supervisor

Date of Submission: May 17, 2019

Jacobs University — Computer Science and Electrical Engineering

With my signature, I certify that this thesis has been written by me using only the indicates resources and materials. Where I have presented data and results, the data and results are complete, genuine, and have been obtained by me unless otherwise acknowledged; where my results derive from computer programs, these computer programs have been written by me unless otherwise acknowledged. I further confirm that this thesis has not been submitted, either in part or as a whole, for any other academic degree at this or another institution.

Signature

Bremen, 17.05.2019
Place, Date

Abstract

As the Internet continues to expand and new bigger and more power demanding technologies are developed, the need for a fast, reliable and easily programmable packet processing tool is needed. The conventional way of using in-kernel networking stack is no longer an efficient option, because of the many expensive context switches between kernel and userspace it requires. This is the reason why the network community has turned towards creating packet processing tools using kernel bypass techniques in which the application takes complete control of the networking process by taking raw packets directly from the driver. While these approaches are fast, they lack the kernel well-tested security mechanisms and its application isolation. However, a new packet processing tool has been developed that is leveraging both the in-kernel and the bypassing techniques. It is called eXpress Data Path (XDP) and is developed by the Linux kernel community. XDP is a hook on the driver layer that provides a safe execution environment for a custom packet processing applications. These applications are written in C and are inserted into the kernel thanks to the extended Berkeley Packet Filter (eBPF). XDP can be programmed in many ways. It can drop the packets immediately, forward them to an interface or transmit them to the in-kernel networking stack. Recently, an address family (AF_XDP) was also developed that allows the creation of a raw socket for transmitting packets directly from the XDP program to userspace. In this way XDP can bypass the kernel networking stack, while still being able to use in-kernel functionality through the execution of eBPF helper functions. In this paper I will provide detailed explanation of XDP, its data paths and the newly-developed AF_XDP. Then I will compare different filtering programs by measuring their performance.

Index terms— XDP, AF_XDP, eBPF, Linux, Networking

Contents

1 Introduction **1**

2 Extended Berkeley Packet Filter (eBPF) and its Components **2**
2.1 From BPF to eBPF 2
2.2 eBPF Maps 2
2.3 Function Calls and Service Chains 3
2.4 eBPF Verifier and its System Call 3
2.5 eBPF Compilation 5

3 eXpress Data Path (XDP) **6**
3.1 eBPF in Networking 6
3.2 Integraded XDP in the Linux Kernel 6
3.3 The XDP Program 8
3.4 An Example XDP Program 9

4 AF_XDP **12**
4.1 Rings 13
4.2 UMEM 13
4.3 XSK 14

5 Performance Evaluation **16**
5.1 Basic XDP actions 16
5.2 XDP filtering Program 18
5.3 XDP socket with a Userspace Stack 18

6 Conclusion **21**

1 Introduction

It is more and more evident that the modern computer software requires better packet processing in terms of performance. The traditional way of doing this is by passing the packets through an in-kernel network stack. This approach fails at delivering good enough performance due to the expensive context switches it has between kernel and userspace. Thus, modern approaches were developed, which create data paths from the network driver directly to a userspace application. An example of such approach is the Data Plane Development Kit (DPDK) [10]. It is a userspace application that takes packets directly from the network driver and processes them in userspace with the help of special libraries. In this way performance is significantly improved. The downside of such approaches is that they omit the kernel security mechanisms. Moreover, their integration with the rest of the system is much more difficult. The kernel has a lot of functionality such as routing tables and higher level protocols that is isolated from the application layer. Therefore, when it is bypassed, a proper handling of raw packets by every application receiving them must be ensured.

However, a new approach that takes advantage of both the in-kernel and the bypassing packet processing has been under development since 2016. It is called the eXpress Data Path (XDP). The release of the extended Berkeley Packet Filter (eBPF) in 2013 created the possibility of executing user-defined programs in the kernel. This put the beginning of many new technologies, one of which is XDP. It works by defining a limited execution environment, inside of which an eBPF code can be run. The environment is created at the earliest possible point after a packet is received from the network device, i.e. before the allocation of the socket buffer (SKB) or any other metadata.

XDP is implemented as a hook in the official Linux kernel. There are two XDP hook modes. The first one is the driver mode, which is available only if the network driver is supported by XDP. It is happening before the SKB allocation. The second one is the generic mode, which is driver independent but slower in terms of performance.

The XDP framework provides the functionality to immediately drop packets, forward them to an interface, pass them through the in-kernel network stack or to redirect them directly to userspace through special raw sockets. A whole new address family for XDP raw sockets was created and implemented in the Linux kernel in August 2018. It is called AF_XDP. This address family made it possible for raw packets to be directly transmitted to userspace. It further supported the core idea behind the XDP initiative to get the network stack out of the way as much as possible without bypassing the kernel completely. Kernel functionality is preserved within the XDP programs, which have access to most of the kernel instructions. The result combines minimal overhead with a great deal of flexibility, at the cost of a setting up [7]. Another great advantage of XDP is that a userspace application can indirectly control the in-kernel filtering program through the usage of shared eBPF maps.

This paper is structured as follows. Section 2 explains eBPF and its components. Section 3 and 4 outline a detailed overview of XDP, its design and the newly developed XDP address family. Section 5 provides three experiments, which are proving the efficiency of XDP and its address family. Section 6 concludes the paper.

2 Extended Berkeley Packet Filter (eBPF) and its Components

Extended Berkeley Packet Filter is a generic in-kernel, event-based virtual CPU implemented in the Linux kernel from version 3.15 (June 2014) onward. It is different than virtual machine (VM) due to the fact that it is working only within the kernel space. Despite that, it is often called an in-kernel VM by many sources. There are two main aspects in terms of the usage of eBPF - kernel tracing / event monitoring using kprobes and network programming. It can also be used as a socket filter and Linux traffic control tool. Another interesting feature is the eXpress Data Path (XDP), which achieves very fast in-kernel packet processing.

The most interesting feature eBPF has is the ability to run user-supplied programs inside the kernel. This is an important innovation because it allows the userspace processes to send a filter program specifying the exact packets that are needed. In this way the copying of any unnecessary packets from kernel to user space processes is avoided, which results into significantly improved performance [9].

Another interesting feature is that it reacts to generic kernel events. This is possible because each eBPF code is attached to one kernel event. There may be multiple hook points for networking, which allows the implementation of network functions at different layers of the stack [16].

2.1 From BPF to eBPF

eBPF is an extended version of the original BSD packet filter (BPF) [13], which has two 32-bit registers and understands 22 different instruction. eBPF extends the number of registers to eleven and increases their sizes to 64 bits. Its instruction set is also increased by adding new arithmetic and logic instructions for the bigger register size. The 64-bit registers map one-to-one to the hardware registers on the 64-bit architectures, which enables just-in-time (JIT) compilation into native machine code [12].

Other important features of eBPF are its ability to call functions, its special data structures called eBPF maps, its tail calls and the verifier, which makes sure potentially dangerous code does not enter the kernel .

