COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

COMS 6901 E Sec 028: PROJECTS IN COMPUTER SCIENCE - SPRING SEMESTER 2022

FINAL REPORT

Application of e-BPF to improving system-performance w.r.t network I/O

Souryadeep Sen - MS Computer Engineering - UNI: ss6400

May 10, 2022

Abstract

This report describes my work during the Spring Semester of 2022, at Columbia University, in the Programming Systems Lab. The project entails the application of extended-BPF (Berkeley Packet Filter) at the XDP layer of the kernel network stack, to explore its impact on response times in a simulated multi-hop forwarding router system, and compare its performance with the traditionally used ip/nftable filters that are used as firewalls in Linux systems, to either drop or forward packets.

1. Introduction

1.1 Overview of e-BPF and XDP

e-BPF (extended BPF) is a revolutionary technology with origins in the Linux kernel that can run sandboxed programs in an operating system kernel. It is used to safely and efficiently extend the capabilities of the kernel without requiring to change kernel source code or load kernel modules. Historically, the operating system has always been an ideal place to implement observability, security, and networking functionality due to the kernel's privileged ability to oversee and control the entire system. At the same time, an operating system kernel is hard to evolve due to its central role and high requirement towards stability and security. The rate of innovation at the operating system level has thus traditionally been lower compared to functionality implemented outside of the operating system. With e-BPF, application developers can now add functionality to the kernel without tinkering with the kernel, by running sandboxed programs. The operating system then guarantees safety and execution efficiency of the kernel with the help of a just in time compiler.

The project applies e-BPF to networks, though e-BPF is used for security and observability too. The goal is to study the improvement in response time on multi-hop routing systems (simulated in this project, as experiments are run on a localhost, with multiple spawned and forwarded ports), where forwarding is done at the XDP layer, with the help of a e-BPF XDP filter, vs traditional routing, allowing propagation up the kernel network stack.

The traditional flow of a network packet, from the epoch it is received off the wire on the **NIC** (network interface card), to when it is read by the read system call using a socket descriptor for the socket, where the packets destination is, and finally used by the application the packet was meant for, is a highly inefficient operation. To explain the steps involved in a network packet propagating up the kernel network stack would require more than just a networking course, but this article [1] does good justice to describing the steps, though the linux kernel has since evolved, so some out of data structures and functions are mentioned. On a high level, when the packet is first received on the NIC, an interrupt is raised, that requires immediate attention of the kernel, to read the data off the wire and into a kernel buffer, to avoid overwhelming the buffer. As highlighted in red in the below image (***figure 1***), the packet is put on a back-log queue, from where it is picked up by the kernel for further processing, based on the whims and fancies of the linux scheduler. This constitutes the 'Bottom Half' processing, since this is a time consuming process, and not as urgent as taking the packet off the NIC. The packet is passed up the kernel networking stack (***figure 2***) which constitute the layers making up the OSI model. Each layer requires stripping off headers and allocating data structures (such as the sk_buff data structure [2], [3]), and all in all, is a bottleneck when compared to the high processor speeds nowadays. In figure1, we can see that for an operation such as routing a packet to a different host, the packet would need to propagate up to the network/IP layer, reference the IP tables to be forwarded to its destination. Think about the internet, which is made up of multiple routing points, each router having a scaled down version of linux running on it, these packets would need to go through this path at each and every router, which is expected to slow down the response time.

Now let us understand where in the stack the XDP filter would sit, and how/why this can potentially address the bottleneck of allowing propagation of the packet up the linux networking stack. The diagrams w.r.t the OSI model, networking stacks are a bit overloaded and look similar, but vary a bit, causing some confusion when comparing them to understand where exactly each layer is when juxtaposed with the other diagram, and I will attempt to give some clarity while referencing these images. In ***Figure 3*** we can see where the XDP [4], [5] layer sits. It is just off the NIC, and nowadays, there is hardware support to run the XDP filters on the NIC too. Just above the XDP layer, we can see the tc (traffic control) layer, above which we can see the TCP stack, which is essentially the network layer and up, in the OSI model. The difference between XDP and tc layer is that the tc layer has better packet mangling capabilities and access to more meta-data than the XDP layer. This itself makes the XDP layer more efficient, with a trade-off of less meta-data. The tc layer has access to the sk_buff struct, referenced earlier, that requires kernel memory allocation which is expensive. The XDP layer accesses the xdp_buff struct, and does so at an earlier hook point than the tc layer. So as we can see, the XDP layer and its hook point sit several layers earlier than the networking stack.

As mentioned above, 'hook points', is a good segway into the next topic of interest, i.e. how does e-BPF work, and what are hook points w.r.t e-BPF? eBPF programs are event-driven and are run when the kernel or an application passes a certain hook point [6]. Pre-defined hooks include system calls, function entry/exit, kernel tracepoints, network events, and several others. If a predefined hook does not exist for a particular need, it is possible to create a kernel probe (kprobe) or user probe (uprobe) to attach eBPF programs almost anywhere in kernel or user applications (***figure 4***). As we can see in Figure 4, e-BPF programs can be hooked on to system calls such as read and write, send/recv socket related system calls, and even all the way down at the NIC. This is where we hook the XDP filter in this project.

Briefly touching on the process of getting an e-BPF program to run. When the desired hook has been identified, the eBPF program can be loaded into the Linux kernel using the bpf system call. This is typically done using one of the available eBPF libraries. As the program is loaded into the Linux kernel, it passes through two steps before being attached to the requested hook: Verification and JIT compilation. The verifier ensures that the e-BPF program is safe to run. Loading the e-BPF program requires root privileges, unless configured for unprivileged operation, the program does not crash or harm the system (sandboxed operation), and the program always runs to completion (infinite loops will be rejected by the verifier). The Just-in-Time (JIT) compilation step translates the generic bytecode of the program into the machine specific instruction set to optimize execution speed of the program. This makes eBPF programs run as efficiently as natively compiled kernel code or as code loaded as a kernel module.

This is just about sufficient information to delve into the specifics of the project, without clogging the report with too much literature.

