

École polytechnique de Louvain

Accelerating the Linux TCP stack via eBPF kernel bypass

Author: **Noa QUENON**
Supervisor: **Tom BARBETTE**
Readers: **Clément DELZOTTI, Matthieu PIGAGLIO**
Academic year 2025–2026
Master [120] in Computer Science

Abstract

The Linux TCP stack, although feature-complete and extensively tested, imposes a significant per-packet overhead that quickly becomes a bottleneck in high-throughput, low-latency workloads. Existing acceleration techniques either abandon the kernel entirely, sacrificing security, observability, and application compatibility, or preserve the kernel at the cost of modest performance improvements. This thesis explores a middle ground by using eBPF and its XDP and TC hooks to bypass the most expensive parts of the data path while retaining the kernel's control plane, security guarantees, and native tooling.

We design and implement a hybrid system combining a small loadable kernel module that exposes the kernel functions necessary for memory management and packet injection with user-supplied eBPF programs. Two bypass variants are presented: a stateless variant that accelerates established flows unconditionally and a stateful variant that interacts with Linux's Netfilter subsystem to implement flow offloading. A comprehensive evaluation of throughput, latency, CPU utilization, and scalability shows that the bypass delivers up to a 4x increase in throughput and reduces CPU cycles per megabyte of throughput by a similar factor in configurations that are usually unfavorable to the standard stack. The results confirm that eBPF provides a safe foundation for TCP acceleration while requiring no modification of existing applications or kernel rebuilds.

Acknowledgments

I would like to thank Prof. Barbette, who guided me throughout my research and offered invaluable expertise on kernel bypass techniques. My sincere thanks also go to Clément Delzotti, Ahmed Lahmer and Vladimiro Paschali, who followed my progress week by week and provided essential feedback; to Matthieu Pigaglio for kindly accepting to review my thesis and sit on my defense jury; to Matthieu Baerts for his thorough explanations of the Linux networking stack; to Maxime Just for his preliminary work on the subject, on which this thesis builds; and to Nikita Tyunyayev, who first introduced me to the wonderful world of eBPF and XDP programming.

Finally, I am profoundly grateful for the unwavering support of my family, my girlfriend and my friends. This academic year presented its share of challenges, and I could not have reached this milestone without their continuous encouragement.

Regarding the use of artificial intelligence

During the research phase of this thesis, artificial intelligence (and specifically generative AI) was utilized as a supplementary tool to gather, discuss, and compare ideas. However, the code, technical decisions, and the written content of this manuscript are entirely my own original work and reflect my personal understanding of the subject matter.

Additionally, AI-assisted writing tools, namely DeepL Write (<https://deepl.com/write>) and Language-Tool (<https://languagetool.org>), were employed during the drafting process. These tools were used exclusively for proofreading, grammar correction, and stylistic refinement of my original drafts, rather than for generating novel text.

Contents

1 Context and motivations	1
2 Background and state of the art	4
2.1 Introduction	5
2.2 The Linux network stack	5
2.2.1 The New Application Programming Interface	5
2.2.2 The socket buffer structure (sk_buff)	6
2.2.3 User-space copy and socket contention	7
2.3 Traditional kernel bypass and acceleration	7
2.3.1 Linux's optimizations	7
2.3.2 Existing research	8
2.3.2.1 mTCP	8
2.3.2.2 IX	9
2.3.2.3 TAS	10
2.3.2.4 StackMap	11
2.3.3 Summary	12
2.4 The eBPF and XDP ecosystem	13
2.4.1 eBPF maps	13
2.4.2 The eBPF verifier	14
2.4.2.1 Validating memory accesses	15
2.4.2.2 Validating pointer dereferences	15
2.4.2.3 Validating run to completion	15
2.4.3 XDP	16
2.4.4 Traffic control hook	17
3 Implementation	19
3.1 Introduction	20
3.2 Architectural overview	20
3.3 Kernel module and functions	21
3.3.1 Finding an entry point in the TCP stack	21
3.3.2 XDP kernel function	22
3.3.2.1 Verifying socket state	23
3.3.2.2 Allocating the socket buffer	23
3.3.2.3 Populating the socket buffer	25

