Aalto University
School of Science
Master's Programme in Security and Cloud Computing

Felix Maurer

Investigating Causes of Jitter in Container Networking

Master's Thesis
Stockholm, July 30, 2021

Supervisors: Prof. Marco Chiesa, KTH Royal Institute of Technology
Prof. Mario Di Francesco, Aalto University
Advisors: Simone Ferlin-Reiter Ph.D., Ericsson AB
Tom Barbette Ph.D., KTH Royal Institute of Technology

Aalto University
School of Science
Master's Programme in Security and Cloud Computing

ABSTRACT OF
MASTER'S THESIS

Author: Felix Maurer

Title:
Investigating Causes of Jitter in Container Networking

Date: July 30, 2021 **Pages:** vii + 68

Major: Security and Cloud Computing **Code:** SCI3084

Supervisors: Prof. Marco Chiesa
Prof. Mario Di Francesco

Advisors: Simone Ferlin-Reiter Ph.D.
Tom Barbette Ph.D.

Clustered container infrastructures are increasingly popular for deploying applications. The networking in these clusters is provided by specialized container networking solutions that often lead to complex network configurations on the nodes hosting the containers. Thereby, they can have a significant impact on the performance of the applications hosted in the cluster. While the throughput that can be achieved by the container networking solutions is regularly studied, the latency and subsequently jitter introduced by them is often underreported.

This thesis investigates the latency and jitter introduced by the packet processing in the Linux kernel using different container networking solutions. This requires very detailed data about the processing of packets, which existing tracing tools for Linux fail to provide. Therefore, a custom tracing application is developed using eBPF that focuses on the flow of packets through the kernel. The application is evaluated and then used to compare the latency and jitter behavior of commonly used container networking solutions.

The results show that the choice of transport protocols for real-time applications has a significant impact on the latency introduced by the kernel irrespective of the container networking. Also, some container networking solutions fall short of providing their proclaimed benefits in their default configurations. This highlights the need for performance evaluation in environments representative of the production setting and the need for tuning the configuration of container networking solutions and system resources to match the requirements of real-time use cases. The data also show that there is a need for more lightweight tracing technologies for packet processing.

Keywords: Container Networking; Tracing; eBPF

Language: English

Acknowledgements

My sincere thanks to everyone who supervised me during this thesis, Marco Chiesa, Mario Di Francesco, Simone Ferlin-Reiter and Tom Barbette, for their helpful suggestions and ideas on my work, the inspiring discussions, and the feedback to this thesis. I also thank my opponent, Max Crone, for challenging my work and providing valuable feedback; the Linux kernel developers Toke, Jesper, Jiri, and Arnaldo for answering my questions about the networking stack and BPF; and Valentin, Simon, Carolin, and Wolfgang for suggesting improvements to this thesis document. Finally I would like to thank my friends and family for supporting me, either directly or indirectly, during this thesis.

Stockholm, July 30, 2021

Felix Maurer

Contents

Glossary

BPF Originally appeared as a abbreviation for Berkeley Packet Filter. Nowadays it is used as technology name for extended BPF (eBPF). In this thesis, it always refers to eBPF.

conntrack A netfilter module implementing connection tracking in the Linux networking stack.

DWARF A file format to store debugging information in the object file of an application.

iptables A netfilter module that can filter, redirect and modify IP traffic.

netfilter The netfilter framework in the Linux kernel.

POSIX A set of standards for interfaces to ensure compatibility between different operating systems. Many Unix systems implement them at least partially.

sk_buff The data structure in the Linux kernel that manages packet data. It is implemented as the struct `sk_buff` in the kernel source code.

tc A subsystem of the Linux networking stack to perform operations for quality of service of network communication, including traffic shaping, packet scheduling, and dropping packets.

Acronyms

ABI Application Binary Interface.

API Application Programming Interface.

BPF Berkeley Packet Filter.

BTF BPF Type Format.

cBPF classic BPF.

CDF Cummulative Distribution Function.

CNI Container Networking Interface.

CO-RE Compile Once - Run Everywhere.

CPU Central Processing Unit.

DMA Direct Memory Access.

eBPF extended BPF.

FIB Forwarding Information Base.

GRO Generic Receive Offload.

JIT Just-In-Time compilation.

NAPI New API.

NAT Network Address Translation.

NIC Network Interface Controller.

PDV Packet Delay Variation.

PMU Performance Monitoring Unit.

RPS Receive Packet Steering.

RSS Receive Side Scaling.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

VM Virtual Machine.

XDP eXpress Data Path.

XPS Transmit Packet Steering.

Conventions

Throughout the thesis, the first mentions of uncommon names are set in *italic font*. Subsequent mentions of the same names are set in normal font. Identifiers from source code are set in `monospaced font` and the surrounding text explains from which code base it was taken. The only exception to this rule is the data structure `sk_buff` from the Linux kernel. It is frequently used throughout this thesis and therefore written as sk_buff for the sake of readability. System commands are always set in `monospaced font`.

Chapter 1

Introduction

Software architecture and the way software is deployed is constantly evolving. One recent change was the shift towards cloud computing and especially containerization of workloads. While this at first appears to be a change in deployment, it also affected the software architecture towards *cloud-native* applications. As this approach promises benefits for software development and infrastructure management, it is relatively common nowadays [1]. Therefore, many new applications are built in a cloud-native way and even existing applications are brought into cloud infrastructures, often starting out with containerization, to benefit at least from the management advantages.

At the same time, the number of applications requiring low or ultra low latency is growing and is projected to grow further [2]. The applications range from entertainment purposes, such as cloud gaming or collaborative content creation, to highly critical use cases, including industry automation or even remote healthcare. The GSM Association [3] argues that especially virtual and augmented reality applications will drive the need for low latency mobile data connections. Most often when the latency needs to be low, also the *jitter*, i.e., the variance of the latency, is required to be low. High jitter in multimedia applications often indirectly leads to higher latency as well, because buffer sizes are increased to be able to present a continuous stream to users.

When these two developments meet, a new set of challenges arises. Cloud environments often improve utilization by sharing resources between different workloads. But resource sharing naturally adds some unpredictability to the performance, which is opposed to maintaining a constant latency, i.e., low jitter. Also, in efforts to create a uniform environment to deploy applications, many layers of abstraction are introduced that all add their penalty to the observed latency and jitter. Therefore, the applicability of cloud infrastructures to low latency applications is an area of ongoing research.

This work focuses on a part of this issue. In particular, it considers networking aspects of *Kubernetes*, a container orchestration system that is commonly used in industry. Container networking is an interesting aspect because it often uses many different features of the networking stack of an operating system and creates rather complex configurations to provide networking to containers. The more complex the configurations get and the more different features are involved, the more places can potentially introduce jitter.

1.1 Problem Statement

To allow latency-sensitive applications to be built on systems such as Kubernetes, it is important to understand how the container networking impacts the latency and jitter of network connections of the applications. As the container networking is provided by different software systems, it is necessary to investigate how their impact on latency and jitter differs. In order to optimize these systems in the future, the causes of latency and jitter should be known.

1.2 Goals

This thesis collects detailed data about the latency of packets in the Linux networking stack when the machine is used as a Kubernetes node. The data is collected for different container networking providers. An analysis of the data compares the different providers to show how the different implementations impact the latency and jitter but also to highlight common behavior. The data and analysis can help to select a container networking solution for a specific use case and to adapt the settings of the system components to better suit the needs of the applications. It can also support the improvement of container networking implementations to achieve a better performance for systems that require low latency and low jitter.

1.3 Methodology

The collection of detailed latency data from the networking stack can not be performed with the existing tracing tools in Linux. Therefore, a specialized tracing application for the stack latencies is developed using eBPF. This is a new technology in the Linux kernel that can be used to implement specialized applications for tracing the kernel. The results are based on quantitative

analysis of the obtained data. All measurements are performed in a testbed environment with resources dedicated to the experiments. Additionally, the overhead of the tracing application is estimated based on measurements in order to give an intuition of the impact that the execution of the application in a production environment would have.

1.4 Limitations

The thesis focuses on the latency introduced by the networking stack of the Linux kernel when used for container networking. In particular, kernel bypass technologies and userspace networking, such as DPDK, are not subject of this thesis. These technologies are rarely used to provide full container networking capabilities as required by Kubernetes. Instead, they may be configured in addition to the normal container networking. In any way, they are usually dependent on the underlying hardware and therefore break the abstractions that cloud-native applications build upon.

The measurements are performed on a single host because they trace the local stack. Therefore, the experiments are only performed with a single-node Kubernetes cluster. For additional insights into the networking within a multi-node cluster, the analysis could aggregate the results from multiple nodes. The node used for the experiments is a bare-metal machine and there is no virtualization layer in any of the measurements.

In the scope of this thesis, only a small number of container networking solutions could be tested. The selected ones are commonly used and differ in their implementation. If the results of this work should help to select a container networking solution from candidates for a production environment, it is anyways advisable to repeat the comparison with all candidates in a setting that is more representative for that environment.

1.5 Sustainability and Ethics

The direct impact of this work on sustainability issues is limited. However, the results may be used in the future to optimize container networking towards low-latency applications. These applications could in turn also profit from the benefits of cloud infrastructures, including reduced costs and higher energy efficiency resulting from a better utilization of computing resources. Also, if latency-sensitive applications can run on cloud infrastructures, the need for additional infrastructure for these applications decreases which can improve cost efficiency of infrastructure management and development.

The measurements for this work are performed in a testbed using dedicated resources and synthetic network traffic. Therefore, the measurements can not collect any private information that would require especially cautious handling.

1.6 Structure of the Thesis

After this introduction and definition to the topic, the rest of this thesis is structured as follows. Chapter 2 covers the necessary background information for the thesis. It describes the Linux networking stack, Kubernetes and container networking, different tracing tools in Linux, and the technology eBPF and summarizes the related work. Chapter 3 shows the design ideas behind the implemented tool, the development process, and the key features of the actual implementation. Chapter 4 estimates the overhead of the implemented tool and presents the comparison of the container networking solutions. Chapter 5 provides concluding remarks.

Chapter 2

Background

This chapter provides the technical background for the thesis. It starts with an overview of the Linux kernel networking stack, because Kubernetes clusters usually consist of Linux nodes. Then, Kubernetes clusters in general and their container networking are described. Next, different tracing tools for the Linux kernel are presented to show why they can not be used to implement the tracing application. Instead, it is based on the technology eBPF, which is introduced afterwards. The chapter closes with a summary of the related work.

2.1 The Linux Networking Stack

A particular strength of the Linux kernel is its full-featured and flexible networking stack. It supports many different protocols, including all that are commonly used in the Internet, and can perform different network functions including routers, firewalls, and other middleboxes. Giving an overview of all networking features is not possible within the scope of a thesis. Therefore, this section is limited to the parts of the stack that will be relevant in the course of this thesis. For further information, we refer to the available literature, for example the detailed, technical description in *Linux Kernel Networking* by Rosen [4].

The networking stack in the kernel handles functionality up to the transport layer. In contrast, the application layer is usually handled by userspace applications. The interface between applications and the networking stack on Linux is the POSIX socket Application Programming Interface (API) [4]. This API provides an uniform interface to the different transport layer protocols supported by the kernel. Despite the differences of the protocols, the interface is pretty consistent and allows to use the same functions irrespective

of the underlying protocol. In general, the application can use sockets to send data to other endpoints or to receive incoming data. Besides that, there are functions to start outgoing communication for client applications and functions to accept incoming communication for server applications.

Within the kernel, the data of network packets is kept within a data structure called `sk_buff`. An sk_buff maintains the offsets to the beginnings of the headers for the different protocols in the packet data in memory. It also maintains other metadata of the packet that are useful throughout the stack. As sk_buffs are used everywhere in the stack, they also serve as a unified interface for packet handling within the kernel. While they are often described as a single packet, sk_buffs do not necessarily represent one packet as it is transferred over the wire. Instead, it can also combine the data of multiple packets, often due to a technique called *segmentation offloading* where the Network Interface Controller (NIC) handles the segmentation and aggregation of packets that belong to one flow to enable the networking stack to process them at once [5].

The rest of this section will focus on different aspects of the kernel networking stack. It starts with an overview of how the packets enter the stack from physical NICs and is followed by a description of the three main paths a packet can take through the stack.

2.1.1 NAPI

The Linux kernel contains drivers for many different NICs. They are responsible for interacting with the hardware to receive incoming packets and hand them to the networking stack, and to take packets from the stack and send them out of the interface. Generally speaking, there are two ways in which a driver for a NIC can operate. It can either react to interrupts generated by the NIC for an incoming packet or it can constantly poll the network interface for new data from the network [6]. While basing a driver on interrupts achieves low latency, it can lead to significant performance issues at high load, because of the high performance overhead of handling an interrupt. On the other hand, constant polling wastes CPU resources and can increase the latency for handling incoming data, while being able to provide high throughput.

The current prevalent framework for implementing drivers for NICs in Linux is called New API (NAPI). It has been introduced by Salim et al. [7] in 2001, but the core ideas go back to previous work by Mogul et al. [6] from 1997. The main goal of that work is to prevent livelock when receiving network packets in an interrupt driven kernel. Livelock describes a situation where a kernel can not perform useful work any more because a resource, most often CPU time, is fully exhausted by processing the interrupts from

the NIC for incoming network packets. To address this issue, Mogul et al. [6] suggest a combination of interrupts and polling, where the first interrupt triggers subsequent polling. They also suggest trying to drop packets, which can not be handled due to a overload situation, as early as possible. This reduces the time spent on processing packets that would be dropped anyways later on. Salim et al. [7] implement these ideas in the Linux kernel under the name NAPI and extend them to reduce the introduction of packet reordering on multi-core machines.

With NAPI in Linux, when a driver of a NIC receives an interrupt for an incoming packet, it does not handle the packet immediately in the interrupt handler [8]. Instead, it disables further interrupts for new packets from the device and instructs the kernel to start polling on the device shortly. The polling function, which is part of the driver as well, is then called and expected to pass all new packets that arrived at the network interface to the networking stack. This might be just the single packet from the original interrupt, but may also entail additional packets that arrived in the meantime. To prevent the polling from going on for to long, there is a limit in place for how many packets the driver should pass to the stack. If the limit is reached and there are still further packets available, the polling function hands control back to the kernel and indicates that it expects to be run again. If there are no further packets available from the device, the polling function indicates this to the kernel and activates the interrupts from the device again. This process clearly achieves a dynamic combination of interrupt-driven and polling-based drivers, as suggested in the scientific work.

The NAPI design also achieves the goals of early dropping of packets and reduced reordering of packets [7]. This happens due to the way NICs handle the incoming packets: the packets are placed in a Direct Memory Access (DMA) ring buffer. This means that the device writes the data of received packets directly to a ring data structure in the main memory as long as there is space left in the data structure. If the packets arrive faster than the kernel can handle them, the ring fills up. When the ring is full, the NIC silently drops further incoming packets without disturbing the kernel. This means, that packets that can not be processed due to an overload situation do not reach the kernel at all. As the ring is a linear data structure, the packets are placed sequentially within it. This sequential order can be used later in the stack to reduce the reordering of packets [7, 8]. To speed up the processing of the received packets, the DMA ring serves as the memory backing pre-allocated sk_buffs. Thereby, the incoming packets can directly be used by the stack without the need to allocate an sk_buff first. When the driver has completed one polling cycle, a new sk_buff in the DMA ring gets allocated for each received packet to store the next incoming packets.

However, NAPI also has its limitations. Especially in situations with a medium amount of packets, the polling loop is started because a packet arrives and finishes just before the next one arrives [9]. The processing of the next packet will then again be triggered by an interrupt, which is costly. Due to the high number of interrupts, the load on the machine is already very high at a packet rate that is low compared to the maximum capacity of the host.

2.1.2 netfilter

The *netfilter* framework is a very powerful subsystem of the Linux networking stack. It provides various hooks throughout the packet processing path of the kernel [4]. At the hooks, various actions can be performed with the packet, including rewriting addresses or ports, dropping packets, and logging them. These functionalities are provided by kernel modules, which can register callbacks to be executed at the hooks. The most common usage of the netfilter infrastructure is probably *iptables*, which allows system administrators to configure rules for filtering and processing IP traffic.

Arguably, examining just a single packet may often be not enough for deciding which action to take upon it [10]. Therefore, the connection tracking system *conntrack* has been built, utilizing netfilter to support stateful filtering and similar operations. It identifies connections by their 5-tuple (source address and port, destination address and port, and protocol) and stores the current state of the connection. The stored state does not resemble, for example, a full TCP state machine, but rather distinguishes mainly if there has been just one-way, or already two-way traffic for a flow.

A particular strength of netfilter is that the modules can also be combined. Thus, it is possible to base the decisions in iptables rules on the current state of the flow in conntrack. This makes it possible to express complex network functions, including stateful firewalls or load balancers, using netfilter modules.