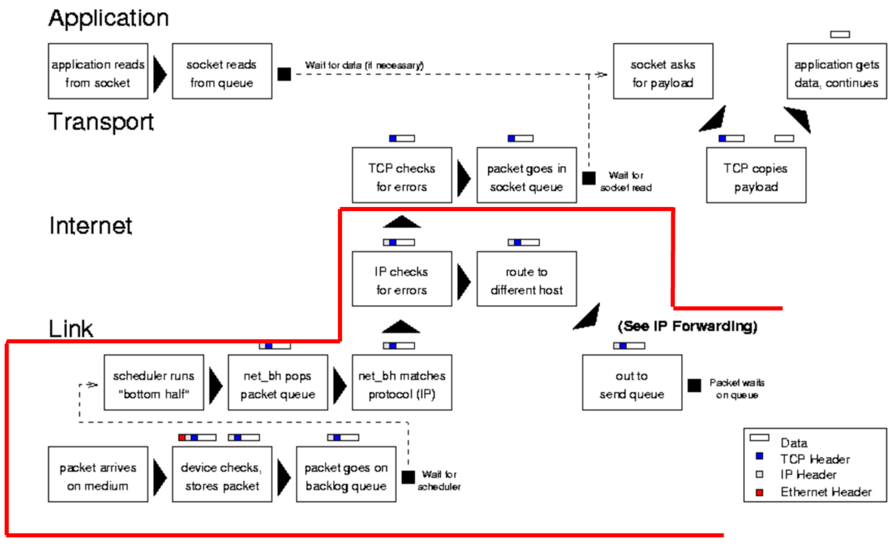
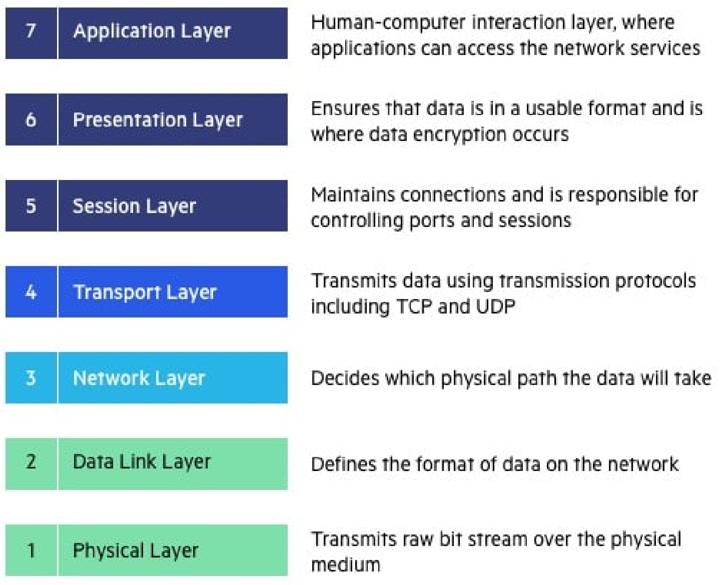


Figure1: Life of a packet through the linux network stack



7	Application Layer	Human-computer interaction layer, where applications can access the network services
6	Presentation Layer	Ensures that data is in a usable format and is where data encryption occurs
5	Session Layer	Maintains connections and is responsible for controlling ports and sessions
4	Transport Layer	Transmits data using transmission protocols including TCP and UDP
3	Network Layer	Decides which physical path the data will take
2	Data Link Layer	Defines the format of data on the network
1	Physical Layer	Transmits raw bit stream over the physical medium

Figure 2: The 7 layers of the OSI model

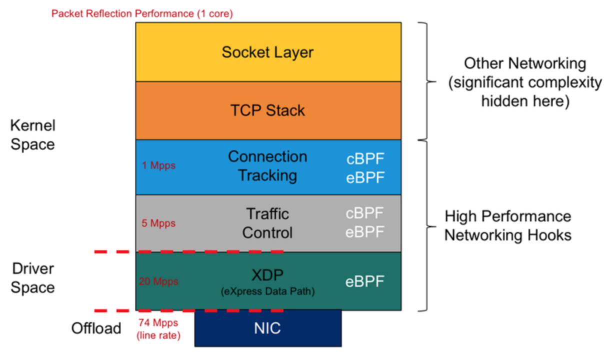


Figure3: Points of attachment of e-BPF (XDP) filter

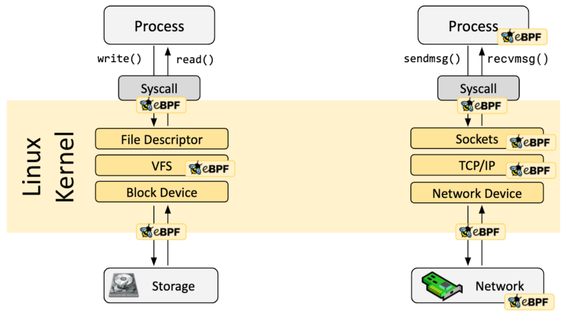


Figure4: e-BPF hook points

1.2 Motivation

The previous section gives a high level overview of why e-BPF filters (programs), hooked to earlier layers of the network stack can be more efficient than letting the packets flow up the networking stack. This, in fact, is highly application dependent. For example, if the application requires the packet, it would need to go through the linux networking stack to be accessed. In this project, the goal is to study the response time of a **simulated routing system**, just like how the internet functions, with multiple routing points, each running the linux OS on them. The infrastructure used for this project is basic. Instead of having a network of multiple

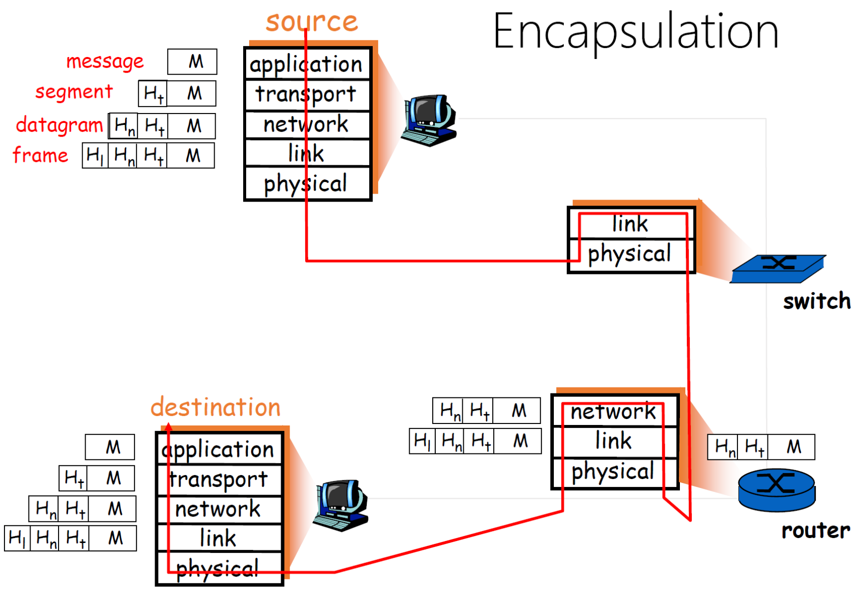


Figure 5: Packet encapsulation and flow across the network

routing points, we set up the experiments on a localhost machine, running linux on a VM. The simulated route points are ports setup on localhost using socat, netcat and a server application developed during my Operating Systems course at Columbia. Since we want to simulate multiple routers, multiple ports are set up, and forwarded to the next, from client to server as shown in figure 6 below.