3.3.3 TC kernel function	25
3.3.4 Injection into the TCP state machine	26
3.3.4.1 A note on socket ownership semantics	27
3.4 Stateless eBPF bypass	27
3.5 Stateful flow offloading	28
4 Performance evaluation	31
4.1 Introduction	32
4.2 Experimental setup and methodology	32
4.3 Throughput analysis	32
4.4 Latency analysis	34
4.5 CPU utilization and efficiency	35
4.5.1 Profiling	35
4.6 Scalability and connection dynamics	39
4.7 Summary	42
5 Conclusion	43
Bibliography	45
A Partial flamegraphs	49
B Complete flamegraphs	51

List of Figures

2.1	A standard TCP/IP packet.....	5
2.2	A standard TCP/IP packet with its socket buffer.....	6
2.3	Thread model of mTCP, running entirely in user space [1].....	9
2.4	The IX dataplane operating system [2].....	10
2.5	TAS receive flow [3].....	11
2.6	StackMap architecture overview [4].....	12
2.7	Overview of eBPF's inner workings [5].....	13
3.1	Structure of the implementation. Red arrow indicates normal packets, green arrow indicates accelerated ones.....	20
3.2	Path of an established-flow TCP packet through the kernel, grouped by responsibility. The TCP state machine is highlighted in green.....	21
3.3	Step-by-step sk_buff pointer adjustments to encapsulate raw XDP page memory.....	24
3.4	A standard TCP/IP packet and its associated sk_buff pointers before injection.....	27
3.5	Structure of the flow offloading implementation. Red arrow indicates normal packets, green arrow indicates accelerated ones, dotted arrows indicate cache flow.....	29
4.1	Throughput per variant across different configurations (higher is better).....	33
4.2	Throughput per variant across different configurations during a worst-case scenario (MSS set to 80 bytes, higher is better).....	34
4.3	Average and P99 latency per variant across different configurations (lower is better).....	35
4.4	Branch misses per megabyte of throughput per variant (log scale, lower is better).....	35
4.5	Cycles per megabyte of throughput per variant (log scale, lower is better).....	35
4.6	Cache misses per megabyte of throughput per variant (lower is better).....	35
4.7	CPU cycle breakdown by functional category for configurations (offloading disabled, Netfilter enabled).....	37
4.8	CPU cycle breakdown by functional category for configurations (offloading disabled, Netfilter disabled).....	38
4.9	Mean throughput per variant across all flows of 100 MB with hardware offloading disabled (higher is better).....	40
4.10	Mean throughput of each bypass variant relative to the baseline (offloading disabled, Netfilter enabled) versus concurrent flow count of 100 MB (higher is better).....	40
4.11	Per-flow throughput across variants with offloading disabled (higher is better).....	41
A.1	Flamegraph of the baseline during a throughput test zoomed on the stack entry point (offload disabled, Netfilter enabled).....	49

A.2	Flamegraph of the XDP bypass during a throughput test zoomed on the BPF entry point (offload disabled, Netfilter disabled).....	50
A.3	Flamegraph of the TC bypass during a throughput test zoomed on the stack entry point (offload disabled, Netfilter disabled).....	50
B.1	Flamegraph of the baseline during a throughput test (offloading disabled, Netfilter disabled)...	51
B.2	Full flamegraph of the baseline during a throughput test (offloading disabled, Netfilter enabled).	52
B.3	Full flamegraph of the XDP bypass during a throughput test (offloading disabled, Netfilter disabled).....	52
B.4	Full flamegraph of the XDP bypass during a throughput test (offloading disabled, Netfilter enabled).....	53
B.5	Full flamegraph of the XDP bypass during a stress test (offloading disabled, Netfilter enabled).	53
B.6	Full flamegraph of the TC bypass during a throughput test (offloading disabled, Netfilter disabled).	54
B.7	Full flamegraph of the TC bypass during a throughput test (offloading disabled, Netfilter enabled).	54

List of Algorithms

3.1 Basic flow offloading bypass.....	29
---------------------------------------	----

Listings

2.1	Annotation of the <code>bpf_map_lookup_elem</code> eBPF helper in the Linux kernel v6.19.....	15
2.2	An XDP program that drops IPv4 packets.....	17
3.1	Kernel function signature for the XDP fast path.....	22
3.2	Implementation of the <code>tcp_inject_packet</code> function.....	26
3.3	Implementation of the stateless XDP program.....	28

