Synergistic Server-Based Network Processing Stack

by

Marcelo Abranches

B.Sc., University of Brasília, 2004

M.Sc., University of Brasília, 2016

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Electrical, Computer and Energy Engineering

2022

Committee Members:

Eric Keller, Chair

Tamara Lehman

Joseph Izraelevitz

Eric Wustrow

Dirk Grunwald

Marcelo Abranches, (Ph.D., Computer Engineering)

Synergistic Server-Based Network Processing Stack

Thesis directed by Prof. Eric Keller

Network functions provide the required functionality to interconnect systems while ensuring security, availability, efficiency, and performance. With the recent trend to run network functions in software over commodity servers (instead of using specialized appliances), there is the need to introduce new systems that can provide the required network features while running on top of optimal processing environments. Several software packet processing technologies currently exist (e.g., in-kernel/kernel-bypass, XDP, and SmartNICs) and each of them provide different features in terms of available functionality, processing capabilities, performance, and efficiency. In this thesis, we break down network application processing needs and by characterizing the processing features provided by each technology, we verify that no single technology can cover all network application requirements. With this observation, we provide new systems that often break the boundaries between different technologies, allowing the building of optimal packet processing environments that can meet countless requirements of modern networks.

Using this as the foundation of our work, we build systems to address many network function needs – layer 2 to layer 7 processing and monitoring. In our first work, we introduce a new packet I/O subsystem to a high-performance userspace TCP stack. This subsystem is provided by new programmable in-kernel features allowing the TCP stack to have a better resource consumption profile and to build cooperation mechanisms between the kernel and userspace. In our second work, we address the needs of monitoring systems by introducing new primitives that allow for building high-coverage monitoring systems with high performance and efficiency. We optimize those primitives by building an efficient division of work between SmartNIC offloads, XDP on the host, and userspace processing. Finally, in our third work, we rethink the Linux networking stack to address the inneficiencies that prevent it to support the performance requirements of modern

applications. We propose to break down its processing in a minimal and efficient fast path and a in a robust and feature-rich slow path provided by the Linux kernel. The fast path is built on demand, based on current processing needs for a given set of services configuration and gets assistance from the slow path for processing completeness. This allows avoiding unnecessary processing inside the kernel, minimizing overheads and increasing performance, while still maintaining Linux's rich set of features. With these contributions, we believe that we can provide new foundations to help both academia and industry to build optimized systems that can address many modern network needs.

Dedication

To my lovely family, Paula, Julia and Miguel. Your courage and love were my fuel during this journey.

Acknowledgements

First, I would like to thank my parents Alice and Fernando, my stepparents, my grandparents and brothers for all their support during my life.

I would like to thank my loved wife Paula, for all her support, love, care, and lightness of heart. Words and thanks will never be enough to express my love for you.

A very special thanks to my advisor, Eric Keller, for all the insights, for always giving me something to think about, and for being an awesome mentor. I also thank my committee, Tamara Lehman, Joseph Izraelevitz, Eric Wustrow, and Dirk Grunwald for the teachings, and insightful suggestions that helped build this dissertation.

I would like to acknowledge my lab mates, Mohammad Hashemi, Azzam Alsudais, Oliver Michel, Sepideh Goodarzy, Greg Cusack, Karl Olson, Maziyar Nazari, Dwight Browne, Erika Hunhoff, and Bashayer Alharbi. Thanks for your friendship and for teaching me amazing stuff during our meetings. Also, it was great to have some of you as my co-authors. I extend those thanks to my co-authors Shivakant Mishra and Stefan Schmid.

I also would like to thank the people who have worked with me in Brazil at the University of Brasília, Priscila Solis, Marcelo Ladeira, Rommel Carvalho, and Eduardo Alchieri. Also, many thanks to my co-workers at CGU, Marcelo Polo, Salatiel Robson, Thiago Paysan, Rafael Dias, André Fonseca, Thimotheo Oliveira, Gustavo Moura, Antonio Maroysio, Henrique Rocha, and many others.

Thanks for all the support, without you guys, this work would not have been possible.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

Modern networks are challenged with workloads that demand high throughput and low latency with ever-increasing intensity. Those networks are at the core to support a highly-dynamic cloud infrastructure where applications can be rapidly deployed and adjusted as needed. Network functions (NFs) like proxies, switches, routers, intrusion detection systems (IDS), access control lists (ACLs), firewalls, and monitoring applications are composed and deployed on the infrastructure. This is done to enforce the desired network behavior, security, scalability and availability. To support those highly-dynamic environments, network services are shifting from running in physical appliances to commodity servers. Running on commodity servers means that network functions now run as software components, which enables great flexibility, but at the same time, imposes new challenges related to providing high performance. One fundamental challenge in this context, is that with the end of Moore's law [72] and Dennard scaling [7] commodity processors (CPUs) are not able to keep up with the performance requirements of the networks that support modern systems, such as big data, artificial intelligence and video on demand. This leads to the necessity of avoiding as much unnecessary processing as possible, so that network packets can be processed promptly.

The Linux network stack has been proven to do too many operations per packet [51], preventing it to have high performance. One recent trend is to build high-performance packet processing systems on top of kernel-bypass technologies, like DPDK [8]. This type of technology enables building custom packet processing solutions, that only execute highly optimized logic that is needed for

each use case; thus avoiding many overheads and inefficiencies inherent to full kernel network stack processing in a generic operating system. However, avoiding a feature rich operating system kernel like the one provided by Linux is a missed opportunity in terms of available functionality (e.g., resource sharing and protocols). To programmatically avoid some Linux network stack overheads, recently a new technology called the eXpress Data Path (XDP) [51] was introduced in the Linux kernel. XDP allows building lighter data paths, while still having access to some kernel functionality, enabling it to balance performance and functionality for NFs. Another trend is to leverage hardware offloads in SmartNICs to assist NFs performance and efficiency on commodity hardware. SmartNICs can help networks by executing some or all the tasks needed by a packet processing pipeline.

Previous works tended to consider the mentioned technologies as independent or competing among each other. In this work, we argue that network systems should be designed considering the potential synergy of combining those technologies to build optimal systems (Figure 1.1). We observe that currently, there is a gap in systems that can effectively enable high-performance network applications while coexisting with and leveraging the rich features provided by the Linux kernel. In this direction, we propose to address several dimensions of network application needs finding a suitable feature/performance balance between kernel-bypass, in-kernel processing, and finally SmartNIC offloads.

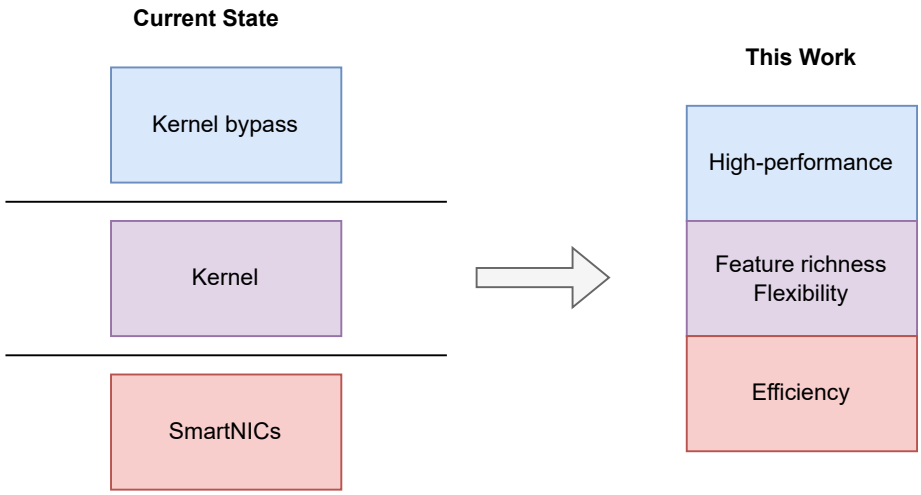

Figure 1.1: Breaking packet processing boundaries.

1.1 What Network Applications Need?

Network applications are fundamental components of a datacenter infrastructure. They provide connectivity among different services, protect the infrastructure against cyberattacks and ensure service scalability and performance. Those applications are responsible for different tasks in a network, providing essential functionality for each layer of the open systems interconnection (OSI) model [110]. Layer 2 (L2) and layer 3 (L3) services are responsible to provide connectivity among hosts on the same network segment (Ethernet) or on different networks (IP). Those services are provided by NFs like routers and bridges. Layer 4 (L4) to layer 7 (L7) services operate above the previously mentioned layers, and are responsible for functionalities such as multiplexing data among different services or applications on a host (e.g., UDP), ensuring reliable communication in addition to multiplexing (e.g., TCP), and providing a human-computer interaction layer (e.g., HTTP).

Those services need security mechanisms that can operate on several of those layers. For example, we can have filtering and ACLs operating on layer 2 (MAC addresses) and layer 3 (IP addresses and subnets) and firewall rules operating on layer 4 features (e.g., ports and connection state). We can also have security functionality operating at layer 7, like access controls based

on HTTP data and IDSs. In the same way, we can have other functionality, like load balancing to ensure scalability and availability, operating on features provided by different layers. Another essential feature of a network infrastructure is having the ability to monitor network events and means to derive useful knowledge about them.

Given the requirements for modern network services (i.e., high performance, feature richness, flexibility and efficiency) and the constraints imposed by network "softwarization" (i.e., run on commodity servers), in this work we take a new approach to build systems covering several dimensions of network application needs; we do not see in-kernel, kernel-bypass processing, and SmartNIC offloads as unrelated or competing technologies. Instead, we investigate cooperation mechanisms among them to build optimal packet processing and monitoring systems covering several network layers. This is crucial, as we argue that different network functions inside an application have different processing and performance needs, and, at the same time, no technology can cover all the requirements needed by each of them.

To find an optimal mapping between different network applications requirements and available technologies, in Table 1.1, as a first step, we break down network application processing functions going from L2 to L7, monitoring features, and also functionality that is relevant to all applications (i.e., system and resource management features). On the left part of Figure 1.2, we show examples of each of those functions. After breaking down network applications in processing functions, we characterize their different processing requirements in terms of performance, functionality, capability and efficiency. In this work, we define each of these requirements as follows:

- **Performance**. Required behavior in terms of throughput and latency. "High" means high throughput/low latency, being required by time-sensitive functions that process each packet on a network. For example, L2 and L3 forwarding and L4-L7 functions like web servers and proxies.

- **Functionality**. Required ecosystem to support a given function. For example, the slow-path functionality required to provide bridging and routing for L2 and L3 is classified as

"high", as it depends on routing daemons, protocols like spanning tree and many configuration tools.

- **Capability**. Complexity in terms of logic required to support the function itself. L4-L7 processing is classified as "high" in this requirement, as processing protocols like TCP requires supporting congestion and flow control, retransmissions, data reassembly, and buffer management.

- **Efficiency**. Ability to perform the function while having a fair resource consumption profile (e.g., CPU and memory). In this work, we assume that high efficiency is a desired requirement for all processing functions.

Packet Processing Requirements				
Processing functions	Performance	Functionality	Capability	Efficiency
L4-L7	High	High	High	High
L2-L4 (slow path)	Low	High	High	High
L2-L4 (fast path)	High	Low	Low	High
System management	Low	High	High	High
Resource management	Low	High	High	High
Mon (counters)	High	Low	Low	High
Mon (sketches)	High	Low	Medium	High
Mon (stateful)	High	Medium	Medium	High
Mon (routing)	High	Medium	Medium	High
Mon (analytics/alerts)	High	High	High	High

Table 1.1: Processing needs for different network functions.

In the next step, we characterize each of the main technologies available to provide software packet processing, in terms of their ability to ensure the desired packet processing requirements (Table 1.2). SmartNICs (2.4) have limited CPU and fast memory resources, however they can provide high performance, for example, in packet forwarding applications, specially if a system can ensure that all processing will be done on the NIC as this allows avoiding PCI crossing overheads. They can also preprocess packets, reducing the load on hosts. XDP (2.3) allows high-performance programmable packet processing inside the Linux kernel. Its high performance comes from avoiding some kernel processing – which makes XDP oblivious of many kernel functionalities. To ensure

safety, XDP runs on a constrained environment, meaning that this may not be the right environment to build functions with high capability requirements. The Linux ecosystem (2.1) provides high functionality, capability and efficiency, however it suffers with low performance. This causes it to be inadequate for executing tasks with high-performance requirements. User space packet (2.2) processing technologies can ensure high performance and capability, but at the cost of disallowing resource sharing, having inefficient CPU consumption and being oblivious to Linux management tools and features.

Packet Processing Technologies				
Processing Features	Linux	User Space	XDP	SmartNICs
Performance	Low	High	High	High
Functionality	High	Low	Medium	Low
Capability	High	High	Medium	Low
Efficiency	High	Low	High	High

Table 1.2: Processing features for different packet processing technologies.

Now, it is clear that no packet processing technology can optimally host all packet processing and monitoring tasks. In this work, we use our observations from Tables 1.1 and 1.2 to provide enabling technologies to map applications to an optimal system implementation across these technologies and associated developer goals (Figure 1.2). To this end, as our main contributions, we propose three systems that synergetically integrate available packet processing technologies to optimize performance, efficiency and functionality of L2-L7 network applications and monitoring. In the following subsections, we give a brief overview of such systems.

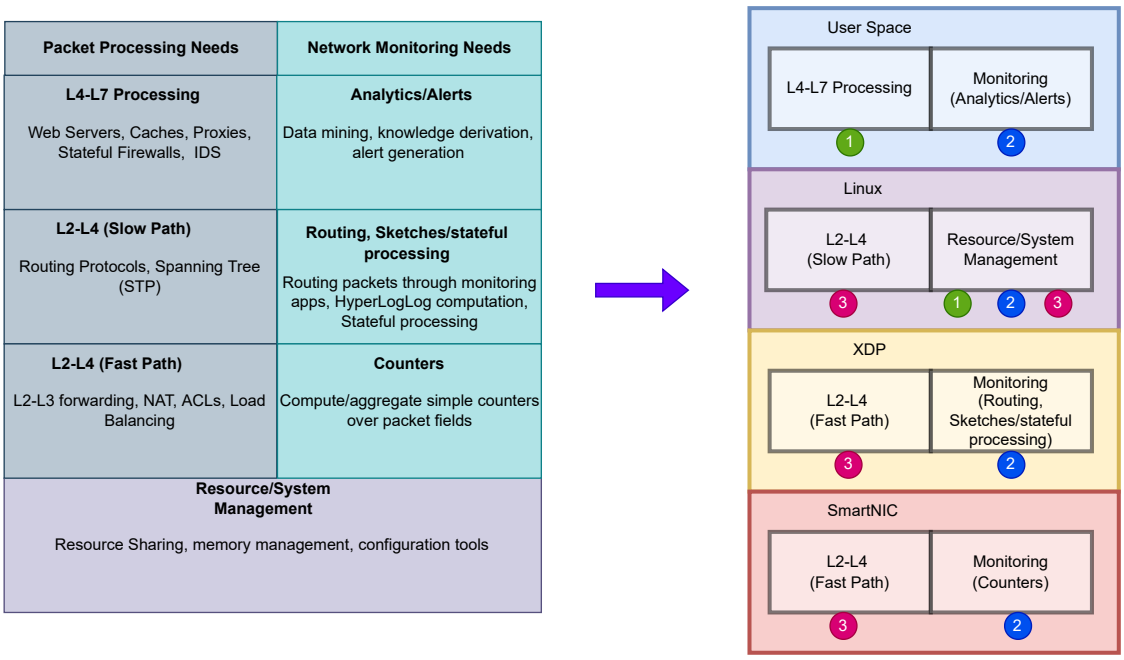

Figure 1.2: Network applications needs and optimal technology mapping through the following systems: (1) A Userspace Transport Stack Doesn't Have to Mean Losing Linux Processing. (2) Efficient Network Monitoring Applications in the Kernel with eBPF and XDP. (3) Getting back what was lost in the era of high-speed software packet processing.

1.1.1 L4-L7 Performance and Feature Richness

L4-L7 network services play a fundamental role on the modern Internet and datacenters. They provide functionality like web servers, caches, proxies, stateful firewalls and IDSs. Traditionally, those services run on top of L4 functionality provided by an operating system (e.g., TCP over Linux). However, Linux TCP is built with a series of inefficiencies like, for example, lack of connection locality, heavyweight data structures and system call overheads that cause its number of transactions per second to peak at 0.3 million while packet processing rates can reach several millions of packets per second [58].

To ensure high-performance, as already mentioned, a new trend is to build those services on top of kernel-bypass technologies. While this approach helps to avoid the Linux kernel overheads and inefficiencies, it brings some challenges inherent to losing Linux processing, as we discuss next.

1.1.1.1 Challenges with Current Approaches

As noted in systems like *mOS* [57] and *Microboxes* [69], kernel-bypass technologies can indeed provide good performance to L4-L7 applications. Those systems are built on top of a high-performance userspace TCP stack, called *mTCP* (multicore TCP) [58]. *mTCP* is designed with a series of optimization techniques to improve TCP scalability on multicore systems. Some examples of such techniques are lock-free, per-core cache-friendly data structures which in combination with a high-performance kernel-bypass packet I/O subsystem enable *mTCP* to be up to 320% faster than the Linux TCP stack.

However, gaining performance using this type of system comes at a price. The first issue is that kernel-bypass systems cannot leverage the packet I/O features provided by the Linux kernel (i.e., IRQs and NAPI). This prevents resource sharing as there is the need to dedicate NICs and CPUs to the packet processing application. To perform packet I/O, the dedicated CPUs need to busy poll the NIC queues, causing 100% CPU consumption all the time, even when the load is low. This has negative implications in terms of efficiency, power consumption and support to CPU intensive applications [17]. The second issue is that kernel-bypass applications are oblivious to other Linux functionality, like sub-L4 protocol implementations, security features and management tools. This requires reimplementing those protocols and security features in userspace, and also prevents kernel-bypass systems to leverage the rich management tools provided by Linux.

1.1.1.2 A Userspace Transport Stack Doesn't Have to Mean Losing Linux Processing [17]

To address the mentioned challenges, in Chapter 3, we introduce a new packet I/O subsystem to a high-performance userspace TCP stack (i.e., *mTCP*). This subsystem is built on top of a new high-performance socket type called AF_XDP [60]. This new type of socket allows sending raw packets directly from the XDP layer to user space [51]. This allows hybrid kernel/kernel-bypass L2-L7 programmability. The userspace TCP stack brings high-performance to L4-L7 network

functions and applications, while the in-kernel stack ensures efficient resource usage, lower-layer protocol processing, and ACLs. Consequently, our system avoids the need to reimplement countless networking features in userspace, and by having a better resource consumption profile (i.e., CPU), our system makes the userspace TCP stack more suitable to support CPU-intensive applications with high performance.

In our evaluation, we show that our approach enables building cooperation mechanisms between the XDP layer and the userspace TCP stack. For example, we protect a L7 application using a distributed denial of service (DDoS) defense mechanism built on top of XDP. We also show that by avoiding the need to busy poll the NICs, our system can have up to 64% more throughput for a CPU intensive application, comparing to the DPDK implementation.

1.1.2 High-Coverage Monitoring

Continuous traffic monitoring and analytics are fundamental to modern networks. They allow getting insights about the current state of network elements so that we can perform diverse tasks, such as detecting performance, security and availability issues allowing executing actions to drive the network to the desired state and also doing other fundamental tasks, like accounting for billing purposes.

One important aspect in this scenario is that different network conditions are manifested differently, so monitoring applications tend to be highly specialized. Consequently, to achieve high coverage, several of such applications need to be deployed in parallel, making monitoring tasks expensive.

1.1.2.1 Challenges with Current Approaches

Programmable switches play a significant role in this context, as some monitoring tasks can be offloaded to them [84, 50, 98]. However, as this type of equipment have a constrained programming model and limited resources, ultimately telemetry data and packets need to be processed by analytics applications in software [79]. Hence, those applications need to ingest and process millions

of packets. To allow high performance in this scenario, several network analytics frameworks rely on kernel-bypass technologies [8, 79, 102]. This makes monitoring expensive, as at least one CPU core and network interface needs to be dedicated to the monitoring application, making it infeasible to deploy systems alongside other applications and services, e.g., at the network edge. Added to that, the fact that we need to deploy several monitoring applications in parallel, we see the need for new mechanisms to efficiently ingest packets and orchestrate monitoring applications so that we can reduce their resource footprint while providing high performance.

1.1.2.2 Efficient Network Monitoring Applications in the Kernel with eBPF and XDP [18]

With the previous observations in mind, in Chapter 4, we introduce a new network monitoring framework that intelligently orchestrates the deployment and execution of monitoring applications. Our implementation leverages modern kernel-level packet processing (i.e., XDP), so we can avoid many issues related to kernel-bypass technologies, such as high CPU consumption and poor resource sharing capabilities. Orchestration is done by new shared network monitoring primitives that collect lightweight high-level metrics that may indicate that a condition of interest may exist. In this case, packets go through a deeper (and heavier) analysis that can confirm and get more details about a possible issue. In this way, we reduce the resource footprint of software network analytics, consolidating the logic that all monitoring applications require and only running them when needed. This enables avoiding processing packets that are not related to a critical issue, leaving processing power to perform analytics and extract important information that actually needs attention. This also avoids extra delays on packet processing due to in-band monitoring, which ensures a faster network to support applications. Finally, we show that an efficient division of work between SmartNIC offloads, XDP on the host and user space can ensure an even higher degree of efficiency, performance and functionality to our system. In our evaluation, we build three example applications; a SYN flood detector, a DNS flow analyzer and a traffic accounting application. Using those three applications, we show that our system can provide high-coverage

network monitoring with high performance and efficiency.

1.1.3 L2-L4 Performance and Feature Richness

The Linux network stack provides a rich set of L2-L4 functionality such as routing protocols, bridging, packet filtering and policies that drives the core network infrastructure in the cloud, edge, private datacenters, the Internet, and more. [55]. However, the Linux network stack is known to be heavyweight – one of the consequences of its generality, which allows it to provide a rich set of features for diverse use cases. With the advent of faster networks (i.e., 10-100 Gbps) the Linux kernel is proving to do too many operations per packet, preventing it to support such speeds [51]. A new technology called XDP provides a mechanism to build custom (and lighter) data paths inside the Linux kernel using eBPF (extended Berkeley Packet Filter). By being customizable, XDP data paths opens up the possibility to programmatically avoid some overheads inside the Linux kernel, but unlike full kernel-bypass technologies, XDP allows integration with in-kernel features, making it a good candidate to build feature-rich high-performance data paths.

1.1.3.1 Challenges with Current Approaches

Recently, several proposals have shown the value of building high performance data paths on top of XDP/eBPF [31, 44, 64, 77]. However, those works generally models the XDP data path as being mostly independent of the kernel, implementing custom functionality targeted to specific use cases.