2.2 eBPF Maps

eBPF programs use pre-formatted shared memory. The data stored in this memory is accessed by structures called maps. They serve as generic storage structures for different data types shared between kernel and userspace or between different eBPF programs. Data types are generally treated as binary blobs. At map-creation time only the size of the key and the size of the value is specified. In other words, a key/value for a given map can have an arbitrary structure. No access to a specific offset within a map is allowed, thus the concurrency issues when multiple entities are trying to access the same data are avoided. The maps can be private per CPU core or global. If they are private, no data synchronization between CPU caches is needed [12].

Maps are referenced by global id. A userspace process can create multiple maps (with key/value-pairs being opaque bytes of data) and access them via file descriptors. Different eBPF programs can access the same maps in parallel. It is up to the user process and eBPF program to decide what is necessary to be stored inside maps.

Using these maps there is no need to store and parse the data for post processing. We can simply return our own map structure directly from the eBPF program to the user application in the form we prefer (hash,array, etc.). Then we can eventually collect statistics directly from the eBPF program [14].

There are different types of maps. The eBPF data structures can be generic hash maps, arrays, radix trees, special types containing pointers to other eBPF programs or pointers to other maps [12]. The maps are created based on a request from userspace via the `bpf()` system call. The required arguments are the type of the map, the size of the key, the size of the value, the maximum number of entries and whether or not to pre-allocate the map in memory. The call returns a file descriptor that refers to the newly created map. The map is deleted once all file descriptors are closed and no loaded eBPF program is referencing it.

2.3 Function Calls and Service Chains

The eBPF instruction set is increased with one more instruction called *call*. It enables the eBPF programs to make fixed amount of function calls and use their return values. Moreover, such function calls have very small additional overhead due to the one-to-one mapping to the hardware registers and the fact that eBPF uses the same calling convention as the C language [12].

Many eBPF programs can be run at the same time, even attached to different hooks. They can operate in isolation, which means that after the execution finishes they are returning the packet to the hook they are attached to. Additionally, eBPF programs can also be chained together to create complex service chains. This is achieved by using a low-overhead linking primitive called *tail call*. The tail calls are simply big jumps from one program to another. However, they are different than normal functions due to the fact that they do not allow returning to the previous context. eBPF programs can also be chained to any physical/virtual Network Interface Card (NIC), or any tunnel adapter [16].

2.4 eBPF Verifier and its System Call

As described before an important feature of eBPF is the ability to run user-supplied programs inside the kernel. As a consequence, these programs bypass a lot of security checks made automatically by the system. Therefore, an error in an eBPF program can result in a kernel panic, which can actually halt the system. Moreover, an infinite loop in the code can freeze the entire system because the scheduler located in userspace cannot stop the program. This is the reason why it is critical that the safety of eBPF programs is ensured. To achieve this the kernel enforces a single entry point for loading all eBPF programs. This is done with the `bpf()` system call. It is a multiplexor for a range of different operations on eBPF. Below the `bpf()` system call is introduced.

```
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

The `bpf_attr` union is a C union which allows for different C structs to be passed to the `bpf()` system call, i.e it allows data to be passed between the kernel and user space. The exact formats of the structs depend on the command being used (`cmd` argument). The size argument gives the size of the `bpf_attr` union object in bytes. [8]

There are ten commands for the `bpf()` Linux system call which are listed below.

```
enum bpf_cmd {
    BPF_PROG_LOAD,
    BPF_OBJ_PIN,
    BPF_OBJ_GET,
    BPF_PROG_ATTACH,
    BPF_PROG_DETACH,
    BPF_MAP_CREATE,
    BPF_MAP_LOOKUP_ELEM,
    BPF_MAP_UPDATE_ELEM,
    BPF_MAP_DELETE_ELEM,
    BPF_MAP_GET_NEXT_KEY,
};
```

These commands determine what is possible for a user to do with the eBPF system.

When using `BPF_PROG_LOAD`, the program a userspace process is trying to load is first analyzed by the in-kernel eBPF Verifier through certain checks. This Verifier statically determines if the program will terminate and be safe to execute. Some of its checks are presented below [16].

- **Limited program size of 4096 assembly instructions :** It is important that the loaded eBPF programs are small to guarantee that they will terminate within a bounded amount of time and will not become a bottleneck for the system. This also means that eBPF does not support completely arbitrary programs, i.e. it is not Turing complete.
- **Adding checks for loops and cyclic flows :** The verifier builds a Directed Acyclic Graph (DAG) of the control flow of the program with the root at the first instruction. All branches end at the last return instructions and all instructions are reachable [20]. In order to guarantee termination, simple non-recursive depth-first-search verifies that there are no back-edges, i.e. there are no loops in the program.
- **Preventing unreachable/ illegal instructions :** This check ensures that an eBPF program has just what is strictly necessary to run, which makes it small and fast to load and execute.
- **Preventing invalid memory access :** The verifier disallows jumps that happen outside the program boundaries into any arbitrary memory location. This is important because eBPF programs should not be able to harm the system, for instance, by moving the control flow or writing memory belonging to other processes. The verifier is able to do that by tracking data types, pointer offsets and possible value ranges of all registers. It stores the range information in state variables and uses it to predict the possible amount of memory that each load instruction can potentially need. For packet data access this is done by looking for comparisons with the special `data_end` pointer that is available in the context object; for values retrieved from a BPF map the data size in the map definition is used; and for values stored on the stack, accesses are checked against the data ranges that have previously been written to [12]. In this way the eBPF verifier ensures that only save memory accesses are performed.
- **Path and flow evaluation :** The verifier walks through the program, starting from the first instruction and descending all possible paths. It also keeps tracks of register and stack statuses. In case of a registry overflow/underflow or stack overflow, the

verifier does not execute the code ensuring again that the eBPF program does not harm the system.

- **Argument validity :** This guarantees that a program will not access a register that does not exist, write in a read only register, write a chunk of data into a smaller register or call unauthorized kernel functions.

If the criteria are not met, the verifier rejects the eBPF program. Thus, it guarantees that any eBPF program is sandboxed. It can neither slow down, nor harm the system [20].

2.5 eBPF Compilation

We can write eBPF programs using assembly and generate eBPF bytecode through the kernel's `bpf_asm` assembler, but doing that is difficult and often results in many mistakes. However, Clang/LLVM compiler allows using restricted-C instead of assembly by providing a backend, which compiles an eBPF C program into bytecode. The object file containing this bytecode can then be directly loaded into the kernel. Since eBPF is designed to be compiled with JIT using one to one mapping, it is possible for Clang/LLVM compiler to generate optimized code through an eBPF backend that performs almost as fast as natively compiled code.

The command for compiling the eBPF program using the Clang/LLVM compiler is:

```
clang -march=bpf  -o mybpf_out mybpf.c
```

The eBPF bytecode (object file) produced by Clang compiler needs to be loaded by a program that runs natively on the machine. The kernel provides the `libbpf` library, which includes helper functions for loading programs, creating and manipulating eBPF objects. Once loaded, programs can be attached to a hook in the kernel. When the program is attached, it becomes active and starts analyzing, capturing or filtering information depending on its purpose. Userspace programs can manage the running eBPF code by using eBPF maps and manipulating them to alter the behavior of the program. [9].

3 eXpress Data Path (XDP)

Linux 4.8 kernel was released on 2 October 2016. What is interesting about this specific kernel is that it introduces a new framework for eBPF that enables high-performance programmable packet processing in the networking data path. This technology is called eXpress Data Path.

3.1 eBPF in Networking

Since eBPF is event-driven, program execution can be triggered by the arrival of a packet. Two kernel hooks are available to intercept, modify, forward or drop packets - Traffic Control (TC) and eXpress Data Path (XDP). Programs using TC hook intercept data when it reaches the kernel traffic control function either in RX or TX mode.This hook provides all functionality of the Linux kernel since it is within the network stack. However, the packets still go through the traditional in-kernel processing and many context switches between kernel and userspace are involved. Therefore, this hook does not provide very good time performance.

On the other hand, programs hooked to XDP intercept received (RX) packets right out of the NIC driver, possibly before the allocation of the socket buffer (skb), which allows features like early packet drop, firewalling or forwarding. This mechanism to run eBPF programs at the lowest level of the Linux networking stack allows early enough filtering to significantly improve the packet processing performance. There are two XDP operating modes at this hook. The first one is called Driver (Native) mode. It is the primary mode of operation and can be used only if the driver of the netdevice supports XDP, because it requires driver modifications in order to work. Running network applications in XDP produces significant performance benefits, because they can perform operations (redirect, drop or modify) directly on the packet before any allocation of kernel meta-data structures. It spends fewer CPU cycles for processing the packet compared to the conventional stack delivery. The other mode is called Generic XDP. It allows using the tool with drivers that do not have native support for it [16]. It is important to note that a program attached to XDP generic is faster than the ones attached to the TC hook, but slower than the XDP native mode [20].

3.2 Integraded XDP in the Linux Kernel

XDP eBPF programs are executed by the eBPF virtual CPU at the earliest point a packet is available for processing. This is the point when it is just received from its RX Ring by the driver (Figure 1). The infrastructure to execute the program is contained in the kernel, which means it is executed without any context switches to userspace. In the Driver mode there are no expensive operations, such as the allocation of the socket buffer for pushing the packet further up the stack. The metadata structure is also not yet added to the packet. This leads to a performance that is approximately four times faster than the same operation done by the network stack. Unfortunately, not all network device drivers have support for XDP. In case an XDP eBPF program is run on unsupported device driver, it falls back to a generic XDP hook, which is implemented by the core kernel. This hook is available regardless of the specific network device driver. The Generic mode is done immediately after the socket buffer allocation, which makes it faster than the eBPF TC hook or the traditional in-kernel packet processing, but significantly slower than the Driver mode, because of the context switches done to allocate the socket buffer. [2].

Network device drivers supporting XDP hook in Linux 5.0.10 and later are *ixgbe, ixgbevf, bnxt, qede, mlx5, mlx5e, i40e, tun/tap, thunderx, virtio_net, mlx4, nfp, sfc, intel e1000, intel e1000e* and *veth* [3].

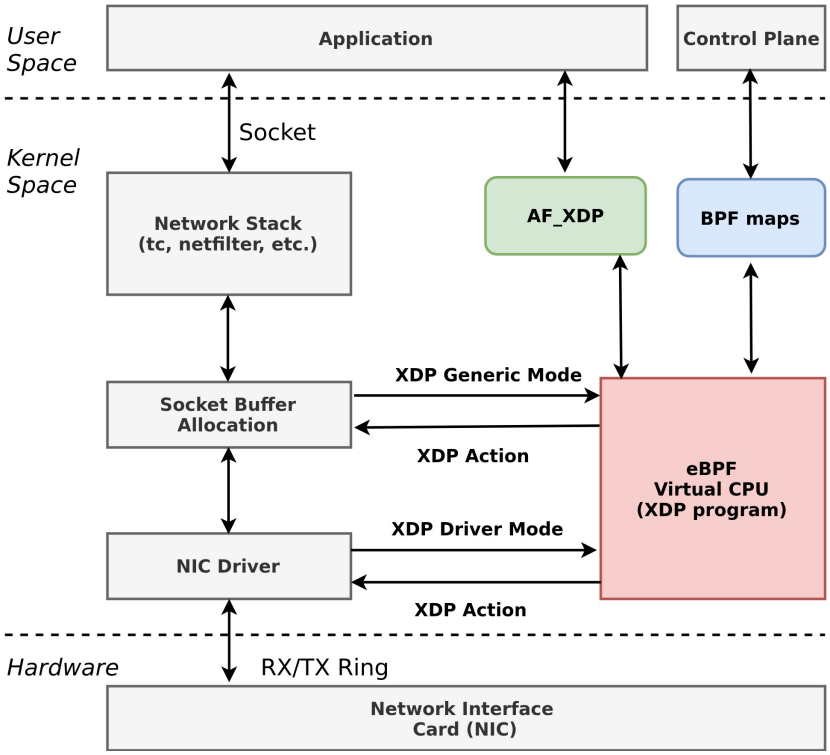

Figure 1: XDP in the Linux Kernel

What makes XDP distinct from other modern approaches to packet processing such as the ones which completely bypass the kernel is that it does not replace or bypass the network stack. It augments it and works in concert with its infrastructure by keeping the packets within kernel space [2]. Some of XDP advantages are listed below.

- No switching between kernel and user space.
- Guaranteed stability of the eBPF instruction set and the API exposed along with it.
- The same security model is used by XDP as the one used by the rest of the kernel.
- XDP is able to reuse all the upstream developed kernel networking drivers, user space tooling and other available in-kernel infrastructure such as routing tables, TCP stack, sockets, etc. This is possible through the usage of eBPF helper functions.
- XDP offers transparency to the applications running on the host. It enables new deployment scenarios from userspace, which can be used for various purposes such as creating inline protection against denial of service (DoS) attacks on servers [12].
- The use of eBPF allows for full programmability. Moreover, XDP programs can be re-programmed in a dynamic way without interrupting the network traffic processing, because of the eBPF JIT compiler.

- It does not require any special hardware features. However, the drivers must be modifies to work with the XDP driver hook (software modification).
- No third party kernel modules or licensing are required. It is a long-term architectural solution, a core part of the Linux kernel, and developed by the kernel community.
- XDP does not require a whole CPU core for packet processing. It takes as much CPU power as it needs to process the current traffic. This has many important efficiency and power saving implications [12].

On top of all this functionality, XDP also offers a path bypassing the Linux kernel. Thanks to a new address family called AF_XDP, which was introduced in August 2018, a special XDP raw socket (XSK) can be initialized and used to directly transmit raw packets from the XDP program to a userspace application. Since the filtering program is still executing in the kernel, it can take advantage of the kernel functionality and security mechanisms. Moreover, XDP can still filter the raw packets as much as the current running XDP program is capable of and transmit only desired packets to userspace. This new functionality makes XDP better than other kernel bypassing technologies, which lack the kernel security and ability to do basic in-kernel filtering.

3.3 The XDP Program

An XDP program can be written both in assembler or restricted-C. Since writing in assembler is much more complicated and advanced, this section will use the restricted-C language. The compilation process is the same as the one described for an eBPF program.

When a packet arrives in the device driver, the XDP hook is activated and the XDP program starts its execution. It consists of a function with a single argument, which is a context object describing the current packet. This object contains pointers to the raw packet data, along with metadata fields describing which interface and RX queue the packet was received on [12]. The program is allowed to parse and manipulate this packet in arbitrary ways as long as it respects the constraints imposed by the eBPF verifier, i.e. it cannot contain loops, it should access only valid memory and it should comply with the maximum number of eBPF instructions allowed. By modifying packets, the XDP program is able to expand or shrink the packet buffer to add or remove headers. Thus, it can rewrite address fields for forwarding

In case there are more packets, the XDP program processes them linearly. The context object also gives access to a special memory area that is adjacent to the packet data. The program can use this memory to attach its own metadata to a packet that will be carried with it as it traverses the system. Finally, the program must return a controlling predefined action to the owning device driver, specifying what the driver has to do with the packet afterwards [1].

XDP program actions:

- Immediately drop a packet with the predefined action **XDP_DROP**.
- Pass a packets to the in-kernel network stack with **XDP_PASS**.
- Send a packet out through the same interface it was received on with **XDP_TX**.

- Redirect a packet to another NIC or CPU using **XDP_REDIRECT**.
- Redirect a packet to userspace with the help of the special **AF_XDP** socket.
- Indicate eBPF program error with the action **XDP_ABORTED**. When this action is returned, it is treated by the driver as a packet drop [11].

Because of the eBPF maps, XDP programs are allowed to define and access their own persistent data structures. In this way they can communicate with the rest of the system. This is particularly useful for the cases when an userspace process wants to exert control over the XDP program. Another interesting feature is that the XDP programs can use helper functions, through which they can access a lot of kernel functionality and security mechanisms.

3.4 An Example XDP Program

The example code below is a simple XDP eBPF program that takes different actions on different packets. In case the received packet is of type IPv6, it is sent back to the interface it was received from. If it is an IPv4 packet, it is passed through the kernel stack. Any other valid ethernet packet is dropped. If an invalid packet is received, an eBPF program error is raised with the XDP_ABORTED return value and the packet is dropped. The code is shown below.