Glossary

flow offloading: A performance optimization where the first packet of a flow is validated and routed by Netfilter, while all subsequent packets bypass the standard network stack for faster forwarding. 20, 28

hard IRQ – hardware interrupt: A physical signal from hardware sent to the CPU that immediately preempts it to run a minimal, time-critical handler. 5

kfunc – kernel function: A function within the kernel that has been annotated and designed to be called from eBPF programs. 3, 14, 20, 21, 22, 23, 25, 26, 27, 28, 38

soft IRQ – software interrupt: A deferred kernel task, scheduled by a hard IRQ, that performs a time-consuming work when the CPU has available cycles. 2, 6, 7, 16, 23

socket – socket: A software endpoint that serves as a bidirectional communication interface, allowing user-space applications to send and receive data across a network through the kernel's networking stack. 2, 3, 7, 9, 11, 20, 23, 26, 27, 38, 39

sk_buff – socket buffer: The core data structure used by the Linux kernel's networking stack to manage individual network packets. 2, 6, 7, 16, 17, 18, 21, 22, 23, 24, 25, 26, 33, 38, 39

zero-copy: An optimization technique that eliminates the need for the CPU to copy data between memory spaces during I/O operations. 7, 11, 12

Abbreviations

BTF – BPF Type Format: A metadata format that provides type information for the Linux kernel and eBPF programs. It enables the kernel to describe its own internal data structures, facilitating portability across different kernel versions through mechanisms like CO-RE. 21

DMA – Direct Memory Access: A mechanism that allows hardware subsystems to access main system memory independently of the CPU, greatly increasing data transfer speeds and reducing CPU load. 5

eBPF – Extended Berkeley Packet Filter: A technology that safely executes sandboxed programs within the operating system kernel, dynamically extending its networking, security, and observability capabilities. 2, 3, 5, 13, 14, 15, 16, 17, 18, 20, 21, 23, 26, 27, 28, 32, 38, 44

GRO – Generic Receive Offload: A software-level optimization improving on LRO. It groups multiple incoming packets from the same TCP connection at the entrypoint of the kernel's network stack. 7, 8, 32, 44

GSO – Generic Segmentation Offload: A software-level optimization extending TSO. It groups generic packets into a massive one (up to 64KB) and splits it right before passing it to the NIC. 7, 8, 32, 44

LKM – Loadable Kernel Module: A piece of code that can be dynamically loaded into the Linux kernel to extend its functionality. It has full access to kernel data structures. 3, 14, 20, 21, 25, 27

LRO – Large Receive Offload: A hardware-level optimization that groups multiple incoming packets from the same TCP connection at the NIC before passing it to the kernel's network stack. 7, 8, 32

MSS – Maximum Segment Size: A TCP parameter specifying the largest amount of payload data (in bytes) that a host is willing to receive in a single segment. 33

NAPI – New Application Programming Interface: A framework in the Linux networking stack designed to improve high-speed packet processing. It reduces CPU overhead by mitigating hardware interrupt storms through a hybrid model of hardware interrupts and software polling. 5, 6

NIC – Network Interface Controller: A hardware component that connects a computer to a network. It is responsible for transmitting and receiving network packets at the physical and data link layers. 2, 5, 6, 7, 8, 11, 12, 16, 20, 25, 32, 36

RFS – Receive Flow Steering: A software-level optimization improving on RPS. It tracks on which core applications are listening for packets and steers the relevant flows to them. 8

RPS – Receive Packet Steering: The software equivalent of RSS. It allows one-queue NICs to distribute the load on multiple cores despite not having the required hardware to use RSS. 8

RSS – Receive Side Scaling: A hardware-level feature that allows NICs that have multiple queues to distribute the load efficiently, each queue being pinned to a particular CPU core. 8, 10

TC – Traffic Control: A kernel subsystem designed for packet scheduling, shaping, and policing. It provides eBPF hooks on both the ingress and egress paths of a network interface, allowing packet manipulation with full access to the socket buffer structure. 3, 5, 17, 18, 21, 25, 33, 38, 39