2.1.3 Incoming Packets

The path that incoming packets take through the stack can be seen on the left side of Figure 2.1 in green. New packets enter the stack as sk_buffs that the NIC driver passes to the stack. The first operation relevant in the scope of this thesis is the processing of packets by traffic control (tc) ingress hooks. This can be used to perform operations related to quality of service [11]. Next, the callbacks attached to the `NF_INET_PRE_ROUTING` hook are executed, including running conntrack for the incoming packet [4]. They may change

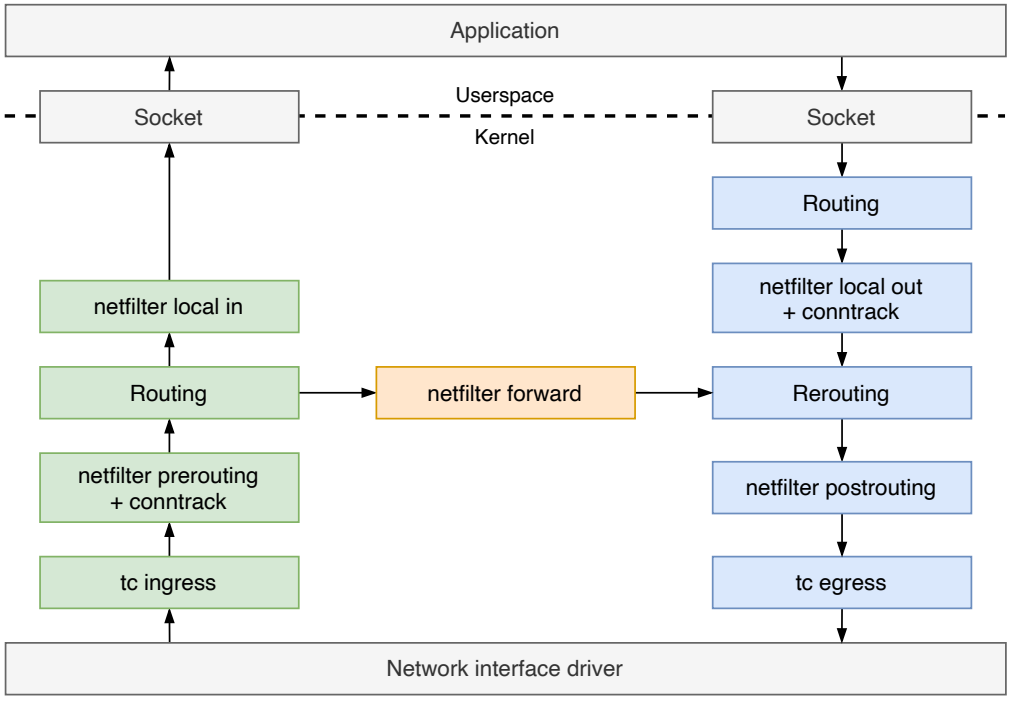


Figure 2.1: Packet flow in the Linux networking stack, based on Rosen [4]

the packets when they, e.g., perform Network Address Translation (NAT). After that, the Forwarding Information Base (FIB) of the system is consulted to decide if the packet is destined for the local host, or should be forwarded. Packets for the local host go through the callbacks for the `NF_INET_LOCAL_IN` hook. Afterwards, the socket that the packets should arrive at is looked up and if such a socket exists, the sk_buffs are added to the receive queue of this socket. Then, the data is ready to be received by the userspace application. If the packets are not for the local host, they instead follow the forwarding path described in the next section.

2.1.4 Forwarding Packets

The forwarding path is simple and shown in the center of Figure 2.1 in orange. After the routing decision in the input path was made, the packets are processed by the callbacks for the `NF_INET_FORWARD` hook [4]. Then, the packets join the output path at a particular step that is described in the next section.

2.1.5 Outgoing Packets

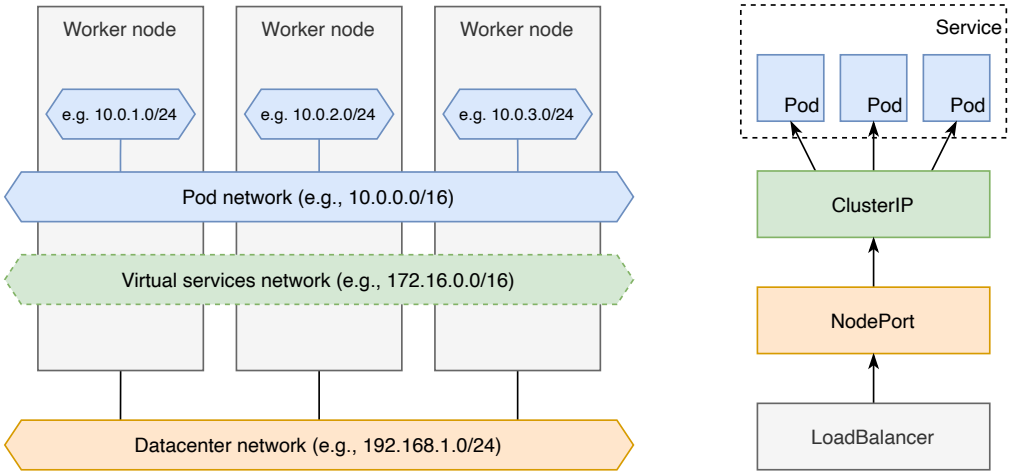
The path of outgoing packets is shown in Figure 2.1 on the right in blue. Outgoing packets can be generated by userspace applications [4]. When an application writes to a socket, an sk_buff is allocated for the data and passed to the networking stack. For these sk_buffs, first a lookup in the FIB is performed to find out if the kernel knows how to send the packet to its destination. If this lookup is successful, the packets are processed by the callbacks for the `NF_INET_LOCAL_OUT` hook, which includes running conntrack. This step may change parts of the packet, including the destination address when performing NAT. Therefore, another routing decision is made now to decide on the network interface through which the packet leaves the system. From this step onward, the packets that are forwarded by the kernel follow the same steps.

Next, the packets are processed by the callbacks for the `NF_INET_POST_ROUTING` hook [4]. Then they are handled by the tc egress hooks. After that, they are finally handed to the driver of the interface through which the packet is supposed to leave the system.

2.2 Kubernetes and Container Networking

Kubernetes is a software system to manage containerized applications [12]. The desired state of the deployed applications is defined in a declarative way and the system constantly compares the desired state to the actual state to take action if needed. Kubernetes has a focus on being scalable to big application deployments. For example, it usually distributes the running applications across a cluster of worker nodes, can start multiple instances of an application, and load balance the traffic towards them.

The containers running the applications are organized in *pods* [13]. Each pod can contain one or more containers. The containers and pods are isolated from the system and other pods using Linux namespaces, cgroups, and potentially other technologies. From a networking perspective, it is important that all containers in a pod share the same network namespace. Each network namespace appears to have its own networking stack with its own set of interfaces, a routing table, and all the other features of the Linux networking stack. Within one network namespace, the applications can also communicate with each other over localhost.



(a) The different networks used in a cluster

(b) The different layers for exposing a service

Figure 2.2: Networking in a Kubernetes cluster

2.2.1 Kubernetes Networking

In general, there are three different networks in a Kubernetes cluster. They are shown in Figure 2.2a. First, there is the datacenter network. Each of the nodes in the cluster has an IP address from this network to be able to communicate with other nodes. Then there is the pod network. The pod network is defined when the cluster is setup and each node in the cluster gets assigned a subnet of the pod network. Finally, there is the virtual services network. Services running in the cluster can get assigned one IP address from this network. They are described in more detail later in this section.

Kubernetes puts up a few rules for how the networking within the cluster should work [14]. Each of Tthe pods gets assigned one IP address from the pod subnet assigned to the node the pod is running on. Kubernetes requires that a pod can reach all other pods in the cluster without any NAT happening in between. Also, software running in the host network namespace must be able to reach the pods running on the same node. On Linux, this rule is extended so that software in the host network namespace must also be able to reach all other pods in the cluster.

In addition, Kubernetes also defines how pods can be made reachable from outside the cluster [15]. This is shown in Figure 2.2b. There are *Services* that group one or more pods which are usually running instances of the same application. A Service can have a *ClusterIP* assigned to it which is taken from the virtual services network. The ClusterIP is accessible from all

pods in the cluster and the traffic to it is load-balanced to the pods of the service. To make a ClusterIP accessible from outside the cluster, Kubernetes offers *NodePorts*. A NodePort opens a port on each node of the cluster and forwards the traffic arriving at this port to the underlying ClusterIP. However, NodePorts are still not optimal for users to access because they require the users to memorize the port number. Also, the nodes of the cluster might just be connected to a private network that is not accessible by the users. Therefore, Kubernetes allows to configure *LoadBalancers* that distribute the traffic to the different NodePorts. The LoadBalancers are usually external to the cluster, e.g., operated by the cloud provider where the Kubernetes cluster is hosted.

While Kubernetes specifies all this, it does not implement the networking itself. Instead, this is delegated to other software systems that follow the Container Networking Interface (CNI) specification [16]. There are many of these CNI plugins available. The installation of one of them is required for a working Kubernetes cluster. To achieve a network configuration that follows the rules imposed by Kubernetes, the CNI plugins make use of various elements of the Linux networking stack, such as iptables to configure NAT or different tunnel protocols to transport the inter-node traffic. They might also use additional software to provide additional functionality, such as routing daemons to integrate with the datacenter network or userspace proxies to implement network policies on higher layers. The software kube-proxy is often used by CNI plugins to perform the load balancing from ClusterIPs to the pods of a service. While there are other options, kube-proxy usually configures iptables to perform the load balancing.

2.2.2 CNIs in This Thesis

This thesis considers the CNI plugins Calico and Cilium for the comparison of their packet processing latency and jitter. Both are commonly used systems with many different features, often exceeding the requirements of Kubernetes. In their default configurations they differ in how they configure the networking: while Calico uses iptables for most of the configuration [17], Cilium aims to implement many features in BPF and bypass the networking stack in these cases to achieve better performance [18]. However, Calico also provides a implementation of its data plane in BPF [17]. Both BPF implementations are capable of replacing kube-proxy and its iptables rules completely [17, 19]. Calico and Cilium both use virtual ethernet pairs to transfer packets between network namespaces. A virtual ethernet pair can be imagined as a virtual ethernet cable. It is created when a pod is created and one side of the pair is

moved into the network namespace of the pod while the other side remains in the host namespace.

Of course, the two CNI plugins have many other features that are not covered here and are not considered in this thesis, including different mechanisms to transfer traffic between nodes, encryption of this intra-cluster traffic, integration into the data center network, network observability, or network policies. For these topics, the reader is referred to the extensive documentation of both projects.

2.3 Tracing in the Linux Kernel

Tracing is a way to obtain detailed data about a software system by providing a framework for efficient and configurable logging of events [20]. The events are usually timestamped and concern the execution flow of the system itself instead of more high-level business metrics. As such, tracing has proven itself to be a very helpful tool for developers to debug complex systems and identify performance issues. Therefore, there are several tracing tools available for the Linux kernel [20]. Some of them are directly integrated into the kernel, while others are developed outside the kernel source tree and can be added to a system as a kernel module. The rest of this section describes the capabilities of the different tracing systems that are directly integrated into the kernel, *ftrace* and *perf*, while the next section presents eBPF in detail.

The so called *tracepoints* are an important feature for tracing the Linux kernel. They are essentially hooks in the Linux kernel code that are placed there at development time [21]. At runtime, user defined programs, called probes, can be attached to tracepoints. If there is no probe attached, a tracepoint adds very little overhead, i.e., just one check of a branch condition. The probes also get passed some pre-defined parameters, which describe the context of the current invocation of the function. For example, in networking-related tracepoints, this context might include a pointer to the sk_buff that is being processed. In addition to tracing, the tracepoints may also be used for performance accounting.

2.3.1 ftrace

ftrace is a tracing system that focuses on the internal behavior of the kernel [20]. It can trace events from two main sources: kernel functions and tracepoints. For kernel functions, it can generate events when a function is entered and exited [22]. This allows, for example, to build call graphs for the kernel and record the time spent in each individual function. When it works on

tracepoints, ftrace just generates an event when it encounters tracepoints in the kernel.

ftrace it not available as a single program on a Linux system [20]. Instead, it is integrated into the kernel and controlled using the *debugfs* filesystem. The files, that this virtual filesystem presents, can be used to enable and disable the tracing, filter the events, and select which information should be collected. As interacting with ftrace through the filesystem is rather inconvenient, there are command line frontends, such as `trace-cmd`, that can be installed in most Linux distributions [23].

2.3.2 perf

perf is usually known to be a tool for sampling and profiling of Linux applications [20]. For this use case, it uses the `perf_events` subsystem to gather information about the performance of applications. Most notably, it can interact with the Performance Monitoring Units (PMUs) of CPUs to collect data on various low-level metrics, including the number of processed instructions, the busy and stalled cycles of the processor, cache hits and misses, and missed branch predictions, and thereby helps with fine-grained, down to the line of code or even instruction, performance analysis of applications. But perf can also interact with tracepoints in the kernel [20]. This allows it to collect more high-level data on how an application uses different Linux kernel subsystems. An example for this use case would be to check if an application uses the `read()` system call efficiently or if it issues a high number of small reads.

2.4 The Extended Berkeley Packet Filter

The original Berkeley Packet Filter (BPF) was introduced in the early 1990s to improve the performance of packet captures in the userspace [24]. Before capturing packets, a userspace application provides a set of rules to the kernel. The kernel evaluates these rules for each packet to decide if the packet should be copied to the userspace. This way, the performance-wisely expensive copy over the kernel/userspace boundary is limited to the packets that are of interest to the application. BPF provided a new way to write the rules and evaluate them in the kernel. They are provided as program consisting of BPF instructions. The kernel executes these instructions in a register-based pseudo machine. Combined with additional optimizations, this improved the performance of the in-kernel filtering by up to 100 times. The most notable use of BPF is probably *libpcap* that underlies `tcpdump` and generates BPF code from the traffic filters given by users on the commandline [25].

BPF has later been substantially reworked [26]. The reworked version was released with kernel version 3.18 in December 2014 and publicly called eBPF because it contained so many improvements. Therefore, the original BPF design is sometimes referred to as classic BPF (cBPF) when it is required to highlight the differences between the versions. The kernel development community continued to call the new version BPF but does consider it a name of a technology now and no longer an abbreviation for Berkeley Packet Filter. Throughout this thesis, the term BPF always refers to the current implementation, eBPF.

eBPF includes an updated instruction set and changes to the pseudo machine used to execute them [27]. These changes were mainly introduced to align the BPF instruction set more closely with current native hardware instruction sets. This potentially improves the performance, simplifies the Just-In-Time compilation (JIT), and allows for direct interaction with kernel functions. These low-level changes allowed to introduce additional features that made BPF more versatile, including maps to support stateful processing and a library of in-kernel helper calls [28]. Thereby, it evolved into a flexible technology to add functionality in different places in the Linux kernel and is sometimes considered an "universal in-kernel virtual machine" [29]. The following sections describe the capabilities that BPF has at the moment.

2.4.1 Programs

Usually, BPF programs are written in a limited C dialect and compiled to BPF bytecode by LLVM [26]. The most important limitation of the programs is that their complexity is limited by the verifier in the kernel. Therefore, control structures can not be arbitrarily deeply nested. Programs may only contain bounded loops, i.e., the loops must have a maximum number of iterations that is known at compile time. Therefore, the code shown in Listing 1 is a common pattern to implement loops. Up until kernel version 4.16 and LLVM version 6.0, a BPF function could not directly call another one. Instead, all function calls needed to get inlined, which leads to increased size of the bytecode. The only exception to this were tail calls, by which the end of a BPF program trigger the start of another one. In contrast to an ordinary function call, the execution flow never returns to the calling function in case of a tail call.

The programs can not directly access arbitrary kernel memory [26]. Direct access is only possible to memory in the stack of the program, in maps, or of the network packet the program is working with. Even these accesses need to be checkable by the verifier, which sometimes makes it necessary to add explicit bound checks before accessing the memory. Programs can only read

```
#define MAX_ITERATIONS 4096
for (i = 0; i < iterations && i < MAX_ITERATIONS; i++) {
        // do work
}
```

Listing 1: Example for a bounded loop that can be used in BPF

kernel memory using helper functions such as `bpf_probe_read` but they are not allowed write to the kernel memory to maintain the integrity of the kernel.

In addition to the memory access functions, there are many more functions available to developers. They are all called *helpers* and documented in the man page `bpf-helpers(7)` [30]. Some helpers provide convenient access to commonly needed functionalities, such as the computation of packet checksums, while many others can be used to access kernel functionality that could not be expressed in pure BPF. Examples for this include redirecting packets, modifying socket options, reading perf event counters, performing lookups in the kernel routing table, and many more. Many of the helpers are focused on operations related to the networking stack, because packet processing is a common use case for BPF. However, the usage of some of the functions comes with a potential caveat: some helpers require the BPF program to be licensed under the GPL. To allow the verifier to check the license, the BPF object files contain a special section to denote the license, which can be set from the BPF source code.

The process of loading BPF programs into the kernel is depicted in Figure 2.3. First, the bytecode of the program is loaded into the kernel by the userspace application by issuing a `bpf()` system call. Then the verifier runs to ensure that the program is safe to run in the kernel [25]. This most importantly means, that the program always runs to completion, does not crash, and only performs safe memory accesses as described before. To verify the completion of all executions, the verifier parses the program into an directed acyclic graph. This step requires that the program does only contain bounded loops and performs no backward jumps. Afterwards, the verifier follows all possible execution paths of the program to make sure they all pose correct behavior using a state machine. This step limits the overall complexity of the BPF programs to prevent a state explosion caused by too many different paths through a program. If the program accesses packet data, the verifier makes sure that it performs a border check beforehand to prevent illegal access to kernel memory. Thereby, it is possible to allow writing to the packet data while making sure that the program can not write to arbitrary kernel memory.