Using this setup, response times will be studied, sending GET requests from client to server with multiple varying sizes, to see if there is an improvement in response time, by setting up a XDP filter to forward packets at each port, vs using traditional methods of forwarding ports using ip/nftables.

This way, we are trying to study if we can make routers more efficient, since their primary job is to route and forward packets.

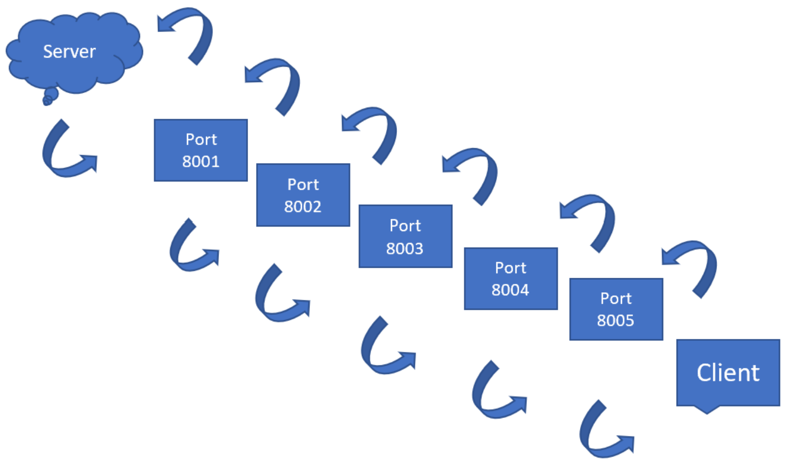


Figure 6: Simulated routing system on localhost machine

Another look at the network traffic path in figure 7 below, from [1] makes it clear why the multiple stages of processing of the linux network stack can make operations such as routing/forwarding inefficient. Once again, the resource [1] is great to understand the networking stack and the various layers, structures and allocations along the stack.

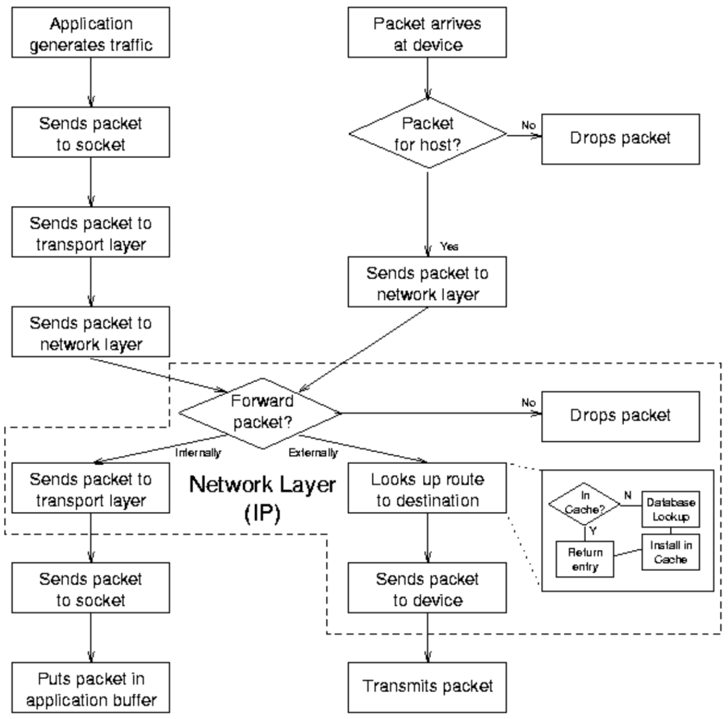


Figure 7: Network traffic path

1.3 Some Details about e-BPF

We touched on some aspects of using e-BPF, which is verified and loaded into the kernel, running in a sandbox environment. In this section, we discuss a bit more on the technology and its features with the use of some code snippets (taken from [8]), but stay as brief as possible and defer the reader to resources such as [7], [8], which is a good starting point. This section is just to understand what those hook points are, that was mentioned earlier, and how to specify them in the code.

`#include <linux/bpf.h>` includes some basic BPF-related types and constants necessary for using the kernel-side BPF APIs (e.g., BPF helper functions, flags). This header is needed for the bpf_helpers.h header, included next. bpf_helpers.h is provided by libbpf and contains most-often used macros, constants, and BPF helper definitions, which are used by virtually every existing BPF application. `bpf_get_current_pid_tgid()` above is an example of such a BPF helper.

`SEC("tp/syscalls/sys_enter_write") int handle_tp(void *ctx) { ... }` defines the BPF program which will be loaded into the kernel. It is represented as a normal C function in a specially-named section (using SEC() macro). Section name defines what type of BPF program libbpf should create and how/where it should be attached in the kernel. In this case, we define a trace point BPF program, which will be called each time a write() syscall is invoked from *any* user-space application.

The next interesting feature is the global variable `my_pid`. This global variable can in fact be read and written from user space. The code snippet shown below is the BPF side program. The BPF programs can run in pairs, one which would be the user side program, and the other is the BPF side, which is loaded into the kernel by the verifier. It can also be used to pass data back-and-forth between in-kernel BPF code and user-space control code.

The **`bpf_get_current_pid_tgid`**`()` is used to ensure that the program proceeds only if the sys call was generated by the process. This is an important point to note, as the BPF program is not specific to a process, but applies to all processes making the sys call, such as the write sys call. Similarly in this project, the e-BPF filter essentially is hooked to XDP point, but applies to all ports. So we need to specify the ports to filter. **`bpf_trace_printk`**`()` is similar to using printf in user space, and printk in the kernel for debug purposes and can be viewed by using:

```
sudo cat /sys/kernel/debug/tracing/trace_pipe
```

```c
// SPDX-License-Identifier: GPL-2.0 OR BSD-3-Clause
/* Copyright (c) 2020 Facebook */
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

char LICENSE[] SEC("license") = "Dual BSD/GPL";

int my_pid = 0;

SEC("tp/syscalls/sys_enter_write")
int handle_tp(void *ctx)
{
    int pid = bpf_get_current_pid_tgid() >> 32;

    if (pid != my_pid)
        return 0;

    bpf_printk("BPF triggered from PID %d.\n", pid);

    return 0;
}
```

The user space side defines a number of bpf map structs to communicate with the bpf side application, but what is most worth mentioning, is the below snippet:

```
skel = minimal_bpf__open_and_load();
```

This helper function is from the libbpf library, used to load the BPF program into the kernel. This is an important point, as the BPF user space program is the one that loads the BPF side program into the kernel, for verification and JIT compilation.