TCP – Transmission Control Protocol: An internet protocol that establishes a connection between devices to ensure the reliable, ordered, and error-checked delivery of data packets across a network. 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 16, 20, 21, 22, 23, 24, 25, 26, 27, 29, 32, 33, 36, 38, 39, 44

TSO – TCP Segmentation Offload: A hardware-level optimization that groups multiple TCP segments into a massive one (up to 64KB) and hands it to the NIC, which is responsible of breaking it into MTU-sized packets. 7, 8, 32, 44

XDP – eXpress Data Path: A Linux kernel feature that allows eBPF programs to process incoming network packets directly at the network driver level for maximum performance and minimal latency. 3, 5, 13, 15, 16, 17, 18, 21, 22, 23, 24, 25, 27, 28, 32, 33, 37, 38, 39, 44

1

Context and motivations

Since its initial publication by Vint Cerf et al. in December 1974, the Transmission Control Protocol (TCP) [6] has evolved far beyond its original specifications. Originally designed as a reliable transport protocol for a few interconnected research machines, TCP now supports the vast majority of global Internet traffic, including web browsing, video streaming, distributed storage, and high-frequency trading. Despite the emergence of alternatives like QUIC [7], it remains the de facto transport protocol of the modern Internet [8]. Its performance directly shapes the experience of billions of users, as well as the operating costs of data centers worldwide.

Over the past five decades, TCP has undergone numerous optimizations, including the implementation of new congestion control algorithms [9], [10], selective acknowledgments [11], window scaling [12], segmentation offloads [13], and zero-copy primitives [14] among many others. The Linux kernel implementation, which serves as the basis for most production deployments, comprises approximately 144,000 lines of code within a network subsystem containing nearly a million lines [15]. This complexity is the price paid for the protocol's generality, security, and feature completeness, but it is also the source of a growing performance gap.

The performance of a single CPU thread has not advanced nearly as quickly as network hardware. A modern 100 Gbps Network Interface Controller (NIC) can deliver a 64-byte packet every 5.12 nanoseconds, leaving the operating system only a few dozen CPU cycles to classify, demultiplex, and deliver each packet. Empirical studies have shown that processing a single bit of TCP traffic in the Linux stack uses approximately one hertz of CPU time [16], meaning that fully saturating a 100 Gbps link to a single application would require nearly the entirety of a 100 GHz processor's worth of cycles (a clear impossibility on currently available hardware). The causes are well documented: packet-based interrupts, soft IRQ scheduling, deep call chains, cache pollution due to large `sk_buff` structures, and lock contention on shared `socket` states [17]. While each factor individually has a modest impact, collectively they cause the kernel to become the limiting factor in high-throughput, low-latency workloads rather than the hardware.

The networking community has responded with a variety of solutions. On one end of the spectrum, full kernel-bypass frameworks such as DPDK [18] and Netmap [19], as well as user-space TCP stacks like mTCP [1], IX [2], and TAS [3], deliver order-of-magnitude increases in throughput and reductions in latency. These frameworks achieve this by removing the kernel from the data path entirely. Applications poll the NIC directly from user space, manage their own memory, and implement the protocol themselves. However, these approaches introduce drawbacks: existing applications must be rewritten or relinked, kernel tooling (e.g., `tcpdump`, `nftables`, `netstat`) stop working, and the security and completeness guarantees provided by the kernel no longer apply. At the opposite extreme are incremental optimizations around the kernel, such as StackMap [4]. These optimizations preserve the existing stack but yield more modest gains and still require changes to applications. What is missing is a mechanism that accelerates common use cases without compromising the security, observability, and compatibility that make Linux networking attractive in the first place.

The Extended Berkeley Packet Filter (eBPF) [20] offers exactly this capability. Originally designed as a packet filtering mechanism, eBPF has evolved into a general-purpose, security-checked, in-kernel execution environment exposed through a growing set of hooks. Programs run within the kernel, with

native access to packet data and kernel structures, while remaining sandboxed and dynamically loadable without recompilation or reboot.