While this opens up opportunities for innovation, in our vision, this approach introduces challenges. The first is that, we are starting to see non-standard communication mechanisms powering datacenter connectivity [31, 77]. This introduces complexities in troubleshooting as, for example, tracing tools need to incorporate system-specific knowledge to detect unexpected behavior. In addition, managing many non-standard network systems requires that operators learn specific nuances of unrelated technologies, which leads to steeper learning curves and may make the whole environment more susceptible to errors. Another challenge, is that many networking features, need

to be (re-)implemented in XDP/eBPF form. This is a missed opportunity, as the Linux kernel already provides rich forwarding and security functionality.

1.1.3.2 Getting back what was lost in the era of high-speed software packet processing

The need for high performance and custom software-based packet processing has resulted in decades of research. Common between them is that in order to obtain performance, these approaches bypass or replace the Linux networking stack. This, in turn, has the unfortunate consequence of sacrificing the rich and robust functionality within the Linux network stack and the ecosystem of management programs and control plane software that is built on top of the Linux interfaces and data structures. In chapter 5, we take the position that we should rethink the design of the Linux network stack to address its shortcomings, rather than creating alternative pipelines. This re-design would involve (1) decomposing processing into a fast path and slow path processing, where each has a different execution environment (a fast path is efficient and focused on throughput, a slow path is focused on completeness of functionality and state management), and (2) transparently creating a custom fast path dynamically that consists of only what is needed based on the current configuration of Linux.

This is seemingly counter to the monolithic design of Linux, but recent work enabled dynamic and safe loading of processing, so we can load only what is needed at that particular time, while allowing processing to interact with the Linux kernel. This enables is Linux, as exists today, to serve as the slow path. What is missing is a system to build the fast path. For this, we introduce TNA, a prototype system that is able to automatically generate a minimal data path, based on what is current in used on a Linux box, avoiding many of its overheads, ensuring high-performance while maintaining its rich set of functionalities. We show in our evaluation that TNA is able to transparently accelerate many Linux kernel network subsystems.

1.2 Outline

The remainder of this work is organized as follows. In Chapter 2 we present background information about the main technologies that we explore in this work to build feature-rich high-performance packet processing systems. In Chapter 3 we show how we build a system that synergetically integrates in-kernel and kernel-bypass processing to benefit L4-L7 network functions and applications. After that, in Chapter 4, we address the needs of many networked systems – a high-coverage monitoring system, carefully designed to ensure high efficiency and performance leveraging user-space, in-kernel and SmartNIC processing. In Chapter 5, we introduce a new system that enables having feature-rich and high-performance network stack. To this end, we propose to automatically break Linux packet precessing in a fast path and a slow path. The fast path is provided by XDP and can programmatically bypass some Linux processing, avoiding its overheads. The slow path complements the fast path, executing more complex tasks that need full stack processing and state management. Finally, in Chapter 6 we conclude this work.

Chapter 2

High-Performance Networking Overview

In this chapter, we briefly introduce the main technologies currently available to build high-performance network applications, before we continue to the next chapters, where we do a deep dive on systems that integrate the described technologies in order to provide optimal processing environments for different network application profiles and needs.

2.1 Linux Networking

The Linux kernel provides feature rich and secure network services. It has been in use for several years, in different contexts and scales. It has powered corporate networks, cloud, edge and telecom infrastructures [55]. This high degree of adoption, combined with the fact that Linux is an open source project maintained by the community, creates a synergetic environment where its capabilities and security are in constant progress and scrutiny. However, the features that enable its popularity are also a potential source of inefficiency. To cover this wide range of use cases, Linux implements several layers of functionalities, allowing it to support different kinds of hardware, network protocols and security features. This means that its network stack usually performs too many operations per packet and needs to rely on heavy-weight data structures for doing so (i.e., *sk_buff*), causing overheads that could be otherwise avoided if some features are not necessary [28, 51].

2.2 User Space Networking

To avoid those mentioned overheads, a recent trend is to build network solutions on top of kernel bypass technologies, like DPDK [8]. Kernel bypass networking enables building custom network applications in user space. This opens up the possibility to avoid the multiple layers of overheads present on the Linux kernel stack. However, this gain in performance comes at high costs. Bypassing the Linux kernel means losing many of its good features, like resource sharing, robust protocol implementations and security features. For example, an application built on top of DPDK have no access to the packet I/O features provided by the Linux kernel (e.g., IRQ handling and NAPI). This requires DPDK applications to busy poll the network interface card (NIC) queues, dedicating CPU cores to the packet processing application and causing 100% CPU consumption all the time, even if there are no packets to process. Another concern related to kernel bypass solutions is that they can not leverage the battle proven protocol and security features available in the Linux kernel. For example, if we are interested in building a L4 kernel bypass application, we need to add code support all the lower layer functionalities like ARP (Address Resolution Protocol) handling and IP (Internet Protocol). If we add to this, the necessity to re-implement security features already provided by Linux and the fact that the management tools available to Linux cease to work on kernel bypass applications, the challenges and inefficiencies brought by this scenario are obvious.

2.3 XDP

To strike a balance between the high performance provided by kernel bypass technologies, while enabling network applications to leverage some rich features available on the Linux kernel, recently, the Linux community introduced a new technology in the Linux kernel called the eXpress Data Path (XDP). This technology enables programmable packet processing, leveraging some good features of the Linux kernel while having performance close to kernel bypass. XDP enables safe and dynamic code injection at the device driver level on the host, or even directly on a SmartNIC itself. Dynamic code injection means that we can add new logic to XDP with no packet processing

interruption. Safe code injection means that the injected code will not crash the system. To ensure safety, XDP programs can be written in a restricted version of C and a compiler converts it to BPF byte code (which stands for Berkeley packet filter). The BPF byte code runs on a sandboxed environment inside the kernel called the eBPF VM (virtual machine). This byte code will only be loaded in the Kernel, if it passes some safety verifications done by the BPF verifier. The Verifier will ensure that the byte code does not have infinite loops, does not try to access unsupported instructions and do not perform out-of-bound memory accesses. XDP is fast, as it executes at the device driver level, or even on a SmartNIC that supports it. This enables XDP to take actions/decisions early in the network stack, avoiding overheads like *sk_buff* allocation and other unnecessary packet processing steps, depending on the use case.

With XDP we can deploy a customized data path that can take actions like rewrite packet headers, access some kernel functionality via kernel helpers, persist data using a data structure called BPF map and finally decide the fate of a packet using one of the many XDP actions. The XDP actions allow dropping packets, forward them to another interface, or to the same one as the packet was received, send the packet to be processed by the Linux kernel network stack, and more. XDP is fast, as it executes at the device driver level, or even on a SmartNIC that supports it. This enables XDP to take actions/decisions early in the network stack, avoiding overheads like *sk_buff* allocation and other unnecessary packet processing steps, depending on the use case.

2.4 SmartNICs

SmartNICs (smart network interface cards) enable programmable packet processing on the NIC hardware itself. It allows moving from the unflexible world of classic ASIC NICs, where functionality is hardwired on the NIC, to a more flexible one, defined by user provided logic. Several types of SmartNICs are available on the market today, and despite their different architectures, generally, they will provide hardware/software constructs for the following basic packet processing functionalities: hearders parsing/deparsing, packet classification, re-writing, scheduling and forwarding. They may also have a set of general purpose accelerators (e.g., for compression and

encryption tasks), and provide APIs for configuration and programmability. Currently, SmartNICs use FPGAs (Field Programmable Gate Arrays) [109], embedded CPU cores [3] or NPUs (Network Processing Units) [86] as their main computing substrates. They can be programmed in a variety of languages such as P4 [12], Verilog [16] (for FPGA NICs), C or eBPF [86, 27]. Being able to run eBPF leads to an interesting opportunity to create cooperation mechanisms between the SmartNICs, and software running on the host while using a common development environment and language. This enables sharing processing responsibilities between the NIC and the host, reducing load on the host, which ultimately leave more processing power to execute both packet processing and user applications.

Chapter 3

L4-L7 Performance and Feature Richness

While we cannot question the high performance capabilities of the kernel bypass approach in the network functions world, we recognize that the Linux kernel provides a rich ecosystem with an efficient resource management and an effective resource sharing ability that cannot be ignored. In this work, we argue that mixing kernel-bypass and in kernel processing can benefit applications and network function middleboxes. We leverage a high-performance user space TCP stack and recent additions to the Linux kernel to propose a hybrid approach (kernel-user space) to accelerate SDN/NFV deployments leveraging services of the reliable transport layer (i.e., stateful middleboxes, Layer 7 network functions and applications). Our results show that this approach enables high performance, high CPU efficiency, and enhanced integration with the kernel ecosystem. We build our solution by extending mTCP which is the basis of some state-of-the-art L4-L7 NFV frameworks. By having more efficient CPU usage, NFV applications can have more CPU cycles available to run the network functions and applications logic. We show that for a CPU intense workload, mTCP/AF_XDP can have up to 64% more throughput than the previous implementation. We also show that by receiving cooperation from the kernel, mTCP/AF_XDP enables the creation of protection mechanisms for mTCP. We create a simulated DDoS attack and show that mTCP/AF_XDP can maintain up to 287% more throughput than the unprotected system during the attack.

3.1 Introduction

Stateful middleboxes and Layer 7 (L7) network functions (NFs) are fundamental elements of modern networks and datacenters [57]. Stateful middleboxes are responsible for services like proxying, TCP splicing, stateful network address translation, firewalling, application load balancing, network intrusion detection systems (IDS), and content caching. These elements rely on services provided by the transport layer (e.g., TCP) to track Layer 4 (L4) state and inspect data content at the flow level (through data reassembly). With the increased pressure on the network for video conferencing, group collaboration [11], and digital entertainment [6], datacenters and cloud providers need to offer adequate infrastructure to support these trends. Given the dynamics and need for scalability, this trends towards software-based NFs, in the form of network functions virtualization (NFV).

Recent works [89, 58, 22, 51] have shown that it is currently hard for the operating system's kernel to provide the necessary performance to support these modern network services. This is mostly because today's operating systems add non-negligible overheads to packet I/O due to inefficiencies in data-structures and memory allocation (e.g., sk_buff, file descriptors, etc.), extra memory copies, unnecessary protocol processing, and an inability to react to microsecond scale bursts due to coarse temporal granularity in scheduling CPUs to the network applications threads.

To overcome these inefficiencies and enhance packet processing programmability, user space network processing toolkits, such as DPDK [8], have been introduced and are gaining in popularity. These toolkits give complete control of the networking hardware to the user space network processing application, enabling the development of high performance packet processing applications as many of the those kernel network stack inefficiencies can be avoided. Several projects have been built on top of DPDK [89, 22], and they generally show that this approach can indeed improve the performance of NFs and network applications substantially. Following this trend, mTCP [58] proposed a highly scalable user space TCP stack that is optimized to multi-core systems and able to outperform the Linux TCP stack by up to 320%. This created opportunities to L4-L7 net-

work functions frameworks [57, 69] to innovate in terms of functionality while also achieving high performance.

However, this gain in flexibility to build high performance network functions comes with costs. First, there is a need to dedicate CPU cores and network interfaces to the network application. Further, the application needs to busy poll the network interface queues in order to receive packets. Each of these causes high CPU consumption and leaves less CPU cycles for the network function logic. Second, as the kernel is completely bypassed, all the configuration, monitoring, security, and network (protocol processing, bonds, etc.) features provided by modern kernels are also bypassed - leading to NFs needing to completely re-implement them in user space.

Observing the challenges introduced by network kernel bypass technologies, the kernel community introduced a new programmable packet processing framework, called the eXpress Data Path (XDP) [51], that enables efficient and safe custom packet processing inside the kernel. The main idea of XDP is to provide custom programs early access to the packet (before the packet reaches the Linux kernel network stack), giving the XDP program the ability to modify the packet and also to define a verdict to it (including rewriting packets and applying drop or redirect actions). As XDP is part of the kernel, the packet processing application runs inside the kernel context, having access to some capabilities provided by it. With XDP there is no need to dedicate resources (e.g., CPU and network interfaces) to the packet processing application, and the XDP program can selectively make use of kernel services and therefore avoid the need to re-implement functionality.

On top of XDP, the kernel community proposed a new high-performance network socket type called AF_XDP [60] that enables sending raw packets received at the XDP layer to user space. This opens up the possibility of building hybrid packet processing and NFV solutions where packets can be processed by user space applications, but with cooperation and support of XDP, NFs can leverage all the integration and features that the kernel provides.

Leveraging the addition of XDP and AF_XDP to the Linux kernel, in this chapter we argue that accelerating the transport layer using a hybrid (kernel-user space) approach is of benefit to SDN/NFV deployments as this scenario can leverage the performance of a high performance user

space TCP stack without completely bypassing the kernel. This work is the first that we are aware of to introduce a high-performance TCP stack to AF_XDP. In particular, we make the following contributions:

- We propose an architecture and implementation that provides a hybrid packet processing model in which NFs can leverage both a high-performance user space TCP stack, and rich kernel functionality (Section 3.4).
- We demonstrate that this system is able to achieve a better CPU consumption profile that leaves more cycles to execute application and NFs code – e.g., for a CPU intense workload, mTCP/AF_XDP can have up to 64% more throughput due to the extra CPU cycles available. (Section 3.5)
- We demonstrate the benefit of hybrid processing where we show that a network application is able to maintain up to 287% more throughput in the face of a DDoS attack, through using filtering at the kernel level. (Section 3.6)

The remainder of the chapter is organized as follows: Section 3.2 describes related work and depicts the main challenges of their current architecture. Section 3.3 describes several scenarios that the hybrid kernel-userspace transport approach can benefit the network functions and applications world. Section 3.4 describes the architecture and implementation of the solution. In Section 3.5, we compare the hybrid kernel-userspace transport stack (mTCP/AF_XDP), with a full userspace transport stack that uses DPDK as the packet IO subsystem (mTCP/DPDK). We show in Section 3.6 the ability to protect network applications from DDoS attacks through the hybrid-processing. Finally, in Section 3.7 we conclude and discuss future work.

3.2 Related Work and Challenges

3.2.1 Kernel Bypass Approach for NFVs

The Linux operating system was designed to be as general as possible, and to support a wide range of applications and configurations. This means that the kernel network stack will perform costly processing and allocate heavy weight data-structures even if the packet only needs a few steps to processes lower-level protocols [51]. For example, for every packet that arrives, a data structure called *sk_buff* will be allocated to enable further protocol processing leveraging the rich semantic provided by this data structure [19]. After that, other costly processes (e.g., *__netif_receive_skb_core*) will be triggered to process the layered protocol stack, and to filter packets. As observed in [51, 19], this process slows down packet processing and could be avoided if it is possible to build custom packet processing applications that shortcut unnecessary processing.

To avoid these and other inefficiencies while providing maximum performance, modern NFs frameworks are built on top of kernel bypass technologies [58], [57], [59], [69]. This approach has the benefit of allowing highly customizable packet processing applications to avoid, for example, unnecessary protocol processing and provide more efficient processing pipelines.

mTCP is a high-performance user space TCP stack that is built on top of kernel bypass technologies, to improve performance, and in turn, can leverage multicore systems to improve scalability. To enable multi-core scalability mTCP is built with a series of optimization techniques (e.g., lock-free, per-core cache-friendly data structures). The mTCP process operates by running as distinct threads (one for the application and one for the mTCP logic) on each CPU. mTCP leverages RSS to distribute incoming packets from different flows among different CPU cores, while handling core affinity. Being a user level implementation, mTCP decouples the TCP logic and development from the kernel complexity, which smooths the development of new features to the stack itself and enables building new solutions on the top of it. For example, mOS [57] is a framework built on top of mTCP that allows building stateful middleboxes with full support for L4-L7 processing. mOS currently supports DPDK [8] and Netmap [93] as packet I/O subsystems.

3.2.2 Challenges of the Current Approach

As we saw in the previous subsection, several network function frameworks have benefited from the performance enabled by the kernel bypass approach. However, it is important to recognize possible limitations of a complete kernel bypass approach and look for new opportunities to evolve the current solutions. In the next paragraphs, we will list the limitations that motivated this work.

Inefficient CPU usage: Being a kernel bypass framework, DPDK does not rely on the kernel networking mechanisms to receive packets (e.g., interrupts, ksoftirqs and NAPI) [51, 60]. Instead, DPDK needs to busy poll the NIC queues in order to receive packets. Although this mechanism can provide better latency profiles to applications, this causes high CPU consumption, leaving less cycles to process the NFs logic, as we will demonstrate in Section 3.5.

Lack of system integration: The Linux kernel has a rich ecosystem to provide network connectivity, monitoring, configuration, resource sharing, isolation, and security. Generally, kernel bypass technologies are blind to this ecosystem [19], which may slow down progress in this context as much of this functionality needs to be re-implemented in user space. As we will see in the next sections, a better alternative is to selectively use kernel functionalities, while still leveraging the high performance and flexibility achieved by user space technologies.

3.3 Motivation

In this work, we ask if we can leverage the recent additions to the Linux kernel to address the challenges listed in Section 3.2 and benefit a high performance user space TCP stack [58], which in turn extends to the NFs frameworks built on the top of it that provide for processing capabilities all the way up to the application layer [57, 69].

XDP enables flexible and efficient programmable packet processing inside the kernel [51]. The key enabler for XDP is the eBPF virtual machine, that allows only verifiable eBPF code to be loaded inside the kernel. XDP enables attaching eBPF programs to process packets at the earliest point inside the kernel (i.e., at NIC driver level, before the packet reaches the kernel network stack).

If the NIC supports it, the eBPF programs can be offloaded to the NIC hardware. XDP programs allow, for example, rewriting packet headers and accessing packet metadata (e.g., queue number on multiqueue NICs and custom metadata). The XDP hook execution finishes by assigning a verdict to a packet. Possible verdicts are to drop the packet, transmit the packet back on the same interface as it arrived, pass the packet to be processed by the kernel stack, and redirect the packet to another interface (physical or virtual), another CPU for further processing, or even to a special socket that sends the packet to user space (i.e., AF_XDP).

AF_XDP is another addition to recent Linux kernels. It enables sending raw packets to user space at high-rates through zero-copy transfers (as long as the NIC driver supports this [60]). To send and receive packets, AF_XDP interacts with the kernel via specialized rings (i.e., fill, completion, Tx and Rx rings), and uses a special memory area called UMEM. Those rings are used by the userspace network application and kernel to switch control of UMEM areas (which stores packet data) between each other (i.e., fill and completion rings). They are also used by the application to receive packets, and inform the kernel the packets that should be sent (i.e., Rx and Tx queues).

We use XDP and AF_XDP to provide a new packet I/O subsystem for mTCP. This new subsystem provides an efficient CPU consumption profile for mTCP applications and NFs, and also provides better system integration.

Providing efficient CPU consumption: Middleboxes need to perform packet I/O, but they also need available CPU cycles to process the network function's logic, and as we will see in Section 3.5, this is a challenge for DPDK. On the other hand, XDP enables a better CPU consumption profile as it does not need to rely on busy polling to perform packet I/O, because it has the Linux interrupt infrastructure and syscalls available.

Providing system integration: While kernel bypass packet I/O systems like Netmap [93] may bring a better CPU consumption profile, it lacks good Linux system integration. For example, currently Netmap is not part of the Linux kernel, so it may be a burden to maintain Netmap based applications [51]. Moreover, it does not support XDP, which limits its data-path programmability.

XDP does not take over the ownership of the NIC as DPDK, so it is possible to share the interface among multiple applications (providing the necessary XDP/eBPF logic). It is also possible to use the Linux network configuration and monitoring tools like ethtool, iproute2 which may easy the integration of XDP based network solutions with automation tools like Puppet, Ansible and Chef. Container technology plays an important role in NFV deployments [59], and as it relies heavily on kernel functionalities to provide resource isolation and configuration, XDP based deployments can easy the integration of fast packet processing and enhanced networking capabilities to the containers world. As DPDK completely bypasses the kernel, enabling these functionalities to DPDK based applications is challenging [14].

Leveraging the support of the rich kernel ecosystem: As AF_XDP sockets can send raw packets to userspace after they are processed on the XDP hook, they enable a hybrid networking stack approach. XDP can be used as a first layer that provides enhanced network functionalities to the high performance transport layer running in user space. This first layer can be used to protect the upper user space stack [5] and also to provide kernel integration functionality leveraging BPF maps and kernel helper functions [51, 19]. Kernel helper functions can be used, for example, to support packet checksum calculation and also to access kernel routing tables [10, 19]. BPF maps can be used by the XDP redirect logic to react to events occurring at different kernel subsystems and different resource monitoring points, including in user space (e.g., CPU load and cgroups) [51, 13] opening up opportunities to create new load-balancing mechanisms. Furthermore, XDP can also provide a flexible mechanism to implement access control lists (ACLs), packet filters, and other functionalities to protect the user level transport layer.

This approach brings flexibility to NFs and applications that leverage high performance user space transport stacks, as the XDP logic allows selecting only the needed kernel network functionalities to be used. It does this via kernel helpers and does not require the packet to traverse the whole Linux network stack.

3.4 Architecture and Implementation

3.4.1 mTCP/AF_XDP Integration

Now that it is clear the motivation behind having a hybrid kernel-userpace TCP stack, we present the architecture of the mTCP/AF_XDP stack in Figure 3.1. This figure shows the basic interactions between the different components of the solution, as we will explain in the next paragraph. Notice that to obtain maximum performance, we decided to have one UMEM per AF_XDP socket, and also one AF_XDP socket per mTCP thread, so we could completely avoid synchronization overheads and obtain maximum performance.

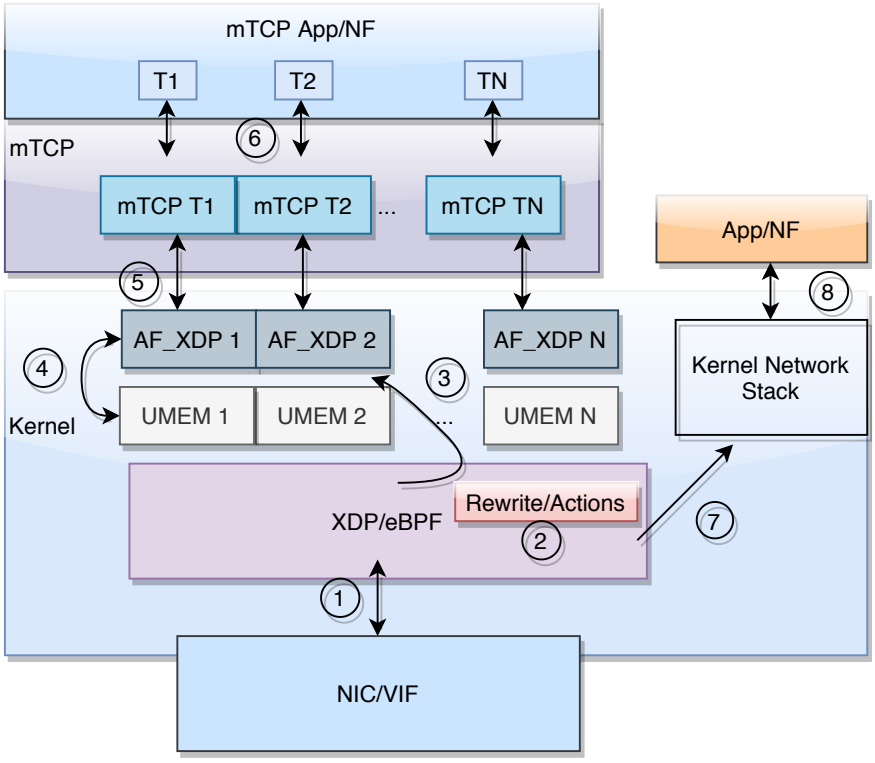

Figure 3.1: mTCP/AF_XDP architecture.