After the program has been approved by the verifier, it may be processed

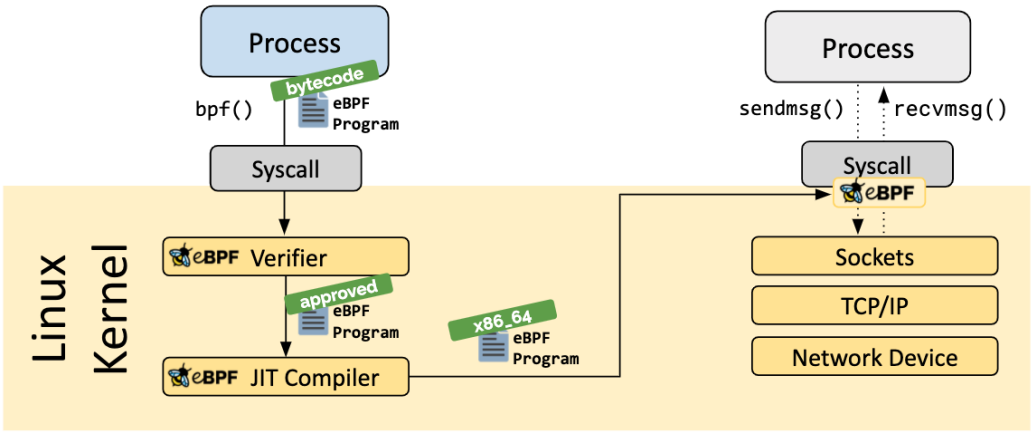


Figure 2.3: The process of loading a BPF program into the kernel[1]

by a JIT compiler [26]. The JIT compiler is supported on many architectures, including the most common Arm and x86 on 32-bit and 64-bit processors, and can be disabled at runtime. It translates the BPF instructions into native instructions for the processor. This enables high performance as if the code has been compiled for the hardware platform. If the JIT compiler is not available or disabled, the bytecode is executed by an interpreter in the kernel.

When this process is completed, the program is ready to be executed. The execution of BPF programs is event driven, i.e., a program is executed when a particular event occurs in the kernel. The different events that could trigger the execution of a program are called *hooks*. The hook that a program should be *attached* to is set when the program is loaded into the kernel. It is determined by the program type [25]. Often, the type is accompanied by additional parameters to choose the exact hook. Besides the hook, the program type also determines the context which is passed to the program and the helper functions it may use. The program types can be divided into two groups according to their use cases: networking and tracing.

The networking program types have in common that their context is a network packet, a data stream, or other networking related data. These contexts are usually defined for the BPF programs and pose a compatible API and Application Binary Interface (ABI) across kernel versions. The helper functions for these program types offer functionality often needed for network processing, such as redirecting packets and calculating header checksums, and sometimes allow the modification of the passed context or related data structures. Some of the networking related program types and their contexts are described below.

[1]From `https://ebpf.io/what-is-ebpf`, under Creative Commons BY 4.0

eXpress Data Path (XDP) Programs of type `BPF_PROG_TYPE_XDP` can run before a packet does even enter the networking stack [31]. They only have access to the raw packet, which has not yet been parsed by the stack. Therefore, XDP programs can process packets at very high throughput rates at low CPU utilization. To achieve this, the program is called directly by the driver, which thus needs to be modified. Such modifications have been introduced to many drivers, especially to the drivers for NICs with a high bandwidth. If the driver does not have XDP support, generic XDP can be used instead. However, this only runs the programs after the packet has been parsed by the stack and therefore does not offer the same high performance.

An XDP program is attached to a network interface and only handles incoming packets [31]. Within the programs, the content of the packets can be read and modified. At the end of the program, it can determine how the packet should be processed further. The options are to drop the packet, transmit the modified packet out of the same interface it arrived at, pass the packet to the normal networking stack, or to redirect the packet. The redirection can happen to a different interface for transmission, a different CPU for processing, or directly to an `AF_XDP` userspace socket.

Traffic Control Hooks Programs of type `BPF_PROG_TYPE_SCHED_CLS` can be attached to the ingress and egress tc hooks [25]. The ingress hook runs the program after it has been parsed into an sk_buff but still before most of the networking stack; the egress hook runs the program after most of the networking stack. As with XDP, the programs are able to read and write directly to the packet in memory and can the determine the further processing of the packet with the return code. Options for further processing are, among others, to accept the packet to the tc queue, to drop the packet, or to start the classification again, e.g., when the packet was modified.

In addition to these, there are program types that can be attached to various networking related hooks in the kernel, including the opening of sockets, handling of socket options, packet filtering from cBPF, and parsing of data streams to and from sockets.

The other group of program types deals with tracing. The context of these programs is usually a set of register values that refer to data structures in the kernel. Therefore, the BPF bytecode for tracing programs is not necessarily compatible between different kernel versions. A current approach to increase the compatibility is called Compile Once - Run Everywhere (CO-RE) and

described in Section 2.4.3. The tracing programs do not have access to many of the helpers related to networking, but to other helpers that allow more low-level interaction with the kernel, including, reading kernel or user memory, or triggering signals to tasks. The points to attach the programs fall into two categories.

Tracepoints As described before, there are tracepoints in the kernel for various events [21]. BPF programs can be attached to these tracepoints at runtime. The programs can read all the parameters that are defined as the context of the tracepoint. However, they can not modify the parameters or any other kernel memory.

Kernel functions It is also possible to attach BPF programs directly to the beginning or end of functions in the kernel [26]. This is more flexible because the programs are not limited to the pre-defined tracepoints. The programs can be attached using the older *k(ret)probe* mechanisms [32] or the newer *fentry/fexit* mechanisms [33]. While they both achieve very similar results, they differ in details. The k(ret)probes can be attached to all functions in the kernel, while fentry/fexit probes need BTF information to be available for the functions. Most notably, this is not available for kernel functions marked as static. The context of programs attached at the beginning of kernel functions, i.e., kprobes and fentry probes, contains all the parameters that were passed to the function. However, the context of programs at the end of a function, i.e., kretprobes and fexit probes, differs in that kretprobes can only access the return value of the function while fexit probes can additionally access the parameters that were initially passed to the function. Both types of programs can only read kernel memory and their respective contexts but not modify it.

In the context of tracing, the programs are often called probes to align with the terminology from other tracing tools. The technical differences between the different hooks in each of these categories are presented in Section 4.1.2.

2.4.2 Maps

Maps are one of the most notable additions that were introduced with eBPF. They are special data structures that can be accessed from BPF programs in the kernel and from userspace applications. In contrast to cBPF, maps make it possible to build stateful applications in eBPF that share state between the invocations of the programs. There are different types of maps available for different use cases. Most of them treat the keys and the values as binary data

and thereby allow the users to freely define their use. The maps always have a maximum number of entries that is set at load time. Thus, the required memory can already be reserved at initialization to prevent exhausting kernel memory at runtime by growing maps. The most important types of maps to store generic data are described below.

`BPF_MAP_TYPE_ARRAY` This map behaves like an array in other programming languages. The values in the map are accessed by their index and all indexes exist from the beginning of the program. This map also exists in a per-CPU version where the map is duplicated for each core. This allows programs to access the map without locking but does not synchronize the values across the different cores.

`BPF_MAP_TYPE_HASH` This map implements a key-value store with user-defined keys and values. The values can be looked up using the key but only after they have been inserted, i.e., there are no pre-allocated entries that can be accessed as in the arrays. When the maximum number of entries is reached, further insertions will fail. This map is also available in a per-CPU version.

`BPF_MAP_TYPE_LRU_HASH` This map type is very similar to the hash maps. But when the maximum number of entries is reached, further insertions do not fail. Instead, the least recently used entry is evicted and the new entry is inserted. This map is also available in a per-CPU version.

`BPF_MAP_TYPE_LPM_TRIE` This map implements a longest prefix matching trie. This is especially useful for networking use cases, such as routing tables.

`BPF_MAP_TYPE_PERF_EVENT_ARRAY` This map is an array where the values can only be file descriptors for perf events. These maps can be used for two different things. Firstly, they can be used to read perf events, e.g., the hardware counters for instructions or cycles, in a BPF program. In this case, the userspace application initializes the map with file descriptors to any perf event obtained from `perf_event_open`. Secondly, the map can also be used as a message queue from the BPF program to the userspace application. For this, the application needs to obtain the file descriptor with a special configuration. The BPF program can then write data of arbitrary size to the file descriptor in the map. The userspace application can then read each of these data blobs as one event.

`BPF_MAP_TYPE_RINGBUF` The new ring buffer map works similar to the second use case of the perf event array, i.e., as a message queue to the userspace application [34]. The main difference from a developer perspective is that the ring buffer guarantees to deliver the events in the order they were added to the buffer.

In addition, there are many map types available that store references to particular kernel data structures, e.g., `BPF_MAP_TYPE_SOCKMAP` stores references to sockets that can be used to redirect traffic to the referenced socket using a helper function, or `BPF_MAP_TYPE_PROG_ARRAY` stores references to other BPF programs to allow tail calls to them. These maps can not be used to store user-defined data and are therefore only suitable in their specific use cases.

2.4.3 CO-RE and BTF

Linux generally offers strong ABI guarantees for user APIs, especially for system calls [35]. This means that the layout of binary data, e.g., structs filled by system calls, is not changed across kernel versions, to maintain compatibility. If data structures need to be changed, existing fields can only be deprecated and new ones added at the end of the structure. As described previously in Section 2.4.1, the contexts of the networking BPF programs usually follow these ABI guarantees which makes the compiled bytecode compatible across different kernel versions. In contrast, tracing programs can read directly from internal kernel data structures. These data structures are not part of the userspace API and thus not guaranteed to be stable. Therefore, plain tracing programs can only work reliably on the single kernel version they were compiled for.

For some time, the solution to run the programs on different kernel versions was to recompile them on the system when they are about to be used. This approach is used for example by BCC, a framework that allows to use BPF from a Python interface [36]. However, this approach has the downside of requiring the compiler to be present on the target system [37]. The only BPF compiler available today is part of LLVM, which is a compiler suite for many different programming languages. Therefore, it requires significant storage space on the target system. Also, compiling the programs on the target system introduces additional load on that system.

To make compiled tracing programs more compatible across kernel versions, a technology called Compile Once - Run Everywhere (CO-RE) has been introduced [37]. It allows to store type information in the BPF programs that get resolved to the actual kernel data structures at load time of the program.

This is achieved by an integration of a new type format, the LLVM compiler for BPF, and the userspace library *libbpf*. The kernel itself only needs few changes to support CO-RE, which makes it available even in non-recent kernel versions.

The type format used for CO-RE is called BPF Type Format (BTF) [37]. It allows to embed type information into object files similar to what is achieved with DWARF debug symbols. The advantage over these debug symbols is, that BTF information are up to 100 times smaller than DWARF symbols for the same object file. This makes it feasible to always include BTF information even in production kernels. They are already included in recent releases of the common Linux distributions. The size reduction is achieved by a simplification of the format to focus on the type information and by a deduplication algorithm that aims to include each symbol just once and work efficiently across multiple compilation units. The algorithm has been created and thoroughly described by Nakryiko [38].

The Linux kernel is most often compiled using GCC. However, this compiler can not emit BTF information directly. Thus, in the kernel build process, `pahole` is used to generate the BTF information from DWARF symbols [39]. When compiling BPF programs, the compiler must emit *BTF relocations* [37]. This is natively supported by LLVM since version 8.0.0. The relocations keep, for example, the information about what field of a struct should be accessed by name, instead of compiling it down to a simple access with an offset from the beginning of the struct. The relocation information are translated at load time into accesses with the offsets that apply to the running kernel. This is done by mapping the names and types stored in the relocations to the BTF information of the kernel.

The loading of CO-RE BPF programs happens through libbpf [37]. This userspace library is part of the Linux kernel source code and performs the aforementioned translation of BTF relocations in the object file of a BPF program to simple memory accesses to generate an executable version of the bytecode that matches the running kernel. However, this is only a small subset of the functionality of libbpf. It actually supports the whole lifecycle of BPF programs, including the complete loading and attachment process of a program, the setup of maps described in the object file, the creation of the necessary data structures, and the actual interaction with the kernel using system calls. The library greatly simplifies the handling of BPF programs and is therefore also used for the implementation of the tracing application in this thesis.

The preceding section has outlined the different functionalities and versatile components that BPF has. They allow developers to program elements of the kernel that could previously only be changed by changing the kernel itself

or by introducing kernel modules. This newly gained programmability of the kernel in combination with userspace software enables many new applications in tracing, networking and security without modifying the kernel that were previously impossible to build.

2.5 Related Work

One example of a specialized tracing tool that makes use of BPF is ipftrace [40]. It allows to observe the functions that process sk_buffs in the networking stack. To achieve this, the tool scans the BTF information of the running kernel for all functions that take a pointer to an sk_buff as a parameter. It then attaches BPF kprobes to each of these functions to register their execution. The advantage that this tool offers over less specialized tools is that it only traces sk_buffs that are marked in a specific way. Such marks can be added to sk_buffs using tc or iptables. Therefore, the scope of the tracing can be narrowed down to network packets that fulfill certain requirements, e.g., belong to a specific flow. Thereby, ipftrace can support debugging that is directly related to the handling of packets [41]. However, while it supports to retrieve some customizable data from the sk_buff, it can not get the full context that is necessary to identify, for example, which netfilter table the sk_buff is being processed by. Also, it does not report accurate per-packet timing information that are necessary for jitter calculation.

There has been previous research on the latency and jitter of networking on a single host. Beifuß et al. [42] study the latency of packet processing in Linux that is caused by NAPI and its interaction with the hardware drivers, i.e., before a packet reaches the networking stack. They simulate the behavior of a NAPI driver and validate their results in a testbed with accurate timing measurements. The machine is configured to perform simple in-kernel layer 2 forwarding using the Open vSwitch included in Linux. Their results show that NAPI leads to a high number of interrupts at low packet rates and a low number of interrupts at higher rates. This is expected due to the design of NAPI. For the latency, they show that the distribution of observed packet latencies is non-trivial. As an example they show round-trip latencies ranging from less than 10 µs to 100 µs with multiple peaks across the whole range but the two highest ones around 80 µs. This latency determined by kernel functionality such as the NAPI design and the concrete driver implementation, but not the implementation of a CNI plugin. Therefore, we note its existence here and do not include it in our measurements.

Herzog et al. [43] measure the latency between the top half and the bottom half of an interrupt handler. While their measurements are not focused on

network packets, this latency applies to NAPI drivers as well before they start polling packets. To obtain the data, they used a custom kernel module that registers an interrupt handler and measures the time difference that occured between its top and bottom half. The results show that this latency can be up to 10 µs and exhibit significant jitter. However, this jitter can also not be influenced by the CNI plugin and is thus excluded from the measurements in this thesis.

Public cloud providers usually offer Kubernetes clusters based on Virtual Machines (VMs). Oljira et al. [44] show that virtualization adds latency to the packets because of buffering and unfair resource sharing in the hypervisor networking. Also, the number and network traffic of VMs co-located on the same hypervisor host have a high impact on the additional delay. Whiteaker et al. [45] also found that co-locating VMs with high network traffic leads to increased latency. This effect greatly exceeds the impact of co-located VMs stressing other system resources. We conclude from the described research that virtualization adds many more potential causes of jitter outside the control of the cluster operator. Therefore, we focus our work on clusters running on bare-metal machines, because they are more feasible for workloads with strong requirements in terms of low latency and jitter.

Emmerich et al. [46] present an analysis of round-trip latency for UDP connections up to the application layer. The application running on the test machine is supposed to resemble a game server, i.e., real-time communication. They as well see a high impact of running the application in a VM and see a huge increase in latencies and jitter when the host is under high load. Also, they show how the socket buffer size affects the trade-off between latency and throughput for sending packets.

Blake et al. [47] focus their latency measurements on an application that uses TCP connections and measure the latency introduced by the networking stack. To collect the internal data from the networking stack, they have built a kernel module using SystemTap. This is a tracing framework that is not part of the Linux source code and allows to create custom tracers that are compiled into kernel modules and loaded for the data collection. They show that the 99^{th} percentile latency until the packets are placed in the receive queue does not vary a lot with increasing load while the 99^{th} percentile latency until the packet is received by the userspace application can increase by more than two orders of magnitude. Therefore, they conclude that the queuing behavior is responsible for a significant share of the receive latency. They also observe that outgoing packets show a significant increase in the 99^{th} percentile latency under high load.

While the related work described so far focuses on the latency of network communication and thereby detects jitter as variance in their data, it has

also been tried to identify sources of jitter more directly. De et al. [48] used instrumentation of the Linux kernel to observe operating system jitter, i.e., jitter that applications running on the system experience that is not necessarily related to networking. They used custom kernel modules to obtain data on which process or interrupt handler was active on the processor and when it started or ended. Their data shows that most of the interruptions of a synthetic test application happened due to the timer interrupt, but they also identified several daemons running on the system that were responsible for interruptions of the application and therefore jitter.

Gonzalez et al. [49] also investigate the operating system jitter but using a different method. Instead of using custom kernel modules, they use perf to collect data from tracepoints related to the scheduler. While this removes the requirement of kernel modules, it also misses some sources of jitter, most notably the handling of interrupts. However, it is possible to observe when the system executes parts of the interrupts, i.e., the bottom half, in kernel threads as they are managed by the scheduler. The data shows that the majority of jitter from other processes can usually be attributed to a small number of processes and thus potentially optimized with reasonable effort.