This master's thesis investigates the feasibility of implementing a kernel bypass for TCP in eBPF that retains most of the control plane (e.g., socket APIs, Netfilter rules, observability tools, security guarantees) while bypassing the most expensive portion of the data path. To this end, this thesis makes the following contributions:

1. A bypass design combining an eBPF program linked to the XDP or TC hook with a small loadable kernel module that exposes the necessary kfuncs for memory management, socket synchronization, and packet injection.
2. Two implementations illustrating the architecture's versatility: a stateless bypass optimized to minimize CPU overhead, and a stateful variant that maintains additional metadata about the connection for richer processing.
3. A comprehensive evaluation across four metrics (throughput, latency, CPU efficiency, and scalability), comparing the different bypass variants to the unmodified Linux network stack under various configurations.

The remainder of this manuscript is organized as follows: Chapter 2 reviews the Linux network stack, analyzes existing work on TCP acceleration, and introduces eBPF and its hooks in detail. Chapter 3 presents the design and implementation of the proposed architecture, including the kernel module, the kfuncs it exposes, and the two bypass variants for each hook. Chapter 4 evaluates the system experimentally and analyzes the results. Chapter 5 concludes and discusses possible directions for future work.

2

Background and state of the art

Contents

2.1 Introduction	5
2.2 The Linux network stack	5
2.3 Traditional kernel bypass and acceleration	7
2.4 The eBPF and XDP ecosystem	13

2.1 Introduction

This chapter explores the various concepts necessary for a thorough understanding of the thesis' subject. First, it explains how the Linux network stack processes a TCP packet, the various optimizations that have been implemented over the years, and the research conducted to improve the stack's throughput and latency. Next, it introduces eBPF and related concepts (such as maps and the verifier), the various hooks that will be used in the implementation (i.e., XDP and TC) in Section 3, as well as the limitations imposed by eBPF's design.

2.2 The Linux network stack

The Linux network stack is a highly complex piece of software. As of version 6.19, the entire `net` directory contains 965,176 lines of code, with the TCP/IP stack alone accounting for 143,719 of them [15].

To understand why bypassing this stack yields significant performance gains, we must first understand the default path a packet takes. We will focus on the receive path, also known as the ingress (packets traveling from the network into our application). Let's consider the case of a TCP segment, carried within an IPv4 packet, as shown below:



Figure 2.1: A standard TCP/IP packet

The packet first enters the computer via the NIC, which confirms that the host is the legitimate recipient by comparing the destination MAC address with the one present in the Ethernet header (marked “MAC” on Figure 2.1). It then performs hardware checksum verification and copies the packet directly into the system's memory via Direct Memory Access (DMA).

Once the packet is in memory, the CPU must be notified of its arrival. Historically, the NIC achieved this by triggering a hardware interrupt (hard IRQ) for every single packet [21]. However, the context switching required to execute the interrupt handler introduces significant overhead. On high-speed networks, this per-packet interruption often leads to a phenomenon known as an “interrupt storm”, a state where the CPU spends more time managing interrupts than processing actual workloads, resulting in severe performance degradation. To mitigate this issue, the New Application Programming Interface (NAPI) was introduced.

2.2.1 The New Application Programming Interface

First published in 2001, NAPI [21] uses a clever hybrid model that smoothly transitions the system between interrupt-driven and polling-driven modes depending on the network load. The process begins with the arrival of the first packet into an empty buffer. The NIC raises a hard IRQ, often referred to as

the “top half” of the interrupt handling process. Upon its reception, the CPU jumps to a minimal handler that has two main roles: it disables all future hardware interrupts from the NIC and schedules another handler, known as the `soft IRQ`, or “bottom half”. By deferring the heavy lifting to this other handler, the CPU is instructed to process the packets when it has the appropriate resources, leaving it free to work on more critical tasks in the meantime.

Once the kernel executes the deferred `soft IRQ`, it invokes a specialized `poll()` function declared in the NIC's driver. At this stage, the system has effectively switched into polling mode. The `poll()` function iteratively traverses the buffer, processing packets one by one. Because a busy network could theoretically keep the buffer full (and thus keep the `poll()` function running indefinitely), `NAPI` implements a budget mechanism to prevent a single device from monopolizing the CPU. The driver is effectively given a maximum quota of packets it is allowed to process in a single polling cycle. If the budget is exhausted while there are still packets waiting in the buffer, it gracefully yields the CPU to allow other tasks to run, and reschedules the `soft IRQ` for later execution.