2. Experimental Setup and Code

2.1 The Setup

All experiments in this project were carried out on a personal computer running Windows native OS, and a VM loaded with Linux.

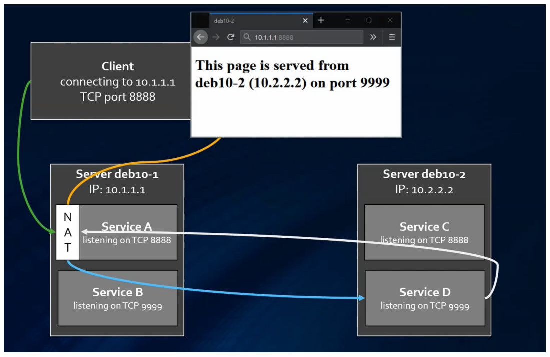
The Linux Kernel version is 5.10.57 and the Debian OS distribution 11 is used on a VMWare Workstation Pro Virtual Machine. The PC is a Dell XPS 13 9360, using Intel Core i7 8th Gen 2GHz Quad Core processor, with 8GB of DDR4 SDRAM.

To simulate the multi-hop routing system on localhost, multiple ports are spawned using either netcat or socat. Socat was preferred, since its command line options allow port forwarding, as well as simply just opening the port in listening mode.

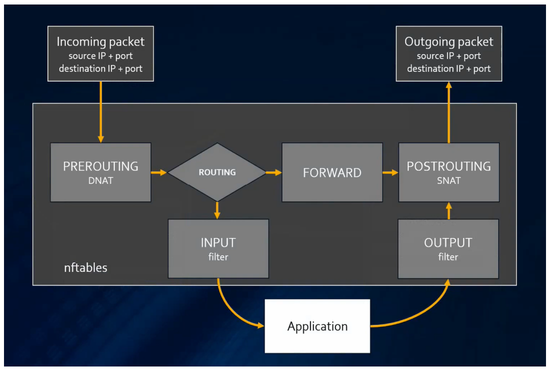
The server program is established on a particular port, to which the client tries to connect. However, the client does not connect to the server directly, rather, it connects via intermediate ports, that are port forwarded to the next in chain.

For example, the chain looks like: `7996 -> 7997 -> 7998 -> 7999 -> 8000`

A glimpse at what the data path of a forward chain would look like, is shown in the images below.



This page is served from deb10-2 (10.2.2.2) on port 9999



The figures above are an example of nftables used for forwarding, nftables are used in the linux kernel nowadays for packet filtering/forwarding, and is something the socat ports will reference. They are at a lower level in the kernel stack compared to the IP tables (traditional firewall in Linux), and hence are used to compare to XDP, which is at an even lower level.

Response time statistics are measured using the siege HTTP load tester Linux command line tool, for both socat port forwarding and XDP filter port forwarding.

2.2 Code

All the code developed for this project can be found at the Programming Systems Lab github repo: https://github.com/Programming-Systems-Lab/souryadeep

There were several programs developed/used for this project. They include a HTTP TCP server (developed during my OS course). This constitutes the server with which siege is tested, through port forwarding. We will not delve into the details of this code, and it suffices to know that the server receives a GET request string from the client and responds with the requested data. The server can be found in the multi-server directory in the repository. A HTML creator program was written to create HTML formatted files, each of double the size of the previous, to be used for testing response time with varying workloads. A python script to parse through the raw data files generated by siege. And lastly, the XDP filter, which is the purpose of implementing this project.

Section 1.3 touches on some basics of the structure of e-BPF programs. Keeping that in mind, we discuss the e-BPF XDP filter. The filter can be found in the packet-route-filter directory in the github repo mentioned above.

Like all e-BPF programs, this too consists of 2 parts. 1. The user space side program, and 2. the BPF side program. To be more specific, the user space side program in this case is `main.py,` which is created with the help of the **bcc** [9] library. In section 1.3, we spoke about using **libbpf**, which is another BPF library widely used. For the filter used in this project, the user space side is purely used for loading the BPF side program into the kernel, and not for communicating with the BPF (kernel) side program, as we are only attempting to forward packets, which can be handled on the kernel side completely. In the case of applications like observability, we would want to use **BPF MAPS** to communicate with the BPF side program.

In the below code, which constitutes the user space side code, we see that the device is set to the loopback device. The source file is set as `filter.c`, which is the kernel side program and is loaded into the verifier and finally into the kernel. The function is `udpfilter,` and is loaded as a BPF XDP type function and attached to the loopback device, with the help of the **bcc** library routines below.

```
#!/usr/bin/env python3
from bcc import BPF
device = "lo"
b = BPF(src_file="filter.c")
fn = b.load_func("udpfilter",BPF.XDP)
b.attach_xdp(device, fn, 0)

try:
  b.trace_print()
except KeyboardInterrupt:
      pass
b.remove_xdp(device, 0)
```

Next, we discuss the **`filter.c`**, kernel side program, which is the crux of the problem being attempted to solve. The aim is to create a simulated multi-hop filter on a localhost machine, with the aid of a loopback device, having spawned multiple ports that represent routers in the internet, and compare/observe performance benefits if any.

The function below is what we have registered and loaded into the kernel at the loopback device, using the user space program. The struct xdp md is available for use at this level.

```
int udpfilter(struct xdp_md *ctx) {
```

Three structs that constitute the headers of the packet at various layers, can be extracted from the xdp_md struct. `struct ethhdr *eth, struct iphdr *ip, struct tcphdr *tcp.` These are used in conditional checks to ensure that they are falling within the designated boundaries, else the packet is of a different type, and is allowed to pass through the kernel networking stack (`return XDP_PASS`). If the packet is of the protocol `IPPROTO_TCP,` then the packet is a candidate for forwarding. The forwarding chain is hardcoded for experimental purposes, but can be made generic by using BPF MAPS to pass in the port numbers from the user space side, once those ports have been opened and is known in the user space side.