Toussaint et al. [50] determine the networking jitter under different Linux system configurations. They use an external video device to generate a stream of network packets with constant inter-packet gaps. On the device under test, timestamps are generated when the kernel receives a packet and when the userspace application receives a packet. The kernel timestamps are generated by a tracepoint in the interrupt handler of the NIC and observed using perf. The variation of the difference of consecutive timestamps is the jitter introduced by the hardware (kernel timestamps) or the kernel (application timestamps). To see the effects of different parts of the system, they changed the system configuration in various ways and observed the resulting jitter. They conclude that they see the biggest impact on the jitter from hardware effects, such as the interrupt throttling of the NIC, and these effects exceed the impact of the scheduler.

Qi et al. [16] recently published an extensive study comparing five different CNI plugins, including Calico with the default data plane and Cilium. They compare the features and describe how each plugin fulfils the Kubernetes networking requirements on a technical level. In addition, throughput and latency is measured in two-node cluster in a testbed. However, they only measure the round-trip latency and do not dissect the latencies in the networking stack. Instead, perf is used to observe how many cycles are spent in the different kernel subsystems for each of the CNI plugins. While this correlates with the latency introduced by a subsystem, the data can not be used to determine the causes of jitter in the stack.

Chapter 3

Implementation

This thesis aims to obtain data that allow to identify the causes of jitter in the Linux kernel networking stack. The data collection is performed by an application that is implemented as part of this thesis, because the existing tracing tools in the Linux kernel are not sufficient for this use case. This chapter first presents the main design ideas of the implemented application and summarizes why the existing tools are not sufficient. Then, the process of identifying feasible probe points is outlined and finally, the details of the implementation are described.

3.1 Design

The core idea is to implement a specialized tracing application to measure the latency introduced by different parts of the networking stack of the Linux kernel. The application is called *lattrace* and focuses particularly on the components of the stack that are used by CNIs.

Inside the kernel, BPF probes generate timestamped events when a packet passes specific parts of the stack. An event is always related to one particular sk_buff and may be augmented with additional information where it is feasible. This may include data to further describe the event or details of the packet, such as the flow 5-tuple. The events are transferred to a userspace application which collects them and calculates the latency that each packet experienced. This per-packet data is stored to be analyzed later on.

The probes are attached at different points in the networking stack to trace different subsystems. In the following, it is explained which probes, on a high level, were identified to obtain useful insights into the stack latencies. The concrete hooks to attach probes are not readily listed anywhere but needed to be found through a manual process described in Section 3.3. Section 3.4

presents the hooks that were selected for the tracing application by this process and explains the probes being attached to them. The application should collect data for the following events:

New packet The time when a packet is seen for the first time in the stack. This may be due to the packet arriving from a NIC or being sent out from an application.

Begin of processing The time when the stack starts processing the packet in the main part of the stack. This can show if there are significant delays between the time we see a new packet and the time its processing starts.

tc The start and end time, and subsequently the duration, of the processing of tc hooks. Cilium attaches its BPF programs here to process packets before most of the stack is executed [51].

conntrack The start and end time, and subsequently the duration, of the normal connection tracking based on netfilter. This can be bypassed by Cilium [52].

iptables The start and end time, and subsequently the duration, of the processing of one hook in one table. This also requires additional information to identify the processed hook and table. Many CNIs use iptables but they often differ in how they use it.

End of processing The time when the stack is done processing the packet. This can be because the packet has been handed to the NIC to be sent to the network or because the packet has been placed in the queue of a userspace socket.

Packet finished The time when the packet has been completely handled and is not needed any more. This can differ from the end of processing for packets that are destined for a socket where the packet is only finished after its data has been read by the socket.

During the processing, a probe retrieve the information of the flow 5-tuple from the packet to be able to attribute it to a connection. The flow information is just obtained once and not by every probe to reduce the impact of the probes on the system performance. This happens at the end of processing the sk_buff to make sure that all packet headers have already been parsed by the stack to be able to reuse this information. Obtaining the flow information just once means that some fields in the packet have already been rewritten by

previous actions of the stack. Thus, the manual process of identifying relevant flows from the output data requires some knowledge about the communication that is happening in the cluster.

3.2 Existing Tracing Tools

The main tracing tools that are integrated into Linux are perf and ftrace as described in Section 2.3. While they provide great value in many situations, they are not feasible to obtain the data necessary for the latency tracing. For this, the tracing tool needs to fulfill three criteria. First, it needs to be able to instrument the kernel in a very fine-grained way. It needs to be able to record when the execution of certain functions starts or ends. Second, it must obtain rich data from the kernel. The traced function executions need to be attributed to particular packet. Some functions may provide even more context, such as the currently processed table for functions that belong to iptables. In addition, the packet should not only be identified but also attributed to a flow. Third, the tracing should be limited to the necessary information to limit the overhead introduced by performing the measurement. The functions in the networking stack are notoriously often executed because they process all network packets that arrive at the machine. Therefore, the implemented tracing tool should obtain the previously described fine-grained, rich data only in the few necessary spots and not influence any other parts to keep the overhead low.

The tracing tool perf handles the performance counters and kernel tracepoints. Performance counters are optimized for userspace applications instead of the kernel and not optimal for latency tracing. On the other hand, the kernel tracepoints do not cover every part needed in the networking stack. That requires tracing the execution of kernel functions, which is not supported well with perf. Also, the context received from tracepoints is very limited and does not fulfill the requirement for rich data. perf does only provide information about how the processing is happening but not about what data has been processed.

ftrace can also trace arbitrary kernel functions in addition to tracepoints. The tracing can also be limited down to certain functions and tracepoints of interest to reduce the overhead that is not necessary for the latency tracing. However, also with ftrace it is not possible to obtain further data for the traced events.

Therefore, the tracing application for this thesis is implemented using BPF. It can attach probes to tracepoints and the beginning and end of functions. Each of the attachment points needs to be selected manually which keeps the

overhead low compared to probes being attached to all kernel functions by default. The probes can read arbitrary data that can increase the level of insights that can be obtained from the data. As BPF is not limited to tracing, functionality that the other tracing tools provide out of the box needs to be implemented, e.g., generation of timestamped events and the transfer to the userspace application.

3.3 Identifying Hooks for Probes

To trace the latencies in the networking stack, timestamps should be taken when packets traverse important points in the stack. The high-level ideas for the points is described in Section 3.1. For the implementation it is necessary to identify concrete hooks in the kernel where the tracing probes could be attached. This requires a thorough understanding of the whole networking stack for which the literature can only serve as a basis. Thus, significant effort has been put into obtaining knowledge about how exactly packets are processed by the networking stack with different CNI plugins. The process of identifying the hooks that are used in the implementation is described in the following. It can also serve as guide to adapt lattrace to new use cases. There are two types of hooks relevant for the tracing applications: attaching a probe at an existing tracepoint or attaching it at the beginning or end of a kernel function.

The initial idea for probe points to measure the lifetime of an sk_buff was taken from Gregg [53]. As there are no tracepoints that could be used for the allocation of new sk_buffs, the probes were attached to the function that allocates memory from a kernel memory cache. This approach was abandoned later, because this function is used by many other kernel subsystems apart from the network stack and therefore attaching probes to it introduced a lot of overhead. Instead, the probes were attached to the functions for allocating sk_buffs in the networking stack. For freeing the sk_buffs there are two tracepoints already available, but they turned out not to cover every code path. Therefore, an additional probe at a low-level function for freeing sk_buffs was added to not miss out on any attempts to free an sk_buff. These hooks were all identified by reading the kernel source code and discussing with kernel developers. But this is a rather slow and inconvenient process to identify hooks for all kernel subsystems.

To improve the process of identifying further hooks for probes, ftrace was used to observe how the kernel processed packets and narrow down to a few candidates to attach probes. The next paragraphs describe how tracing with ftrace was used to identify the hooks for lattrace. First, a single-node

Kubernetes server was installed with one of the CNIs of interest to be able to follow the traffic in this setting and an iperf server was deployed to the cluster. Next, the node was configured to perform all the packet processing on a single core. This simplifies the analysis of the trace later on because there are no concurrent activities in the trace.

To keep the processing on one core, there are three elements to consider: on which core are incoming packets processed, on which core is the application running that receives the packets, and on which core are outgoing packets processed. All three aspects can be configured on a Linux machine. If a NIC supporting multiple queues is used, the number of send and receive queues should be set to one for the tracing. To restrict the incoming packet to a specific core, the `irqbalance` daemon is stopped to prevent it from overriding any manual configuration. The system is then configured to handle interrupts for incoming packets from the single queue of the NIC on a specific core using Receive Side Scaling (RSS) [54]. Usually, RSS is used by the network interface to distribute incoming traffic to different queues that are processed on different cores. In addition, Linux supports Receive Packet Steering (RPS), which is similar to RSS but implemented in software [54]. It is disabled by default and should stay disabled for the tracing. Outgoing packets are processed on a core determined by Transmit Packet Steering (XPS) [54]. This allows to configure the queue of the interface that is used for outgoing packets. If the interface has just a single queue, XPS has no effect and does not need to be configured. The core that executes a userspace application is selected by the scheduler. This selection can be restricted to a subset of the available cores using CPU affinity. It can be configured for a running process using `taskset` and should be set to allow the iperf server to run only on the core that also handles the incoming packets as configured using RSS. The described configuration is just used to understand the networking stack better by tracing its execution. It was reverted afterwards and is not required to perform the measurements with lattrace.

With the system prepared as described, the processing of packets can be traced with ftrace. To simplify the interaction with ftrace, `trace-cmd` is used. The tracing can be restricted to the single core that has been configured to perform all the packet processing. The function tracer from `trace-cmd` generates an event every time a new function in the kernel is entered and is therefore well suited to understand the code path a packet takes. The output contains a line for each function entry in the kernel in the order the functions were executed. In addition, the function names in the output are indented according to the stack depth to visualize which function called which one. This helps to quickly identify potential functions for the latency tracing. To find the actual hooks to be used, it is necessary to read the source code of

```
<idle>-0 [021] 976125.386983: function: ip_list_rcv
<idle>-0 [021] 976125.386984: function:    ip_rcv_core.isra.0
<idle>-0 [021] 976125.386984: function:    ip_rcv_core.isra.0
<idle>-0 [021] 976125.386984: function:    ip_sublist_rcv
<idle>-0 [021] 976125.386984: function:       nf_hook_slow_list
<idle>-0 [021] 976125.386985: function:          nf_hook_slow
<idle>-0 [021] 976125.386985: function:             ip_sabotage_in
<idle>-0 [021] 976125.386985: function:             ipv4_conntrack_defrag
<idle>-0 [021] 976125.386986: function:             iptable_raw_hook
<idle>-0 [021] 976125.386986: function:                ipt_do_table
<idle>-0 [021] 976125.386986: function:                   comment_mt
<idle>-0 [021] 976125.386987: function:                   comment_mt
<idle>-0 [021] 976125.386987: function:                   mark_tg
[...]
<idle>-0 [021] 976125.386994: function:             iptable_mangle_hook
<idle>-0 [021] 976125.386994: function:                ipt_do_table
<idle>-0 [021] 976125.386994: function:                   comment_mt
<idle>-0 [021] 976125.386995: function:                   comment_mt
```

Listing 2: Snippet from the ftrace output used to identify hooks for probes

the potential functions in addition to the tracing output. They may already contain tracepoints that can be used. Otherwise, the function entry or exit needs to be traced. As functions that are marked as static can not be traced with fentry or fexit probes but need k(ret)probes, functions that are not marked as static should be preferred.

Listing 2 shows a short snippet from the tracing output to exemplary demonstrate the process of finding a hook. The trace was generated with incoming packets and the output starts soon after the networking stack starts processing one of the packets. For the sake of this example, we are looking for hooks to trace the time that is needed to evaluate the iptables rules. Each line of the output contains the process name (here always identified as idle), the process ID (here always zero), the CPU ID of the core the function is running on, a timestamp, and finally the name of the function that was executed. It can be observed that iptables uses different functions that

```
unsigned int
ipt_do_table(struct sk_buff *skb,
             const struct nf_hook_state *state,
             struct xt_table *table)
```

Listing 3: Signature of `ipt_do_table` from `net/ipv4/netfilter/ip_tables.c` of the Linux kernel source code

are themselfes hooks, e.g., `iptable_raw_hook` and `iptable_mangle_hook`, which could both serve as potential hooks for our tracing probes. But both functions call `ipt_do_table`, which could also be a hook for our tracing tool. While this would have the advantage of just requiring one probe, we want to maintain the context in which it was executed, i.e., the netfilter hook. The context must be identified from the parameters of the function. To make a decision for one or the other potential hooks, we can take a look at the source code of the functions. None of the functions contain useful tracepoints that could be directly utilized. While the whole `ipt_do_table` function is too long to be shown here, Listing 3 displays its signature. The first parameter is a pointer to the sk_buff being processed, which is always needed as context to identify the packets across the networking stack. By looking at the definitions of the passed structs, it can be seen that the struct `nf_hook_state` contains a field with the current netfilter hook and the struct `xt_table` contains a field the name of the iptables table that is processed right now. As this provides enough context, the duration of the evaluation of one iptables table can be traced by attaching probes to the beginning and end of `ipt_do_table`. The two calling functions do nothing more than calling `ipt_do_table`, so we are not excluding any relevant processing time from our tracing.

This is just one example of how a hook was identified. In some cases, tracepoints could be found somewhere in the executed functions that can be used instead of attaching directly to functions. In other cases it was in the end necessary to trace a static functions, because there are no other options available. However, all of the hooks that are used in the tracing application were picked by the described combination of tracing and studying the kernel source code.

3.4 Implementation

This section describes the implementation of the tracing application in detail and provides remarks about the kernel version compatibility and potential improvements at the end.

The tracing application was implemented using the Rust programming language using the library *libbpf-rs* which offers a safe interface to libbpf. This is a library which manages the full lifecycle of BPF programs and provides additional utility functions. At the start of the application, it loads the BPF probes into the kernel and attaches them to the different hooks across the network stack. When the probes are executed, they create an event data structure, fill it with the type of the event, the current time and potentially additional information, and send the event into a perf event array.

A perf event array can be used as unidirectional communication channel from the probes in the kernel the userspace application and behaves similar to a message queue. In the application, one thread is polling the perf event array. When a new event is received, the bytes representing it are read from the perf event array, parsed into a data structure, and send to another thread for further processing. There, the events for each packet are collected over the whole lifetime of the sk_buffs. When the application is stopped, the collected events are further processed to calculate the duration spent in the different subsystems of interest for the latency tracing. The resulting data are saved per-packet into a CSV file to be analyzed later on. In addition, the collected events can be saved to repeat the post-processing with new versions of the application.

The sk_buffs are identified by its memory address in the tracing application. Usually, the memory for the sk_buffs is allocated using a slab allocator [55]. These allocators are commonly used in operating system kernels to improve the performance of allocations of many objects of the same type and to reduce the fragmentation of the memory [56]. The core idea of a slab allocator is to maintain a cache of allocations for each type of objects. New allocations are made from this cache and are therefore fast. Instead of freeing the memory after usage, the allocation is returned to the cache.

For the tracing application, this means that the sk_buffs with the same memory addresses, and therefore the same identifiers, are often reused. Therefore, it is very important not to miss any allocation and free of an sk_buff to make sure that all collected events are attributed to the right sk_buff and not an earlier or later packet. As described before, adding probes at the functions managing the cache is not desirable, because these functions are used more often than just from the network stack. The key observation to trace all allocations of sk_buffs was that an sk_buff might be cloned while traversing the networking stack. This is used, for example, to be able to perform a TCP retransmit after the original packet has already been sent out. The cloning of an sk_buff is handled in the tracing application by cloning the history of already collected events as well. Thereby, the recorded allocation time and the beginning of the history of the clone is equal to the original sk_buff. After the cloning event, the recorded events can differ.

3.4.1 Selected Hooks

This section describes the actual hooks that are used in the tracing application to obtain the information outlined in Section 3.1. In general, attaching probes to tracepoints is often preferable over attaching them to function entries and exits. Tracepoints are guaranteed to be called when the code is executed,

while function calls might not be present in the binary anymore because the function has been inlined by the compiler to optimize the performance. Also, the same tracepoint can cover many different execution paths where it would be easy to miss a path when attaching probes directly to the function entry or exit. Of course, tracepoints are not always available and, especially when the duration of the execution of a function should be measured, the probes need to be attached directly to functions. Attaching probes to fentry or fexit hooks is claimed to introduce less overhead than k(ret)probes [33]. The difference in overhead is thoroughly investigated in Section 4.1.2. The hooks used by lattrace are listed in Table 3.1 and the reasoning behind them is described in the following.

New sk_buff To trace the allocation of a new sk_buff, there are two fexit probes at the functions `__alloc_skb` and `__build_skb`. These functions cover all allocations of sk_buffs from the slab allocator. The probes need to be attached at the exit of the functions, because they both return a pointer to the fresh sk_buff, which is used as an identifier for the whole lifetime of the sk_buff. While these probes cover all allocations of sk_buffs, the first measurements showed overly long lifetimes in the order of 60 ms for the packets arriving at a NIC. This happens, because NAPI drivers maintain a buffer of sk_buffs. They are allocated before they are put into the buffer but only used after a while when they have been filled with packet data by the network interface. To account for this, another probe is attached to the tracepoint `napi_gro_receive_entry`. This tracepoint is located at the beginning of the first function in the networking stack that handles incoming packets and does not belong to the driver anymore.