Conversely, if the network load is manageable and the `poll()` function successfully drains the buffer before reaching its quota, the system returns to its interrupt-driven state: the polling mechanism is deactivated, and the driver re-enables the interrupts on the NIC. As soon as a new packet arrives, the cycle will repeat.

2.2.2 The socket buffer structure (`sk_buff`)

As a packet is processed by the kernel, it passes through several subsystems, each handling a specific protocol layer. For instance, using the example packet in Figure 2.1, after the layer 2 (i.e., Ethernet) processing is complete, the stack forwards the packet to the IPv4 subsystem, which in turn hands it off to TCP. If each subsystem had its own internal representation of the packet and its metadata, the kernel would need to translate these before passing them to the next layer. Instead, the kernel relies on a central data structure, called the `sk_buff`, which acts as a universal container for the packet throughout its traversal of the network stack.

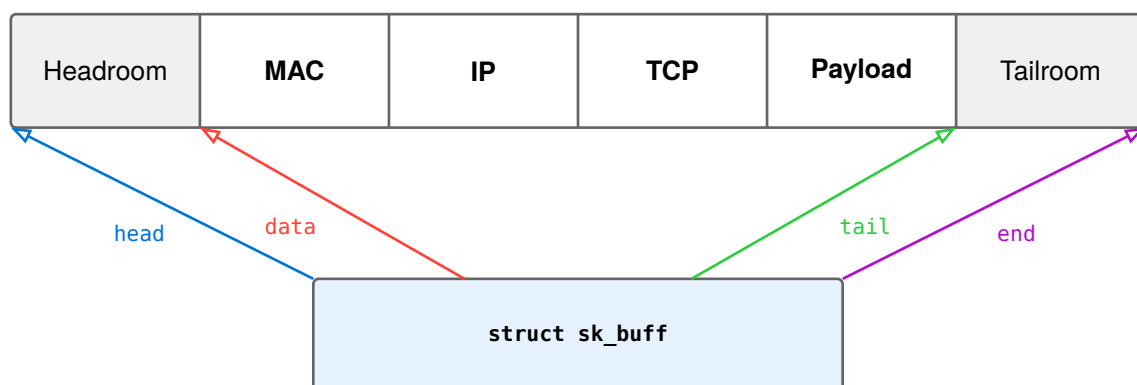


Figure 2.2: A standard TCP/IP packet with its socket buffer

A key advantage of the `sk_buff` is its ability to offer easy, zero-copy header manipulation. As illustrated on Figure 2.2, each structure has four core memory pointers: `head`, `data`, `tail` and `end`. These point, respectively, to the start of the allocated memory buffer, the beginning of the active packet data, the end of the active packet data, and the end of the allocated buffer. By simply moving the data pointer, the network stack encapsulates or decapsulates protocols without reallocating memory. For instance, to strip an Ethernet header, the kernel would only need to advance the data pointer forward to the start of the IP header.

While having a single data structure to represent the packet is practical from a software engineering perspective, it forces the `sk_buff` to act as a “kitchen sink” for all possible networking features. Because it must be universal, it is packed with hundreds of bytes of metadata to support everything the kernel might need to do: as of version 6.19, it counts over 80 distinct fields [22]. Even if a single packet requires none of these features, the kernel must still allocate, populate, and manage this massive structure. This constant allocation and management is one of the primary bottlenecks in high-speed networking [17].

2.2.3 User-space copy and socket contention

Assuming the packet successfully traverses the IP and TCP layers without being dropped, its payload finally reaches the receive queue of the application’s `socket`. At this stage, the packet still resides entirely in kernel memory. To retrieve it, the application must call `read()` or `recv()`, which forces a costly context switch from user mode to kernel mode. Furthermore, reading this data requires synchronization: since the network stack processes packets asynchronously in the `soft IRQ` context, the kernel must acquire locks on the `socket` to safely dequeue the payload without corrupting the memory. This creates a potential point of contention, where the application and the `soft IRQ` are fighting to lock the queue to read and write data. Even after the application thread successfully acquires the locks, the kernel must perform a CPU-intensive memory copy to move the packet from kernel memory to user-space memory [23], [24].