```
1  // Based on the code  "xdp1_kern.c" from Linux Kernel 5.0 release
2  #define KBUILD_MODNAME "xdp_basic_filter"
3  #include <uapi/linux/bpf.h>
4  #include <linux/in.h>
5  #include <linux/if_ether.h>
6  #include <linux/if_packet.h>
7  #include <linux/if_vlan.h>
8  #include <linux/ip.h>
9  #include <linux/tcp.h>
10 #include <linux/ipv6.h>
11 #include "bpf_helpers.h"
12
13 static __always_inline
14 int parse_eth(struct ethhdr *eth, void *data_end, __u16 *eth_type)
15 {
16     __u64 offset;
17
18     offset = sizeof(*eth);
19     if ((void *)eth + offset > data_end)
20         return 0;
21     *eth_type = eth->h_proto;
22     return 1;
23 }
```

```
24 SEC("xdp_prog")
25 int basic_xdp_filter(struct xdp_md *rx_packet)
26 {
27     void *data_end = (void *)(long)rx_packet->data_end;
28     void *data     = (void *)(long)rx_packet->data;
29     struct ethhdr *eth = data;
30     __u16 eth_type = 0;
31
32     if (!(parse_eth(eth, data_end, &eth_type))) {
33         return XDP_ABORTED;
34     }
35
36     if (eth_type == ntohs(ETH_P_IP))
37         return XDP_PASS;
38     else if(eth_type == ntohs(ETH_P_IPV6))
39         return XDP_TX;
40     else
41         return XDP_DROP;
42 }
```

The function `basic_xdp_filter()` has to be executed for every RX packet in order to filter them all. Therefore, it resembles a main function of a normal C program. A new section `SEC()` called `xdp_prog` is defined immediately before this function (line 24). It serves as a connection between the program and the hook. This section points to the location where the execution of the code must start whenever XDP receives a packet. The definition of `SEC()` is provided in the header file `"bpf_helpers.h"` together with many more eBPF helpers. This header is usually a part of the Linux kernel, but may not be packaged by some distributions. Therefore, it needs to be manually downloaded and included [1].

The program accepts a single argument `rx_packet`, which is a pointer to the struct `xdp_md`. This struct represents the context object containing all the data necessary to access a packet. Its definition is given below.

```
// Source: <uapi/linux/bpf.h>
// User accessible metadata for XDP packet hook
struct xdp_md {
    __u32 data; // start of packet data
    __u32 data_end; // end of packet data
    __u32 data_meta;
    /* Below access go through struct xdp_rxq_info */
    __u32 ingress_ifindex; // rxq->dev->ifindex
    __u32 rx_queue_index;  // rxq->queue_index
};

struct xdp_rxq_info {
    struct net_device *dev;
    u32 queue_index;
    u32 reg_state;
} ____cacheline_aligned;
```

In lines 27-28 the beginning and the end addresses of the packet are cast and assigned to

void pointers. Then the void pointer to the beginning of the packet is assigned to a pointer to a struct representing an Ethernet frame header (`struct ethhdr`). The definition of this struct is listed below.

```
// Source: <uapi/linux/if_ether.h>
struct ethhdr {
    unsigned char   h_dest[ETH_ALEN];      /* destination eth addr    */
    unsigned char   h_source[ETH_ALEN]; /* source ether addr*/
    __be16      h_proto;                  /* packet type ID field*/
} __attribute__((packed));
```

Afterwards, the function `parse_eth()` is called (line 32). It just takes the beginning and the end of the packet and parses its content. The if-statement in its definition (lines 18-19) performs a simple straightforward check whether the size of the ethhdr struct is valid and if it is not, it returns 0 (failure). Then it extracts the packet type field in the 16-bit unsigned integer `eth_type` and returns 1 (success).

In case the function returns 0, an `XDP_ABORTED` action is returned, which raises an eBPF program error and drops the packet. If `pasrse_eth()` returns 1, the program checks whether the header type is IPv4 or IPv6. It is doing that by comparing it with the converted from network byte order to host byte order IPv4 (ETH_P_IP) or IPv6 (ETH_P_IPV6) header using the function `ntohs()`. Then a corresponding action is returned.

In order to compile the program, all the header files need to be linked to the source code. It is important that some of the libraries the code is using are kernel version specific, which means that they need to be properly linked to the code depending on their locations in the used kernel version. The more functionality an XDP program is using, the more complicated the linking is.

After a successful compilation an object file is created. In order to run the program, this object file must be loaded inside the kernel and attached to an interface. For more complicated programs this is done by a controlling user-space program, which also monitors and interacts with the XDP program via eBPF maps. However, for less complex codes like the example presented in this section `iproute` can be used to attach the XDP program to a network interface. All available network interfaces can be found by the shell command `ip link show`. Then the XDP program can be loaded using the command:

```
ip link set dev <net device name> xdp object xdp_obj_file.o sec xdp_prog
```

This command attaches the XDP program to the specified network interface (device), trying to use the device hook or falling back to the generic one otherwise [1]. In case the user wants to force a specific xdp mode, the keywords `xdpdrv` or `xdpgeneric` can be used in place of `xdp` in the command above. If a specific keyword is used and the specified mode cannot be successfully loaded, the command returns a corresponding error message. Provided a successful execution of the command is achieved, an xdp module will be listed along with the other loaded modules for this network interface when the command `ip link show` is run. For unloading an XDP programs the following command is used:

```
ip link set dev <net device name> xdp off
```

4 AF_XDP

AF_XDP is a new address family added to the Linux kernel on 12.08.2018 with the release of version 4.18. It is intended to enable userspace code to perform highly efficient packet processing with as much hardware support as possible and a minimum kernel overhead.

The XDP address family makes it possible for a userspace process to create XDP raw sockets (XSK), which allow direct transfer of packets from the XDP program. The transfer is ensured by the usage of shared memory. This new functionality avoids the in-kernel data path and allows an eBPF program that processes packets to directly forward them to an application in a very efficient way. Moreover, the performance is further improved by gradually adding zero-copy (ZC) driver support, which ensures that there is no copying of data packets between kernel and userspace [4]. In this way much faster transport of packets can be achieved. However, the new approach is restricting the control of the kernel and gives more power to the userspace applications, which decreases the security of the procedure [18]. It is also important to note that ZC is currently implemented to work only with limited network drivers.

AF_XDP can operate in two modes - XDP_SKB and XDP_DRV. If the network driver does not have support for XDP, or the generic mode is explicitly chosen when loading the XDP program, XDP_SKB mode is used for copying the data packets to userspace. This is a fallback mode that works for any network device. However, if the driver has support for XDP_DRV mode, it will be used by the AF_XDP code and much better performance will be achieved [6]. The two modes are illustrated in the figure below and compared with the standard kernel family for raw sockets - AF_INET.

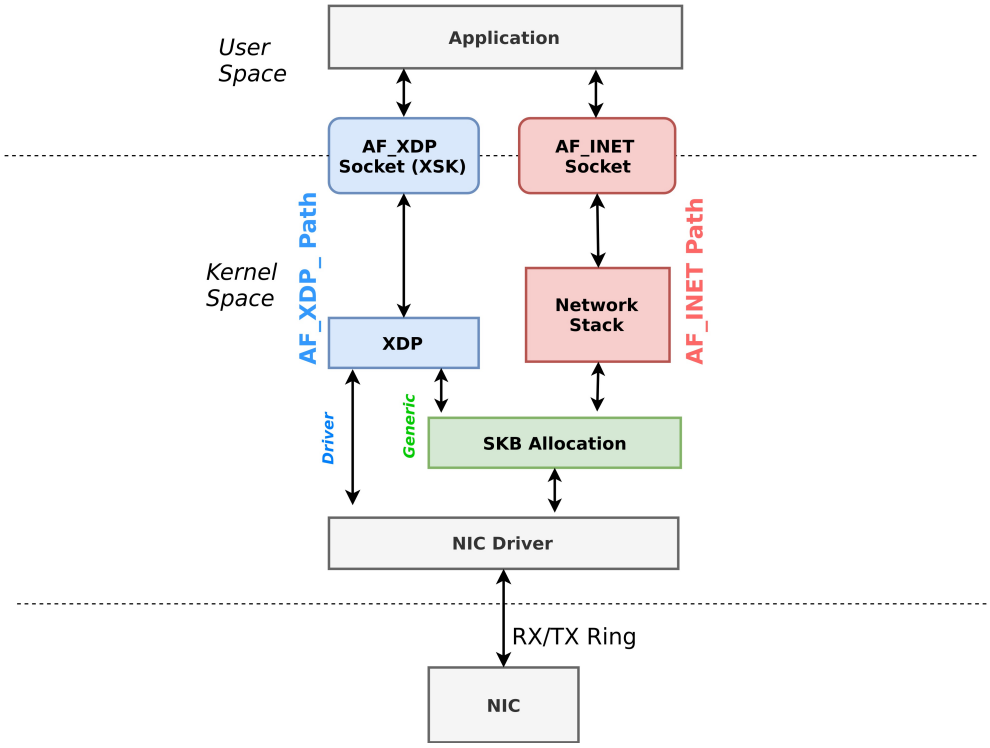

Figure 2: AF_XDP Modes

The XDP address family works by using special rings and shared memory called UMEM.

4.1 Rings

There are four different kinds of rings related to AF_XDP: Fill, Completion, RX and TX. The rings are head(producer)/tail(consumer) based rings. A producer writes the data ring at the index pointed out by `struct xdp_ring producer` member and increases the producer index. A consumer reads the data ring at the index pointed out by `struct xdp_ring consumer` member respectively and increases the consumer index. It is important that the rings are single-producer/single-consumer, therefore, synchronization is needed if multiple processes/threads are reading or writing to them. The size of each ring is always a power of two. The **RX ring** is the receiving side of a socket. Each entry in the ring is a `struct xdp_desc` descriptor. The descriptor contains UMEM memory offset (addr) and the length of the data (len). The **TX Ring** is used to send frames. The `struct xdp_desc` descriptor contains index, length and offset of the frame and is passed into the ring. To start the transfer a `sendmsg()` system call is required. The user application produces `struct xdp_desc` descriptors to this ring. **Fill** and **Completion Ring** are explained in the UMEM section below [6].

The rings are configured and created via the `_RING setsockopt` system calls and mapped to userspace using the appropriate offset to `mmap()` - XDP_PGOFF_RX_RING, XDP_PGOFF _TX_RING, XDP_UMEM_PGOFF_FILL_RING and XDP_UMEM_PGOFF_ COMPLETION_RING. An RX/TX descriptor ring points to a data buffer in a special memory called UMEM. It is possible that RX and TX rings are using the same UMEM. In such cases the packets are not copied between the two rings, which ensures even better performance [6].

4.2 UMEM

UMEM is a region of virtual contiguous memory, divided into equal-sized frames/chunks. It is associated to a netdev (abstraction layer for network device) and a specific queue id of that netdev. UMEM memory can be created by using the `XDP_UMEM_REG setsockopt` system call. This system call is also used to set important configurations of the memory such as the size of the chunk, the total size of the memory or the starting address. Before the UMEM creation the userspace process has to allocate memory for it using standard library functions such as `malloc()`, `calloc()`, `mmap()`, etc. After that the memory needs to be associated to a netdev and a queue id. This is done by using the bind() system call [6].

The UMEM has two kinds of rings - **Fill** and **Completion**. They are used to transfer ownership of UMEM frames between kernel and userspace.

The **Fill ring** is used by the application to send down `addr` to be filled in with RX packet data, i.e. it transfers ownership of UMEM frames from userspace to kernelspace. These frames are used for the ingress path (RX Ring). There are descriptors referencing the addresses of the filled frames, which are recorded in the RX ring once each packet has been received. The `addr` of a frame is simply an offset within the entire UMEM region [6]. The userspace application can request the kernel to place an incoming packet into a specific frame within the UMEM array by adding that frame's descriptor to the queue [7].

The **Completion ring** contains frame addresses that the kernel has transmitted completely using the TX ring and are now available again to userspace for either receiving or sending data packets. The userspace application takes UMEM addresses from this ring. They are used to transfer ownership of UMEM frames from kernel to userspace. The

passed frames are part of the egress path (TX Ring) [6]. The whole AF_XDP datapath is also illustrated in the figure below.

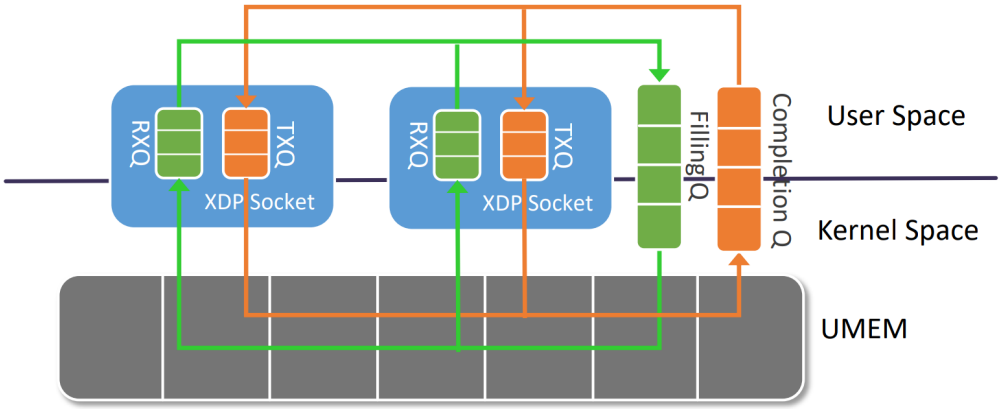

Figure 3: AF_XDP Datapath [18]

4.3 XSK

An AF_XDP socket (XSK) is created with the `socket()` system call. It can receive packets on the RX ring and it can send packets on the TX ring. These rings are registered using the `setsockopts` XDP_RX_RING and XDP_TX_RING. Each XSK needs to be connected to at least one TX/RX ring. Finally, the socket needs to be bound with a bind() call to a device and a specific queue id on that device. After the binding is completed the traffic can start to flow [6]. The process of creating and configuring an XSK is outlined in the listing below.