The life of a packet inside mTCP/AF_XDP (see Figure 3.1): The XDP program ②, decides to which AF_XDP socket a packet should be sent. In our implementation, we use hardware packet steering, and the NIC queue in which the packet arrived ① is used as the index in the BPF map to select the target AF_XDP socket for the packet ③. We avoid extra cache overheads by

pinning a mTCP thread on the same core that handles the *ksoftirq* on behalf of a packet received by the multi-queue NIC. The kernel places the packet on the UMEM area using one of the addresses available on the fill ring associated to that socket (if the NIC driver supports zero-copy, the NIC will place the packet at UMEM via DMA). After that, the kernel places a file descriptor on the socket RX ring. The mTCP/AF_XDP packet I/O subsystem uses the *(*poll) system call to monitor this ring ⑤, and our implementation enables sending and receiving packets in batches for best performance. The batch of packets is received by the mTCP's stack main loop which performs the TCP stack logic, and makes data available to the application threads through the mTCP events system and userspace function calls (e.g., mtcp_read) ⑥. After successfully receiving the packets mTCP/AF_XDP returns ownership of these UMEM areas to the kernel by posting their descriptors in the fill ring.

The sending path is similar. The mTCP/AF_XDP packet I/O subsystem uses the TX ring to place file descriptors pointing to the packet buffers it wants to send ⑤. The kernel then assumes control of this UMEM region and sends the packet to the NIC which sends the packet out. After successful transmission, the kernel makes this UMEM area available for sending new packets by posting a memory descriptor on the completion ring ④. mTCP/AF_XDP consumes these descriptors and uses them on the next iteration of the packet sending routine. Finally, the XDP program may also be customized to provide extra functionality (e.g., stack protection features), or even to redirect the packet to an NF or application using the Linux kernel network stack, allowing coexistence ⑦, ⑧.

To implement this solution, we added about 500 lines of C code to the mTCP code base. We have specific eBPF/XDP code (afxdp_kern.c), which is responsible for sending/receiving the desired packets to/from the AF_XDP sockets, making them available at the mTCP layer. We leverage the modular mTCP packet I/O design, to add a new packet I/O module (afxdp_module.c) and make targeted modifications to other mTCP components to support it. The code is available in our mTCP fork [2], and we expect to merge it to the main mTCP repository soon.

3.4.2 NFV Deployments

We envision that L4-L7 network functions built on top of mTCP based frameworks (e.g., [57, 69]) can benefit from our proposal by leveraging cooperation scenarios with XDP/eBPF (see section 3.6 for an example), the high performance provided by mTCP [57, 69] and the better CPU consumption profile enabled by our solution - see section 3.5 - (which will ultimately produce more resource and power efficient solutions). In this scenario, NFs like L7 caches, protocol accelerators, and IDS can leverage services provided by XDP/eBPF to control how packets flow, determining, for example, which packets should be sent to a specific NF, which of them should be sent to the Linux stack (e.g., to handle corner cases), which packets should be dropped and which of them should be routed to the next hop or forwarded to an application in the case the flow does not need NF processing. Interactions between the userspace NFs and XDP/eBPF should occur via eBPF maps, and enhanced integration with the kernel should be achieved through the kernel helpers available to the XDP layer.

3.5 Evaluation

To demonstrate the value and feasibility of the proposed approach, in terms of performance and added functionality, in our evaluation we answer the following questions regarding using an AF_XDP based packet I/O subsystem on a high performance user space TCP stack:

- Can the proposed system provide high performance?
- Can we have a better CPU consumption profile that enables more CPU cycles to be consumed running application code?

Experimental Setup: To answer these questions, we set up two testing environments on Cloudlab Wisconsin [15], one for mTCP/DPDK and another for mTCP/AF_XDP. Each environment is composed by 1 physical server machine that runs mTCP code (type c220g5 [15], Ubuntu 18.04.1 LTS Kernel 5.3.0-61-generic) and 5 physical client machines (type c220g1 [15], Ubuntu 18.04.1 LTS Kernel 5.3.0-61-generic). The server machines run the HTTP server that ships with

mTCP (*epserver*). The client machines run *ab* (Apache Benchmark). As each client host has 16 cores available, we run 16 instances of *ab* on each host. We have observed that in our setup, each client *ab* instance has maximum performance when sending 50 parallel HTTP connections, so we use this configuration on all tests. The server machines have 2 sockets with 10 cores each, and each socket is attached to one NUMA node. These machines have only one dual-port 10 GbE NIC attached to NUMA node 0, so we only report results for threads running on the processor on the first socket. We used the I40e Intel NIC driver, and all of the AF_XDP experiments use zero-copy mode. Also, we observed that DPDK performs poorly when the number of cores dedicated to mTCP is not a power of two (we do not observe this limitation on the AF_XDP implementation). So, to have a fair evaluation, we run our experiments using up to 8 cores on the server machines. In this work, we did not implement hardware TCP checksum offload for mTCP/AF_XDP, so we also report results for DPDK with it disabled (referred to as DPDK on the labels). For maximum performance, we disable hyper-threading and CPU power saving for each core. To isolate these cores, and avoid the kernel scheduling other user level threads on them, we use the *isolcpus* statement at boot time.

The Spectre and Meltdown mitigations affected the performance of eBPF programs, so AF_XDP is also impacted [60]. Users in controlled environments, where only trusted code can be executed, may opt to disable related mitigations. As we saw maximum performance for mTCP/AF_XDP when we disabled the mitigations, we include this scenario in the results on Figure 3.2. In our experiments, DPDK performance does not seem to be affected by those mitigations, so we only include these results for AF_XDP. As we cannot expect that all environments to be controlled and only run trusted code, for all other experiments we leave the mitigations enabled. In [60] the authors proposed a socket option called *XSK_ATTACH*, that automatically loads a minimal XDP code that only redirects packets that arrive at a *queue_id* to an AF_XDP socket avoiding the user to have to provide a custom XDP code. This code is optimized and minimizes the impacts of the mitigations. In our tests, we do not use this socket option, as we want to maintain the flexibility of having custom XDP code in our hybrid stack.

For each test, we set up each client instance to send 1 million requests (50 in parallel for each instance). Unless specified differently, in each test we use all five client hosts with 16 *ab* instances and each client instance sends HTTP requests to download a 64B file from the server. Each test is repeated 5 times, and we report our results using the average of each metric and standard error (although the bars are too small to be noticed on the graphs). In [58], the authors show that mTCP can outperform the Linux TCP stack by several orders of magnitude (up to 25 times for small messages), so we do not include Linux TCP in our evaluation. Finally, we use Linux *Perf* tool to analyze the overheads of each implementation and other metrics that may affect their performance (e.g., number of CPU cache misses, context switches and so on).

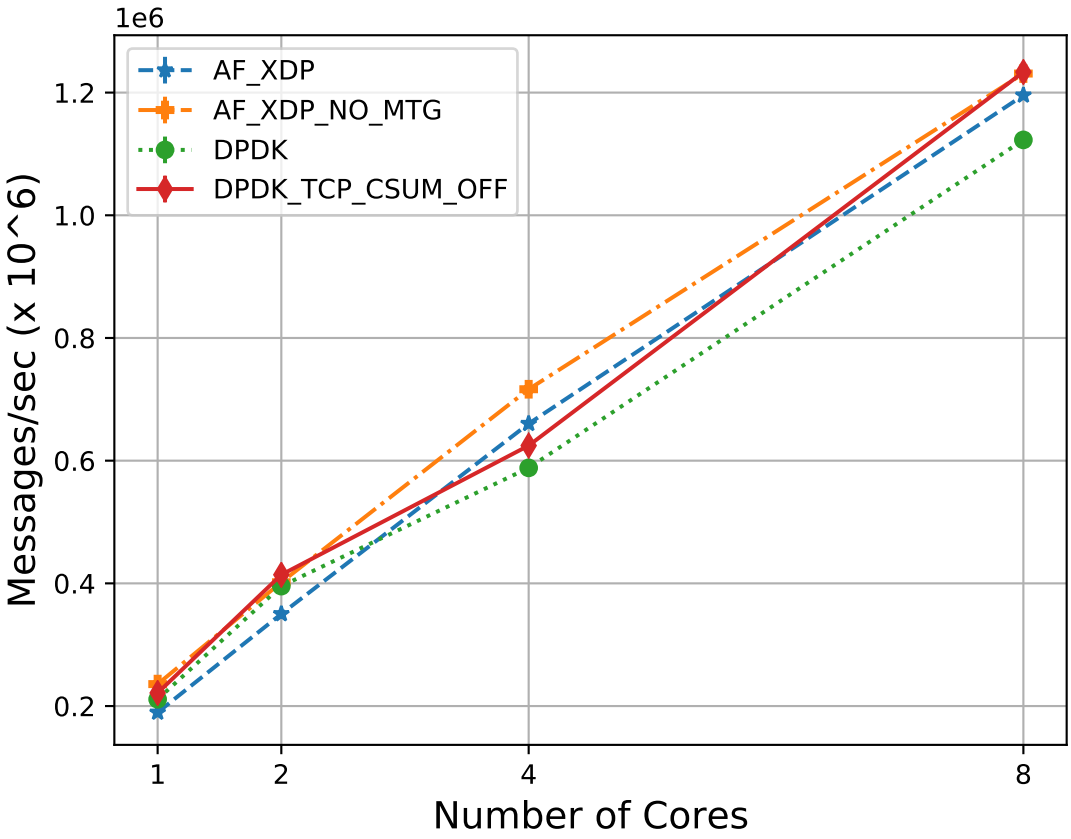

Figure 3.2: Different number of cores.

Raw performance evaluation: To demonstrate that mTCP/ AF_XDP can support high

performance and scale, in the first experiment we compare mTCP's core scalability for mTCP/DPDK and mTCP/AF_XDP. We can see in Figure 3.2 that mTCP's throughput scales almost linearly with the number of cores for all implementations. mTCP/DPDK with HW TCP checksum offload enabled (DPDK_TCP_CSUM_OFF) has the best performance for 2 and 8 cores, with AF_XDP with mitigations disabled (AF_XDP_NO_MTG) having performance almost as good as it for 1, 2 and 8 cores. We observe that for 4 cores, mTCP/DPDK has some drop in performance. Because of that, mTCP/AF_XDP outperforms mTCP/DPDK for 4 cores. It is important to notice that as we do not implement hardware TCP checksum offload to AF_XDP, the application has to spend CPU cycles to calculate it. In fact, we observe that when we disable the hardware TCP checksum offload on mTCP, it can spend up to 8% of its processing time performing those calculations. Observing these results, we expect mTCP/AF_XDP to improve its performance as we integrate hardware TCP checksum offload for mTCP/AF_XDP, which we leave as future work.

Efficient CPU consumption profile: In this experiment, we evaluate if mTCP/AF_XDP can provide an efficient CPU consumption profile (figures 3.3 and 3.4) and the effects of having more CPU cycles available to execute application code (figure 3.5). We can see in Figure 3.4 that mTCP/AF_XDP gradually increases CPU consumption as the number of client hosts and messages per second increase. In contrast, mTCP/DPDK relies on busy polling to receive packets, so it always consumes 100% of CPU. mTCP/AF_XDP does not rely on busy polling to perform packet I/O, which saves precious CPU cycles that can be spent to run the mTCP stack and application code. We analyze where each implementation spends more time, and we observe that mTCP/DPDK spends non-negligible time on the receiving path busy polling loop. In contrast, mTCP/AF_XDP spends more time executing application code and also handling important events on the mTCP stack. We can also observe in this figure that mTCP/AF_XDP does not hit 100% CPU consumption for 5 client hosts, even though the server is saturated. This is because our CPU measurements start when there is no load on the server, and goes until all clients finish sending the HTTP requests, so at the end of the experiment there is also a drop in load and this reflects on the average CPU consumption in this test.

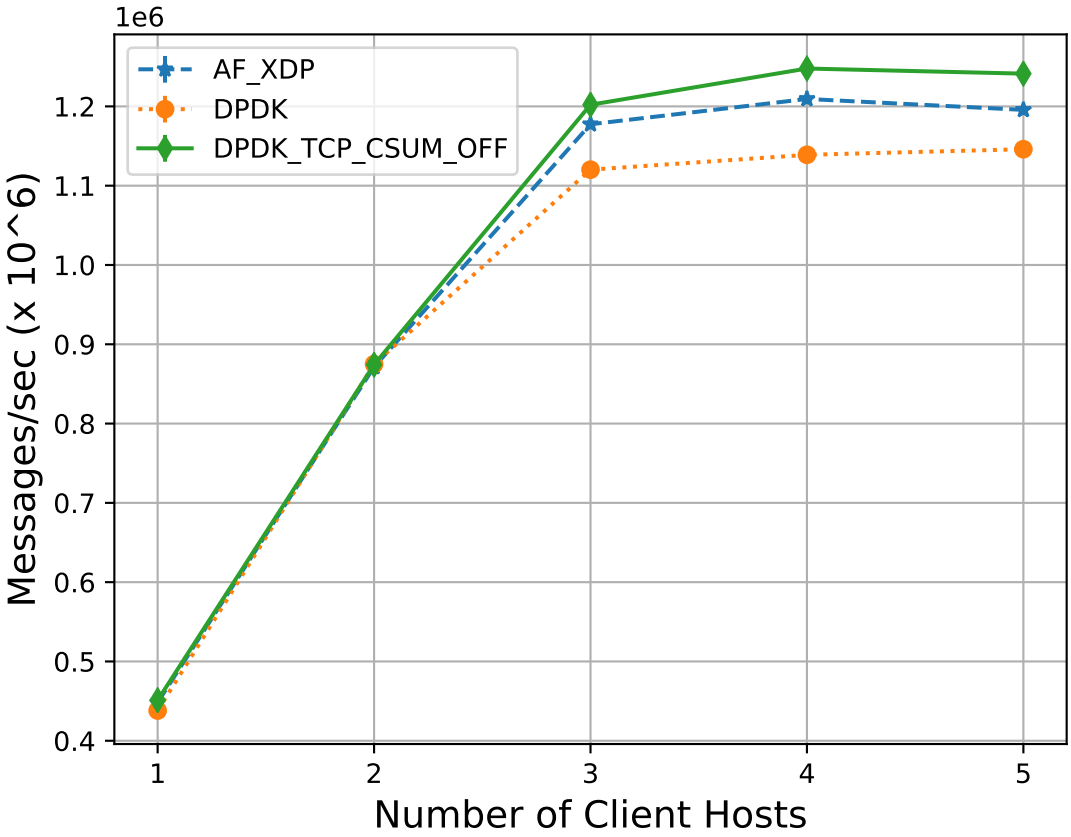

Figure 3.3: Throughput vs Number of Clients.

To observe the mTCP/AF_XDP benefits of having more available cycles to process application logic, we add a simulated CPU intensive HTTP application by executing a function to find all the prime numbers smaller than a given X in each HTTP request. The results can be seen in Figure 3.5, which shows the normalized throughput using mTCP/DPDK with HW TCP checksum enabled as the baseline. We start the server with 8 cores. To find primes lower than 100, the application does not get CPU bound enough for the mTCP/AF_XDP benefits to be perceived. But, as we increase X above 200, we observe that more CPU power is needed to find the prime numbers, and in this scenario mTCP/AF_XDP can have up to 64% more throughput than mTCP/DPDK (for $X = 800$).

Having more CPU cycles available may benefit mTCP to run CPU intensive applications (e.g., node.js) and network functions logic, and also enables mTCP to better support SSL/TLS.

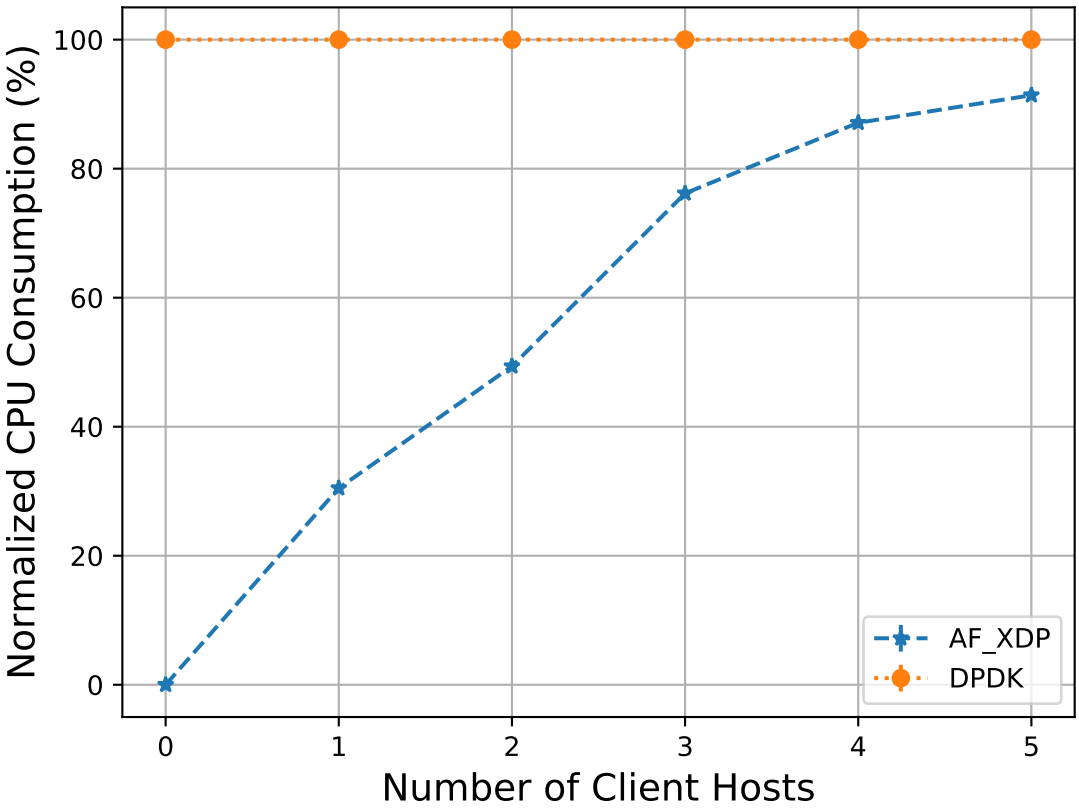

Figure 3.4: CPU consumption vs number of Clients.

3.6 Protecting the userspace TCP stack

As we have shown in section 3.3, one of the benefits of mTCP/AF_XDP is the possibility to leverage XDP/eBPF to enhance and protect the mTCP stack. To show this, we implement a simulated DDoS attack and an XDP DDoS protection similar to the ones described in [51] and [5]. In this experiment, we use 4 of the 5 client hosts to generate UDP packets targeting the mTCP server. Each attacking host uses 16 *nping* instances (one for each core) to generate UDP packets at a rate of 10 thousand packets/second. We gradually increase the intensity of the attack by joining new client hosts to the attack. The other client host runs *ab* to generate HTTP requests to the server. We start the mTCP servers on a single core to make the attack more pronounced.

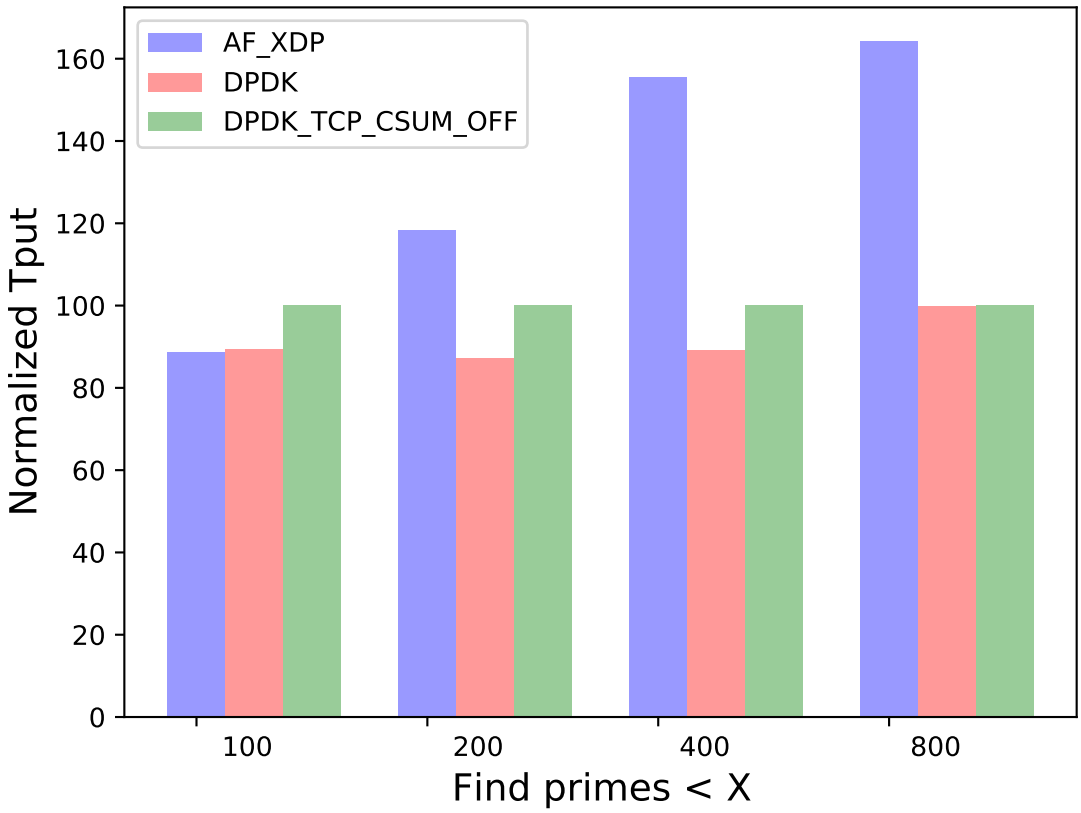

Figure 3.5: CPU Intensive Workload.

As mTCP is a user space TCP stack, all the security and isolation mechanisms provided by the Linux kernel are bypassed, so mTCP is responsible for dropping the UDP packets. This is not the case with mTCP/AF_XDP. To protect the mTCP stack from this attack, we change the XDP/eBPF code that sends the packets to the AF_XDP sockets (see Section 3.4) to parse each received packet and if it is a UDP packet, apply the XDP_DROP verdict, and otherwise send the packet to the AF_XDP socket, so it can be normally processed by the mTCP stack.