Cloned sk_buff Sometimes, an sk_buff is created by cloning an existing one, e.g., to keep it ready for potential TCP retransmits. To observe the creation of clones, an fexit probe is attached to `skb_clone`. This function returns a pointer to the new clone which is why the probe is attached at the end of the function, similar to the functions creating fresh sk_buffs.

Begin of processing A probe is attached to the tracepoint `netif_receive_skb` to observe when the handling of packets from a network interface starts in the networking stack. This is not necessarily identical to the point of time where the packets are retrieved from the interface, because after the retrieval of the packet the buffer of sk_buffs in the driver may be refilled or other interrupts may arrive. As many CNI plugins use virtual ethernet pairs, which are virtual network interfaces, this probe

Event	Probe Type	Function/Tracepoint
New sk_buff	fexit	`__alloc_skb`
	fexit	`__build_skb`
	tracepoint	`napi_gro_receive_entry`
Cloned sk_buff	fexit	`skb_clone`
Begin of processing	tracepoint	`netif_receive_skb`
tc	fentry	`tcf_classify_ingress`
	fexit	`tcf_classify_ingress`
	fentry	`tcf_classify`
	fexit	`tcf_classify`
conntrack	kprobe	`nf_conntrack_in`
	kretprobe	`nf_conntrack_in`
iptables	kprobe	`ipt_do_table`
	kretprobe	`ipt_do_table`
	kprobe	`ip6t_do_table`
	kretprobe	`ip6t_do_table`
Queued to NIC	tracepoint	`net_dev_queue`
Queued to socket	kprobe	`tcp_queue_rcv`
	kprobe	`udp_queue_rcv_skb`
Freed sk_buff	tracepoint	`consume_skb`
	tracepoint	`kfree_skb`
	fentry	`__kfree_skb`

Table 3.1: Hooks that are used by the tracing application

can also be used to observe when the sk_buff is picked up by the stack in a different network namespace.

tc Cilium attaches BPF programs to tc hooks to process packets before they hit the main part of the stack. The programs are executed by the functions `tcf_classify_ingress` for incoming packets and `tcf_classify` for outgoing packets. While they both call the same function to perform the actual processing, the context, i.e., incoming or outgoing, is not accessible in the called function and the called function is marked as static which would require to use k(ret)probes. Therefore, fentry and fexit probes are attached in the beginning and end of each of the two

initially mentioned functions.

conntrack The connection tracking is performed for packets received from a network interface and for packets created on the host. Both times, the main work is performed by the function `nf_conntrack_in` which allows to identify the context, i.e., incoming or outgoing, from one of the parameters. Therefore, the probes can be attached directly to this common function. However, the netfilter modules, including conntrack, are developed as kernel modules. While kernel modules can contain BTF information, those were not exposed by the running kernel until very recent kernel versions. But the BTF information of a function are necessary to attach fentry and fexit probes to that function. Therefore, the k(ret)probes are used in the beginning and end of `nf_conntrack_in`.

However, this introduces another issue: In contrast to fexit probes, kretprobes can not access the parameters of the function but just the return value. The return value of `nf_conntrack_in` are just constants indicating what should happen to the packet after connection tracking. Therefore, the kretprobe is neither able to identify the currently processed sk_buff nor any other context. To address this issue, a new BPF map is introduced to the program. The map is a per-CPU array that has only one entry on each CPU core. The kprobe at the beginning of `nf_conntrack_in` creates its event with all information and stores it in the map in addition to sending it to the userspace application. The kretprobe at the end of the function can than reuse the event in the array, replace the type and the timestamp of the event, and send it to userspace. This way, the event already contains all important information like the identifier and the context. The approach works because a core does not execute functions in parallel, the probes at the beginning and end of the function run on the same core, i.e., the core the function runs on, and each core has its own value in the array that is not overwritten from other cores.

iptables The iptables rules can be added to different tables, such as raw, filter, or mangle, and netfilter hooks, which appear as the default chains to users. However, not all tables are present at all hooks. Together, table and hook are the necessary context to identify which rules are being evaluated. Depending on the table and the hook, the rules are evaluated in different places in the stack, but always by the same functions, `ipt_do_table` for IPv4 and `ip6t_do_table` for IPv6 packets. The parameters of the two functions allow to identify both elements of the context. The table can only be identified by its name which

is stored as a string. This name is sent to the userspace application as a string which is not optimal because it increases the size of the transferred events unnecessarily.

As conntrack, iptables is also maintained as a kernel module. Therefore, k(ret)probes are attached to the beginning and end of the functions as well. The issue of the missing context in the kretprobe is also addressed with a BPF per-CPU array map, in the same way as for conntrack.

Queued to NIC To observe when the network stack is done processing an outgoing sk_buff and adds it to the queue of a network interface, a probe is attached to the tracepoint `net_dev_queue`. With the virtual ethernet pairs used by many CNI plugins, this probe can also be used to observe when the sk_buff has been fully processed in a network namespace. In combination with the probe at the beginning of processing, this allows to determine the latency introduced by the network namespace switches.

Queued to socket When an sk_buff is destined for an application on the local host and is fully processed, it is added to the receive queue for the respective socket to be read by the application. For TCP, this is done by the function `tcp_queue_rcv`; for UDP by `udp_queue_rcv_skb`. While both functions represent the queueing operation well for their respective transport protocol, they are both marked as static. Therefore, no BTF information are generated for the functions and kprobes must be used instead of attaching fentry probes.

Freed sk_buff When an sk_buff is not needed anymore, it is freed, i.e., returned to the slab allocator. This can happen through many code paths. Probes are attached to the tracepoints `consume_skb` and `kfree_skb` which already cover many of these paths. Many of the functions with the tracepoints call the function `__kfree_skb` to perform the actual work. However, this function is also called directly from the TCP stack, bypassing the tracepoints. Therefore, an fentry probe is attached to `__kfree_skb` as well. In theory, this could lead to two free events being observed for one sk_buff, but this case is never observed in practice. This is probably due to the function often being inlined and therefore not covered by the fentry probe.

One important aspect of the probes for freeing sk_buffs is that they are the last operations in the stack. It is reasonable to assume that all headers in the packet have been identified and parsed by now. Therefore, these final probes are also used do retrieve the flow information from the packet and include them in the event sent to the userspace application.

3.4.2 Kernel Requirements

The tracing application does not require one specific kernel version because it uses BTF. However, it does require a kernel with included BTF information. This is the case for many current releases of the popular Linux distributions. In addition, it requires kernel version 5.5 or higher to make use of fentry/fexit probes for more efficient tracing of the start and end of function executions. In the future, the minimum required version could be increased to 5.11, because this version makes it possible to expose the BTF information of kernel modules through the existing kernel interface. This would allow to use fentry/fexit for functions related to iptables and conntrack which are maintained as kernel modules.

This also highlights the automatic adaptation to the kernel version as one potential area of improvement for the tracing application. The current code was written for the kernel version 5.8 and at least the BPF probes should be compatible with other kernel versions thanks to BTF. However, bigger changes in the kernel are currently not accounted for. For example, it could be implemented that the probes for iptables and conntrack are attached as fentry and fexit probes if supported by the kernel with a fallback to the current k(ret)probes. Similarly, when parts of the networking stack are rewritten, the tool could have probes for the old and the new code paths. In many cases, this would not even mean to maintain a large list of kernel versions and their features. Instead, the application could simply try to attach the probes and use potential fallbacks if the attachment fails.

Chapter 4

Evaluation

This chapter consists of two sections. It begins with a evaluation of the performance overhead introduced by running lattrace. This is followed by a comparison of the latency and jitter characteristics of three Kubernetes CNIs. The data for the comparison is collected using the tracing application.

For the evaluation, two different hosts were used. Their specifications are shown in Table 4.1 and they were connected via an Ethernet switch for the measurements. The main host was running the experiments while the traffic generator was used as a client for generating network traffic. The setups of each experiments are described more extensively in the respective sections. In general, the experiments do not require the powerful machines used and should be reproducible with less resources.

	Main Host	Load Generator
Processor	2 × Intel Xeon 8276M	2 × Intel Xeon E5-2695 v4
Memory	12 × 16 GiB	16 × 16 GiB
NIC	Intel I219	Intel X540
Operating System	Ubuntu 20.04.2	Ubuntu 20.04.2
Kernel Version	5.8.0-45	5.11.22

Table 4.1: Details of the machines used for the evaluation

In the context of networking, jitter usually describes how much the latency differs over a set of packets. Despite being commonly used, the term is not well defined as a metric. Different standards, such as *RFC 5481* [57] and *ITU-T Y.1540* [58], instead define the metric Packet Delay Variation (PDV) of a packet as the difference between the delay of that packet and the lowest observed delay of a packet. The distribution of the PDV is thus the

distribution of the latency, shifted towards zero by the minimum latency. To summarize the PDV of a set of packets into one value, *ITU-T Y.1540* [58] recommends to choose two quantiles and use the difference between the delay variation values at these quantiles as the summary. For this chapter, whenever numbers for the jitter or PDV are reported, the difference between the 0.01 quantile and the 0.99 quantile, i.e., the 1^{st} and 99^{th} percentile, of the PDV distribution are used.

4.1 Performance Overhead of the Tracing Application

The tracing application was developed for this thesis and thus has no existing validation available for it. Therefore, it is important to measure the overhead introduced to the packet processing by the application. This is necessary to assess the obtained data later on. However, the impact on the overall system performance is not of great importance, because the application is not supposed to run on production machines.

The most significant overhead when running *lattrace* results from the fact that there is the additional code of the probes that is executed for every packet that is processed by the kernel. This overhead is estimated in two different ways in the next section. Often, probes can be attached with different mechanisms, e.g., kprobes, fentry probes or tracepoints. Section 4.1.2 investigates the impact of the different attachment mechanisms on the overhead.

4.1.1 Overhead of the Probes

The first way to estimate the overhead of the latency tracing application is to run a simplified version of the application. This simplified version contains only the probes at the beginning and the end of the stack, but the probes for the different subsystems in the kernel are removed. The remaining probes are for the allocation, cloning, and freeing of sk_buffs and for adding an sk_buff to the queue of a network interface. The results of the two versions of the tracing application can then be compared. The difference in the measured stack latency can be attributed to the probes that are missing in the simplified version of the application. It includes the the overhead of the attachment mechanisms and the probes themselves.

The differences in the number of probes and the observed latency is shown in Table 4.2. The traffic during the measurements was generated using iperf sending out 50 Mbit/s of UDP traffic. The data shows that the number of

		Observed Latency			Difference	
CNI	Probes	Median*	PDV*	Events	Latency*	Events
Calico	all	65.3	57.8	17	44.0	13
	start/end	21.3	39.2	4		
Calico eBPF	all	78.1	65.7	21	59.3	17
	start/end	18.9	40.1	4		
Cilium	all	30.6	37.9	9	15.7	5
	start/end	14.9	27.6	4		

* Times displayed in µs

Table 4.2: Measured stack latency and jitter with all probes attached or only probes at the beginning and end of the stack ("start/end") and the number of events observed for each sk_buff, i.e., executed probes

probes has a big impact on the overall latency. Each probe adds around 3 µs overhead to the packet processing which is around 15 % to 20 % of the latency of the whole networking stack as measured with probes at the beginning and end of the stack. In addition, an increase of the jitter can be observed as well. However, this effect is smaller than the increase in latency. Both observations apply regardless of the CNI plugin.

Another way to measure the overhead of probes is to profile their execution directly. This allows to reason about the overhead of one specific probe more accurately than with the former method. `bpftool` can profile, beside others, the number of instructions the CPU executed and the number of cycles needed for the execution. Due to the internals of the tool, the profiling does not work on fentry and fexit probes, but for the sake of this measurement, they can be converted into k(ret)probes. This does not impact the executed code of the probe.

The probes used by the latency tracing tool can be categorized by their expected overhead: simple probes that emit events without reading memory, probes that need to read memory before emitting an event, the kprobes that emit an event and cache it in a map, and the kretprobes that use the cached events. To recall, the two latter ones are needed to retain information for the kretprobe when it is not possible to attach fentry and fexit probes, for example, in kernel modules.

Four probes from the tracing application were selected to represent the categories: a kprobe at `udp_queue_rcv_skb` for the simple probe, a kprobe at `__kfree_skb` that reads the flow information from the sk_buff in memory, and

k(ret)probes around `ipt_do_table` employing the caching mechanism. Each of the probes was profiled for a period of time to accumulate about 40 000 to 50 000 runs of the probe. The measurement was repeated 10 times for each of the probes. The obtained results are shown in Table 4.3 Unfortunately, `bpftool` only reports the number of executions and the sum of cycles and instructions over all executions which only allows to calculate a mean but does not provide information about the variance. The numbers reported here are means of the means and the standard deviation of the means to provide at least some intuition about the stability of these numbers.

		CPU Cycles		Instructions	
Function	Probe	Mean	Std.Dev.	Mean	Std.Dev.
`udp_queue_rcv_skb`	kprobe	3947.7	93.3	2370.0	123.4
`__kfree_skb`	kprobe	6566.6	74.7	3201.5	11.7
`ipt_do_table`	kprobe	4285.0	201.3	2865.5	215.8
`ipt_do_table`	kretprobe	1357.1	154.3	1379.7	1.5

Table 4.3: Cycles and instructions as reported by `bpftool` for different probe categories

The data from the profiling confirm the results from the previous latency measurements that the probes add significant overhead. Taking into account a CPU frequency of 1 GHz to 2 GHz as observed during the experiments, the time derived from the number of cycles is similar to the times observed in the previous measurement. While the overhead is in the same range for all probes, the impact of the probes differs depending on their category. Probes that need to read a lot of memory, e.g., to gather the flow data, generally show a higher overhead in terms of cycles and instructions than the simple probes. Also, a kprobe that caches an event has slightly more overhead than a simple probe. However, the kretprobe reading the cached event uses much less cycles and instructions. The sum of the overhead of such a pair of kprobe and kretprobe is still lower than the combined overhead of two simple probes. Especially the number of cycles is significantly lower for the k(ret)probe pair, because the kretprobe can execute more instructions per cycle, which is probably related to the event generated by the kprobe still being in the CPU cache.

Considering the results from both measurement approaches, it becomes clear that the application introduces significant overhead to the processing of network packets. As expected, it is thus not suitable for continuous monitoring of stack latencies on production hosts. But the overhead is relatively constant per probe and can therefore be estimated when the number of probes in the

Probe Type	Attach Point	Data Access
kprobe	Begin of any function in the kernel	All parameters
kretprobe	End of any function in the kernel	Return value
fentry	Begin of kernel functions with BTF information	All parameters
fexit	End of kernel functions with BTF information	All parameters and the return value
tracepoint	Pre-defined tracepoints in the kernel	Context of the tracepoint

Table 4.4: Comparison of BPF probes for tracing from a developer perspective

processing path is known. The impact on the jitter appears to be lower than the impact on the latency. Still, both effects should be kept in mind when assessing the results. To limit these effects, some probes are disabled in the experiments in Section 4.2. Chapter 5 presents some further ideas for how to improve the application to reduce the overhead.

4.1.2 Performance of the BPF Probe Types

In addition to the overhead from the actual probe code, the probes also need to be attached to hooks in the kernel. This section evaluates the performance overhead of the different attachment mechanisms.

The Linux kernel offers different mechanisms to attach BPF probes. The mechanism used for a particular probe is determined by the type of the probe. There are three general types of probes that are interesting for tracing: tracepoints, kprobes/kretprobes, and fentry/fexit probes. For a developer of probes, they differ in the places where they can be attached and in the data they can access. These differences are shown in Table 4.4. From a kernel point of view, the probe types also differ in the mechanism that is used to execute attached probes. The mechanisms are described in the following.

k(ret)probe In Linux, kprobes can be attached to the kernel for a long time already, since kernel version 2.6.9 [59]. It was originally designed to be used from kernel modules. Later, this was extended to allow to attach kretprobes at the end of kernel functions. The k(ret)probes accessible

through BPF are slightly limited but come with all the benefits of BPF programs. All kernel versions that support BPF, support k(ret)probes.

To attach a kprobe to a kernel function, the first instruction of the function is copied to a different location in memory [32]. The original instruction in the function is then replaced with a breakpoint instruction. When the CPU reaches this breakpoint, all registers are saved. Then, the BPF program is executed, which receives a pointer to the saved registers as a parameter. Afterwards, the copy of the single instruction is executed before the CPU returns to the normal execution flow of the kernel function.

When a kretprobe is attached to a kernel function, first a kprobe is attached to the beginning of the function [32]. This kprobe saves the return address from the stack and replaces it with the address of a so called *trampoline*. Therefore, upon the return from the probed function, the code of the trampoline is executed. The trampoline itself has another kprobe attached to it. This kprobe then runs the code that the user supplied for the kretprobe. Afterwards, the kprobe restores the saved return address to the instruction pointer of the CPU. Thereby, the execution continues normally in the kernel function that called the function with the attached kretprobe. Attaching a kprobe to a function that already has a kretprobe attached should add less overhead because the user-defined kprobe and the internal kprobe at the beginning of the function can be chained without an additional breakpoint.

Due to the way the BPF programs are attached to k(ret)probes, there is a bit of additional, constant overhead for the call to the program.

fentry/fexit This attachment mechanism has been added recently in kernel version 5.5 [33]. It works by creating customized *BPF trampolines* for the attached probes and making use of a compiler feature that was originally used for dynamic tracing with ftrace.

The Linux kernel is compiled using an option that inserts a function call to a special function in the beginning of each kernel function. Thus, in the compiled Linux kernel, the first instruction in each function is a `call`. At boot time, these `call` instructions are replaced with `nop` (no operation) instructions of the same size to minimize the overhead as long as tracing is disabled.