```
1  sfd = socket(PF_XDP, SOCK_RAW, 0);
2  buffs = calloc(num_buffs, FRAME_SIZE);
3  setsockopt(sfd, SOL_XDP, XDP_MEM_REG, buffs);
4  setsockopt(sfd, SOL_XDP, XDP_{RX|TX|FILL|COMPLETE}_RING, ring_size);
5  mmap(..., sfd, ......); /* map kernel rings */
6  bind(sfd, "/dev/eth0", queue_id,....);
7  for (;;) {
8      read_process_send_messages(sfd);
9  };
```

Listing 1: Creation of XSK [18]

It is important to note that one XSK can be linked to one UMEM, but one UMEM can have multiple XSK sockets, i.e. common UMEM memory can be shared by more than one XDP socket. If a process wants to share UMEM, it has to skip the registration of the memory and its corresponding two rings and instead to set the XDP_SHARED_UMEM flag in the struct `sockaddr_xdp` member `sxdp_flags`. After that it has to submit the XSK descriptor of the process it would like to share UMEM with as well as its own newly created XSK socket to the struct `sockaddr_xdp` member `sxdp_shared_umem_fd`. The new process will then receive frame address references in its own RX ring that point to the shared UMEM. Since the ring structures are single-consumer / single-producer for performance reasons,

the new process has to create its own socket with associated RX and TX rings, which cannot be shared with any other processes. This is also the reason why there is only one set of Fill and Completion rings per UMEM [6].

The incoming packets distribution from an XDP program to the XDP sockets is ensured with the BPF map type `BPF_MAP_TYPE_XSKMAP`, which is a simple array of XDP sockets. An userspace process that is attached to the UMEM can call `bpf()` to store its descriptor in an arbitrary place within the map. There is also an XDP program loaded into the driver, which is able to classify incoming packets and redirect them to a specific index in this map. Thus, the packets are enqueued in the RX queue of the socket in the chosen map entry. If the XDP socket in the chosen map index is invalid (not bound to netdev or queue id) or the map is empty at that index, the packet is automatically dropped [7].

5 Performance Evaluation

In this section three experiments are presented, which serve as a proof of the fast performance and good functionality of XDP and AF_XDP.

The experiments are done on a laptop machine with a quad core Intel Core i7-6700HQ CPU running at 2.5GHz. The operating system is Arch Linux with kernel version Linux 5.0.13. The network device driver used is veth. It is used by a docker container. Veth has support for the XDP driver mode, but does not support Zero-Copy (ZC). The traffic generation tool used is TRex [5], because it manages to generate high enough traffic to reach the limits of the different packet processing techniques.

5.1 Basic XDP actions

The first experiment is to measure the packet processing performance of an XDP program that immediately drops all incoming packets or send them out through the same interface they were received on. One CPU core is allocated to the processing and the the network traffic [pps] is gradually increased until it reaches the maximum per CPU core. The XDP program is attached to a hook located immediately after the veth driver. Docker is used for the creation of a container, inside of which there exists a virtual interface (veth) for testing. A container environment is used, because it easily creates network namespaces that are accessible through the ip-command.

In this experiment three different setups are tested - XDP program in Driver mode, XDP program in Generic mode and in-kernel networking stack performance. Two XDP programs are used - one that contains a returning XDP_DROP action and one that has an XDP_TX action. The programs are at first compiled using the Clang/LLVM compiler into object files. Then the object files are attached to the desired interface using

```
ip link set dev <interface name> xdpdrv obj xdp_program.o sec xdp_prog
```

for driver mode, or

```
ip link set dev <interface name> xdpgeneric obj xdp_program.o sec xdp_prog
```

for the generic XDP mode. In the case where the packet processing of the kernel stack is measured, it is configured to drop all incoming traffic with iptables by using the commands listed below.