The filter was originally written for the UDP protocol, which was considerably easier than developing the one for TCP. The UDP protocol, being a connectionless protocol, does not require establishing a connection between client and server. Once the packet is sent, there is no guarantee of being received at the server, and as a result, there is no `ACK` from the server back to the client. So for the UDP case, if a destination port of the packet is detected when received off the wire, the filter checks whether the `dest` port is a specific port, and if it is, changes the `dest` port and forwards the packet (`XDP_TX`), without propagating up the network stack. This is pretty straightforward, but the server developed was for TCP (HTTP), and siege cmd line utility, which measured performance, was also for TCP. In general, the idea of testing for TCP seemed better, so that we know that data was reliably reaching its intended destination, to profile the XDP filter. The challenge of porting the filter to TCP, was it is a connection oriented and reliable protocol. Unlike UDP, changing only the `dest` port is not sufficient.

To understand this issue a bit further (required quite a lot of debugging), let's consider the forwarded chain `8000 -> 8001 -> 8002`. When the packet is sent to 8001 from 8000, the UDP filter would change its `dest` port to 8002. So far so good, the packet gets forwarded to 8002 and TCP changes `source` port to 8001 (done by the protocol). The packet reaches 8002, which is the destination. Unlike UDP, this is not the end of the lifecycle of a TCP packet, it needs to send back data an ACK message. TCP changes the `dest` port to 8001 and `source` port to 8001. Now once the packet reaches 8001, it is no longer being forwarded, so TCP thinks its current `source` port is the `dest` port, and starts to ping between 8001 and 8002 forever. This was debugged using the useful `bpf_trace_printk() []` helper macro, to see how the `dest` and `source` ports were being modified at each port.

Finally, the solution was to modify both `dest` and `source` ports at each port in the chain.

The final challenge was, in the chain of forwarded ports, we can never know the source port in advance, and it cannot be hardcoded, as the siege cmd line tool does not bind to a port. Tools like netcat bind to a port, but they are lacking stress testing capabilities of siege. So a method needed to be devised to track the source port in the chain, and this is where we utilized the **BPF HASHmap** [9] to store this port as a global variable, so that it could be tracked.

When the packet reaches the first port in the chain, the previous, i.e. the starting port is currently set as the packet's `source` port. It is updated with a key = 0, into this map, and in the reverse path, on the last path, is extracted from the map using key = 0, and set as the final `dest` port.

2.3 Commands to execute tests

1) Port forwarding using **socat:**
    a) Setup the 5 ports in the port hop chain
    b) Open multiple tmux windows, one for each port, one for the server and one to run siege
        i) Open multi-server: `./multi-server 8000 ../html`
        ii) Listen and forward port 7999 to port 8000: `sudo socat TCP4-LISTEN:7999,fork TCP4:localhost:8000`
        iii) Listen and forward port 7998 to port 7999: `sudo socat TCP4-LISTEN:7998,fork TCP4:localhost:7999`
        iv) Listen and forward port 7998 to port 7999: `sudo socat TCP4-LISTEN:7997,fork TCP4:localhost:7998`
        v) Listen and forward port 7997 to port 7998: `sudo socat TCP4-LISTEN:7997,fork TCP4:localhost:7998`
        vi) `siege -c 1 -r 100 http://127.0.0.1:7996/index15.html`

2) Port forwarding using **XDP:**
    a) Open a new tmux window and start the XDP filter
        i) `sudo python3 main.py`
    b) `siege -c 1 -r 100 http://127.0.0.1:7996/index15.html`
3) Measure run time statistics for both

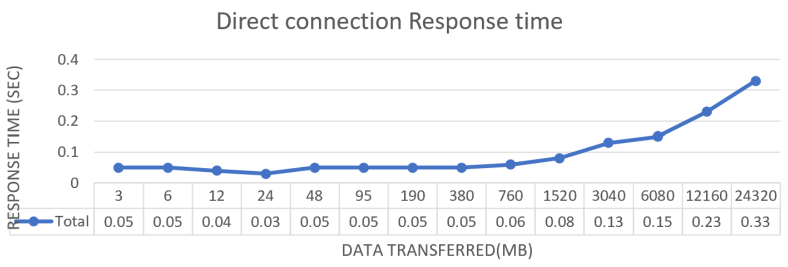
3. Tests and Results

The tests, as mentioned before, were run on a simulated multi-hop routing system, on a localhost machine, using a loopback device. Two sets of forwarding filters were used, one with Socat, which uses the nftables to look up the next destination, and the other, a XDP filter, that forwards packets at the XDP layer based on a predefined route.
Each Data size was run 100x with siege, for each port hop system. This gives us 15 (data sizes) * 4 (different hops) * 100 (iterations) = 600 data points to analyze.

3.1 Plots and Tables

Below are the results (*graphs and tables*) for response `time (seconds) vs data transferred (MB)`, for different levels of hop systems:

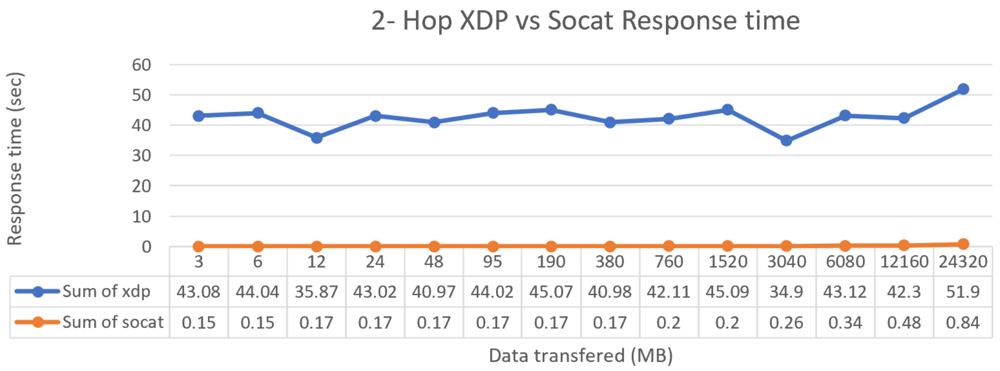
- Direct connection between client and server (`no filters`)



	3	6	12	24	48	95	190	380	760	1520	3040	6080	12160	24320
Total	0.05	0.05	0.04	0.03	0.05	0.05	0.05	0.05	0.06	0.08	0.13	0.15	0.23	0.33