To attach an fentry probe to a kernel function, a customized trampoline is created that saves just the parameters of the function to the stack, calls the fentry probe with a pointer to the stored parameters, and returns [60].

The creation utilizes the BTF information of the function to store the parameters and ensure the stack stays intact. The first instruction of the function is then replaced with a `call` to the trampoline.

Attaching an fexit probe to a function works similarly by replacing the first instruction with a `call` to a customized trampoline, which is created differently [60]. The trampoline also stores the parameters on the stack, but then `call`s the function with an offset to skip the first instruction. When the function returns, the execution of the trampoline continues, which stores the return value on the stack and calls the fexit probe with access to the parameters and the return value. Afterwards, the execution returns to the kernel function that called the probed function to continue the normal execution.

If both, an fentry and an fexit probe, are attached, there is just one trampoline that is a combination of two described before [60].

tracepoint Tracepoints can be used to attach BPF programs as well. When an enabled tracepoint is hit, a special function is called. This function executes the BPF program, besides other operations that are required for other tracing tools.

As some of the operations are not necessary when just a BPF program is attached, raw tracepoints were introduced [61]. They rely on the same tracepoints that are present in the source code, but do as little additional work as possible to just run the program.

The different attachment mechanisms lead to different overheads when running the probes. The measurement of these overheads is described in the following sections.

Measurement approach

To measure the overhead of the different probe types, a small benchmarking application was implemented. The idea behind the benchmark is to pick two hooks in the kernel and attach measurement probes to them. The probes observe how many instructions have been processed in between them and how many cycles this took. Then, additional dummy probes, which do as little work as possible, are attached between the two measurement probes. This allows to compare the number of instructions of a scenario where no probe is attached and scenarios where different types of probes are attached. The difference between the baseline measurement and the measurements with probes is the overhead of attaching a probe.

The measurement probes are an fentry and an fexit probe attached to the same function. This allows simple reasoning about what happens between the two probes. The fentry measurement probe reads the relevant perf counters (instructions and cycles) and stores them in a per-CPU array map. The fexit measurement probe reads the perf counters again and the measurements from the fentry probe, calculates the difference, and sends the difference to the userspace application. The application then reads the measurements and stores them in a CSV file for later analysis. The measurement probes are based on the code that `bpftool` uses to profile BPF programs[1] but were modified to send every measurement to the userspace application.

The benchmarking imposes some requirements on the function that the measurement probes are attached to. The function needs to contain a tracepoint that dummy probes can be attached to and it also needs to call another function which dummy probes can be attached to. Ideally, the function with the measurement probes is simple and contains no loops and not too many conditionals to reduce the noise introduced by running different code paths. The function `consume_skb` matches these requirements quite well and is shown in Listing 4. It has a tracepoint called `consume_skb` and calls the function `__kfree_skb` next to the tracepoint. It has the additional advantage that it is run often by the networking stack, which reduces the overall execution time of the benchmark, and that it can easily be triggered by sending network traffic to the machine.

```
void consume_skb(struct sk_buff *skb)
{
        if (!skb_unref(skb))
                return;

        trace_consume_skb(skb);
        __kfree_skb(skb);
}
```

Listing 4: Function `consume_skb` used for benchmarking of the overhead of the different probe types, from the Linux kernel source, `net/core/skbuff.c`

However, during the measurements it turned out that dummy probes attached to `__kfree_skb` were never executed. Inspecting the disassembled kernel binary revealed that the compiler had optimized the code by inlining `__kfree_skb` into `consume_skb`. Thus, there was no function call anymore that could trigger the execution of the dummy probes. To address this, the dummy probes are instead attached to the function `skb_release_head_state`,

[1] See the Linux kernel source, `tools/bpf/bpftool/skeleton/profiler.bpf.c`

which is unconditionally called by `__kfree_skb` and can therefore serve as a replacement hook.

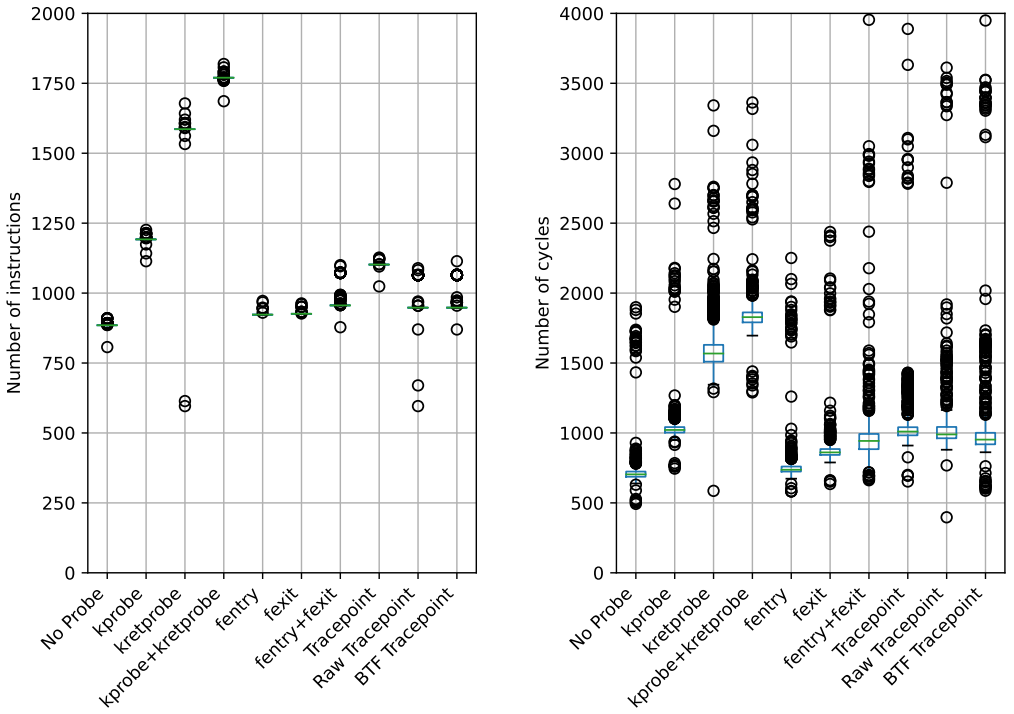
The benchmarking process works as follows: First, the dummy probe to be measured is installed. Then, the application waits for a few seconds to exclude effects of the attachment process from the benchmark. After that, the measurement probes are attached and the results are polled and saved by the userspace application. When the execution has been benchmarked about 10 000 times, first the measurement probes and then the dummy probe are removed. The application repeats this cycle for each of the dummy probes to measure. During the whole benchmarking, there is an iperf server running on the main host and the traffic generator is sending 50 Mbit/s of TCP traffic to the server.

Results

The results obtained from the benchmarks are shown in Figure 4.1. The graphs for instructions and cycles are both boxplots to visualize the spread of the measurements. In Figure 4.1a, the boxes are just lines, because almost all executions had the same number of instructions. This is reasonable because the executed code is always the same. The graph for the cycles in Figure 4.1b shows a similar pattern as the number of instructions because these metrics are strongly correlated. However, the variance of the observed measurements for cycles is higher, because the number of cycles needed for one execution is influenced by many other factors, including CPU caches.

Figure 4.1 shows that kprobes already have a high overhead and that kretprobes have more than twice the overhead of a kretprobe. If a kprobe and a kretprobe are used in combination at one function, the additional overhead over a kretprobe is significantly lower than the overhead of attaching just a kprobe. This is reasonable considering that attaching a kretprobe to a function automatically also attaches a kprobe for technical reasons as described earlier. A kretprobe actually consists of two kprobes, one of which performs work to store the return address. This explains why the kretprobe needs more than twice the instructions of a kprobe. When a kprobe and a kretprobe are used, the user-defined kprobe and the internal kprobe from the kretprobe are chained. Thus, the additional kprobe introduces less overhead than adding the first kprobe to a function.

The fentry and fexit probes both have a very low overhead compared to k(ret)probes while offering similar flexibility. In terms of instructions, both have a very similar overhead while the fexit probe appears to need more cycles. The number of instructions differs only minimally because the trampoline of fentry and fexit is very similar. The overhead of a combination of an fentry



(a) Instructions (b) Cycles

Figure 4.1: Number of instructions an cycles with different attachment mechanisms; the difference to "No Probe" is the overhead of the mechanism; the benchmark results are randomly sampled down to 1000 data points to improve visibility

and an fexit probe is a bit less than the sum of the overheads of the two probes, because they are called from the same trampoline that can reuse the stored parameters for both probes. However, this setting is probably less often needed as with k(ret)probes because fexit probes can access the parameters of the function directly.

Attaching a BPF program to a tracepoint has a high overhead but not as high as a kprobe. The number of instructions that are needed when the program is attached to a raw tracepoint or a BTF tracepoint is equal but significantly lower than for a normal tracepoint. The number of cycles does not differ much between the different tracepoint types in this experiment. BTF tracepoints and raw tracepoints have the same number of instructions, because they use the tracepoints in the same way internally and only differ in the mechanism for loading the probe into the kernel. The difference in

instructions between normal tracepoints and raw/BTF tracepoints are the additional operations that are not necessary to just run a BPF program.

As a summary, it is advisable to use fentry or fexit probes over k(ret)probes when it is possible for the respective function and the kernel version supports it. Tracepoints have the benefit of covering many code paths that perform the same operation, e.g., the tracepoint for consuming an sk_buff is placed in four different function of the networking stack. In addition, tracepoints will never be removed during the optimization of the compiler, while function calls as the hooks for fentry/fexit and k(ret)probes might be inlined and thereby be removed. Thus, (raw) tracepoints do have an advantage compared to k(ret)probes if they are available. In comparison with fentry/fexit probes, the situation is more nuanced and might be best decided by weighting the different advantages in every case.

4.2 Comparison of CNI Plugins

The thesis aims to compare the latency and jitter of different CNI plugins for Kubernetes. This section describes the measurement setup used for the comparison and presents the results obtained from tracing the latency of packet processing in the kernel.

The Kubernetes cluster used for the measurements is a single-node cluster on a bare-metal machine, the main host described earlier. The cluster is freshly set up for each of the different CNI plugins using *kubeadm*. After that, one of the CNI plugins is installed: Calico, Cilium, or Calico with the eBPF data plane. Calico and Cilium were selected because they are both commonly used in the industry but follow different approaches by default. Calico provides the CNI functionality relying on iptables while Cilium implements many features using eBPF. More recent versions of Calico also implement an eBPF data plane that can be activated. This is included in the comparison as well.

The configuration of the cluster uses mostly the provided defaults. It deviates only in three aspects: It uses a different IP subnet for the pods; the single node is untainted, i.e., Kubernetes control plane and worker node at the same time, to allow scheduling normal pods on it; and the number of usable NodePorts has been increased to exceed 10 000. The latter was necessary to be able to test with a very high number of services configured in the cluster. Calico is installed following the installation guide from its documentation[2]. This installs the Tigera operator to the Kubernetes cluster which in turn installs Calico in version 3.19.0. For Calico with eBPF data plane, Calico

[2]Calico installation guide:
`https://docs.projectcalico.org/getting-started/kubernetes/quickstart`

is installed as before but the guide for activating the different data plane is followed after the installation[3]. Cilium is also installed using their installation guide[4]. The guide installs Cilium directly in version 1.9.7. All CNI plugins were configured to follow the pod subnet of the cluster and the increased number of NodePorts but otherwise left in their default configuration.

For each of the CNI plugins, the following measurement process is performed. After the Kubernetes cluster and the CNI plugin have been set up, there are 10 000 services with NodePorts created that all point to the same pod. The services are created in batches to prevent the control plane from overloading. These services should serve as an overhead to measure how the CNI plugin performs with many services. Then, an iperf server is deployed to the cluster that is later used as the testing service. It is created after the overhead services to add its respective rules at the end of iptables chains after the rules of the overhead services. Next, the iperf client on the traffic generator starts the traffic, which is given 10 seconds to stabilize before the latency tracing application is run for 10 seconds. This is repeated for incoming TCP traffic to the cluster, outgoing TCP traffic from the cluster, incoming UDP traffic, and outgoing UDP traffic. The number of overhead services is then reduced to 1000, 100, 10, and finally 0 with all measurements being repeated for each number of services.

However, in none of the experiments, did an increased number of overhead services result in relevant changes to the latency or jitter behavior. Thus, all the results reported in the following refer to the configuration with no additional overhead service. This stands in contrasts to previous results [62, 63] which show an increased latency with more services, but with significant increases only when running more than 2000 services. A reason for this might be the configuration that all overhead services map to a single pod in our experiments. Running one or even multiple pods for each service in the experiments was not possible, because a Kubernetes node does not support running that many pods.

4.2.1 Incoming Traffic

This section presents the results obtained from the measurements with incoming traffic. Figure 4.2 shows Cummulative Distribution Functions (CDFs) of the times it took for the sk_buffs from reception at the NIC to arrive at specific points in the stack, with different subgraphs for the different CNI

[3]Calico eBPF data plane activation:
`https://docs.projectcalico.org/maintenance/ebpf/enabling-bpf`

[4]Cilium installation guide:
`https://docs.cilium.io/en/v1.9/gettingstarted/k8s-install-default/`

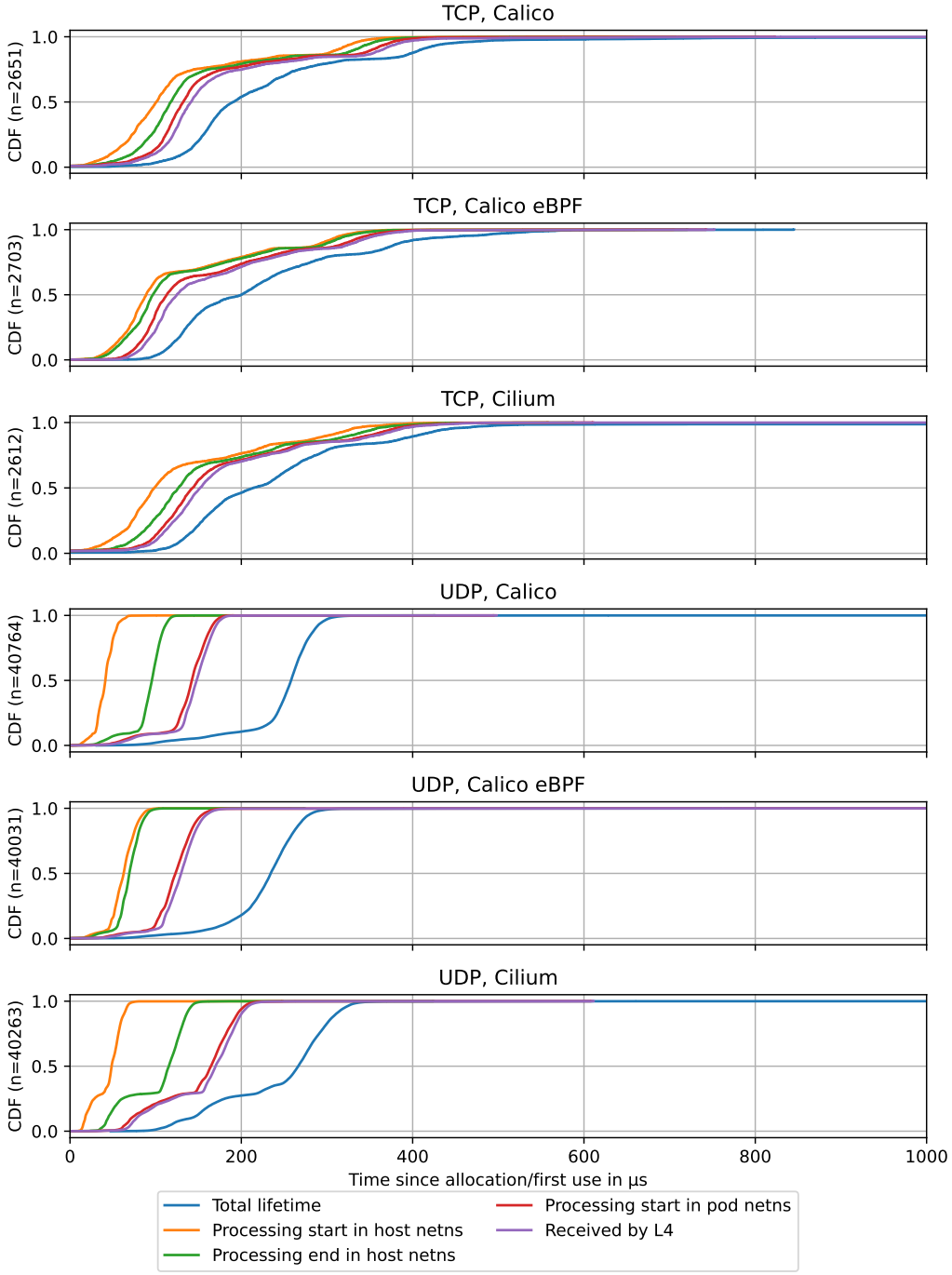


Figure 4.2: Times from retrival of an incoming sk_buff from a network interface until it has passed certain parts of the networking stack

	Received by Socket[1]		Host Network Namespace[2]	
CNI, Protocol	Median (µs)	PDV (µs)	Median (µs)	PDV (µs)
Calico, TCP	141.24	467.67	18.02	28.38
Calico eBPF, TCP	124.64	336.53	5.50	14.11
Cilium, TCP	151.99	448.37	22.46	44.98
Calico, UDP	147.82	132.26	53.85	54.40
Calico eBPF, UDP	129.71	128.19	4.49	14.92
Cilium, UDP	169.91	158.88	65.32	61.64

[1] Time from reception of a packet until it is added to the queue of the socket (purple lines in Figure 4.2)
[2] Time that is actually spent processing the packet in the host network namespace (between orange and green lines in Figure 4.2)

Table 4.5: Latency and jitter for incoming traffic over the whole networking stack and for the processing in the host network namespace

plugins and transport protocol. The line for the lifetime (blue) is equivalent to the time the data is read from the socket. All probes that are not shown in the figure (i.e., iptables, conntrack and tc) were disabled to prevent them from introducing additional latency and jitter. Figure 4.3 shows CDFs of the durations spent in specific subsystems of the networking stack, again with subgraphs for the different CNI plugins and transport protocols. If there is a line missing for one subsystem, this means that this subsystem was not used by the plugin.