```
# Set default chain policies
iptables -P INPUT DROP
iptables -P FORWARD DROP
iptables -P OUTPUT ACCEPT
# Accept on localhost
iptables -A INPUT -i lo -j ACCEPT
iptables -A OUTPUT -o lo -j ACCEPT
# Allow established sessions to receive traffic
iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
```

A packet size of 64 bytes (minimum-sized packets) is used in the experiment, because the challenge with packet processing technologies is the amount of packets processed per second, not their sizes as others have also noted [19]. Furthermore, the CPU power

consumed for increasing traffic is measured until it reaches its maximum (100% CPU core utilization). The results are presented in the line chart below.

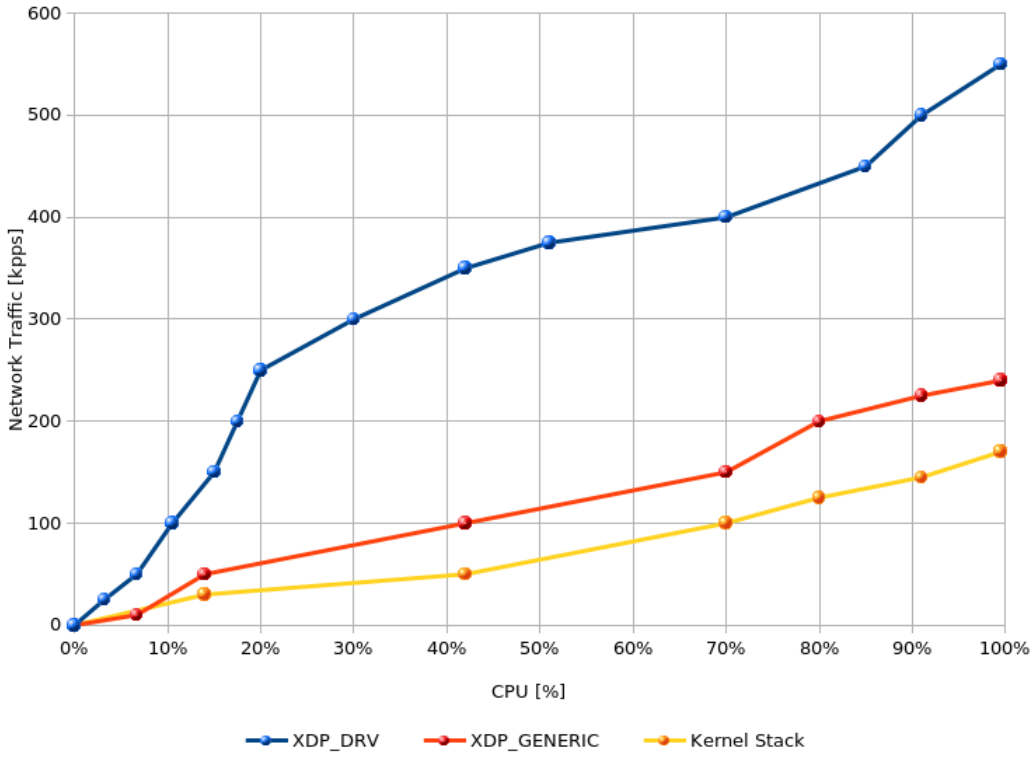

Figure 4: XDP_DROP evaluation

As it can be seen from the chart, XDP in driver mode provides much faster traffic processing than the kernel stack. In fact it is four times faster. This huge performance benefit is due to the fact that the XDP driver mode uses much less context switches. The figure also shows the performance of the generic mode. Although it is far less efficient than the driver mode, it is still faster than the kernel stack. The reason for this is that the generic XDP program is executed immediately after the sk_buff allocation and before other context switches that the kernel performs to allocate more metadata per packet. Another XDP feature that can be deducted from this experiment is that it uses CPU power in linear proportionality to the amount of traffic it processes. Although this is not different than the kernel stack, it makes XDP better than other technologies such as DPDK, which always allocate a whole CPU core to processing packets [12].

The second part of the experiment is to measure the performance of the XDP_TX action. After performing the same procedure as with the XDP_DROP action, the obtained measurements turned out to be approximately the same as the ones shown in Figure 4. The reason is that when the XDP program is executed, it does not make any switches to userspace. Thus, it can be concluded that the time needed to process a packet does not depend on the XDP return action.

5.2 XDP filtering Program

For this experiment the example XDP filtering program presented in section 3 is used. Its performance is measured both in driver and generic mode and the results are compared against the performance of the networking stack. The in-kernel stack is programmed to do the same filtering as the XDP program by using iptables and ip6tables. Identical testing environment to the one in the first experiment is used. The results when full CPU power is consumed of one core by the packet processing are presented in the chart below.

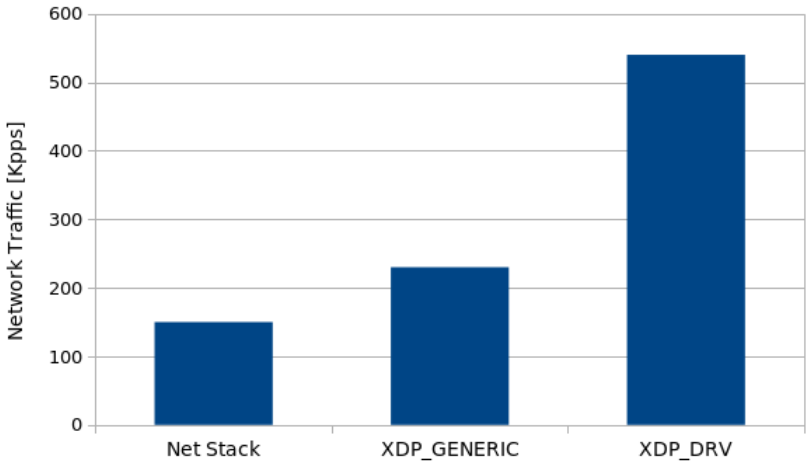

Figure 5: XDP Program evaluation

Here the measurements are again almost the same as with the basic XDP actions. This experiment proves that there is not going to be significant performance difference between the different XDP programs, as long as the desired filtering can be done in a code that is compliant with the eBPF Verifier. This makes XDP very good fit for applications that are requiring limited filtering. Moreover, the XDP performance drop rate is less than the one with the network stack. Therefore, if more CPU power than 2.5 Ghz is used, the rate will be preserved. This means that XDP performance will always be better than the kernel stack one no matter how complicated the XDP program is.

5.3 XDP socket with a Userspace Stack

In this experiment the performance of the new address family AF_XDP is evaluated. It is allowing the creation of XDP sockets (XSK) for redirecting network packets from the XDP program directly to a userspace application. The technology how the XDP sockets work is outlined in section 4.

Additionally, a userspace stack that is processing the raw information received from the socket is combined with the userspace application, which is receiving the raw data. Level-ip [21] is the chosen userspace stack, because it is simple and easily adjustable. It is not processing packets in the best and most efficient way, but for the sake of the experiment, its performance is good enough.

Level-ip is combined with the sample bpf program `xdpsock`, which is distributed with the official Linux kernel [15]. The program provides the code for the communication between the XDP program and a userspace application, which is modified by adding an option to

use the Level-ip userspace stack. If this option is chosen, the program extracts the packets from the XDP hook and enqueues them in a linked-list queue, which is defined as a global variable. Then the queue is read by the `netdev_receive()` function, which consequently is passing the packets to the stack. In order to avoid the global queue getting too big and taking a lot of volatile memory, the application is limited to 64 packets. However, this restriction can be easily modified by changing the macro MAX_QUEUE_SIZE. If the userspace stack is slower than the XDP socket performance and the queue gets completely filled, the process of extracting packets from the hook is going to be slowed down. This ensures that a bottleneck in the data path will not occur.

For this experiment the modified `xdpsock` program is run on the veth docker interface and traffic generation is also done with the TRex tool. The packets are of size 64 bytes and their number is increased until full power of one CPU core is used. Then this is compared to the regular network stack performance. The measurements are outlined in the line chart below.

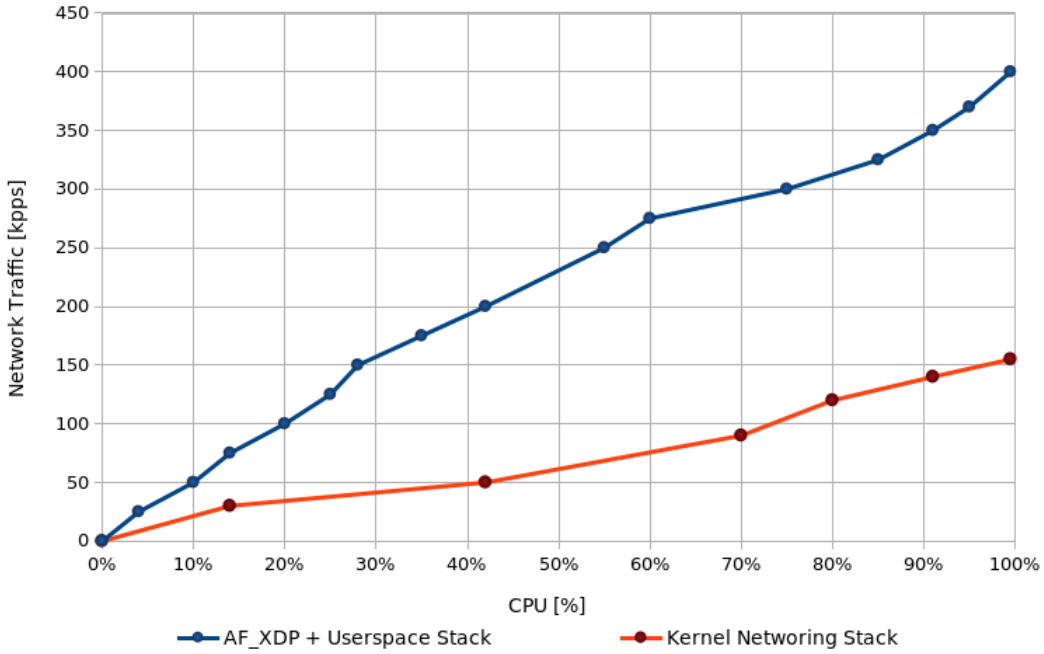

Figure 6: XDP Program evaluation

The results are showing that the performance of the XDP socket combined with a userspace stack is worse that the XDP driver mode in the first two experiments. However, it is still much better than the kernel network stack. The maximum achieved performance is 400-410 kpps with the XDP socket and 150-160 kpps with the networking stack (no explicitly defined filtering actions of the stack). The reason for this is the fact that the packets are copied to userspace, which makes the process slower and they are also processed by the usespace application (stack) afterwards.