Figure 3.6 shows the impacts of the attack. In this experiment, we measure the total HTTP requests completed for each attack intensity. For mTCP/DPDK with hardware TCP checksum offload enabled, mTCP's throughput can drop 3.9 times when the attack is on its maximum intensity. At the same time, mTCP with XDP DDoS protection is able to maintain 2.87 times more through-

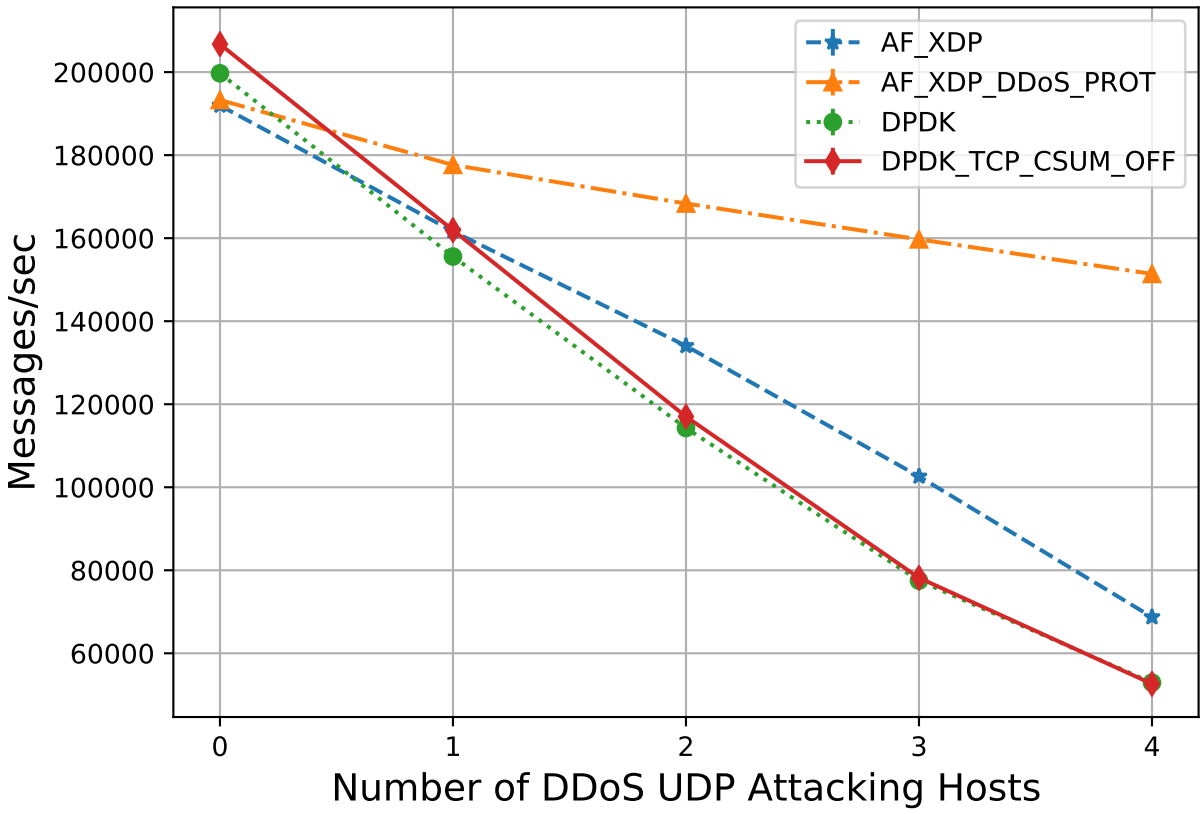

Figure 3.6: XDP Protection to DDoS Attack.

put than mTCP/DPDK versions when the attack is at its maximum load. It is interesting to notice that mTCP/AF_XDP with no DDoS protection (AF_XDP label) can handle the attack better than mTCP/DPDK versions. This is because mTCP/AF_XDP has more CPU cycles available to drop the malicious packets.

By applying this XDP protection mechanism, we free mTCP threads from the burden to process the malicious UDP packets, so the impact of the attack is minimized.

3.7 Conclusion and Future Work

In this work, we have enabled the power of eBPF and Linux system integration to cooperate with a high-performance user space transport layer. We have shown that this approach can have performance compatible with a high-performance kernel bypass approach, but providing enhanced capabilities that come from the OS kernel. This opens up the opportunity to innovate L2-L7 network functions in terms of functionality, deployment, performance and security.

One avenue for future work, is to enable TCP hardware checksum offloading for mTCP/AF_XDP. We expect that this will greatly improve mTCP/AF_XDP's performance, as we described in Section 3.5. Another opportunity is to work on NF stacks built on top of mTCP ([57, 69]) to investigate cooperation scenarios between XDP/eBPF and network functions such as L7 caches, protocol accelerators, and IDS (e.g., advanced forwarding mechanisms, eBPF hardware offloads, etc.). In this context, it is interesting to investigate new NF deployment scenarios, for example, how can the NF containers world leverage the enhanced networking capabilities provided by XDP while providing state-of-the-art high-performance L4-L7 services.

Chapter 4

High-Coverage Monitoring

Continuous traffic monitoring and analytics are fundamental to the operation of today's networks. Network telemetry allows for performing fine-grained analytics on network flow or packet records for various use cases including intrusion detection and traffic engineering. While some analytics tasks can be offloaded to programmable switches, ultimately, telemetry data needs to be processed by analytics applications in software. These applications are highly specialized, and running many such applications concurrently to achieve high coverage is expensive. To reduce the resource footprint of software network analytics, we present a novel network monitoring primitive that consolidates logic which all monitoring applications require. The primitive can (partially) be offloaded to a SmartNIC and triggers applications only when required based on high-level traffic metrics, avoiding unnecessary and redundant computations. We identify eBPF and XDP as a natural fit for this task, and implement a prototype of our system on top of this novel technology. Our evaluation shows that the combination of conditional execution of analytics tasks and the use of modern packet I/O technologies not relying on expensive busy polling (e.g., as in DPDK) significantly reduces the resource footprint of performing continuous network analytics.

4.1 Introduction

Continuous, fine-grained traffic monitoring is essential to the operation of today's reliable communication networks. In a nutshell, network monitoring and analytics describe the process of extracting information from network devices in the form of statistics or traffic records and

transforming this data into meaningful insights to be used for network management decisions. This enables operators to detect changing demands, performance issues, or attacks and subsequently reconfigure the network or scale network functions [84, 50, 99, 106].

In today's networks, switches and routers continuously export data about the traffic that traverses the network to analytics applications running on general-purpose servers [99, 98]. These applications detect problems or calculate metrics for a variety of use cases. Programmable switches allow for some applications to be partially offloaded to the network [84, 50, 98].

Data centers and wide area networks carry hundreds of millions of packets per second, requiring significant processing performance to enable fine-grained analytics [79, 68]. Offloading parts of the analytics pipeline to programmable switches can significantly reduce the load of the software-based stream processing backend; but even then, the rate of events to be analyzed in software is often still on the order of several million events per second per application [50]. This is because many, especially complex or state-intensive, tasks can only be partially offloaded due to the limited memory and compute resources and constrained programming model offered by line rate hardware [79]. As a result, performing analytics in software using either general-purpose stream processors or specialized packet analytics frameworks is indispensable to deploying complex, parallel, and dynamic network analytics.

Deploying network-wide, fine-grained, software-based analytics in a resource-efficient manner is still a major challenge. In particular, we identified two main issues with the state-of-the art in software-based network analytics. First, operators need to run multiple different network analytics tasks in parallel in order to achieve high coverage across possible failures and attacks, and to have continuous, detailed insight into different aspects of the operation of their infrastructure [98]. Running multiple applications in parallel at all times is costly. Many network conditions (e.g., attacks), however, can be identified by shared lightweight logic and simple, high-level metrics (e.g., overall connection count) that can then be used to trigger finer-grained analysis only when required.

Second, most existing network analytics systems leverage kernel-bypass frameworks for packet input [8, 79, 102]. Kernel-bypass can achieve high packet rates, but is expensive as at least one CPU

core is always entirely dedicated to packet input alone due to busy polling on the network interface card (NIC) [8]. This is wasteful as these CPU cycles cannot be used for the actual task of performing analytics. While receiving high volumes of packets via sockets is also inefficient or impossible, the novel eXpress Data Path (XDP) [52] technology not only provides a resource-efficient way to ingest millions of packets per second without using busy polling; it also provides abstractions particularly suited for orchestrating multiple packet processing applications and efficiently running them in parallel.

To address these challenges, this chapter presents a network monitoring architecture designed around a novel primitive which consolidates logic all monitoring applications require. The XDP technology is a natural fit to realize our architecture, and we show that the resulting system provides several performance and architectural advantages over the state-of-the art. Our work makes the following contributions:

(1) We identify the opportunity to consolidate monitoring system tasks in a novel network monitoring primitive. The primitive efficiently computes high-level metrics and can (partially) be offloaded to a SmartNIC.

(2) We propose an architecture for network analytics systems that allows for dynamic orchestration of analytics applications based on high-level metrics and policies.

(3) We implement a prototype of this architecture on a Netronome NFP SmartNIC [86] and for the Linux kernel using eBPF and XDP to demonstrate its feasibility.

(4) Using benchmarks, we show the high performance and small resource footprint of our approach using three example applications.

In the remainder of this chapter, we motivate our work by elaborating on the challenges in software network analytics in Section 4.2. We then present an architecture for efficient and practical network analytics in the kernel in Section 4.3. Section 4.4 describes our prototype and challenges

we encountered during its implementation. We demonstrate the performance of our system in Section 4.5 before discussing related work and concluding in Sections 4.6 and 4.7, respectively.

4.2 Motivation

As previously described, two main challenges in operating software-based network analytics are related to (a) reducing the system resource footprint when running analytics tasks in parallel and (b) efficiently ingesting and processing high rates of network records. We will now elaborate on both challenges.

Efficiently Deploying Parallel Applications. Analytics applications are usually highly specialized and focus on one particular type of scenario, such as a specific attack or common performance problem [84, 50, 106]. Accordingly, applications must be used in parallel to achieve high coverage across monitoring tasks such as intrusion detection [99], analyzing performance issues [84, 102], or traffic classification [87]. Even monitoring a network for just the most common attacks or anomalies therefore requires continuously running many specialized applications thus incurring high cost [98].

Despite the heterogeneity of analytics tasks, all share common tasks and logic. All applications read in network records from a NIC and decide which records carry measurements and which are control (or other) traffic. Then, especially those applications detecting some condition (e.g., an attack or performance anomaly), are often designed to be triggered based on shared, high-level metrics involving packet, Byte, or flow counters [30]. Such metrics may be used for several applications allowing for deduplicating logic and dynamically enabling and disabling more expensive, finer-grained analysis. As a result, (1) performing shared tasks, (2) computing metrics relevant to all applications, and (3) conditionally executing applications based on these metrics can be consolidated at the system-level; an overview of this is shown in Figure 4.1.

For example, a SYN flood toward a particular host would manifest itself not only in the particular pattern of a high amount of unanswered SYN+ACK segments, but also initially in an increase of overall flows. This basic higher-level metric can efficiently be computed on all

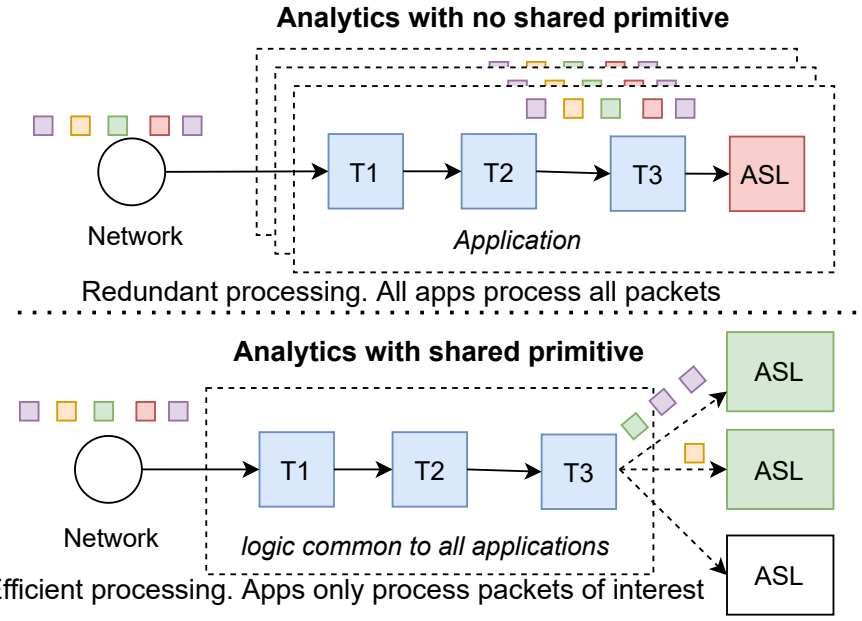

Figure 4.1: Efficient analytics with shared primitive. T1: Receive and select records, T2: compute high-level statistics, T3: conditionally execute app specific logic, ASL: Application-specific logic.

traffic using, for example, probabilistic data structures. A change in a metric can then trigger the activation of a series of more fine-grained analytics applications that are designed to mine the required and more useful information, such as the origin of the attack to subsequently configure filtering. Today, we are missing an architecture including a common primitive that consolidates tasks needed by all analytics tasks and enables the use of high-level metrics to dynamically enable, disable, and orchestrate downstream analytics applications.

Efficiently Ingesting High-volume Packet Streams. To cope with high traffic rates in software, existing software frameworks leverage kernel-bypass technology (e.g., by using DPDK [8]) to ingest network packets at high rates [79]. While this provides high input rates, the use of busy polling in these implementations causes significant CPU consumption, leaving fewer cycles for actual analytics. Furthermore, using kernel-bypass renders all in-kernel network processing capabilities (e.g., use of routing tables, firewalls, sockets) useless on the particular NIC in use and makes integration with other legacy applications challenging or impossible.

Both issues are especially problematic for distributed telemetry frameworks, such as Switch-

Pointer [102] where in-network measurements are processed and stored on all hosts in the network. Using kernel-bypass here would unnecessarily waste CPU cycles on all servers. Also, in this architecture, the telemetry sinks are not dedicated analytics servers and must also perform their regular purpose requiring NIC access via sockets and kernel network processing.

To enable high-performance user-defined packet processing while integrating with the OS and still allowing socket-based applications on the same NIC, the eXpress Data Path (XDP) [52] has been introduced in the Linux kernel. XDP allows to attach Extended Berkeley Packet Filter (eBPF) programs early in the kernel's packet processing path, enabling programmability at performance close to kernel-bypass technologies while leaving the kernel's packet processing functions usable. eBPF programs can be loaded and dynamically chained at runtime; they support stateful processing and offloading to compatible NICs to further boost performance.

While this novel technology is promising for a wide range of packet processing applications, we believe that eBPF and XDP are particularly useful as a platform for the practical deployment of high-performance network analytics applications and can solve many of the above outlined challenges for several reasons. First, common analytics functionality can be implemented in a shared eBPF program that can even be offloaded to a SmartNIC. This common logic can easily be changed and written using the same language and programming model (eBPF programs in C) as the analytics tasks themselves, simplifying development and adoption. Second, monitoring tasks implemented as eBPF programs can be injected and activated at runtime from user space. Multiple such applications can be orchestrated as a chain where each task feeds its results into the next one. Third, this mechanism of high-performance packet processing is lightweight and saves CPU cycles compared to kernel-bypass solutions [52]. It does not take ownership of the NIC and is transparent to kernel-based packet processing and user space networking applications, making this technology particularly suitable for distributed measurement systems.

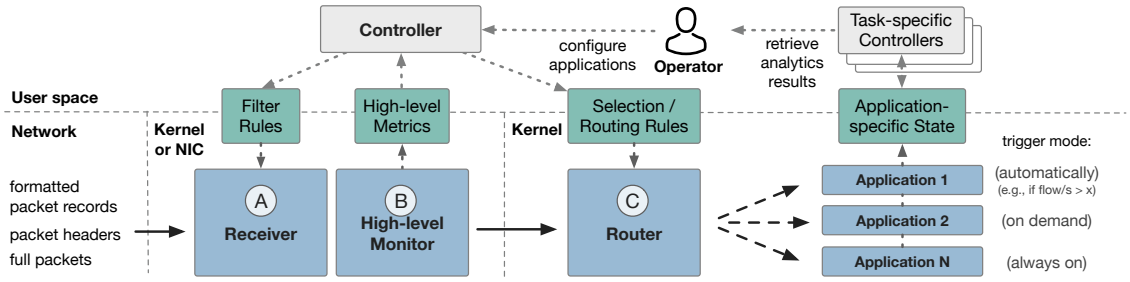

Figure 4.2: System Architecture Overview.

4.3 A Primitive for Network Monitoring Systems

At the heart of our proposed system lies a novel network monitoring primitive that manages a set of monitoring applications and conditionally executes them based on a set of basic metrics in conjunction with operator-specified policies. Performing these tasks at the system-level rather than in each individual application has several advantages. First, application developers can write slimmer applications that focus on the measurement task and do not require logic for input/output, decapsulation of records, and computation of triggers to decide whether a record needs to be processed or not. Second, it provides a simple abstraction for orchestrating and triggering chains of applications reducing the complexity required to build practical, high-coverage monitoring systems, increasing network security and reliability. Finally, consolidating these operations avoids duplicated logic and ultimately saves resources which increases performance and saves cost.

Network Monitoring Applications. Before presenting our system in more depth, we first define what a *network monitoring application* is and elaborate on when and how an operator might want to run a specific application. A network monitoring application is a piece of logic that transforms a high-volume stream of measurements collected in the network into a lower-volume stream of data that provides useful insights for the operator. An insight is useful if it provides enough detail for the operator to make network management decisions with the goal of improving or restoring the correct and reliable operation of the network or to perform other required tasks, such as billing. Network management decisions usually result in reconfiguring a network function (e.g., a router or firewall) or adding, scaling, or removing functions. As previously explained,

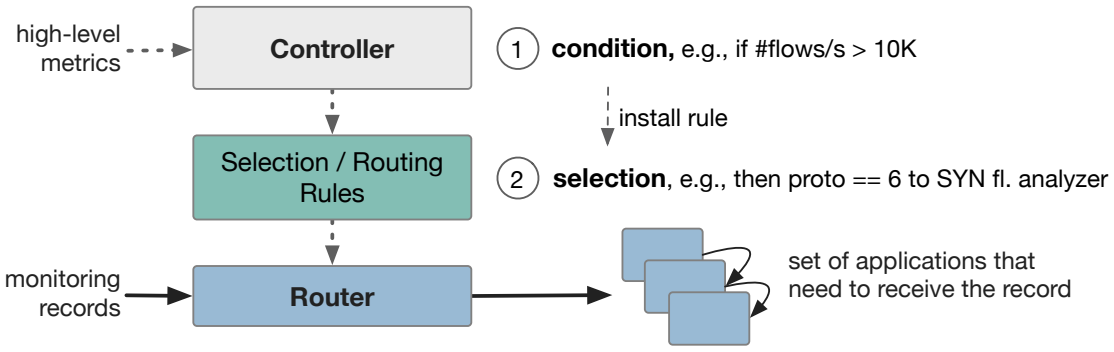

Figure 4.3: Router overview.

applications serve diverse use cases ranging from intrusion detection and traffic engineering to profiling and debugging. While the applications are highly specialized, their logic alone is often relatively simple; many applications add, update, or delete state based on some logic for each received measurement and generate an event if a condition is met.

As applications and their measurement and analytics tasks are diverse, when and how an application should run can also differ depending on the use case. We identified three main cases how an operator might want to deploy an application. First, an application might need to run at all times. This is the case for lightweight monitoring applications that go beyond basic device-level counters, such as a traffic accounting and billing application in a cloud setup. Second, an application can run only when explicitly activated by the operator. This mode allows for, for example, ad-hoc queries or other profiling and debugging tasks, such as detecting an imbalance in ECMP routing. Finally, an application can be automatically triggered by a higher-level condition, such as a sudden increase in connections or overall traffic volume. Here, just knowing about the increase in a metric is not sufficient to apply configuration changes to the network (e.g., block a host). As a result, more fine-grained applications need to be deployed rapidly to mine more facts, such as the set of hosts affected.

Receiving and Filtering Records. We now describe the three main components of our primitive and explain how an operator uses their respective APIs to configure and orchestrate a set of monitoring applications. The overall system architecture is depicted in Figure 4.2; the three

components of the monitoring primitive which we will explain now are labeled A, B, and C. For the remainder of this chapter we focus on network traffic records, in particular formatted per-packet records where each telemetry packet represents one network packet that traversed the monitored device. Each record contains the original packet's IP 5-tuple, ingress switch port number and queue depth, μs-timestamp, packet size, IP-ID, and (if applicable) TCP flags. The records are 32 Bytes in length. This is an example format for the purpose of this discussion and for our prototype; other formats can easily be supported through minor changes in the packet parsing logic.

The receiver component is the entry point to our system. Streaming telemetry records are usually encapsulated in UDP datagrams and sent to a specific IP and port combination on the analytics server. There, a monitoring system needs to differentiate telemetry traffic from control or other traffic destined for the respective machine. Which traffic should be considered monitoring traffic can be specified using filter rules that perform an exact match on the IP 5-tuple of the received packets (not the carried record). Unmatched packets are passed to the kernel to be received by any other application (e.g., the host's SSH server) or dropped at this point.

Packets for the analytics system are then decapsulated and later required metadata fields are prepended. The metadata fields include a list of monitoring application identifiers that will later be populated for record routing and a field for monitoring data derived from a hash computed over the records' IP 5-tuple (see Section 4.4). These steps are stateless operations and can efficiently be offloaded to programmable hardware, e.g., in our case, a SmartNIC.

Our architecture supports various types of input records including formatted packet or flow records, mirrored packets or headers, and regular data packets that may carry telemetry data (i.e., in-band network telemetry, INT [73]). Regular packets would, of course, have to be injected into the normal kernel path after all analytics tasks have been completed.

High-level Traffic Monitoring. The high-level monitor computes statistics relevant to all applications and required for the subsequent routing process. It runs at all times and can also serve as a baseline network monitor, e.g., if no further analytics tasks are currently required. The metrics are scalar values aggregated over a time period (e.g., a second). Our system computes 8

basic counters: the number of packets and Bytes per interval for all records and for TCP, UDP, and ICMP traffic, respectively. This is useful, for example, to detect a shift in the ratio between the protocol types as it would occur in various flooding attacks. It also computes the number of unique flows (as per IP 5-tuple) seen in each interval.

As this module is executed for every received record, it is important that the computation is lightweight. While the basic counters can be incremented efficiently, for example, computing the number of flows would usually require a more heavyweight set data structure. Our prototype leverages a HyperLogLog sketch (HLL) [46] to estimate the number of unique flows. The counters as well as the sketch data structures are stored in memory shared between the controller and data path. The monitor writes into this memory for each packet while the controller reads the values and resets all state after each time period. As this process requires stateful computations, not every type of metric calculation can efficiently be offloaded to a SmartNIC (see more details in Section 4.4).

Routing Records to Applications. Finally, the routing component, conceptually depicted in Figure 4.3, is responsible for determining the set of analytics applications that should receive a particular record. The inputs to the router are the incoming record stream, the current snapshot of previously computed high-level metrics, as well as policies defined by the operator. The policies describe which records under which condition should be sent to a specific application.