Looking at Figure 4.2, the gap between the reception of the packet and the start of processing in the stack stands out, which is especially big for TCP traffic. In terms of the kernel, this gap occurs between the `napi_gro_receive_entry` tracepoint for packet reception and the function `netif_receive_skb` that starts the processing. From more invasive traces generated with ftrace, it can be seen that this time is mostly spent in the polling function of the NAPI driver. Therefore, other packets may be received by the driver after the first sk_buff has been received from the interface and before it is further processed. Also, after all packets have been received, the driver allocates as many new sk_buffs as it has received from the interface and adds them to the buffer of the NIC so they can be filled with new incoming data. Both of these operations have probes attached to them by the tracing application, which add overhead as described in Section 4.1.1. The trace also shows that the driver generally receives more TCP packets from the NIC during one polling loop than UDP. The UDP packets are then processed

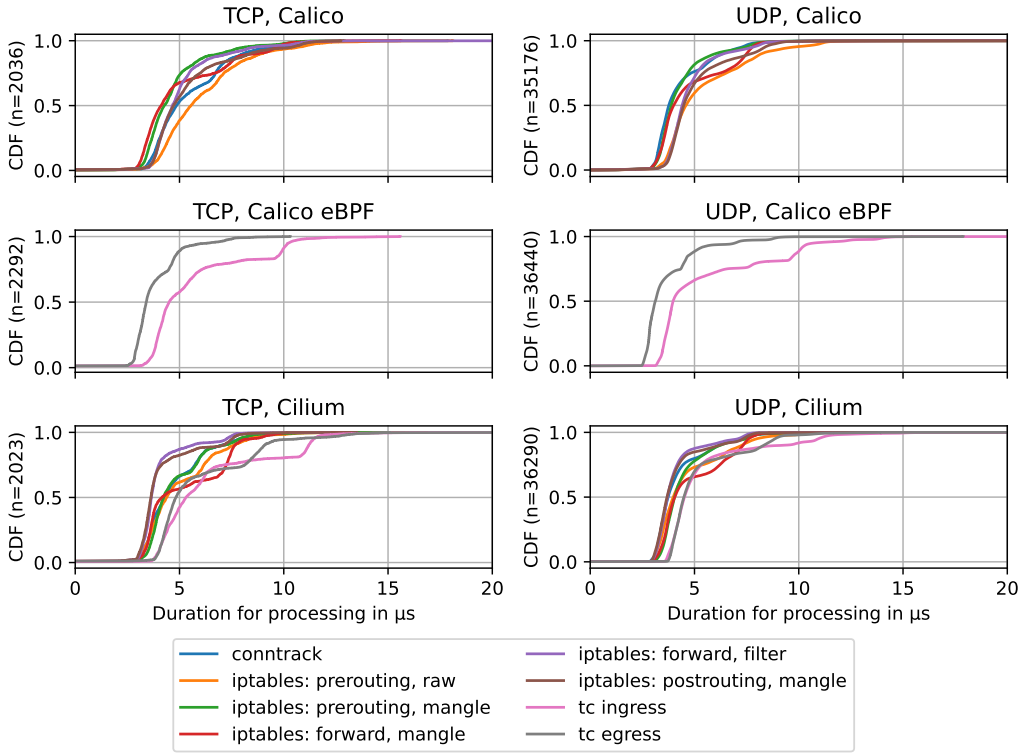


Figure 4.3: Observed processing durations in different subsystems of the networking stack

individually whereas the TCP packets are combined into few big sk_buffs due to a feature of the Linux kernel called Generic Receive Offload (GRO). They are aggregated into the first received sk_buff whose observed waiting time before being processed by the stack then includes the reception of all other packets including the respective probes. In experiments with GRO disabled, the TCP packets show the same, small delay before the processing starts as UDP packets do. GRO is also the reason why there are so much less sk_buffs observed for TCP despite the same measurement duration and the same data rate of the traffic.

For TCP, the gap before the processing starts appears to dominate the latency until the data is received by the application. The lines that mark the times where an sk_buff crosses different points in the stack all follow the shape of the delay until the processing of a packet begins. This also leads to a high jitter when looking a the whole stack as it can be seen in Table 4.5. For UDP, the initial processing delay does not dominate all the other observed timestamps, but does of course contribute to the jitter.

After this initial delay, the packets are processed in the host network namespace. The majority of the configurations applied by the CNI plugins happens here. The time spent in the host network namespace can be seen in Figure 4.2 between the orange and the green line. The data for the introduced latency and jitter are also shown in Table 4.5 in the last two columns. With both transport protocols, the time spent in the host network namespace differs between the CNI plugins. It is especially low for Calico with the eBPF data plane and significantly higher for Calico with the default data plane. For Cilium, it is even higher than for Calico with the default data plane. The same order also applies to the jitter that can be observed for the different CNI plugins. Figure 4.3 points to a reason for this observation. While Calico with the eBPF data plane uses just two tc hooks for the packet processing, Calico with the default data plane uses five different iptables hooks and conntrack. Cilium, in turn, uses iptables and conntrack as Calico does but additionally the tc hooks as Calico eBPF does. The amount of processing also seems to determine the amount of jitter in the host networking namespace. This high latency with Cilium is an unexpected result, because it is focused on providing a performant eBPF data plane that bypasses the kernel networking stack. However, it does not succeed to provide this feature while Calico with the eBPF data plane does so instead.

Further investigations reveal the reasons for this behavior. When Calico eBPF is activated, kube-proxy is explicitly disabled on the nodes, which would normally install many iptables rules. In the default deployment of Cilium as it was used in the experiments, kube-proxy was still running on the node. According to the documentation, this is supposed to be a fallback mechanism in case the kernel is missing features for the eBPF data plane [19]. Cilium is supposed to detect the features the kernel supports and do as much in eBPF as possible. According to the Cilium documentation, the kernel version 5.8 as used in the experiments should support all necessary features to run without kube-proxy. Also, the status report Cilium confirms that all potential eBPF implementations are active. In production setups, this should probably enforced by configuring Cilium to fail if eBPF features are missing (strict mode). However, Cilium relies on the networking stack for routing decisions [64]. Thus the traffic is passed through the stack where it also passes iptables with all the rules from kube-proxy, albeit most of these rules should be skipped. Only starting from kernel version 5.10, Cilium activates a feature called *eBPF-based host-routing* which performs these routing decisions in eBPF and thereby allows the traffic to bypass the stack completely.

The jitter in the host networking namespace clearly correlates with the amount of work to be performed, i.e., the number of kernel subsystems that an sk_buff has to traverse. The data in Table 4.5 show that this difference

between the CNI plugins is present for both transport protocols. However, the plugins that use iptables and conntrack, i.e., Calico with the default data plane and Cilium, the jitter introduced for UDP traffic is significantly higher than for TCP traffic. This may be caused by the higher load on the system due to the more costly processing of the packets and the higher number of sk_buffs that are processed for UDP.

Figure 4.3 shows that a more fine-grained analysis of which subsystems are responsible for jitter is not meaningful due to the latency and jitter introduced by the probes of the tracing application. All lines only start after 2.5 µs which is the same order of magnitude as the overhead of one probe as determined in Section 4.1.1. Therefore, it can be assumed that the detailed graphs in Figure 4.3 show more about the behavior of the probes than of the actually traced functions.

After the sk_buffs have been processed in the host network namespace and before they are processed in the pod namespace, they are not being processed for a while. This can be seen in Figure 4.2 as a gap between the green and red lines, which is more pronounced for UDP traffic. In that period, other packets that were received from the network interface at the same time are processed and added to a queue of the virtual ethernet pair before the whole batch of sk_buffs gets processed in the pod network namespace. The figure also shows that the amount of time spent with processing in the pod network namespaces, i.e., red to purple line, is very low. This is reasonable because the CNI plugins do not configure any networking features inside the pod namespace where they could easily be changed by the applications running in the pod. The delay is slightly higher for TCP traffic because the protocol requires additional work to be done when a packet is received.

When the processing of sk_buffs in the stack is done, they are added to the queue of a socket where they stay until their content is read by the applications. This period is clearly visible for UDP traffic as a gap between purple and blue line. The delay this introduces is not controlled by the CNI plugin but could rather be optimized by the application developers or by tuning the operating system on the host.

4.2.2 Outgoing Traffic

This section presents the results for outgoing traffic with similar graphs as the previous section. Figure 4.4 shows CDFs of the times it took for the sk_buffs to arrive at specific points in the stack, with different subgraphs for the different CNI plugins and transport protocols. As before, probes that are not shown in the figure (i.e., iptables, conntrack and tc) were disabled to prevent them from introducing additional latency and jitter.

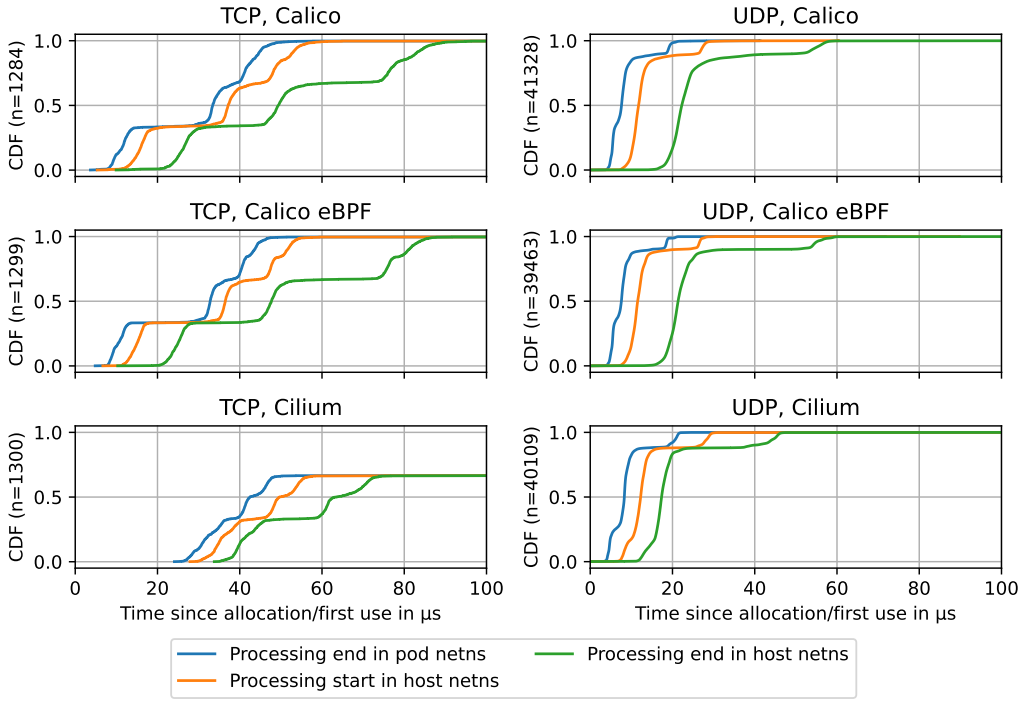


Figure 4.4: Times from creation of an outgoing sk_buff due to a write to a socket until the sk_buff has passed certain parts of the networking stack

In Figure 4.4, the graph of Cilium processing TCP traffic shows a remarkable feature: the CDF do not reach 1 but instead merge into a straight line at 0.7. While this appears to be an issue with the measurement, the effect is reproducible over several runs in the evaluation setup. It does only occur for Cilium with outgoing TCP traffic. A closer analysis of the data shows that almost all sk_buffs whose processing, apart from allocation and freeing, happens on core 21 exhibit the high latency. This is also the core that handles the interrupts from the NIC. The sk_buffs processed on other cores do not show this behavior. Performing the same analysis for the other CNI plugins shows that they almost never process sk_buffs on core 21. Thus, they do not exhibit the high latencies. While this explains, where the differences come from, it is unclear what causes the processing of nearly half of the sk_buffs on core 21 when Cilium is used and what causes the long delays on this particular core. Because of this, TCP with Cilium is not directly compared in the following paragraphs.

The time 0 µs in Figure 4.4 marks the creation of a new packet. With outgoing traffic, this happens when a userspace application writes data to a socket. The figure shows that TCP traffic is subject to a long delay before

	Received by NIC Queue[1]		Host Network Namespace[2]	
CNI, Protocol	Median (µs)	PDV (µs)	Median (µs)	PDV (µs)
Calico, TCP	49.36	69.57	12.47	27.90
Calico eBPF, TCP	47.70	65.56	11.39	24.94
Cilium, TCP	63.50	750.16	9.88	12.90
Calico, UDP	22.26	41.09	10.54	22.87
Calico eBPF, UDP	21.36	42.74	9.98	24.27
Cilium, UDP	17.35	34.29	5.16	12.80

[1] Time from creation of a packet until it is added to the queue of the NIC (green lines in Figure 4.4)
[2] Time that is actually spent processing the packet in the host network namespace (between orange and green lines in Figure 4.4)

Table 4.6: Latency and jitter for outgoing traffic over the whole networking stack and for the processing in the host network namespace

the sk_buffs leave the pod network namespace, i.e., the gap before the blue line. This part of the processing is also responsible for a significant share of the overall jitter. A reason for the initial delay can be found in the buffering of packets at the socket because of TCP congestion control. This can be seen in the more invasive traces from ftrace. For UDP traffic, this delay is consistently small for all CNI plugins and most sk_buffs. This is reasonable because UDP does not employ congestion control or similar mechanisms that would result in buffering of data.

The time between the end of processing of an sk_buff in the pod network namespace and the beginn of processing in the host network namespace, i.e., the gap between the blue and orange lines in Figure 4.4, is pretty constant across all CNI plugins and transport protocols. This indicates that outgoing sk_buffs created through application writes are usually directly picked up by the stack in the host network namespace. Data from ftrace confirm that a software interrupt is raised when an sk_buff is transmitted through a virtual ethernet pair, which immediately triggers the processing of the sk_buff in the host network namespace. For UDP, where no buffering is in place, the data even shows that an sk_buff created by writing to a socket is usually processed completely until it is added to the queue of a physical NIC to leave the machine.

The host network namespace is where the majority of the processing configured by CNI plugins takes place. This introduces latency which can be seen in Figure 4.4 as the gap between the orange and green lines and

in Table 4.6. The data show that the introduced latency is lower than for incoming traffic with the exception of Calico eBPF, which shows a higher latency than for incoming traffic. Also, processing of UDP traffic is generally a bit faster than processing TCP traffic. For both transport protocols, Calico and Calico eBPF appear to introduces similar latency and jitter. In contrast, Cilium only introduces about half of this latency and jitter. This is an interesting observation considering that for incoming traffic Calico eBPF has the best latency and jitter behavior while Cilium has the worst. Looking at the data in more detail reveals that processing outgoing traffic with Cilium uses only two tc hooks whereas Calico eBPF makes use of the two tc hooks, iptables, and conntrack. This is the opposite configuration compared with incoming traffic, where Calico eBPF used just the tc hooks. Here it can be seen again that the latency and the jitter increases when more different subsystems are used for processing.

Chapter 5

Conclusion

This chapter summarizes the work of this thesis and its results, shows how the results can be applied in practice, and presents ideas on how the tracing application could be improved in the future.

In this thesis, a tracing application is developed to observe the latency and jitter introduced by different parts of the Linux networking stack. It is tailored to obtain these metrics from a host performing container networking as in a Kubernetes cluster. The overhead of the application is thoroughly analyzed to allow a better assessment of the results. The application is then used to collect latency data from different CNI plugins which are compared with regard to the latency and jitter they introduce to the network traffic.

Analyzing the overhead of the tracing application shows that its probes introduce significant additional latency. In addition, an impact on the jitter could be observed, albeit smaller than on the latency. To account for that, the number of probes is reduced in the experiments with the different CNI plugins. As the remaining probes still increase the processing latency, the results should not be used to compare absolute values with the results from other measurements. However, they can still be compared relative to each other in the scope of this thesis.

One result of the experiments applies irrespective of the CNI plugin. UDP traffic exhibits similar or lower latency and always lower jitter than TCP traffic considering the processing in the whole stack. This can be attributed to the stack performing less operations in general and absence of buffering mechanisms when using UDP. While it is commonly known that latency-sensitive applications should prefer UDP as the transport protocol wherever possible, the obtained data also enables a nice visualization of the reasons behind this knowledge.