Figure8: Direct Connection Response time

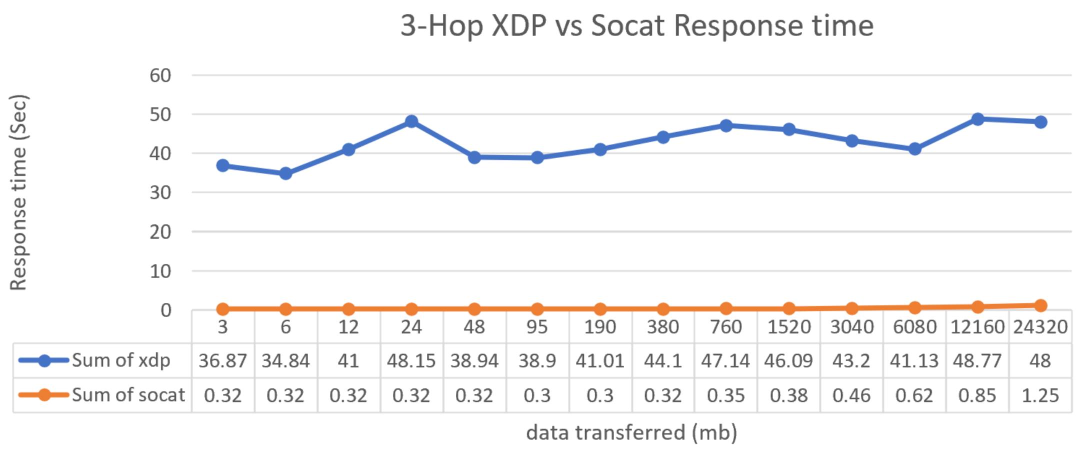
- `2-Hop system: siege port -> 7996 -> 7997`



	3	6	12	24	48	95	190	380	760	1520	3040	6080	12160	24320
Sum of xdp	43.08	44.04	35.87	43.02	40.97	44.02	45.07	40.98	42.11	45.09	34.9	43.12	42.3	51.9
Sum of socat	0.15	0.15	0.17	0.17	0.17	0.17	0.17	0.17	0.2	0.2	0.26	0.34	0.48	0.84

Figure9: 2-Hop XDP vs Socat Response time

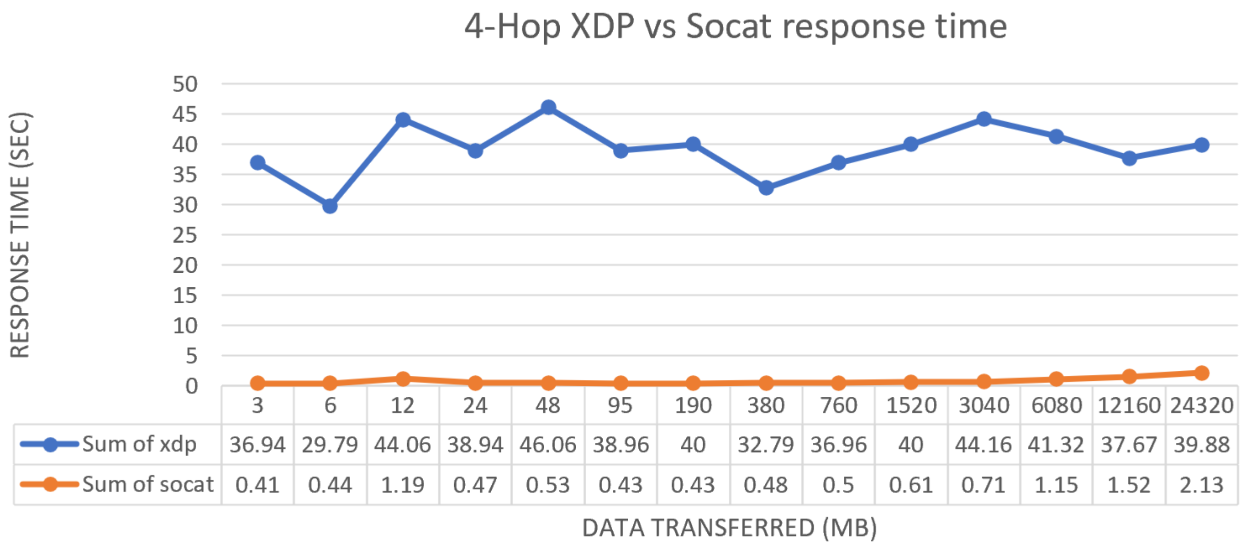
- 3-Hop **system**: siege port -> 7996 -> 7997 ->7998



	3	6	12	24	48	95	190	380	760	1520	3040	6080	12160	24320
Sum of xdp	36.87	34.84	41	48.15	38.94	38.9	41.01	44.1	47.14	46.09	43.2	41.13	48.77	48
Sum of socat	0.32	0.32	0.32	0.32	0.32	0.3	0.3	0.32	0.35	0.38	0.46	0.62	0.85	1.25

Figure10: 3-Hop XDP vs Socat Response time

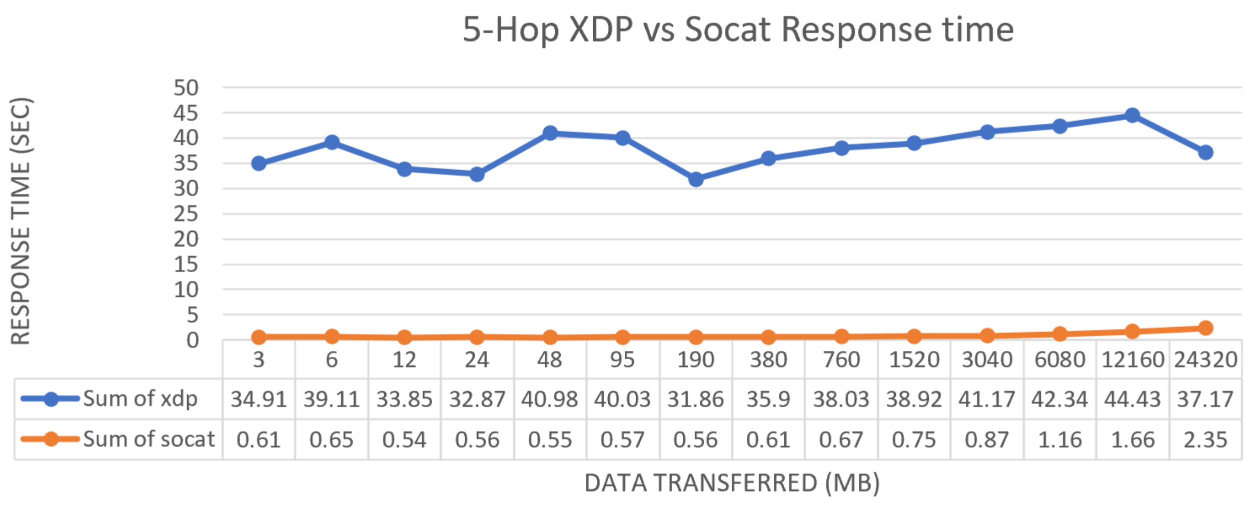
- 4-Hop **system**: siege port -> 7996 -> 7997 -> 7998 -> 7999