At the core of the router is a series of match+action tables for different subsets of the packet's header space (e.g., destination IP address or a combination of fields). An action is a list of monitoring applications that should receive the record. Each record contains the previously initialized list of applications in its metadata. After each match, the router appends the list of applications in the matched entry to the list in the record's metadata and the record will subsequently traverse all applications in this list. The match+action tables are populated by the controller and then read by the router to construct the list of application for the respective record. This allows for a two-step process where first the controller checks whether an operator-defined *condition* is true and then populates the tables for *selection* of the relevant records.

The condition determines when an application should receive a record based on operator policies and previously computed metrics (i.e., when an application should be triggered). The selection step then defines which records, in terms of matches on header fields, are relevant to the triggered application. For example, an application detecting out-of-order TCP segments should not receive UDP traffic at all. The controller provides the operator access to the current state of high-level metrics and exposes an API to add and remove selection rules. The conditions can be implemented either directly in the controller or through an external component (e.g., a script) that consumes metrics and installs rules accordingly.

Above, we outlined three different modes of how and when an operator might want to run an application. Our primitive supports these three modes. First, an application that needs to run at all times, simply has a permanent entry which is installed at system initialization that specifies which slice of the network traffic should always be sent to the application. Second, an application can run only when explicitly activated by the operator (e.g., for ad-hoc queries or debugging). This is possible as the match+action tables can be modified at runtime. Adding or removing an entry does not incur downtime or disruption in the monitoring system. Finally, an application can be automatically triggered by a condition over high-level metrics. A condition is a logical expression, e.g., number of flows per second greater than 10K and is checked after each time interval. If a condition is true, the respective application (or set of applications) is activated for the next time interval. This mechanism is powerful as it allows to (a) execute more computationally expensive applications only when required in order to save resources and (b) autonomously analyze an ongoing issue by deploying operator-defined profiles of more fine-grained applications. More complex conditional execution, e.g., using automated anomaly detection or separate conditions for when an application should be deactivated again are possible in our model but beyond the scope of this work.

4.4 Implementation

We implemented our system and three example applications in approximately 1800 lines of code [1] . The data plane components consisting of the monitoring primitive and the individual applications' data plane parts are written as eBPF programs in C. The main controller and the application-specific controllers operating in user space are written in C++. We will now present the technical details of our implementation, focusing on the computation of high-level metrics, the routing system, and the example applications.

Efficient Computation of High-level Metrics. As we compute a set of high-level metrics for each received telemetry record, this computation must be efficient and not incur unnecessary overheads. Netronome SmartNICs support offloading XDP programs and can be used to maintain simple statistics like counters directly on the NIC without using host CPU cycles. These counters can then be accessed by the controller through an eBPF map. Additionally, our system uses a HyperLogLog (HLL) sketch to estimate the number of unique flows per time interval. The HLL algorithm estimates the cardinality of large sets with negligible memory use [46]. HLL's main idea is that if the binary representation of an element in a set is random and uniformly distributed (e.g., a hash), the number of leftmost zeros in this representation can be used to estimate the cardinality of the set. In this manner, HLL requires the computation of the hash of a given key of interest (e.g., 5-tuple) for every packet in our system.

To reduce variance of the estimation, the hashes are divided in buckets, where the b leftmost bits of the hash value are used as the bucket index on an array. The final cardinality estimation uses the harmonic mean of the cardinality in each bucket. To find the cardinality of a bucket, the HLL algorithm gets the remainder of the hash, calculates the number of leading zeros, and checks if this number is greater than the previously recorded one for the respective bucket. If so, the bucket is updated and the HLL algorithm uses this value to estimate the new cardinality of the set. A smaller b reduces memory requirements for HLL but also reduces its accuracy. For example, by

[1] Our code is available at: https://github.com/mcabranches/xdp-netmon

using $b = 8$, HLL can estimate the cardinality of distinct 5-tuples consuming only 768 Bytes with an accuracy of ∼93.5% (see [46] for details).

Performing the HLL operations is not computationally cheap, so we leverage the processing power available on a Netronome SmartNIC to accelerate them. Despite the limitations of our SmartNIC XDP offloads (i.e., inefficient and slow map update operations from the data plane [9]) we were able to design an efficient division of work between the SmartNIC and the host. In this case, all the stateless operations for the HLL are executed on the SmartNIC and the stateful ones are executed on the host. This is possible as we are able to enrich the packet metadata (using *bpf_xdp_adjust_head()*) with precomputed HLL data on the SmartNIC (bucket index and number of zeros for a given hash), and this will be carried with the packet for further processing on the host XDP layer (HLL state updates) and controller (cardinality estimation).

Routing Packets through eBPF Programs. The match+action tables required for record routing are implemented as eBPF hash maps offloaded to the SmartNIC and populated from the controller to indicate which packets should be processed by each of the applications (traffic selection). When a match occurs in each table, we enrich the packet metadata with a list of file descriptors (FD) that indicates which applications should process the packet in which order. The first field in the file descriptor list is a pointer to the FD of the application that should receive the record next. This FD is used as an index (key) in a `BPF_MAP_TYPE_PROG_ARRAY` on the host's XDP layer to get the memory address of the application to be called using the *bpf_tail_call()* function. Before jumping to the desired application, the pointer is incremented so that the router knows when the chain of applications for a packet has reached the end. After application processing, the packet is returned to the router via a tail call. The router then determines whether the packet needs to be forwarded to the next application or whether it has reached the end of the chain where a XDP action can be applied (e.g., drop, pass, etc.). Figure 4.4 shows an example of this mechanism where a TCP segment is sent through the traffic accounting application and the TCP half-open connection analyzer.

Example Applications. We implemented three example applications to demonstrate the

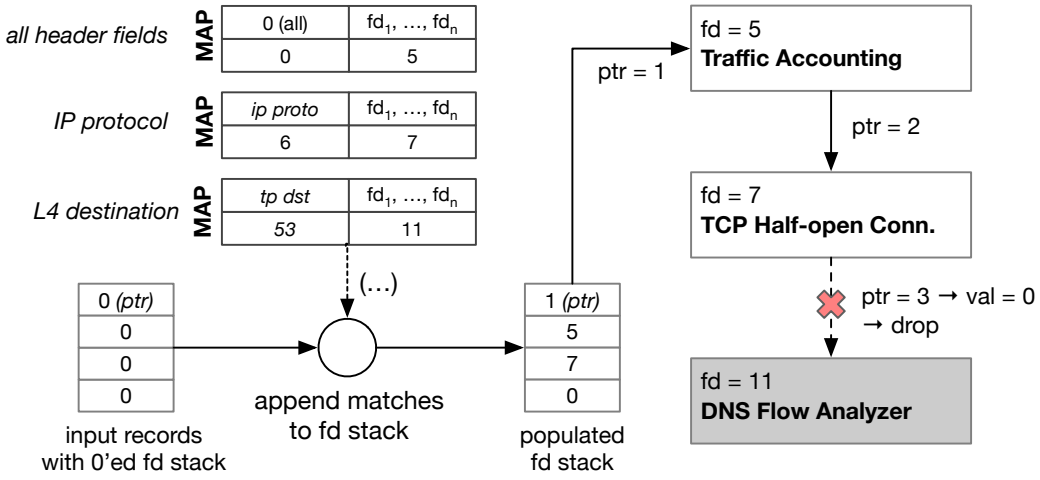

Figure 4.4: Example of record router for a TCP packet.

flexibility of our system. All applications consist of a kernel space (eBPF) component and a user space component. The two components communicate via eBPF maps. The user space component is a standalone application that can access the eBPF maps configured in the kernel space counterpart as soon as the kernel portion is loaded. In order to access a map, the user space application only needs to know a custom identifier string of the respective map set in the eBPF program. We will now briefly describe the applications we used in our evaluation.

Traffic Accounting. This application counts the number of Bytes and packets per destination IP address. This is useful for billing purposes, e.g., in a public cloud. The application performs a lookup and subsequent value increment in a eBPF hash table (`BPF_MAP_TYPE_HASH`) for every single record. The aggregation key and potential filtering can easily be changed to adapt the query to the operator's needs. We envision this application running at all times.

Half-open TCP Connections. A high number of half-open TCP connections are an indicator for various TCP-related attacks, in particular a SYN flood. This application keeps track of all ongoing TCP handshakes with a timestamp of the SYN segment in a hash map indexed by the IP 5-tuple. If the handshake completes, i.e., when the client sends an ACK, the previously installed state for this connection is removed. If an entry spends longer than a configurable threshold (e.g., 5s) in this map, the flow is reported as a half-open connection.

DNS Flow Analyzer. The DNS flow analyzer can be used to confirm a suspected DNS-related attack. It collects the number of packets and Bytes and the timestamp of the first packet for each DNS flow in an eBPF map. This information can be used to block flows where the request rate exceeds a threshold.

4.5 Evaluation

We will now evaluate the efficiency and scalability of our system by measuring CPU consumption and throughput as we vary the number of applications deployed and the number of cores assigned to the system. We also evaluate the ability of our sketch-based high-level monitor to detect an attack.

Experimental Setup. We set up two 12-core servers (Intel Xeon E5-2620v3 at 2.40GHz) with 64 GB of RAM. The first server runs our system and is equipped with a 10 Gbit/s Netronome Agilio CX SmartNIC [86]. The second server has a 10 Gbit/s Intel 82599ES NIC. The servers run kernel versions 5.8 and 5.4, respectively. We use DPDK's pktgen [37] to replay a PCAP file with telemetry records generating up to 12.5 million packets per second (Mpps) between the machines.

Efficiency in CPU Utilization as we add Applications. Our system saves resources (i.e., CPU) by leveraging shared high-level metrics that drive our monitoring application routing decisions. To evaluate this, in our first experiment we apply a conservative load on our system (2 Mpps) and run it on just one core. We also gradually increase the number of monitoring applications that each packet traverses. We can see in Figure 4.5 that as each packet traverses a longer chain of monitoring applications (x-axis), the average CPU utilization on the system only increases in small steps (y-axis) due to our primitive and XDP. With the SmartNIC offloads (see §4.4), our system is even more efficient and able to comfortably process the applied load with all applications enabled, without ever reaching over 60% CPU utilization. This is important for energy efficiency, and also leaves more CPU cycles available for extra monitoring applications and other processes. Systems using DPDK would consume 100% CPU at all times.

Scalability in Terms of Throughput. Now we look at our system scalability in terms of

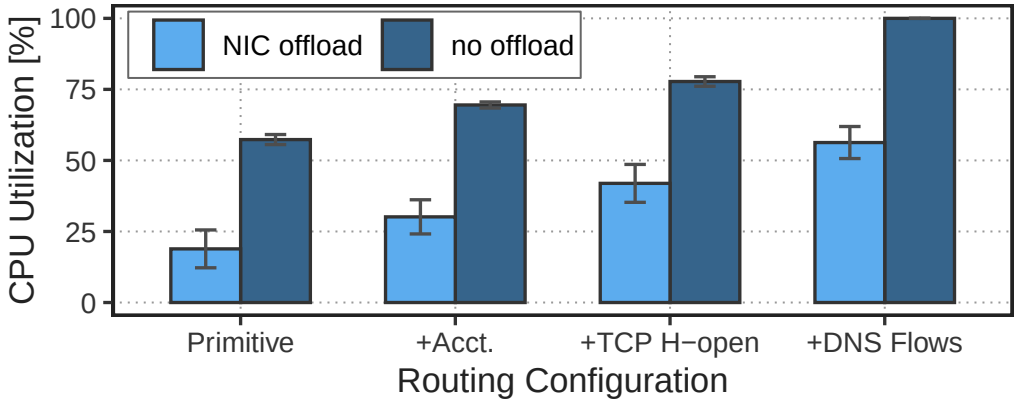

Figure 4.5: Impact of adding monitoring applications on single CPU utilization.

throughput as we add applications. Here, to show a different perspective, we focus on the main limiting factor of throughput in our monitoring applications - the number of eBPF map accesses (i.e., lookups/writes). As we describe in Sections 4.3 and 4.4, most network monitoring applications interact with some state (often in a hash table) to implement their core functionality. For example, our three applications each perform up to two map accesses (lookup and/or write) for each packet.

To evaluate how the number of map accesses affects the system throughput, we gradually increase the number of map lookup/writes on an eBPF hash map for each packet. This XDP application is set to run on just one core, and we can see it as a chain of applications of variable length and of different complexities. Figure 4.6 shows that as we have more map lookups/writes, the system's throughput starts to degrade. By leveraging our system's primitive, monitoring applications can be turned on and off, ensuring that they will only execute when needed, which in turn allows more packets to be processed by the monitoring applications that are running at a given time.

Scalability as we add Resources. Now we show how our system throughput scales with the number of processing cores. In Figure 4.7, we run our system on one and two cores (one and two NIC queues) and set IRQ affinity for each queue/core. The y-axis shows the system throughput in Mpps. Our SmartNIC distributes traffic among cores based on the contents of the telemetry packet (in our case, the 5-tuple of the record). We set pktgen to send traffic at its

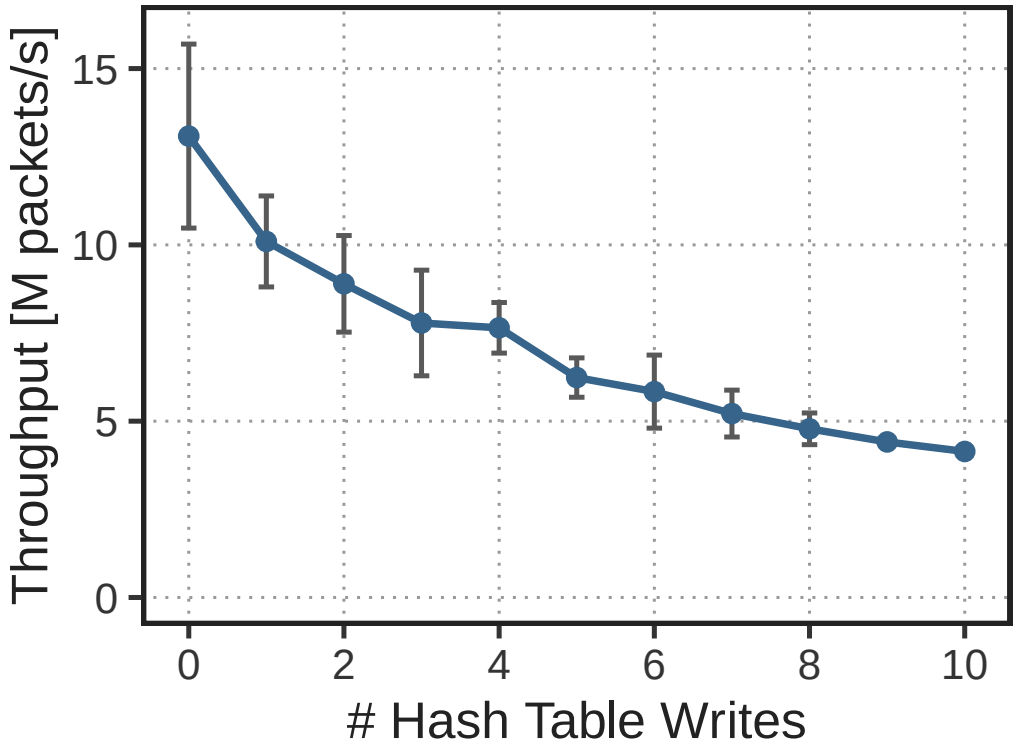

Figure 4.6: Performance impact of per-packet hash table lookups and writes.

maximum rate for our configuration (12.5 Mpps) and run the primitive alone and with each of the monitoring applications (x-axis). Here, we can see that our offloads can speed up our system by accelerating stateless operations, as we describe in Section 4.4. The primitive with one core has slightly higher throughput than with two cores. This is likely caused by memory access contention between multiple memory queues and our logic on the NIC. This cost is, however, amortized as the applications with two cores have higher throughput than those with one core.

High-level Monitoring Accuracy. Finally, to show the accuracy of the HLL-based flow count estimator, we demonstrate a practical example of how our implementation was able to detect an artificially injected flooding attack. In this experiment, we used a WAN trace collected by CAIDA [4] and added an attack packet for each existing packet with probability 0.1 during a 60 second time window. The attack packet had a randomly sampled 5-tuple increasing the number of distinct flows. We computed the exact number of distinct flows seen in every 1 second time interval

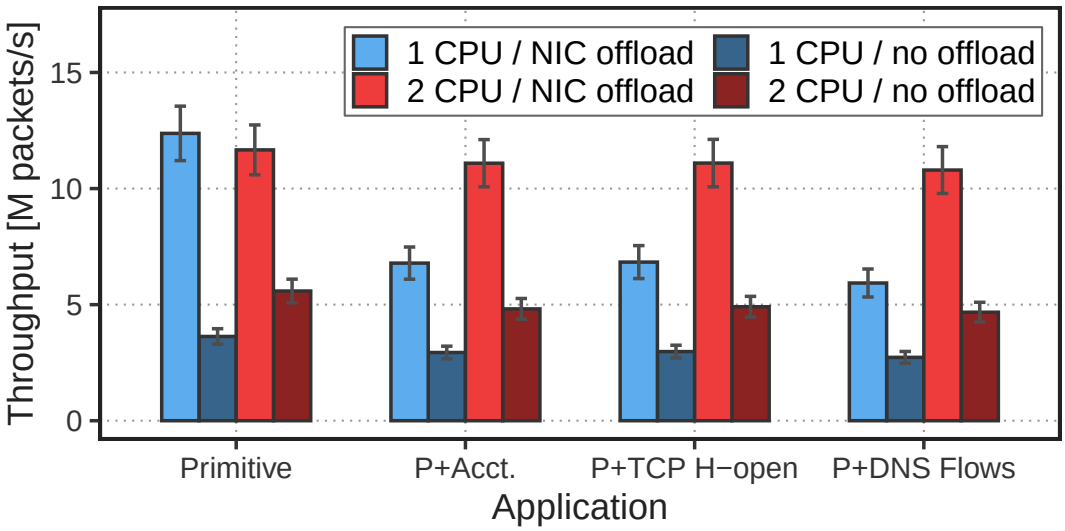

Figure 4.7: Throughput of monitoring primitive and primitive plus individual applications using 1 and 2 CPU cores.

and used our HLL implementation with $b = 8$ to obtain an estimate. Figure 4.8 shows that the estimate closely follows the ground truth; it also immediately reacts to the sudden increase in flow count making this approach suitable for our use case.

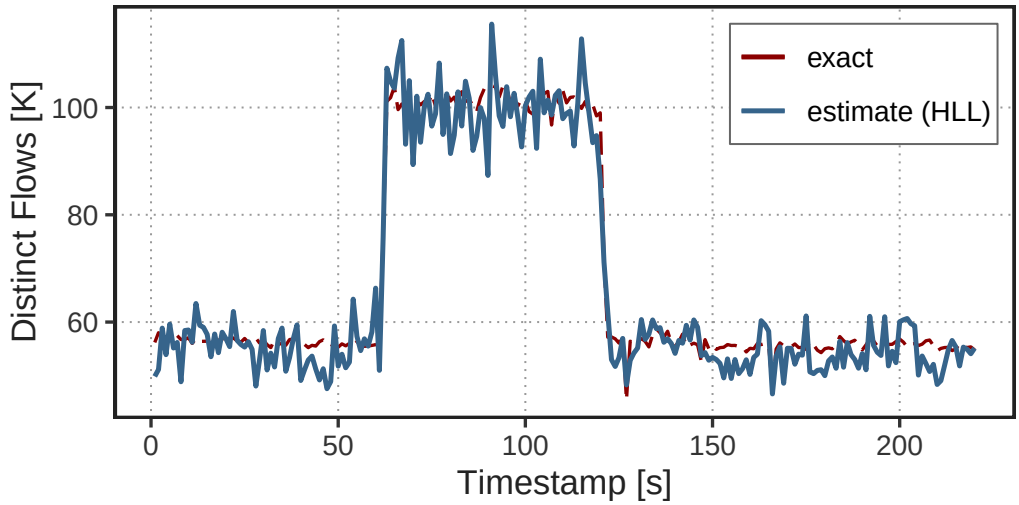

Figure 4.8: HyperLogLog flow count estimate and ground truth during a flooding attack.

4.6 Related Work

The literature around network telemetry and monitoring is vast, yet few works propose solutions for parallel and dynamic queries and analytics tasks. *Flow [98] is a hardware-based telemetry system that partitions an analytics pipeline and performs only those tasks in hardware that are relevant to all queries enabling arbitrary and concurrent queries in software. Jetstream [79] complements *Flow with a fast software processor; the system, however, falls short in providing a server-side routing and dynamic orchestration mechanism for analytics tasks. NetQRE [106] allows for dynamic queries but is not designed for concurrent measurement. BeauCoup [30] is designed for dynamic and concurrent queries at switch line rate but only supports a single class of query (count-distinct).

Most work on orchestrating packet processing functions focuses on network function virtualization (NFV) for a variety of use cases, e.g., [97, 63, 101]. NFV systems and service chains span several hosts and generally dedicate full servers for NF processing, often with all CPU resources being blocked for heavyweight packet I/O frameworks. Our system focuses specifically on network monitoring and is designed to save system resources in order to be deployed alongside other applications and services, e.g., at the network edge.

Over the past years, eBPF and XDP have attracted significant interest in the research community as well as industry. XDP has first been presented in [52] and, since then, the technology has been used for a variety of use cases; most of them revolve around network virtualization [20], load balancing [43], or packet filtering and DDoS mitigation at end hosts [24]. For example, Cassagnes et. al. propose using XDP for DDoS detection and filtering on end hosts [29]. This system, however, also only serves this single use case and does not support orchestration of monitoring tasks.

The perhaps most related work, Polycube [77], goes beyond a single application and provides a framework for realizing general NFV service chains using XDP. Our work focuses on telemetry-based network monitoring applications and is designed and optimized for this use case. Our application

router also uses tail calls to enable chaining and dynamic loading eBPF/XDP applications, but different from Polycube, routing decisions on our system can be based on shared high-level monitoring metrics. Furthermore, our work adds and evaluates offloading XDP logic to a SmartNIC.

4.7 Conclusion

We presented a practical software-based network monitoring framework that significantly reduces resource consumption of network analytics by consolidating tasks relevant to all applications and triggering applications only when required. Our implementation leverages modern kernel-level packet processing capabilities improving efficiency and reducing energy consumption over existing kernel-bypass approaches.

Chapter 5

L2-L4 Performance and Feature Richness

The need for high performance and custom software-based packet processing has resulted in decades of research. Most proposals bypass or replace the Linux networking stack, with the unfortunate consequence of sacrificing the rich and robust functionality available within Linux and the ecosystem of management programs and control-plane software built on top of it. In this paper, we propose to rethink the design of the Linux network stack to address its shortcomings rather than creating alternative pipelines. This re-design involves (1) decomposing packet processing into a fast path and a slow path, and (2) transparently and dynamically creating a custom fast path that only implements the processing tasks currently configured. We leverage Linux's eXpress Data Path to load efficient and small fast-path modules, leaving the kernel stack to serve as the slow path. To materialize this vision, this paper introduces Transparent Network Acceleration (TNA), a prototype system that automatically generates a minimal data path based on introspection of the current networking configuration, avoiding many of the networking stack overheads in Linux while ensuring high performance and maintaining Linux's rich set of functionalities.