Overall, this new approach of using an XDP socket with a userspace stack provides approximately three times better performance than the regular networking stack. Moreover, a filtering XDP program is still hooked to the device driver, which means that there is filtering in the kernel, which can take advantage of the in-kernel functionality and its secu-

rity mechanisms. This is a very important aspect that makes XDP a better choice, when compared with the fast technologies of processing network traffic, which are completely bypassing the kernel.

6 Conclusion

The paper describes a novel packet processing technology called eXpress Data Path (XDP). This technology is working by executing a userspace code in an in-kernel virtual CPU. This is possible thanks to the extended version of the Berkeley Packet Filter (BPF). XDP offers much faster performance than traditional approaches such as the Linux netfilter. Although it is not as fast as new packet processing technologies that are completely bypassing the kernel side, it preserves the kernel functionality and security mechanisms by performing filtering operations inside it. In this way XDP achieves a good trade-off between secure packet processing technologies such as the in-kernel networking stack, and very fast ones, which are bypassing the kernel such as DPDK.

Because of this balance, XDP has been rapidly developed and constantly upgraded in the recent years. An example of an innovative addition is the address family AF_XDP that enables the creation of XDP sockets for transmitting the packets from the XDP program directly to a userspace process. This family further expands the filtering capabilities of XDP.

In order to evaluate the performance of this modern packet processing technology, three experiments have been conducted. First, the performance of the basic XDP filtering actions is measured and it turned out that it can process network traffic approximately four times faster than the the network stack when the driver hook is used. Additionally, the performance of a single-action XDP program is compared to a more complicated one. The results showed that the length of the XDP program code is not reflecting significantly on the performance of the packet processing as long as it complies with the eBPF Verifier requirements. The last experiment is examining the performance of the XDP socket combined with an implementation of a userspace stack. The results achieved showed such combination offers a three-fold performance improvement of the regular Linux kernel networking stack.

The conducted experiments serve as a proof of the advantages XDP offers in terms of performance. However, it also has some weaknesses. Not all network drivers are supporting its faster driver mode. The newly added feature Zero-Copy (ZC), which ensures packets are not copied, but directly transferred from kernel to userspace, is also not supported by many network drivers. Technologies fully bypassing the kernel are still faster than XDP, as pointed by other researchers [12]. Nevertheless, XDP has been expanding a lot since its creation in 2016 and these weaknesses are nothing more than further room for improvement.

As part of the Linux kernel, XDP constantly undergoes changes and improvements. Possible future developments are improving the eBPF limitations, adding more tools for debugging, increasing the driver support for native mode/zero-copy and combining it with other technologies for the creation of more efficient and reliable frameworks.

References

[1] Paolo Abeni. “Achieving high-performance, low-latency networking with XDP: Part I”. https://developers.redhat.com/blog/2018/12/06/achieving-high-performance-low-latency-networking-with-xdp-part-1/, 2018.

[2] Cilium Authors. “BPF and XDP Reference Guide”. https://cilium.readthedocs.io/en/v1.3/bpf/, 2018.

[3] Diego Calleja. “Linux 4.20”. https://kernelnewbies.org/, 2018.

[4] Diego Calleja. “Linux_4.18 - kernel notes”. Article. https://kernelnewbies.org/, Nov. 2018.

[5] Cisco. “TRex: Realistic Traffic Generator”. https://trex-tgn.cisco.com/, 2019.

[6] The kernel development community. “AF_XDP”. Article. https://www.kernel.org/doc/html/v4.18/networking/af_xdp.html, Feb. 2018.

[7] Jonathan Corbet. “Accelerating networking with AF_XDP”. Article. https://lwn.net/Articles/750845/, Apr. 2018.

[8] Jonathan Corbet. “The BPF system call API”. lwn.net/Articles/612878/, 2014.

[9] Matt Fleming. “A thorough introduction to eBPF”. Article. https://lwn.net/Articles/740157/, Dec. 2015.

[10] Linux Foundation. “Data Plane Developement Kit”. https://www.dpdk.org/, 2019.

[11] Diego Pino Garcia. “The eXpress Data Path”. https://blogs.igalia.com/dpino/2019/01/10/, 2019.

[12] Toke Hoiland-Jorgensen, Jasper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. “The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel”. CoNEXT ’18, December 4-7, 2018, Heraklion, Greece, Dec. 2018.

[13] Van Jacobso and Steven McCann. “The BSD Packet Filter: A New Architecture for User-level Packet Capture”. Winter USENIX Conference, Jan. 1993.

[14] Michael Kerrisk. “Linux programmer’s manual. Technical report”. http://man7.org/linux/man-pages/man2/bpf.2.html, 2018.

[15] Inc. the Linux Kernel Organization. “The Linux Kernel”. https://www.kernel.org/, 2019.

[16] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. “Creating Complex Network Services with eBPF: Experience and Lessons Learned”. In: June 2018.

[17] Srivats P. “Ostinato Packet Generator”. https://ostinato.org/, 2019.

[18] Zhang Qi and Li Xiaoyun. “DPDK PMD for AF_XDP”. Article. Intel, 2018.

[19] Luigi Rizzo. “Netmap: a novel framework for fast packet I/O”. Universita di Pisa, Italy, 2012.

[20] Jay Schulis, Daniel Borkmann, and Alexei Starovoitov. “Linux Socket Filtering aka Berkeley Packet Filter (BPF)”. www.kernel.org/doc/Documentation/networking, 2018.

[21] Xiaochen Wang. “Level-IP”. https://trex-tgn.cisco.com/, 2019.