	3	6	12	24	48	95	190	380	760	1520	3040	6080	12160	24320
Sum of xdp	36.94	29.79	44.06	38.94	46.06	38.96	40	32.79	36.96	40	44.16	41.32	37.67	39.88
Sum of socat	0.41	0.44	1.19	0.47	0.53	0.43	0.43	0.48	0.5	0.61	0.71	1.15	1.52	2.13

Figure11: 4-Hop XDP vs Socat Response time

- 5-Hop **system**: `siege port -> 7996 -> 7997 -> 7998 -> 7999 -> 8000`



	3	6	12	24	48	95	190	380	760	1520	3040	6080	12160	24320
Sum of xdp	34.91	39.11	33.85	32.87	40.98	40.03	31.86	35.9	38.03	38.92	41.17	42.34	44.43	37.17
Sum of socat	0.61	0.65	0.54	0.56	0.55	0.57	0.56	0.61	0.67	0.75	0.87	1.16	1.66	2.35

Figure12: 5-Hop XDP vs Socat Response time

The below table summarizes all the data from the plots above:

	Response Time (sec)							
	2 hop		3 hop		4 hop		5 hop	
File Size (MB)	socat	XDP	socat	XDP	socat	XDP	socat	XDP
3	0.15	43.08	0.32	36.87	0.41	36.94	0.61	34.91
6	0.15	44.04	0.32	34.84	0.44	29.79	0.65	39.11
12	0.17	35.87	0.32	41	1.19	44.06	0.54	33.85
24	0.17	43.02	0.32	48.15	0.47	38.94	0.56	32.87
48	0.17	40.97	0.32	38.94	0.53	46.06	0.55	40.98
95	0.17	44.02	0.3	38.9	0.43	38.96	0.57	40.03
190	0.17	45.07	0.3	41.01	0.43	40	0.56	31.86
380	0.17	40.98	0.32	44.1	0.48	32.79	0.61	35.9
760	0.2	42.11	0.35	47.14	0.5	36.96	0.67	38.03
1520	0.2	45.09	0.38	46.09	0.61	40	0.75	38.92
3040	0.26	34.9	0.46	43.2	0.71	44.16	0.87	41.17
6080	0.34	43.12	0.62	41.13	1.15	41.32	1.16	42.34
12160	0.48	42.3	0.85	48.77	1.52	37.67	1.66	44.43
24320	0.84	51.9	1.25	48	2.13	39.88	2.35	37.17

Table1: File size vs Hops vs Response times for XDP and socat

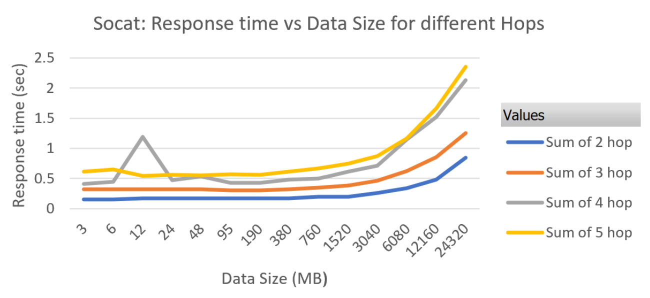


Figure13: Socat: Response time vs data size for different hop

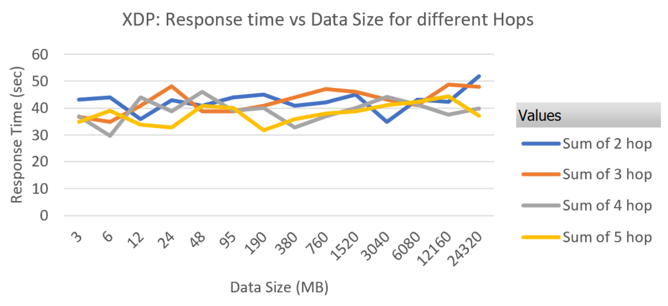


Figure14: XDP: Response time vs data size for different hop

3.2 Analysis and Hypothesis

ANALYSIS

- Looking at section 3.1 results, we clearly see that socat port forwarding worked far better than the XDP filter `-> unexpected`
- For the direct connection filter, as the data size increased, the response time increased `->` **`expected`**
- Across all hop systems (2, 3, 4, 5), **`socats`** `port forwarded system` response time increases gradually with increase in data size `->` **`expected`**
- Across all hop systems (2, 3, 4, 5), **`XDP`** `port forwarded system` response time does not increase with data size, but varies randomly, more or less at a similar average response time across all loads `-> unexpected`
- From **table 1**, for **`socat`**, we can see that as the # of hops increase, for the same file size, the response time increases `->` **`expected`**
- From **tabe 1** for **`XDP,`** we can see that as the # of hops increase, for the same file size, the response time randomly increases and decreses, which averages to ~35 seconds `-> unexpected`

HYPOTHESIS

- The goal of the project was to study whether filtering at the XDP layer, vs the traditional nftable filters would perform better in a port forwarding multi-hop system.
- We see clearly that the results with socat (that uses nftables) performs significantly better than the XDP filter.
- The experiments were started, keeping in mind that the localhost port forwarding system, as opposed to real world routing, would skew the results, though not by this much.
- There are a few reasons we are seeing these kinds of results, and are listed below.