5.1 Introduction

Software-based packet processing is being widely adopted across a number of use cases, such as data center load balancing [41], virtualized networking between containers [103, 104] or virtual machines [62], and in 5G infrastructures [47]. Doing so requires both high-performance and, in many cases, the ability to introduce custom functionality. While Linux is the most widely used

platform for many such services, supporting the packet processing performance required with its out-of-the-box network stack is challenging [28, 51, 58].

This led to new approaches for enabling high-performance custom packet processing through alternative pipelines. These take several different forms, such as kernel bypass (e.g., DPDK [8], netmap [94]) which efficiently copy packets to a user space program for processing, in-kernel network stack bypass (e.g., XDP [51], Click [61]) which run inside kernel space but are still largely an alternative pipeline as performance is only obtained when the traffic does not touch the Linux network stack, or as a new kernel (e.g., x-kernel [54], Demikernel [107]).

Using an alternative pipeline vastly improves throughput over receiving and processing packets through the Linux networking stack and makes it easier to add new functionality; however, it incurs significant drawbacks. First, these alternative pipelines cannot, without degrading performance, leverage the rich and widely used networking functionality of Linux such as its built-in bridging, packet filtering, and traffic shaping subsystems together with their powerful management tools (e.g., iproute2 [48]). The rich ecosystem extends to management software that builds around the Linux APIs and interfaces (and command line tools), such as Infrastructure-as-Code (e.g., Ansible), container networking (e.g., Flannel), and control plane software (e.g., FRR [49] for routing and StrongSwan [100] for IPsec).

In this paper, we take the position that we should rethink the design of the Linux network stack to address its shortcomings, rather than creating alternative pipelines. Moreover, we show that this is practical today.

5.1.1 Overheads in the Linux networking stack.

To understand what is needed as part of a re-design, we need to look at the overheads in the Linux networking stack. The generality that makes Linux networking so powerful is also one of its main sources of inefficiencies. One dimension of this is that there is a long, complex data path that performs too many operations per packet. This includes many different protocols and a wide array of functionality where Linux needs to check whether each block of code to process needs

to be called or not. This leads to a long critical path which, in turn, slows processing. A second dimension is that the processing that does need to be executed is quite inefficient — again, due to the need for generality. This includes both heavyweight protocol implementations that can support all the corner cases (e.g., IP fragmentation), and heavyweight data structures (e.g., *sk_buff*) for which the allocation process is not just a memory allocation, but complete parsing of packets to fill in all the data into the structure.

There are two key insights that we can draw from this — both opportunities for optimization of Linux networking. First, in most cases, only a subset of functionality is actually being used. For example, we might only need to set up a bridge between two interfaces, but Linux still checks if we are using IPsec, packet filters, or traffic shaping. Instead, we believe Linux should be designed much like the models proposed in the x-kernel [54] and Click [61] work — that is, as a **composable design**. However, unlike those in which users explicitly provide a graph of processing, we need to make this transparent from users of Linux.

Second, while Linux does have a degree of fast-path and slow-path processing, such as the inclusion of some control protocols (e.g., spanning tree) in the kernel, it treats all packets the same — with the same pipeline and same data structures being allocated. Instead, we believe Linux should explicitly separate out fast-path functionality and slow-path functionality, and tailor the execution of each.

5.1.2 Rethinking the Linux networking stack is practical.

Redesigning the Linux networking stack, as proposed, would require (1) a decomposition into explicit fast-path and slow-path functionality and execution environments, and (2) an ability to dynamically instantiate only the part of the network stack that is used. While this is counter to the monolithic design of Linux, we believe that it is actually possible today because others have introduced frameworks which can serve as the fast-path execution environment. That said, while not explicitly designed for this purpose, we believe there are existing frameworks that can serve as the fast-path execution environment. The key things we need in an execution environment for fast-

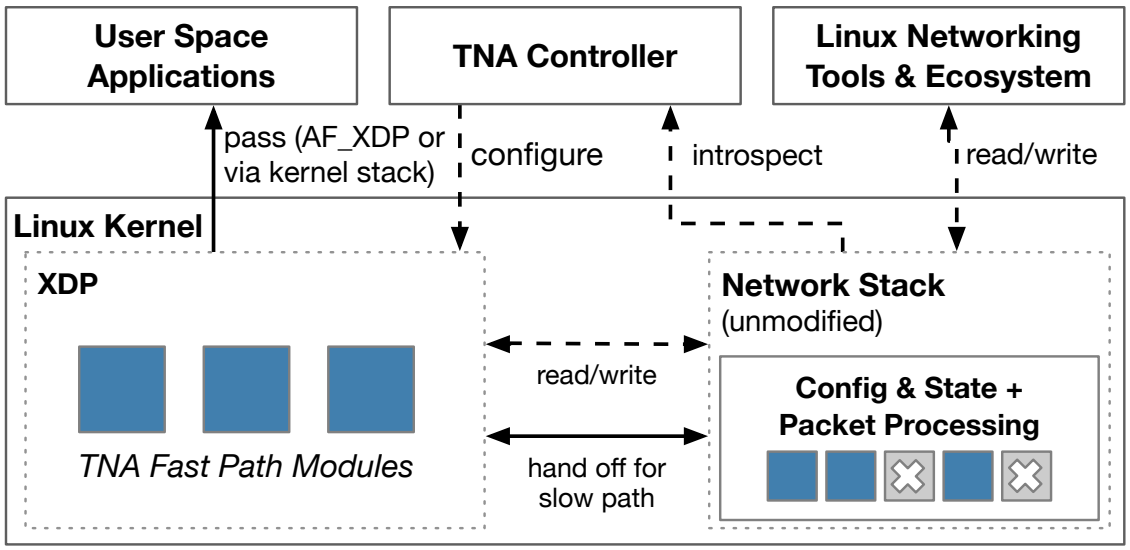

Figure 5.1: TNA Overview.

path processing are (1) that it is designed with efficiency in mind for high throughput processing, (2) that it enables the dynamic loading of processing so that we can load only what is needed at that particular time, and (3) an ability to interact with the Linux kernel to be able to access its data structures and exchange packets. Both Click and, more recently, XDP provide all of these. What having this fast-path execution environment enables is Linux, as exists today, to serve as the slow path. With that, two more challenges remain: (1) how to design the modules such that they are very lightweight and leverage the current Linux network stack as the slow path, and (2) how to dynamically build a graph of Linux processing of what is needed.

5.1.3 Introducing TNA.

As a prototype realization of this vision, we introduce **Transparent Network Acceleration (TNA)**, shown in Figure 5.1, which includes a library of fast-path modules of various Linux networking functions coupled with an orchestration layer that builds a custom data plane on demand based only on what is configured in Linux (with command line utilities or other software that use the Linux interface to set data structures within the Linux kernel, such as the forwarding table). The modules are designed to be lightweight and can be stitched together and loaded into the

kernel with XDP. They are designed to work with the Linux networking stack as the slow path by redirecting any need for corner case processing or state management to the Linux networking stack and accessing state through various helper functions. We support different application needs by multiplexing packets to custom processing based on packet header information. To determine what fast-path processing graph to load at any given time, a lightweight controller continuously introspects the kernel's network configuration, then it composes and places the currently used functions on the data path. It also provides APIs for user-defined logic to be inserted at any point in the data path. The result is a fast Linux networking stack that is (1) **fast** as it only runs a slim data plane of the currently required functions instead of passing packets through a complex, general-purpose stack, (2) **transparent** to the rest of the system as it uses the kernel's configuration and network state together with its ecosystem of tools and third-party software, and (3) **extensible** as it uses an underlying technology (XDP) that was designed to add custom functionality to the Linux kernel efficiently.

To demonstrate the benefits and performance gains with TNA, we accelerate three Linux network subsystems, i.e., bridging, routing and filtering. With this, the user can configure Linux bridges, routing and filtering with unmodified Linux management tools and TNA transparently accelerates their processing. For example, our bridge implementation, depending on the CPU count, improves throughput by up to 3.1 times over the Linux networking stack implementation. We also compare with Polycube [77], which directly leveraged XDP to accelerate bridging by bypassing the Linux networking stack. Unlike with TNA, Polycube is not transparent to the user and requires users to interact with custom control software. We show that even in this case, the TNA-accelerated bridge is up to 1.6 times faster. This largely stems from XDP being used as a bypass of the Linux networking stack in Polycube, versus an explicit leveraging of Linux functionality as a slow path that keeps the TNA modules very lightweight. We also show performance gains for routing, filtering and services composed by a combination of those subsystems.

In the remainder of the chapter, we first discuss how we can decompose Linux functionality into fast-path and slow-path elements (Section 5.2) and how we can then automatically build

a custom fast data path that only includes needed functionality based on the current context (Section 5.3). We then describe the prototype implementation of TNA, along with an evaluation based on accelerating the Linux's bridge, routing and filtering subsystems (Section 5.4). We wrap up with a discussion of conclusions and future work (Section 4.7)

5.2 Building Composable Fast-Path Modules

The first challenge to address is how to design the modules such that they are very lightweight and leverage the current Linux network stack as the slow path effectively. Here, we first provide guidance on how to design the modules and then describe how we decompose Linux functionality into fast-path modules.

5.2.1 Designing fast-path modules.

The Linux networking stack today processes all packets without explicitly distinguishing them as fast-path or slow-path. In this way, any packet that is sent to Linux will be correctly processed and will correctly update internal state. In the trivial case, we can consider that a fast path that is empty and sends all traffic to the Linux networking stack will work correctly.

What we aim for is that elements be inserted such that they can process a majority of packets without needing to send them to the Linux networking stack. These elements should only execute a few simple tasks such that they can be as fast as possible. What this means is different for each function; we provide some examples later in this subsection. In general, it should only include the common-case data plane processing — with corner cases and more complex control protocols being handled by Linux. In XDP, this is enabled through the ability to inject packets into the Linux networking stack. So, modules must include functionality that determines whether a packet can be processed just in the fast path, or needs to be passed to the Linux networking stack.

The functions should not maintain their own state, but instead only use existing Linux networking data structures. In XDP, this can be done through the use of helper functions[1] .

[1] Some kernel helpers are available to XDP today (e.g., *bpf_fib_lookup* [10]), but some are missing. So, to realize

Subsystem	Fast Path	In-Kernel State	Control Plane + Slow Path
Bridging	Parsing, rewriting,FDB lookup/update, forwarding	FDB, port state	Manage FDB (aging), handle FDB misses (flooding), STP protocol processing
Routing	Parsing, rewriting, FIB lookup, forwarding	FIB, neighbor tables	Routing Protocols (e.g., BGP, OSPF), ARP handling, IP (de)fragmentation
Netfilter	Parsing, rewriting, conntrack lookup/update, allow/deny packets	Conntrack, ACLs	Conntrack handling, IP (de)fragmentation, handle rules on unsupported hooks

Table 5.1: Acceleration model for different packet processing applications.

This allows any given packet to be processed by the Linux networking stack without modification, otherwise synchronization would be needed. State will largely be (but does not need to be) read-only as much of the state management exists within the higher-level control protocols or set from management utilities (e.g., Linux command line tools) or user-space control plane software.

5.2.2 Building a library of composable data-plane modules.

To provide examples of these principles in practice, we elaborate in Table 5.1 how we break Linux packet processing into a series of lightweight modules for a set of Linux networking subsystems. While this is not an exhaustive set of networking functionality, this list is highly representative as those services serve as the basic building blocks to implement complex networking applications such as bridges, routers, NAT, firewalls, and container network interfaces (CNIs).

The lightweight modules for each subsystem are responsible for simple tasks, like parsing and rewriting packets, looking up state in the kernel tables, and sending packets for full processing when needed. We call those lightweight modules the *TNA fast-path modules (FPMs)* and we build a library of them to execute tasks needed by each network subsystem. For example, the fast path on a bridge deployment can be composed by a series of pre-built TNA FPMs where each of them performs tasks like packet parsing, forwarding database (FDB) lookups (via a kernel helper), and L2 forwarding (directly from the fast path). The Linux kernel exposes FDB access and port state to the fast path via a kernel helper and performs tasks like aging, STP and FDB misses handling, and packet flooding. FDB misses/flooding should happen only for the first packet destined to an unknown MAC address [26]. STP is responsible for avoiding loops on a network with

our full vision, we will both leverage the ones that are available and build new ones as needed.

redundant paths, by blocking specific ports based on bridge protocol data unit (BDPU) messages that communicate topology changes once every 2 seconds (by default) [32]. In this manner, the fast path is able to process the majority of the packets with higher performance than the full Linux processing (see Section 5.4).

In the same manner, the fast path for a router uses TNA FPM modules to parse packets, perform FIB (forwarding information base) lookups, and L3 forwarding. The fast path gets assistance from Linux by accessing exposed FIB and neighbor data via a helper, supporting routing protocols like BGP (*Border Gateway Protocol*) – in which protocol messages only need to be processed every few seconds [92]), and other unsupported operations like handling fragmented packets (which can be avoided [36]).

The netfilter subsystem exposes access control lists (ACLs) and the table of initiated L4 connections/flows (called conntrack table) to the fast path. This allows building accelerated services like stateful firewalls and NAT[2] .

5.3 Automated Fast-Path Data Plane Creation

The second key challenge is how to dynamically build and load a fast-path processing graph consisting of only what is needed. Before providing a more complete description, to give intuition on how TNA can generate and deploy a minimal XDP fast path to accelerate Linux networking services, in agreement with what is currently configured, we describe a simplified example for one use case. In this example, we show how TNA would automatically instantiate an XDP fast path to transparently accelerate a bridge deployment. In the left part of Figure 5.2 we see that an operator or automation framework (e.g., Ansible) could execute a sequence of commands using Linux configuration tools (e.g., iproute2 and brctl) to configure the bridge. So, the process starts with an operator adding a network interface on the Linux system (NIC1) to a desired VLAN (Y). The second step is to issue a command to create the bridge (br1). After that, NIC1 is added to

[2] Note that this model is essentially different from solutions like [77] which maintains state in BPF maps and reimplements several networking features in BPF form, missing the opportunity to leverage features already implemented in the Linux ecosystem.

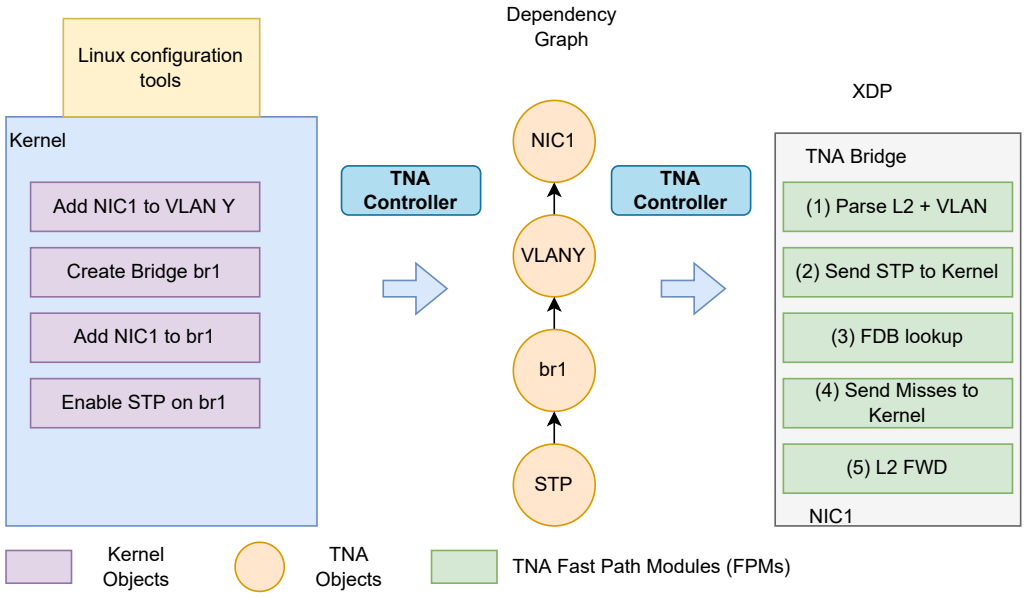

Figure 5.2: Automated Data Path Creation

br1 and finally STP is enabled on this bridge.

To automatically generate the XDP fast path, we need to *introspect the Linux kernel* looking for bridge objects and retrieve the entities that compose them (TNA objects). After that, we *build a dependency graph* with relationships between those objects, allowing us to have a structured view of the current bridge configuration.

The next step is to map the nodes of the graph into one or more small units of pre-build eBPF code (TNA FPMs) that can be stitched together to provide the necessary fast-path logic to accelerate the bridge. In this example, the dependency graph shows a bridge with VLANs and STP configured on it. In this case, we *stitch together and deploy a set of TNA FPMs* on NIC1 to (1) parse L2 header including VLANs, (2) send STP messages for kernel processing (slow path), (3) perform FDB lookups in the kernel, using a kernel helper, (4) when a lookup fails, send the packet to the kernel, (5) when the lookup succeeds, send the packet to the egress port directly from the fast path.

We design the TNA controller (shown in Figure 5.3) with a series of components that work together to allow the steps just described to happen, as we explain next with more details.

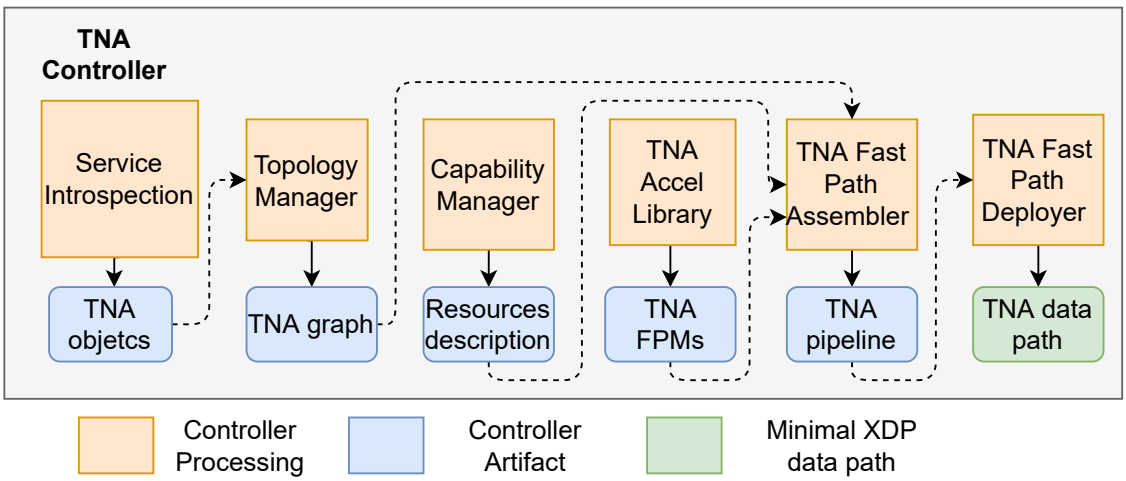

Figure 5.3: TNA Controller

5.3.1 Introspect the Linux kernel

. Our *service introspection* component uses the Netlink protocol [85] by both sending queries to the kernel at the controller startup time to get an initial view of the current configured services and also by joining to multicast groups to get kernel notifications about configuration changes and updates. The received messages are converted into network objects descriptions (TNA objects) containing the type of object and a set of configuration attributes. A network interface, for example, will contain the type of interface (e.g., physical or virtual), its name, its current state (e.g., up or down), IP configuration, and so on.

5.3.2 Build a dependency graph.

TNA starts this process by feeding the TNA objects generated by the *service introspection* to the *topology manager* component, which is responsible for processing each of those objects and establishing relationships among them. This creates the *TNA graph*, which is a dependency graph representing the services that are currently configured and the objects that compose them. To build the dependency graph, we leverage relationships that can be derived from the Netlink messages (e.g., the bridge that an interface is part of). Where this direct mapping is not possible, we apply

domain knowledge to derive other dependencies. [3] For example, when we have an IP address configured on a bridge interface, packets arriving at this interface may need routing, so we add the required objects representing this feature to the dependency graph accordingly.

TNA aims to allow tailoring acceleration code for systems according to the features they have available, such as the presence of SmartNICs with XDP offload capabilities [86] and a kernel version with needed helpers available. For doing so, the *capabilities manager* component builds an inventory of the available assets on a given system, such as the kernel version, and network interface model.

5.3.3 Stitch together and deploy a set of TNA FPMs.

The *TNA acceleration library* has a set of pre-built TNA FPMs. We propose different types of code that can, for example, (if available) leverage SmartNICs to do packet parsing and maintain counters, or leverage a set of available kernel helpers.

The *fast-path assembler* component uses outputs from the topology manager, capability manager, and the TNA acceleration library to generate a minimal fast path to support the services that are currently configured on the system in accordance to its capabilities. Currently, TNA does so by directly mapping one or more TNA FPMs to each node of the TNA graph in a hierarchical way that allows building a fast path that has functional equivalence with the original Linux data path (but is thinner and faster). We aim to generate a data plane that is as thin as possible as this results in fewer instructions per packet (making processing faster), potentially reduces cache misses (as code/data are more likely to fit on processor's cache), and leads to less resource consumption — which is important on SmartNIC XDP offloads. Generating a minimal data plane can be enforced by (1) avoiding code duplication by carefully organizing the TNA graph nodes such that several services on a pipeline can share TNA FPMs (e.g., a packet parser) and (2) avoiding deploying unnecessary code. For example, if there are no VLANs or IPv6 configured on the system, TNA

[3] Currently, we do hard code domain knowledge in TNA. As future work we aim to generalize defining supported kernel objects, services, configurations and dependencies via configuration files.

should not deploy code (i.e., TNA FPMs) to parse those protocols.

Given the minimal fast path that was just generated, the *Fast Path Deployer* is responsible to actually deploy the code on XDP. Currently, we do this by having, at the beginning of the pipeline, an XDP program that leverages the eBPF tail call mechanism to send packets to the minimal data path. Each time the data path is regenerated (which is triggered by changes in configuration), TNA atomically replaces the current XDP data path with the new one, by updating the reference to the new program on the required map eBPF map [51]. The fast path is built by in-lining the required FPM modules to compose the full processing pipeline.

5.3.4 Extensible Fast Path.

While the focus of this paper is how to redesign the Linux networking stack itself to be high performance, we inherit the desire for users to add custom packet processing. For this, TNA provides an API that allows injecting custom code on the packet processing pipeline. This can be done in two ways. The first is to inject custom eBPF code at different points in the XDP processing pipeline. Those attachment points can be at the beginning of the processing, at the end (e.g., before a packet is forwarded), or somewhere in between (i.e., between two TNA FPMs). This allows injecting custom network functionality in the pipeline, e.g., monitoring [18], or load balancing [44], that can work in concert with the deployed pipeline. The second possibility is to add custom packet processing applications on user space (e.g., [17]). This can be done by using a special type of socket, called *AF_XDP*, that allows sending raw packets directly from the XDP layer. This enables creating a hybrid kernel/user space processing environment where lower layer protocol processing and security are provided by Linux/TNA and upper layer processing (e.g., L4-L7) is provided by user space with higher performance than a full Linux pipeline. Full exploration is left as future work.