The data for the different CNI plugins shows that processing packtets using only eBPF can decrease latency and jitter compared to plugins using

iptables. However, none of the compared plugins used eBPF for incoming and outgoing traffic at the same time. While Calico eBPF shows good performance for incoming traffic, Cilium shows good performance for outgoing traffic. With newer kernel versions, Cilium is supposed to use a pure eBPF data plane for incoming and outgoing traffic. It remains to be tested if that can provide the latency and jitter benefits in both directions.

Apart from that, it is hard to make a recommendation for a CNI plugin that provides universally low latency and jitter. Instead, many factors contribute to the latency and jitter that is introduced by the kernel networking stack. The factors are not just limited to the plugin itself but also include the impact it has on the overall system performance and probably aspects like hardware components and other workloads in the cluster. Therefore, it is key to perform measurements assessing the latency in a cluster that matches the production environment as closely as possible. This can as well reveal unexpected side effects of the CNI plugins. It is also important to check if a plugin really processes packets according to its claims or if there are additional operations involved in the processing or if there are unexpected performance issues such as observed for Cilium with outgoing traffic. Of course, the configuration of the container networking solution as well as the system in general should be tuned to be as deterministic as possible when very low latency and jitter are required.

While all experiments are performed using Kubernetes, the tracing application can also work with other container infrastructures, such as *Docker* or *Linux Containers*. If their network configurations are substentially different from CNI plugins, it may be necessary to introduce furter probes in the stack with the technique shown in Section 3.3. This should be especially considered for the way packets are forwarded into the network namespaces of the containers. CNI plugins often use virtual ethernet pairs for this while other container infrastructures use bridges instead. Thus, they may need additional probes to cover this part of the networking stack. However, as long as the workloads and the network processing share the same kernel, i.e., no VMs are used, the tracing application can obtain traces from the network interface to the userspace application.

Chapter 4 shows that the tracing application can have significant impact on the observed latencies. Due to the increased CPU utilization, this may also lead to other parts of the system behaving differently, e.g., the scheduler. To address this high overhead, several areas are worth a look. One option is to reduce the number of probes in the path. While the number of the probes is already limited to the minimum of what is needed to gather the desired data, the probes do not all need to be active at the same time. Instead, they could be grouped and only activated if their measurements are of interest.

For example, the measurement of the duration of iptables, conntrack, and tc could be disabled when the investigation focuses on the latency of the whole networking stack and only activated when the very fine-grained data is needed. This has already been done manually for some of the experiments. However, this could be implemented in a more user-friendly way that does not require to recompile the application with the different sets of probes but rather allow to choose the probes from the commandline interface.

Section 4.1.1 shows that the caching of events that are reused soon, as it is necessary for functions where only kprobes can be attached, has a significant performance benefit. Therefore, this caching could be applied to other probes as well, i.e., the probes for the measurement of the duration spent in tc hooks. However, for all other probes, this approach is not easily applicable and would require more careful design.

The third area of improvements would be to redesign the transfer of events to the userspace application. The different maps that could be suitable for this could be compared in terms of their performance to see if any of them offers an improvement compared to perf event arrays. Also, more data could be aggregated by the probes in the kernel. Here it is important to find a good balance between the time the computations in the kernel take and the time that the transfer to the userspace takes.

However, none of these improvements are expected to reduce the overhead to a level that makes continuous collection of metrics about the stack latency possible, because the tracing of code paths that are as heavily used as the networking stack will always stay expensive. Instead, it may help to reduce the error that needs to be taken into account during the analysis. But even the existing application can provide useful insights as the results obtained in this thesis show.

References

[1] Nane Kratzke and Peter-Christian Quint. "Understanding Cloud-Native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study". In: *Journal of Systems and Software* 126 (Apr. 2017). DOI: 10.1016/j.jss.2017.01.001.

[2] Maria A. Lema, Andrés Laya, Toktam Mahmoodi, Maria Cuevas, Joachim Sachs, Jan Markendahl, and Mischa Dohler. "Business Case and Technology Analysis for 5G Low Latency Applications". In: *IEEE Access* 5 (2017). DOI: 10.1109/ACCESS.2017.2685687.

[3] GSM Association. *Cloud AR/VR Whitepaper*. Apr. 26, 2019. URL: https://www.gsma.com/futurenetworks/wiki/cloud-ar-vr-whitepaper/ (visited on Jan. 17, 2021).

[4] Rami Rosen. *Linux Kernel Networking: Implementation and Theory*. The Expert's Voice in Open Source. New York, NY: Apress, 2014. 612 pp. ISBN: 978-1-4302-6196-4.

[5] Linux kernel developers. *Segmentation Offloads*. The Linux Kernel documentation. URL: https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html (visited on July 1, 2021).

[6] Jeffrey C. Mogul and K. K. Ramakrishnan. "Eliminating Receive Livelock in an Interrupt-Driven Kernel". In: *ACM Transactions on Computer Systems* 15.3 (Aug. 1997). DOI: 10.1145/263326.263335.

[7] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. "Beyond Softnet". In: *Proceedings of the 5th Annual Linux Showcase & Conference*. Vol. 5. ALS '01. Oakland, California, USA: USENIX Association, Nov. 2001.

[8] Jonathan Corbet. *Driver Porting: Network Drivers*. LWN.net. Apr. 28, 2003. URL: https://lwn.net/Articles/30107/ (visited on July 1, 2021).

[9] Torsten M. Runge, Alexander Beifuß, and Bernd E. Wolfinger. "Low Latency Network Traffic Processing with Commodity Hardware". In: *2015 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. 2015 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS). July 2015. DOI: 10.1109/SPECTS.2015.7285296.

[10] Pablo Neira Ayuso. "Netfilter's Connection Tracking System". In: *;login:* 31.3 (2006).

[11] *Tc(8) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man8/tc.8.html (visited on July 29, 2021).

[12] Kubernetes Developers. *What Is Kubernetes?* Kubernetes Documentation. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on July 15, 2021).

[13] Kubernetes Developers. *Pods*. Kubernetes Documentation. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (visited on July 15, 2021).

[14] Kubernetes Developers. *Cluster Networking*. Kubernetes Documentation. URL: https://kubernetes.io/docs/concepts/cluster-administration/networking/ (visited on July 15, 2021).

[15] Kubernetes Developers. *Service*. Kubernetes Documentation. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on July 15, 2021).

[16] Shixiong Qi, Sameer G. Kulkarni, and K. K. Ramakrishnan. "Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability". In: *IEEE Transactions on Network and Service Management* 18.1 (Mar. 2021). DOI: 10.1109/TNSM.2020.3047545.

[17] Calico Developers. *About eBPF*. Calico Documentation. URL: https://docs.projectcalico.org/about/about-ebpf (visited on July 24, 2021).

[18] Cilium Developers. *Introduction to Cilium & Hubble*. Cilium 1.9.8 documentation. URL: https://docs.cilium.io/en/v1.9/intro/ (visited on July 24, 2021).

[19] Cilium Developers. *Kubernetes Without Kube-Proxy*. Cilium 1.9.8 documentation. URL: https://docs.cilium.io/en/v1.9/gettingstarted/kubeproxy-free/ (visited on July 19, 2021).

[20] Mohamad Gebai and Michel R. Dagenais. "Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead". In: *ACM Computing Surveys* 51.2 (June 2, 2018). DOI: 10.1145/3158644.

[21] Mathieu Desnoyers. *Using the Linux Kernel Tracepoints*. The Linux Kernel documentation. URL: https://www.kernel.org/doc/html/latest/trace/tracepoints.html (visited on July 6, 2021).

[22] Steven Rostedt. *Ftrace - Function Tracer*. The Linux Kernel documentation. URL: https://www.kernel.org/doc/html/latest/trace/ftrace.html (visited on July 6, 2021).

[23] Steven Rostedt. *Trace-Cmd: A Front-End for Ftrace*. LWN.net. Oct. 20, 2010. URL: https://lwn.net/Articles/410200/ (visited on July 6, 2021).

[24] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-Level Packet Capture". In: *USENIX Winter 1993 Conference Proceedings*. USENIX Winter 1993. San Diego, California, Jan. 1993.

[25] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications". In: *ACM Computing Surveys* 53.1 (May 29, 2020). DOI: 10.1145/3371038.

[26] Cilium Developers. *BPF and XDP Reference Guide*. Cilium Documentation. URL: https://docs.cilium.io/en/latest/bpf/ (visited on June 25, 2021).

[27] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. *Linux Socket Filtering Aka Berkeley Packet Filter (BPF)*. The Linux Kernel documentation. URL: https://www.kernel.org/doc/html/latest/networking/filter.html (visited on June 25, 2021).

[28] Sebastiano Miano, Fulvio Risso, Mauricio Vasquez Bernal, Matteo Bertrone, and Yunsong Lu. "A Framework for eBPF-Based Network Functions in an Era of Microservices". In: *IEEE Transactions on Network and Service Management* (2021). DOI: 10.1109/TNSM.2021.3055676.

[29] Jonathan Corbet. *BPF: The Universal in-Kernel Virtual Machine*. LWN.net. May 21, 2014. URL: https://lwn.net/Articles/599755/ (visited on June 29, 2021).

[30] *Bpf-Helpers(7) - Linux Manual Page*. URL: https://www.man7.org/linux/man-pages/man7/bpf-helpers.7.html (visited on June 25, 2021).

[31] Toke Høiland-Jørgensen, Jesper 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". In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18: The 14th International Conference on Emerging Networking EXperiments and Technologies. Heraklion Greece: ACM, Dec. 4, 2018. DOI: 10.1145/3281411.3281443.

[32] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. *Kernel Probes (Kprobes)*. The Linux Kernel documentation. URL: https://www.kernel.org/doc/html/latest/trace/kprobes.html (visited on June 15, 2021).

[33] Alexei Starovoitov. *[PATCH v4 bpf-next 00/20] Introduce BPF Trampoline*. E-mail. Nov. 14, 2019. URL: https://lore.kernel.org/bpf/20191114185720.1641606-1-ast@kernel.org/ (visited on July 14, 2021).

[34] Linux kernel developers. *BPF Ring Buffer*. The Linux Kernel documentation. URL: https://www.kernel.org/doc/html/latest/bpf/ringbuf.html (visited on July 16, 2021).

[35] Linux kernel developers. *Linux Kernel ABI Readme*. URL: https://www.kernel.org/doc/Documentation/ABI/README (visited on July 27, 2021).

[36] IO Visor Project. *BCC (BPF Compiler Collection)*. URL: https://github.com/iovisor/bcc (visited on June 30, 2021).

[37] Andrii Nakryiko. *BPF CO-RE (Compile Once – Run Everywhere)*. Feb. 19, 2020. URL: https://nakryiko.com/posts/bpf-portability-and-co-re/ (visited on Mar. 19, 2021).

[38] Andrii Nakryiko. *BTF Deduplication and Linux Kernel BTF*. Nov. 14, 2018. URL: https://nakryiko.com/posts/btf-dedup/ (visited on June 30, 2021).

[39] Linux kernel developers. *BPF Type Format (BTF)*. The Linux Kernel documentation. URL: https://www.kernel.org/doc/html/latest/bpf/btf.html (visited on June 30, 2021).

[40] Yutaro Hayakawa. *ipftrace2*. URL: https://github.com/YutaroHayakawa/ipftrace2 (visited on July 6, 2021).

[41] Yutaro Hayakawa. "eBPF at LINE's Private Cloud". eBPF Summit 2020 (Online). Oct. 28, 2020. URL: https://ebpf.io/summit-2020-slides/eBPF_Summit_2020-Lightning-Yutaro_Hayakawa-eBPF_at_LINE_Private_Cloud.pdf (visited on July 6, 2021).

[42] Alexander Beifuß, Daniel Raumer, Paul Emmerich, Torsten M. Runge, Florian Wohlfart, Bernd E. Wolfinger, and Georg Carle. "A Study of Networking Software Induced Latency". In: *2015 International Conference and Workshops on Networked Systems (NetSys)*. 2015 International Conference and Workshops on Networked Systems (NetSys). Mar. 2015. DOI: 10.1109/NetSys.2015.7089065.

[43] B. Herzog, L. Gerhorst, B. Heinloth, S. Reif, T. Hönig, and W. Schröder-Preikschat. "INTspect: Interrupt Latencies in the Linux Kernel". In: *2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*. 2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC). Nov. 2018, pp. 83–90. DOI: 10.1109/SBESC.2018.00021.

[44] Dejene Boru Oljira, Anna Brunström, Javid Taheri, and Karl-Johan Grinnemo. "Analysis of Network Latency in Virtualized Environments". In: *2016 IEEE Global Communications Conference (GLOBECOM)*. 2016 IEEE Global Communications Conference (GLOBECOM). Dec. 2016. DOI: 10.1109/GLOCOM.2016.7841603.

[45] Jon Whiteaker, Fabian Schneider, and Renata Teixeira. "Explaining Packet Delays under Virtualization". In: *ACM SIGCOMM Computer Communication Review* 41.1 (Jan. 22, 2011). DOI: 10.1145/1925861.1925867.

[46] Paul Emmerich, Daniel Raumer, Florian Wohlfart, and Georg Carle. "A Study of Network Stack Latency for Game Servers". In: *2014 13th Annual Workshop on Network and Systems Support for Games*. 2014 13th Annual Workshop on Network and Systems Support for Games. Dec. 2014, pp. 1–6. DOI: 10.1109/NetGames.2014.7008960.

[47] Geoffrey Blake and Ali G. Saidi. "Where Does the Time Go? Characterizing Tail Latency in Memcached". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Mar. 2015. DOI: 10.1109/ISPASS.2015.7095781.

[48] Pradipta De, Ravi Kothari, and Vijay Mann. "Identifying Sources of Operating System Jitter through Fine-Grained Kernel Instrumentation". In: *2007 IEEE International Conference on Cluster Computing*. 2007

IEEE International Conference on Cluster Computing. Sept. 2007. DOI: 10.1109/CLUSTR.2007.4629247.

[49] Nelson Mimura Gonzalez, Alessandro Morari, and Fabio Checconi. "Jitter-Trace: A Low-Overhead OS Noise Tracing Tool Based on Linux Perf". In: *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*. ROSS '17: International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017. Washingon DC USA: ACM, June 27, 2017. DOI: 10.1145/3095770.3095772.

[50] Arthur Toussaint, Mohammed Hawari, and Thomas Clausen. "Chasing Linux Jitter Sources for Uncompressed Video". In: *2018 14th International Conference on Network and Service Management (CNSM)*. 2018 14th International Conference on Network and Service Management (CNSM). Nov. 2018.

[51] Cilium Developers. *Life of a Packet*. Cilium 1.9.8 documentation. URL: https://docs.cilium.io/en/v1.9/concepts/ebpf/lifeofapacket/ (visited on July 9, 2021).

[52] Cilium Developers. *eBPF Maps*. Cilium 1.9.8 documentation. URL: https://docs.cilium.io/en/v1.9/concepts/ebpf/maps/ (visited on July 9, 2021).

[53] Brendan Gregg. *BPF Performance Tools: Linux System and Application Observability*. 1st ed. Addison Wesley, 2019. ISBN: 978-0-13-655482-0.

[54] Tom Herbert and Willem de Bruijn. *Scaling in the Linux Networking Stack*. The Linux Kernel documentation. URL: https://www.kernel.org/doc/html/latest/networking/scaling.html (visited on July 13, 2021).

[55] Linux kernel developers. *Memory Allocation Guide*. The Linux Kernel documentation. URL: https://www.kernel.org/doc/html/latest/core-api/memory-allocation.html (visited on July 13, 2021).

[56] Jeff Bonwick. "The Slab Allocator: An Object-Caching Kernel Memory Allocator". In: *USENIX Summer 1994 Technical Conference*. Boston, MA: USENIX Association, June 1994.

[57] Al Morton and Benoit Claise. *RFC 5481: Packet Delay Variation Applicability Statement*. Mar. 2009. URL: https://datatracker.ietf.org/doc/html/rfc5481.

[58] *Recommendation ITU-T Y.1540: Internet Protocol Data Communication Service – IP Packet Transfer and Availability Performance Parameters*. Dec. 2019. URL: http://handle.itu.int/11.1002/1000/13933.

[59] Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy, and Masami Hiramatsu. "Probing the Guts of Kprobes". In: *Proceedings of the Linux Symposium*. 2006 Linux Symposium. Vol. 2. Ottawa, Canada, July 2006.

[60] Alexei Starovoitov. *[PATCH v4 bpf-next 04/20] bpf: Introduce BPF Trampoline*. E-mail. Nov. 14, 2019. URL: https://lore.kernel.org/bpf/20191114185720.1641606-5-ast@kernel.org/ (visited on July 24, 2021).

[61] Alexei Starovoitov. *[PATCH v8 bpf-next 6/9] bpf: Introduce BPF_RAW_TRACEPOINT*. E-mail. Mar. 28, 2018. URL: https://lore.kernel.org/netdev/20180328190540.370956-7-ast@kernel.org/ (visited on July 24, 2021).

[62] Alex Pollitt. *Comparing Kube-Proxy Modes: iptables or IPVS?* Apr. 18, 2019. URL: https://www.tigera.io/blog/comparing-kube-proxy-modes-iptables-or-ipvs/ (visited on July 29, 2021).

[63] Shaun Crampton. *Introducing the Calico eBPF Dataplane*. Feb. 25, 2020. URL: https://www.tigera.io/blog/introducing-the-calico-ebpf-dataplane/ (visited on July 29, 2021).

[64] Cilium Developers. *Tuning Guide*. Cilium 1.10.3 documentation. URL: https://docs.cilium.io/en/v1.10/operations/performance/tuning/ (visited on July 24, 2021).