1) Socat tools internal working is not known to us. Since we are using a loopback device, these packets never really reach the network interface card (NIC). No interrupts are raised, and the packets simply pass from one port to the next. The packets don't even need to go further down than the IP layer (***figure 7***).
2) On the other hand, even though for the XDP filter case, packets also don't reach the NIC, the XDP filter is hooked up to the loopback device. Everytime a packet is received at the loopback device, an interrupt is raised and the BPF program is triggered to do its work. With megabytes of data being sent, TCP would be sending this over multiple smaller packets, and sequentially, as this is a reliable protocol. As a result, the filter is going to be invoked numerous times, since the packets can be at multiple stages in the route, i.e. different ports. With so many interrupts getting raised, the scheduler is getting clogged and the number of context switches is likely slowing down the entire transfer process. Remember, as mentioned earlier, the filter, though for a particular port, affects the entire system, since the interrupt is raised.
3) In a real world scenario, where there would be multiple routers, there would be a maximum of 1 port on these routers, and packets would be redirected to their destination. Unlike in our localhost system, we would then have a single filter per router for the specific port on the router, and the system would not get clogged by interrupts, like it is happening on the localhost system.
4) But we may still wonder why the response time does not increase with increases in data size. Based on the hypothesis of interrupts clogging the system (which we do see in the bpf trace logs), the response times are amortized by the number of context switches. Hidden behind those numbers, we may actually see an increasing response time w.r.t loads.

- This would require setting up an infrastructure of routers to test. We defer this work to the future.
- We can conclude that the XDP filter works as expected across multiple hops. Since this is a reliable TCP connection, we can also see that there was no loss in packets.

4. Challenges

- One of the primary challenges of implementing this project was finding good documentation online. Being a nascent field, though there has been some progress in the industry recently, there are only a handful of resources available for e-BPF, and the documentation is dispersed, making it difficult to glue concepts together.
- Understanding the Linux Kernel networking stack. This is a huge piece of the Linux kernel, and understanding it completely would span semesters. A basic understanding was required to deduce what kind of experiments to carry out.
- Understanding how the e-BPF programs work and communicate using BPF maps between user-space and kernel, as well as how and where the programs are hooked in the kernel stack. Another challenge was finding and understanding BPF code, which though written in C, is heavily loaded with macros specific to BPF that are not well documented.
- The infrastructure to run tests, as described in the literature, generally involves testing across the network, with varying processor cores, memories for varying bandwidths, and servers. However, my project was limited to using the localhost loopback device on my PC, on a VM, which has likely skewed results.
- Figuring out how iptables and nftables in the kernel work, to set up port forwarding for them, to experiment with.
- Design of experiments to collect data, and create an infrastructure to process the data for analysis.
- Porting the originally developed UDP protocol XDP filter to TCP protocol, as the server application developed was a HTTP server running TCP/IP protocol and tested with a tool called siege which gives performance statistics that only work with HTTP servers. The challenge here was that TCP, unlike UDP, establishes a reliable connection, and requires a handshake at each hop, unlike UDP, which is unreliable. So for the UDP filter, a simple condition to forward the packet across hops works well, but in TCP, if the destination is changed, but not the source, TCP returns to the original port instead of forwarding the packet. For this, the filter had to be modified, to handle the condition at every hop to re-assign both destination and source port. Another interesting challenge was that the starting port (i.e. siege), does not bind to a port, so there is no known condition in the filter that can be used

for the starting port. For this, a **BPF HASH element** used as a global variable was used to track the state of the starting port.

5. Conclusion and Future Scope of Work

From the experimental data, we see that the XDP filter that is used to forward packets, performed worse than the socat packet forwarding chain that uses the traditional nftables for performing packet forwarding. We believe that a reason for this is that all experiments were being performed on a localhost machine. As we do not have visibility into the socat code, it is possible that when on the loopback device, socat does not pass the data down the networking stack at all, but keeps it in the application layer itself. For the XDP filter, there could be multiple reasons for it performing worse than the nftables. One thing observed is that a large number of interrupts are generated when the packet is received at the loopback device, which is likely clogging the scheduler with multiple context switches. In a real world application, we would have a single port on the system, but here we have 5 ports. Packets flow sequentially, and we are transferring megabytes worth of data, which is broken up into many packets, that could be at any stage in the forwarding chain, raising an interrupt. We also see multiple interrupt messages popping up in the BPF trace, which strengthens our belief that this could be the issue. Having a single filter per port, on separate machines, may really start to see the benefits of XDP over nftables, but we defer this to future work.

We can conclude that the XDP filter can be used successfully for forwarding packets, as well as dropping packets, etc, and that this field of e-BPF is an exciting one with lots of opportunity to explore.

In addition to experimenting across a real network in the future, comparisons can be drawn to forwarding UDP packets using a XDP filter, exploring load balancing with XDP, speculative execution by accessing packet data before it is processed by the kernel networking stack, scheduling them on particular CPUs. There are more opportunities to explore with e-BPF and lots of work being carried out in the industry and academia for the same.

6. Acknowledgements

To say that I learnt a lot in this project, would be an understatement. Entering the project, I only had theoretical knowledge of the Operating System, and was going to pursue the OS course offered by Columbia (a very practical based course). As a result, I was hesitant that I would be able to achieve results by the end of the semester. I also entered the project without any prior knowledge of the linux networking stack and e-BPF. The learning curve was steep, and I still believe that I have much to learn in this exciting field, as it is vast, and up and coming in the industry, with companies such as Facebook and Netflix adopting the technology for their applications.

I would like to thank Professor Gail Kaiser for the opportunity to pursue this research coursework credit towards my degree requirements, and thank Anthony Saieva for his guidance provided.

7. References

[1] https://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html

[2] https://elixir.bootlin.com/linux/latest/source/include/linux/skbuff.h#L751

[3] https://people.cs.clemson.edu/~westall/853/notes/skbuff.pdf

[4] https://liuhangbin.netlify.app/post/ebpf-and-xdp/

[5]https://archive.fosdem.org/2019/schedule/event/xdp_overview_and_update/attachments/slides/2877/export/events/attachments/xdp_overview_and_update/slides/2877/xdp_building_block.pdf

[6] https://ebpf.io/what-is-ebpf

[7] https://docs.cilium.io/en/stable/bpf/

[8] https://nakryiko.com/posts/libbpf-bootstrap/

[9] https://github.com/iovisor/bcc

[10] https://nakryiko.com/posts/bpf-tips-printk/

[11] https://ops.tips/blog/how-linux-creates-sockets/

[12] https://dl.acm.org/doi/pdf/10.1145/3341301.3359628

[13] https://www.tcpdump.org/

[14] https://www.kernel.org/doc/html/v5.12/networking/filter.html

[15] https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/

[16] https://tldp.org/HOWTO/Divert-Sockets-mini-HOWTO-6.html

[17] https://cdn.shopify.com/s/files/1/0177/9886/files/phv2017-gbertin.pdf

[18] https://www.cs.dartmouth.edu/~sergey/netreads/path-of-packet/Lab9_modified.pdf

[19] https://www.hostinger.com/tutorials/iptables-tutorial