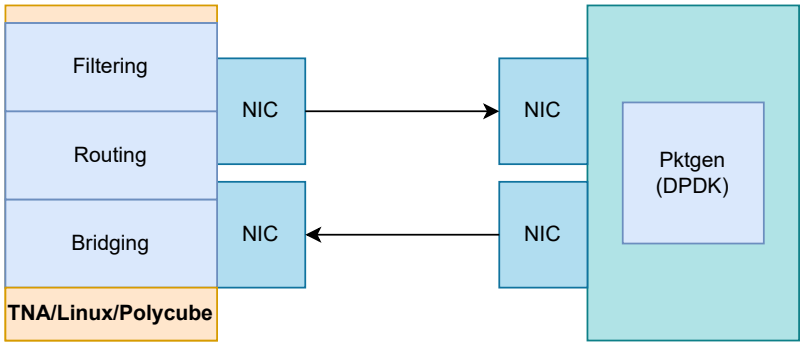

Figure 5.4: Evaluation Scenario.

5.4 Prototype and Evaluation

We built an initial prototype to evaluate TNA's feasibility and performance. For our initial evaluation, we focus on accelerating Linux's bridging, routing and filtering (iptables) subsystems; they have several mature features implemented in the kernel and are widely deployed, for example in data center networks and CNIs. Our prototype is composed by the TNA controller (∼1820 LoC), a library of FPMs (∼200 LoC) and new BPF helpers (∼230 LoC) that we introduce in the Linux kernel, or modify existent ones to support TNA use cases. Those helpers are available on our Linux kernel fork on GitHub [1].

We show our evaluation scenario in Figure 5.4. We set up a test bed composed of two servers connected back to back via two ports of a quad-port 10 Gbps Intel NIC (Intel X710). Those servers have Two 2.4 GHz 64-bit 8-Core E5-2630 "Haswell" processors. We disable Hyper Threading (HT) and power saving. We also perform all tests using CPU cores and NIC interfaces on the same NUMA node. Unless stated otherwise, one of them acts as packet generator/sink using DPDK's Pktgen and generates line rate traffic towards the other server at minimum packet size (64B packets at ∼15 Mpps). In this server, we use one processor core to generate traffic and another one to receive it. The second server is the device under test (DUT) and is used to receive and forward traffic to the other machine. In this server, we deploy and compare TNA with both Linux (kernel 5.15) and Polycube [77] (v0.9.0) using different services — bridging, routing and filtering. Unless stated

otherwise, we run each experiment 10 times for 10 seconds.

5.4.1 TNA Bridge Prototype

As there currently is no helper function available in XDP to interact with bridge state inside the kernel, we needed to add one. When a packet arrives at the XDP layer, the helper adds the packet's source MAC address/ingress port to the kernel FDB table (if not yet present). After that, given a destination MAC address/VLAN ID, if there is a match on the FDB, the helper answers with the output port and also the STP state (e.g., blocked or learning).

The TNA controller is able to introspect the kernel, build the dependency graph representing the kernel objects required for a bridge, including the bridge name, the attached network interfaces, their configuration (e.g., VLANs, STP, etc.). Based on this graph, the TNA fast-path assembler composes a minimal data path. For example, if there are no external routes configured on a system, TNA will not be accelerating L3 forwarding nor will code to parse the IP header be added; the same concept applies to VLAN headers if no VLANs are configured. Similarly, if STP is disabled, there is no code to check STP state on a port and take actions based on it (e.g., drop packets on a blocked port).

As Polycube completely reimplements the bridge in eBPF/XDP and user space, it cannot be configured with standard Linux tools; the command *polykubectl* is required instead. In contrast, we configure the Linux bridge with standard unmodified tools (e.g., *ip*, *brctl*, *bridge*), and let the TNA controller automatically deploy an accelerated XDP data path for this deployment. Figure 5.5 shows the results using 1, 2, and 6 cores for each system. We can see that the Linux bridge with TNA acceleration is up to 3.1 times faster than without. It is also up to 1.6 times faster than the Polycube bridge, with the added benefit of leveraging the Linux bridge implementation and the respective configuration tools. In this scenario, TNA is faster due to its ability to generate a minimal data plane (avoiding unnecessary packet processing overheads) and optimized access to kernel data structures, which allows it to consume 39% and 69% fewer cycles per packet than Polycube and Linux respectively (when using just one CPU core).

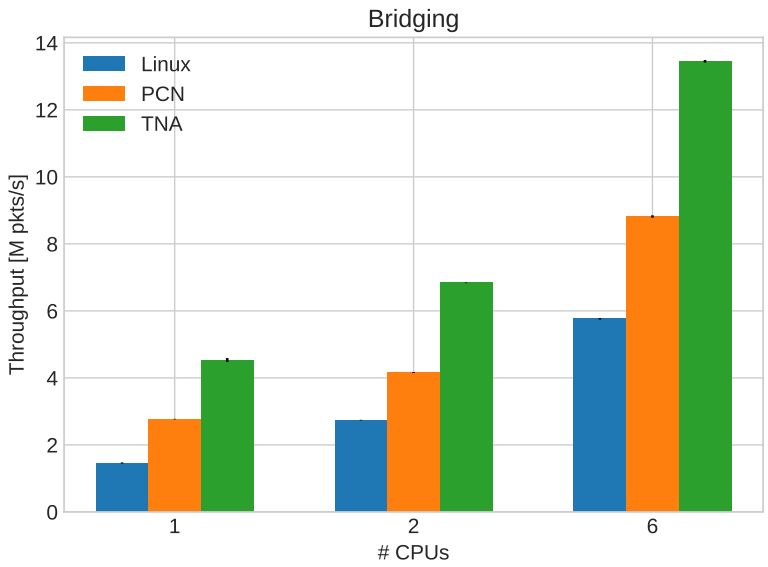

Figure 5.5: Throughput of Bridge Implementations.

5.4.2 TNA Router Prototype

Linux already has a BPF helper that allows interacting with the kernel's FIB, so we leveraged it on our prototype. The TNA controller introspects the Linux kernel and if L3 forwarding is enabled (i.e., *net.ipv4.ip_forward=1*) and there are external routes configured on the system, the TNA composes the fast path and deploys it on the required interfaces. We can see in Figure 5.6 that TNA is up to 3.9 times faster than Linux and 5% faster than Polycube. In this case, TNA's performance is similar to Polycube's and their difference in cycles per packet is almost negligible. We attribute this to the fact that routing is a simpler service than bridging, allowing an eBPF fast path to be naturally thin and optimized. For example, differently than in a bridge, there is no need to process VLANs or spanning tree state for forwarding. In addition, there are fewer interactions between the eBPF data plane and control plane as there is no need to perform complex tasks like, MAC learning, STP state and perform FDB aging. However, we emphasize the fact that TNA can accelerate Linux routing subsystem transparently, where routes can be managed with standard Linux tools (e.g., iproute2), and powerful control plane routing software like Bird, FRR and Quagga. This allows TNA to transparently support and accelerate, for example, BGP and

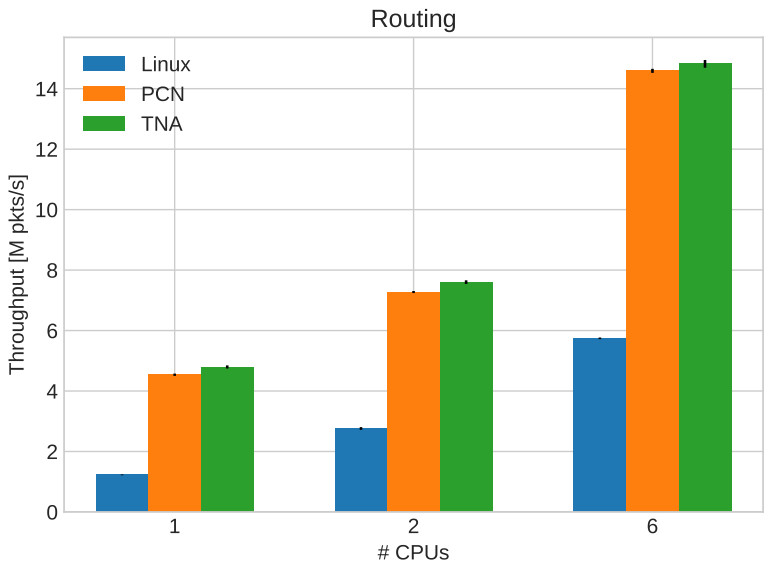

Figure 5.6: Throughput of Router Implementations.

OSPF (*Open Shortest Path First*) deployments. In order for Polycube to support this scenario, the routing control plane software would have to be modified to work independently of the Linux kernel, losing its features and to directly update routes on eBPF maps.

5.4.3 TNA Iptables Prototype

Currently, the Linux kernel does not provide any kernel helpers to allow interacting iptables. As a first step, to allow accelerating iptables, we build a new BPF helper that allows matching iptables rules in its filtering table on the forwarding chain. Currently, our helper enables matching source/destination IP addresses (including longest prefix matches for subnets) and protocol. [4] . Our helper also supports aggregating rules using *ipset*, discussed below. The TNA prototype supports the forwarding chain. This means that TNA is able to accelerate filtering for packets being forwarded by Linux routing subsystem. In this manner, TNA deploys the iptables FPM, following two conditions. The first is that the routing FPM is deployed. The second is that there are supported iptables rules in the forwarding chain. As iptables does not support Netlink, we

[4] As future work, we plan to support matching more packet header fields and support accelerating other chains

introspect its tables using the library *libiptc* [67].

As noted in [75] iptables suffers from scalability issues mainly due to its matching algorithm that performs a linear search on its table until a match is found for a given packet. This causes performance to be degraded when several rules need to be evaluated before applying a verdict (e.g., accept or drop a packet). For example, suppose that a table has 5k rules and none of them matches a given packet. In this case, iptables will try to match the packet against all of them before continuing its processing. To allow better performance and scalability for iptables, the Linux kernel community introduced a new component called *ipset*. Ipset allows aggregating several matching fields (e.g., IP addresses, networks and ports) in one set. This allows dramatically reducing the number of required rules on iptables, which in turn greatly improves its performance. As TNA leverages iptables for filtering, it is able to transparently accelerate firewall deployments using our supported matches and chains. As we will see in this section and the next, this allows accelerating services built on top of a composition of Linux kernel facilities like, for example, bridging, routing and filtering. On the other hand, TNA inherits iptables scalability limitations, which are attenuated as our BPF helper supports ipsets.

In this subsection, we compare TNA and Polycube filtering. In the next section, we compare TNA and Linux iptables filtering, in a scenario where we accelerate a composition of Linux services, which is not supported by Polycube. Polycube implements its own firewall in eBPF form, which is called *pcn-iptables* [75]. Their implementation addresses the iptables' scalability problem by introducing a new search algorithm that instead of performing a linear search on the filtering table, they adopt a more efficient classification algorithm called Linear Bit-Vector Search (LBVS), which was introduced in [65]. LBVS encodes matched rules in bit vectors, and matches for each protocol field are done through separate tables that are matched in sequence. At the end of the pipeline, the LBVS performs a bitwise AND operation with the bit vectors as input to obtain the final verdict for the packet; the most significant bit of the result indicates the matched rule with the highest priority. This allows a divide-and-conquer paradigm where rules can be evaluated in large batches, allowing great lookup speedup when comparing to iptables linear search algorithm. As we can see

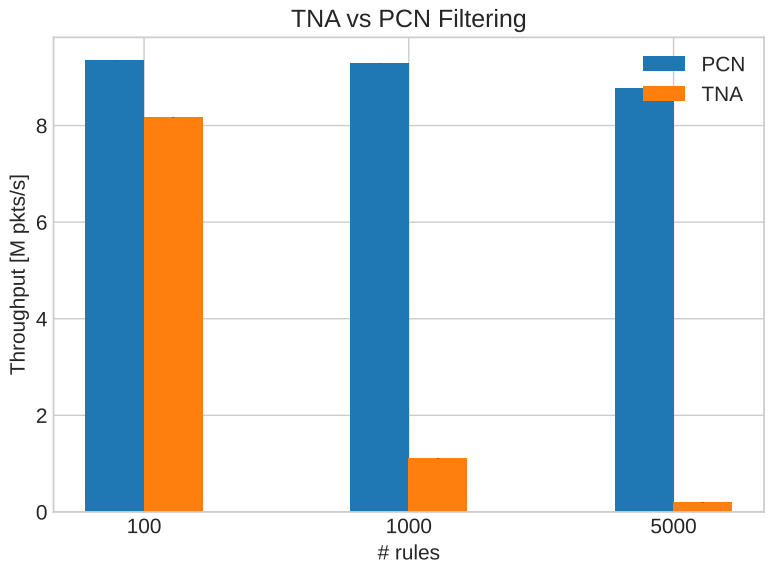

Figure 5.7: Throughput of TNA vs PCN filtering Implementations.

in Figure 5.7, Polycube's filtering is indeed faster than TNA's, specially as the number of firewall rules increase. However, pcn-iptables does not support ipsets, and only supports up to 5000 rules. In contrast, TNA supports several thousands of rules that, if aggregated on a few ipset rules, may lead to performance similar to pcn-iptables. For example, in Figure 5.8 We aggregate 50k different networks subnets on ipsets and create one or two rules matching them. This allows TNA to filter packets for a much larger range of IP addresses and networks than Polycube, while having better or similar performance.

5.4.4 Stacking Different Subsystems Together

We now evaluate the scenario where TNA introspects different subsystems at different levels of its processing stack (i.e., L2, L3 and packet filtering). When possible, we compare Linux and TNA with Polycube. This use case can be seen on Figure 5.9. On the left part of this figure, we build a bridge, with VLAN filtering activated, and we do L3 routing to allow communication between different VLANs. We also configure iptables rules to allow fine-grained filtering between the different VLANs.

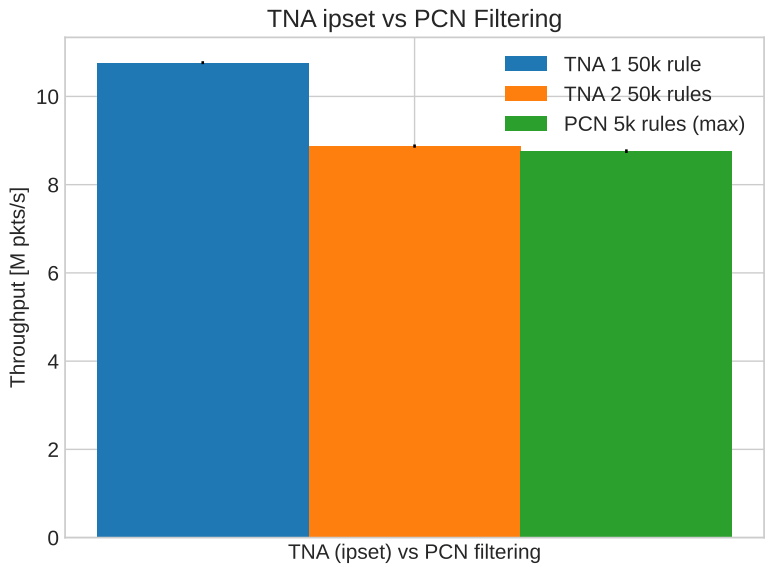

Figure 5.8: Throughput of TNA (ipset) vs PCN filtering Implementations.

In Figure 5.10, we configure bridging and routing using Linux configuration tools for Linux and TNA. TNA introspects the kernel, verifies that the bridge has VLAN interfaces configured with IP addresses, has external routes and also verifies that IP forwarding is enabled. This is translated to a dependency graph that allows TNA to automatically build and deploy a fast path by stitching together the required bridging and routing FPMs. Polycube is configured with *polycubectl*. We test maximum throughput as we add more CPU cores to each deployment. We can see that, in this scenario, TNA is up to 4.2 times faster than Linux, and up to 3.7 times faster than Polycube.

Now we add iptables filtering to this deployment, and set rules to filter traffic between different VLANs. Again, TNA introspects the Linux kernel, and in addition to the dependencies found in the previous example (bridge + routing), it also verifies the presence of supported filtering rules on the forwarding chain. In Figure 5.11, we configure the DUT machine with 6 CPU cores, and we test throughput as we increase the number of filtering rules to be evaluated for each packet. We only compare Linux and TNA, as Polycube does not support this scenario. Here we can see that for 100 rules, 1000 and 5000 rules TNA is 3.5, 1.8 and 1.28 times faster than Linux (note that this plot is in log scale). Now we show how TNA can leverage ipsets to improve filtering

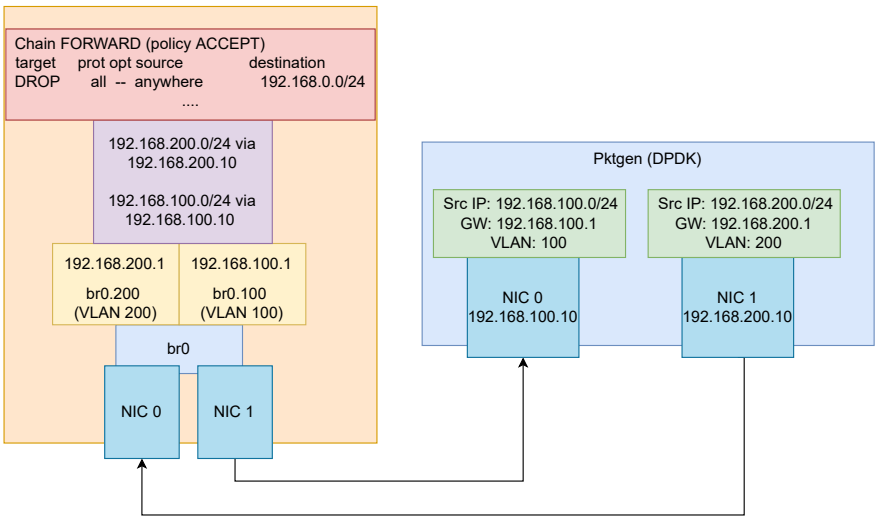

Figure 5.9: Stacking bridging + routing + iptables filtering

scalability while still accelerating Linux. In Figure 5.12, instead of using vanilla iptables rules, we configure ipsets with 50k different network subnets, and add up to 40 rules using them. We can see that TNA is able to increase Linux throughput up to 3.6 times as we add 1, 20 or 40 ipset rules in iptables. In Figure 5.13, we also configure the DUT machine with 6 CPU cores to evaluate compare vanilla iptables rules with Ipset rules in TNA. In this case, we configure 40 ipset rules, with ipsets containing 50k different network subnets. We compare this with 5k vanilla iptables rules. We can see that with ipsets, TNA filtering us able to filter a large range of IP addresses, while having 5.5 times higher throughput.

5.5 Related Work

Kernel-bypass networking. A variety of packet I/O frameworks take the approach of bypassing the kernel in order to scale software packet processing, most notably the Data Plane Development Kit [8], PF_RING [88], and Netmap [94]. Common to these frameworks is that they generally take over control of a NIC, only copy packets a single time from the NIC to pre-allocated memory via DMA, and rely on expensive busy polling instead of interrupts. Snap [74] avoids the need of dedicating hardware resources to network functions by leveraging and customizing Linux

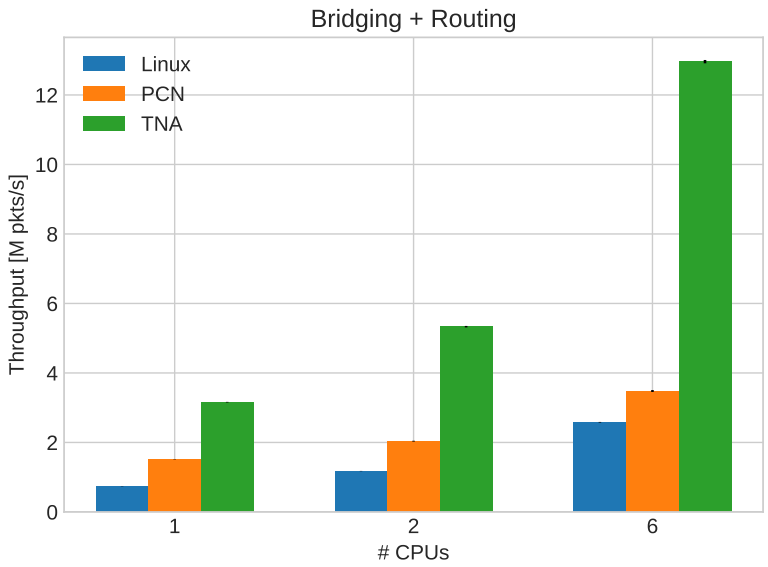

Figure 5.10: Throughput of Bridge + Router Implementations.

kernel features such as scheduling. Snap is a microkernel-inspired approach where core network functions are reimplemented in userspace and managed with custom tools. In contrast, with TNA we believe that the Linux networking stack should not be bypassed, but instead redesigned such that we can leverage the operating system's networking features, and its ecosystem of tools and control plane software.

In-kernel fast packet processing. Alternatively, there has been work that can load custom packet processing functionality into the kernel, providing both the opportunity to access kernel state (e.g., the forwarding table) and exchange traffic with the Linux networking stack. One such framework is the Click [61] modular router, which allows stitching together packet processing elements as a directed graph to build complex network functions from an extensive library of elements and loading that into the kernel as a kernel module. The eXpress Data Path (XDP) [51] similarly provides an in-kernel execution environment, but provides better safety through the use of the eBPF virtual machine (instead of admitting any C++ code to the kernel), and has been integrated into mainline Linux. TNA is complementary to these efforts, as we rely on the capabilities of XDP to provide a fast path execution environment and are inspired by the model in Click where

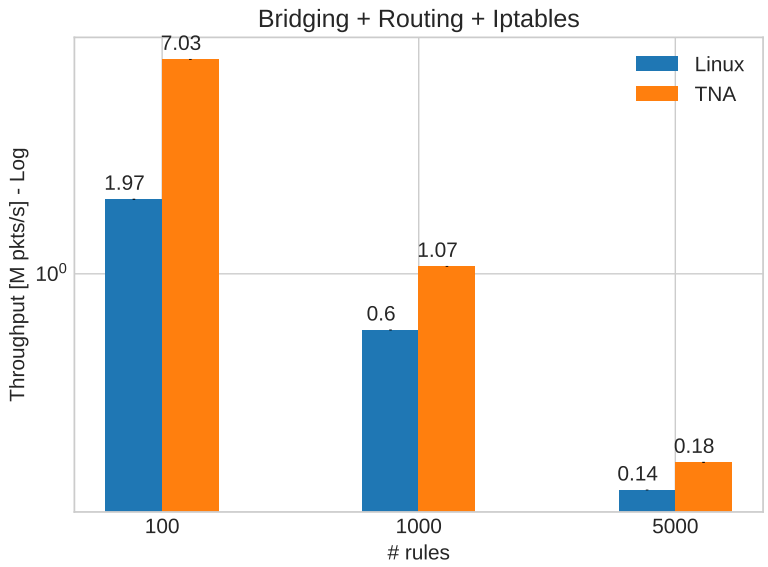

Figure 5.11: Throughput of Bridge + Router + filtering Implementations.

modules can be stitched together. As example applications built with XDP, most related to our work are Polycube (which implemented alternate implementations of some Linux network functions with XDP along with custom interfaces) and Bastion [83] (which implements a CNI with XDP). While providing acceleration, these fall short in that they bypass the Linux networking stack and, in the case of Polycube, slow-path processing needed to be implemented from scratch in user space. In contrast, TNA uses Linux's existing rich functionality as the slow path, avoiding the need for costly reimplementation and non-standard interfaces.

Clean-slate approaches. Finally, entirely new kernel architectures have also been proposed. X-kernel [54] is an early work that proposes an OS designed to simplify building and composing communication protocols; it includes abstractions and building blocks to realize a wide range of protocols to be used within and across hosts. More recently, Zhang et al. proposed Demikernel [107], an OS architecture that aims at integrating legacy control plane software with a fast data path bypassing the kernel. Also motivated by the lack of system integration, Sadok et al. [96] make the case for an interposition layer on SmartNICs and through a custom OS kernel that can see all traffic before software processing starts. All three approaches have in common that they propose

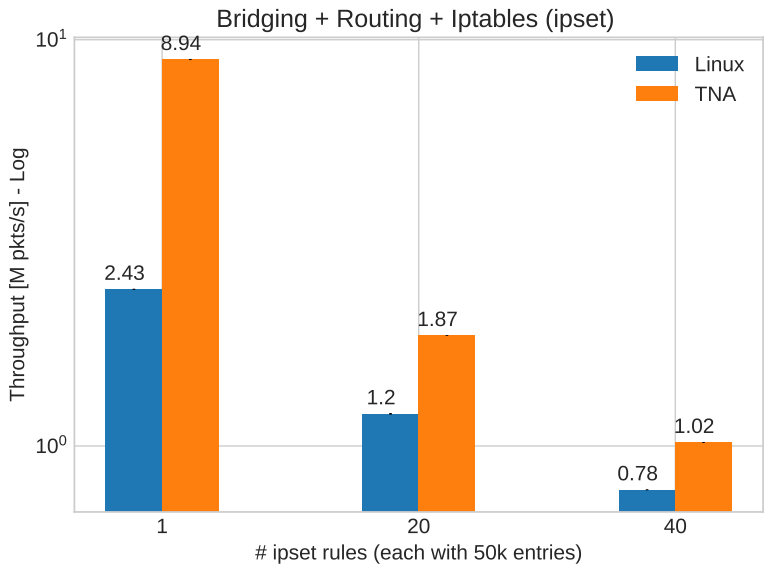

Figure 5.12: Throughput of Bridge + Router + filtering (ipset) Implementations.

completely new kernels and radical changes to OS architecture. While achieving similar goals, TNA can be deployed today, as it can be implemented using mechanisms Linux already provides.

5.6 Conclusion and Future Work

In this paper, we propose a redesign of the Linux network stack, making it more suitable to address the needs of modern network systems in terms of performance, functionality and extensibility. The redesign starts from the observation that it is possible to instantiate a fast data path to Linux, covering only functionality that is actually in use on the system, avoiding many overheads that slow down Linux packet processing. We show that this can be achieved with technology that is currently available in the Linux kernel. To test our proposal, we build a prototype system, called TNA, and show that we can transparently accelerate many Linux network subsystems (i.e., bridging, routing and packet filtering).

There is still work to do to fully realize our vision. First, we need to do a more comprehensive analysis of the Linux kernel network stack to support decomposing more of it with our proposed redesign. Second, we need to investigate techniques for building and optimizing the TNA dependency

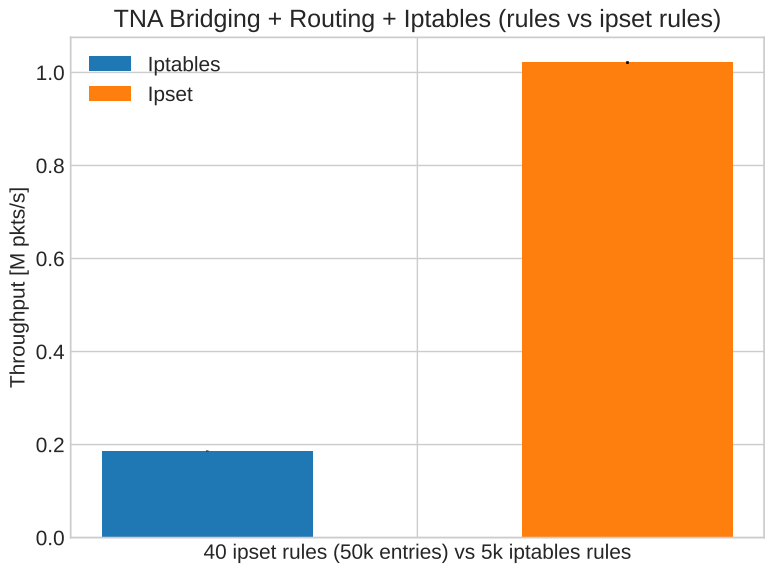

Figure 5.13: Throughput of TNA filtering (ipset vs iptables) Implementations.

graph, and to generate and deploy code based on it. Third, we need to come up with a model that can ensure correctness and consistency in the data plane as we insert custom processing. Finally, we will explore debugging mechanisms considering the new network stack design.

Chapter 6

Future Work and Conclusion

6.1 Future Work

In this section, we discuss avenues to extend this work: Adding support to more use cases for TNA and improving filtering performance.

6.1.1 Adding support to more use cases for TNA

Currently, TNA can accelerate Linux bridging, routing and filtering on the forwarding chain. However, our goal is to rethink all of Linux networking functions. In order to extend the range of services that TNA can accelerate, there is the need to characterize more Linux network subsystems looking for opportunities to break their processing in fast and slow paths. In doing this characterization, it will be possible to generalize the findings and create a taxonomy of processing and state access to build more fast path modules. This can help guide current and future Linux networking functionality, as well as inform how to organize custom processing modules. We believe that one particularly interesting avenue would be to apply XDP/eBPF acceleration to container networking, as other works [103, 31, 77, 83] already do, but fall short in having a closer integration with the Linux network stack, missing several opportunities as discussed during this work.

6.1.2 Improving filtering performance

As discussed in chapter 5, our TNA filtering leverages the widely deployed iptables kernel implementation. This allows TNA to seamlessly accelerate different kernel packet processing ap-

plications that need to perform packet filtering. However, TNA filtering also inherits the iptables' scalability problem that limits its performance when several rules per packet need to be checked. One avenue for future work, would be to improve the iptables matching algorithm itself, which would benefit not only TNA but also other iptables deployments. Another possibility would be to create a caching layer on XDP that could readily apply a verdict on a subset of packets, avoiding the need to traverse the whole iptables tables. As iptables performs a linear search until there is a match for each packet, another alternative would be to have new mechanisms to dynamically reorganizing filtering rules based on frequency of matching. In this manner, rules with higher frequency could be moved up on iptables, so we could avoid several packets to be matched against several rules before processing continues.

6.2 Conclusion

In this dissertation, we identify many challenges faced by modern network systems, most of them related to the impossibility to match their requirements with optimal processing technologies. This leads to system designs that need to trade off having high performance with rich functionality and efficiency. To address this, we start by breaking down the desired processing requirements of many network functions and characterize the processing features of many technologies. After that, we match each processing function with the technology that can most benefit it. In order to have complete packet processing applications in this scenario, we break the boundaries among different technologies, leading to a new foundation that can provide high performance, rich functionality, and efficiency for many network applications. With this foundation, we introduce systems to address many needs of modern data center networks, going from L2 to L7 and monitoring.

We started by introducing a new system that can integrate a high-performance userspace TCP stack with kernel functionality, making it more efficient, allowing better resource sharing capabilities, and enabling cooperation mechanisms between that stack and the kernel, giving it a richer functionality set. We also introduced new monitoring primitives that allow to intelligently orchestrate monitoring applications, only executing them when they are needed. This system is

built using modern in-kernel packet processing, SmartNIC offloads, and userspace processing leading to high-coverage monitoring systems, with efficient resource consumption and rich functionality. Finally, we proposed a redesign of the Linux network stack, which addresses many of its inefficiencies and allows it to have the required performance to support modern services. The redesign starts with the observation that it is possible to break down network processing in a slow path, leveraging kernel features and a fast path. The fast path can be dynamically instantiated, based on the current processing requirements, allowing a shorter processing path that minimizes overheads and greatly improves performance. Unlikely systems that work independently of the Linux network stack, TNA allows leveraging its rich ecosystem composed by management tools, control plane software and protocol implementation.

Bibliography

[1] Author's Linux kernel fork - new BPF helper implementations. https://github.com/mcabranches/linux.

[2] Author's mTCP fork - AF_XDP support to mTCP. https://github.com/mcabranches/mtcp.

[3] Bluefield SmartNIC Ethernet. https://www.mellanox.com/products/ BlueField-SmartNIC-Ethernet.

[4] The caida ucsd anonymized internet traces - mar. 2018. Retrieved July 5, 2021 from `https://www.caida.org/catalog/datasets/passive_dataset`.

[5] Cloudflare, How to drop 10 million packets per second. https://blog.cloudflare.com/how-to-drop-10-million-packets/.

[6] Deloitte, Media and entertainment industry outlook trends. https://www2.deloitte.com/us/en/pages/technology-media-and-telecommunications/articles/media-and-entertainment-industry-outlook-trends.html.

[7] Dennard Scaling and Other Power Considerations. https://pensando.io/dennard-scaling-and-other-power-considerations/.

[8] DPDK, Data Plane Development Kit. https://www.dpdk.org/.

[9] Ever deeper with bpf – an update on hardware offload support - nov. 2018. Retrieved July 5, 2021 from `https://www.netronome.com/blog/ever-deeper-bpf-update-hardware-offload-support/`.

[10] Linux, bpf-helpers(7) — Linux manual page. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[11] Micrsoft, Remote work trend report. https://www.microsoft.com/en-us/microsoft-365/blog/2020/04/09/remote-work-trend-report-meetings/.

[12] P4 Language Consortium. https://p4.org/.

[13] Prototype Kernel, eBPF - extended Berkeley Packet Filter. https://prototype-kernel.readthedocs.io/en/latest/bpf/.

[14] Red Hat, Mobile Networks - Performance and Optimization. `https://access.redhat.com/documentation/en-us/reference_architectures/2017/html/deploying_mobile_networks_using_network_functions_virtualization/performance_and_optimization`.

[15] The Cloudlab Manual, Hardware. https://docs.cloudlab.us/hardware.html.

[16] Ieee standard for verilog hardware description language. IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), pages 1–590, 2006.

[17] Marcelo Abranches and Eric Keller. A userspace transport stack doesn't have to mean losing linux processing. In Proc. NFV-SDN, 2020.

[18] Marcelo Abranches, Oliver Michel, Eric Keller, and Stefan Schmid. Efficient network monitoring applications in the kernel with ebpf and xdp. In 2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), pages 28–34. IEEE, 2021.

[19] David Ahern. Leveraging kernel tables with xdp. In Linux Plumbers Conference, 2018.

[20] Zaafar Ahmed, Muhammad Hamad Alizai, and Affan A. Syed. Inkev: In-kernel distributed network virtualization for dcn. ACM SIGCOMM Comput. Commun. Rev., 46(3), July 2018.

[21] Paul Aitken, Benoît Claise, and Brian Trammell. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. RFC 7011, 2013.

[22] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), pages 5–16. IEEE, 2015.

[23] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine grained traffic engineering for data centers. In Proceedings of the Seventh Conference on Emerging Networking EXperiments and Technologies, CoNEXT '11. ACM, 2011.

[24] Gilberto Bertin. Xdp in practice: integrating xdp in our ddos mitigation pipeline - netdev 2.1. Retrieved June 28, 2021 from `https://legacy.netdevconf.info/2.1/session.html?bertin`.

[25] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating linux security with ebpf iptables. In Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos, pages 108–110, 2018.

[26] Linux, bridge(8) — Linux manual page. https://man7.org/linux/man-pages/man8/bridge.8.html, 2012. Retrieved June 10, 2022.

[27] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hxdp: Efficient software packet processing on FPGA nics. In 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI '20. USENIX, 2020.

[28] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In Proceedings of the 2021 ACM SIGCOMM 2021 Conference, pages 65–77, 2021.

[29] Cyril Cassagnes, Lucian Trestioreanu, Clement Joly, and Radu State. The rise of ebpf for non-intrusive performance monitoring. In Proc. NOMS, 2020.

[30] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In Proc. ACM SIGCOMM, 2020.

[31] Cilium. ebpf-based networking, observability, and security. Retrieved February 22, 2022, from `https://cilium.io/`.

[32] Cisco. Understanding and tuning spanning tree protocol timers, 2006. Retrieved June 14, 2022, from https://www.cisco.com/c/en/us/support/docs/lan-switching/spanning-tree-protocol/19120-122.html.

[33] Cloudflare. Dns flood ddos attack. Retrieved June 25, 2021 from `https://www.cloudflare.com/learning/ddos/dns-flood-ddos-attack/`.

[34] Cloudflare. Slowloris ddos attack. Retrieved June 15, 2021 from `https://www.cloudflare.com/learning/ddos/ddos-attack-tools/slowloris/`.

[35] Cloudflare. Syn flood attack. Retrieved June 15, 2021 from `https://www.cloudflare.com/learning/ddos/syn-flood-ddos-attack/`.

[36] Cloudflare. Broken packets: Ip fragmentation is flawed. https://blog.cloudflare.com/ip-fragmentation-is-broken/, 2017. Retrieved June 10, 2022.

[37] Intel Corporation. Pktgen - traffic generator powered by dpdk. Retrieved February 22, 2022, from `https://github.com/pktgen/Pktgen-DPDK`.

[38] Gregory Cusack, Oliver Michel, and Eric Keller. Machine learning-based detection of ransomware using SDN. In Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks and Network Function Virtualization, SDN-NFV Sec. '18. ACM, 2018.

[39] Andrea di Pietro, Felipe Huici, Nicola Bonelli, Brian Trammell, Petr Kastovsky, Tristan Groleat, Sandrine Vaton, and Maurizio Dusi. Toward composable network traffic measurement. In 2013 Proceedings IEEE INFOCOM 2013, INFOCOM '13, 2013.

[40] Alessandro D'Alconzo, Idilio Drago, Andrea Morichetta, Marco Mellia, and Pedro Casas. A survey on big data for network traffic monitoring and analysis. IEEE Transactions on Network and Service Management, 16(3):800–813, 2019.

[41] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In USENIX NSDI, 2016.

[42] Paul Emmerich, Maximilian Pudelko, Simon Bauer, and Georg Carle. User space network drivers. In Proc. ACM/IEEE ANCS, 2019.

[43] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. Partition-aware packet steering using xdp and ebpf for improving application-level parallelism. In Proc. ACM ENCP, 2019.

[44] Facebook. Katran - l4 load balancer. Retrieved February 22, 2022, from `https://github.com/facebookincubator/katran`.

[45] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI '18. USENIX, 2018.

[46] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In AofA: Analysis of Algorithms, DMTCS Proceedings, June 2007.

[47] Open Networking Foundation. Aether, 2022. Retrieved June 16, 2022, from https://opennetworking.org/aether.

[48] The Linux Foundation. iproute2, 2022. Retrieved June 21, 2022, from https://wiki.linuxfoundation.org/networking/iproute2.

[49] FRR Project. FRRouting project, 2022. Retrieved June 14, 2022, from https://frrouting.org.

[50] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven Streaming Network Telemetry. In Proc. SIGCOMM, 2018.

[51] 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, pages 54–66, 2018.

[52] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, et al. The express data path: Fast programmable packet processing in the operating system kernel. In Proc. ACM CoNEXT, 2018.

[53] Qun Huang, Haifeng Sun, Patrick P. C. Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20. ACM, 2020.

[54] N. Hutchinson and L. Peterson. Design of the x-kernel. In ACM SIGCOMM, 1988.

[55] Fortune Business Insights. Linux operating system market size. Retrieved February 22, 2022, from `https://www.fortunebusinessinsights.com/linux-operating-system-market-103037`.

[56] Van Jacobson and Michael J. Karels. Congestion avoidance and control. In SIGCOMM 1988, Stanford, CA, August 1988.

[57] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mos: A reusable networking stack for flow monitoring middleboxes. In 14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17), pages 113–129, 2017.

[58] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level {TCP} stack for multicore systems. In 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14), pages 489–502, 2014.

[59] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In 14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17), pages 97–112, 2017.

[60] Magnus Karlsson and Björn Töpel. The path to dpdk speeds for af xdp. In Linux Plumbers Conference, 2018.

[61] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. Transactions on Computer Systems, 18(3), aug 2000.

[62] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In USENIX NSDI, 2014.

[63] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, et al. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In Proc. ACM SIGCOMM, 2017.

[64] L3af. Complete lifecycle management of ebpf programs in the kernel. Retrieved February 22, 2022, from `https://l3af.io/`.

[65] TV Lakshman and Dimitrios Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. ACM SIGCOMM Computer Communication Review, 28(4):203–214, 1998.

[66] Rafael Laufer, Massimo Gallo, Diego Perino, and Anandatirtha Nandugudi. Climb: Enabling network function composition with click middleboxes. ACM SIGCOMM Computer Communication Review, 46(4):17–22, 2016.

[67] Leonardo Balliache. Querying libiptc howto, 2022. Retrieved October 24, 2022, from https://tldp.org/HOWTO/Querying-libiptc-HOWTO/index.html.

[68] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: a better netflow for data centers. In Proc. USENIX NSDI, 2016.

[69] Guyue Liu, Yuxin Ren, Mykola Yurchenko, KK Ramakrishnan, and Timothy Wood. Microboxes: high performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, pages 504–517. ACM, 2018.

[70] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-efficient microservices on smartnic-accelerated servers. In 2019 USENIX Annual Technical Conference, ATC '19. USENIX, 2019.

[71] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '16. ACM, 2016.

[72] Chris A Mack. Fifty years of moore's law. IEEE Transactions on semiconductor manufacturing, 24(2):202–207, 2011.

[73] Pilar Manzanares-Lopez, Juan Pedro Muñoz-Gea, and Josemaria Malgosa-Sanahuja. Passive in-band network telemetry systems: The potential of programmable data plane on network-wide telemetry. IEEE Access, 9, 2021.

[74] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A microkernel approach to host networking. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, pages 399–413, 2019.

[75] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. Securing linux with a faster and scalable iptables. ACM SIGCOMM Computer Communication Review, 49(3):2–17, 2019.

[76] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In Proc. IEEE HPSR, 2018.

[77] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. A framework for ebpf-based network functions in an era of microservices. IEEE Transactions on Network and Service Management, 18(1), 2021.

[78] Oliver Michel, Roberto Bifulco, Gábor Rétvári, and Stefan Schmid. The programmable data plane: Abstractions, architectures, algorithms, and applications. ACM Computing Surveys, 54(4), May 2021.

[79] Oliver Michel, John Sonchack, Greg Cusack, Maziyar Nazari, Eric Keller, and Jonathan M. Smith. Software packet-level network analytics at cloud scale. IEEE Trans. on Network and Service Management, 18(1), 2021.

[80] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In Proc. NDSS, 2018.

[81] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13). USENIX, 2013.

[82] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. Acceltcp: Accelerating network applications with stateful {TCP} offloading. In 17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20), pages 77–92, 2020.

[83] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. BASTION: A security enforcement network stack for container networks. In USENIX ATC, July 2020.

[84] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In Proc. ACM SIGCOMM, 2017.

[85] netlink(7) — Linux manual page. https://man7.org/linux/man-pages/man7/netlink.7.html, 2021. Retrieved June 10, 2022.

[86] Netronome Systems Inc. Netronome nfp-4000 flow processor product brief. Retrieved June 17, 2021 from `https://www.netronome.com/media/documents/PB_NFP-4000-7-20.pdf`.

[87] Thuy TT Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. IEEE Communications Surveys & Tutorials, 10(4):56–76, 2008.

[88] NTOP. PF_RING: High-speed packet capture, filtering and analysis, 2022. Retrieved June 13, 2022, from https://www.ntop.org/products/packet-capture/pf_ring.

[89] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In 16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19), pages 361–378, 2019.

[90] Konstantina Papagiannaki, Rene Cruz, and Christophe Diot. Network performance monitoring at small time scales. In Proc. IMC, 2003.

[91] Jiri Pirko and Scott Feldman. Ethernet switch device driver model (switchdev). Retrieved February 22, 2022, from `https://www.kernel.org/doc/html/latest/networking/switchdev.html`.

[92] Cisco Press. Bgp fundamentals. https://www.ciscopress.com/articles/article.asp?p=2756480, 2018. Retrieved June 10, 2022.

[93] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In 2012 USENIX Annual Technical Conference (USENIX ATC 12), pages 101–112, Boston, MA, 2012. USENIX Association.

[94] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In 2012 USENIX Annual Technical Conference, ATC '12. USENIX, 2012.

[95] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the Social Network's (Datacenter) Network. In Proc. SIGCOMM, 2015.

[96] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. We need kernel interposition over the network dataplane. In ACM HotOS, 2021.

[97] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In Proc. USENIX NSDI, 2012.

[98] John Sonchack, Oliver Michel, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With *Flow. In Proc. USENIX ATC, 2018.

[99] Anna Sperotto, Gregor Schaffrath, Ramin Sadre, Cristian Morariu, Aiko Pras, and Burkhard Stiller. An overview of ip flow-based intrusion detection. IEEE communications surveys & tutorials, 12(3):343–356, 2010.

[100] strongSwan Project. strongswan: the opensource ipsec-based vpn solution, 2022. Retrieved June 16, 2022, from https://www.strongswan.org.

[101] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. Nfp: Enabling network function parallelism in nfv. In Proc. ACM SIGCOMM, 2017.

[102] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In Proc. USENIX NSDI, 2018.

[103] Tigera, Inc. Project calico, 2022. Retrieved June 15, 2022, from https://www.tigera.io/project-calico.

[104] weaveworks. Weave Net, 2022. Retrieved June 16, 2022, from https://www.weave.works/oss/net.

[105] Minlan Yu. Network telemetry: Towards a top-down approach. ACM SIGCOMM Comput. Commun. Rev., 49(1), February 2019.

[106] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, et al. Quantitative network monitoring with netqre. In Proc. ACM SIGCOMM, 2017.

[107] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In ACM SOSP, 2021.

[108] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. Newton: Intent-driven network traffic monitoring. In Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '20. ACM, 2020.

[109] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. Netfpga sume: Toward 100 gbps as research commodity. IEEE micro, 34(5):32–41, 2014.

[110] Hubert Zimmermann. Osi reference model-the iso model of architecture for open systems interconnection. IEEE Transactions on communications, 28(4):425–432, 1980.