2.3 Traditional kernel bypass and acceleration

For all the reasons listed previously (i.e., interrupts, massive `sk_buff`, user-space copying, and `socket` locking), the network stack can be slow. With the increasing adoption of 100 Gigabit Ethernet NICs in enterprise networks and the upcoming 1.6 Terabit Ethernet IEEE standard [25], the Linux stack is quickly becoming a significant software bottleneck. To counter this, we’ve seen in recent years a plethora of initiatives aiming at improving processing speed, sometimes at the expense of safety, observability or compatibility with current kernel versions.

2.3.1 Linux’s optimizations

A first logical point of potential optimization is the kernel’s codebase itself. Over the years, seven major improvements have been implemented: TCP Segmentation Offload (TSO), Generic Segmentation Offload (GSO), Large Receive Offload (LRO), Generic Receive Offload (GRO), Receive Side Scaling

(RSS), Receive Packet Steering (RPS), and Receive Flow Steering (RFS) [13]. Because TSO and GSO primarily optimize the transmission (egress) path, they fall outside the scope of this paper, which focuses on receive (ingress) performance.

LRO and GRO are designed to reduce per-packet overhead through packet aggregation. As discussed in Section 2.2, every individual packet traversing the stack incurs a fixed CPU cost. By combining different packets from the same connection in a single, larger packet before passing it to the stack, the kernel only has to evaluate and parse headers once, significantly improving throughput. The earliest implementation of this concept, LRO, worked by stripping the headers of incoming packets and concatenating their payloads. While fast, this was a major issue for routers, since it removed important information that they may want to manipulate (e.g., MTU, TTL) [26]. The system was therefore quickly replaced by the safer GRO, a complete re-implementation purely in software of the same principle, with the addition of strict rules preventing the loss of important metadata.

On the other hand, RSS, RPS and RFS are multi-core distribution optimizations. Historically, a single NIC possessed a single receive queue, which triggered interrupts on a single CPU core. At high speeds, that single core would quickly reach 100% utilization, causing delay or dropped incoming packets. RSS emerged when multi-queue NICs entered the market. It allows these more modern cards to distribute their load across multiple queues, each tied to a different CPU core. To prevent multiple packets from the same connection from being handled by different cores, which would make reordering complex and costly, the distribution is based on a hash of the TCP four-tuple. This guarantees that all segments from the same flow end up in the same queue and, consequently, on the same core. RPS follows the same principle as RSS, but is fully implemented in software, meaning that even single-queue NICs can benefit from this multi-core distribution. RFS pushes RPS further, by tracking on which core applications are listening and dispatching the corresponding packets there to take advantage of the different CPU caches.

2.3.2 Existing research

Despite the several improvements introduced to the kernel over the years, research has been focussed on developing faster and more aggressive optimizations. This paper will cover four major contestants in the field, each having a different approach on the matter: mTCP [1], IX [2], TAS [3], and StackMap [4].

2.3.2.1 mTCP

mTCP [1] is an example of a user-space TCP stack. This type of design has two distinct advantages: it reduces the number of context switches and offers great flexibility in implementation, since the network stack is essentially being redeveloped from scratch. In the case of mTCP, this allowed the authors to incorporate several optimizations into their design.

The most important feature of mTCP is likely its per-thread architecture. As demonstrated by Figure 2.3, each application thread benefits from its own mTCP thread and, consequently, its own TCP buffers. This completely eliminates one of the major issues with the classic Linux stack: cross-core contention/locking. Incoming packets are sorted via RSS and RFS implemented in the NIC, ensuring that those belonging to the application reach the correct thread directly. Another feature highlighted by

Figure 2.3 is the presence of a job queue. As explained in Section 2.2.3, whenever an application wants to read from the socket to retrieve its data, it must make a system call to `read()` or `recv()`. These calls are processed directly by the kernel, resulting in numerous context switches. Since, in this configuration, these system calls are replaced by calls to libraries, mTCP can execute additional logic before actually calling `read()` or `recv()`. One of the choices made was to batch the various calls, so that a costly context switch is performed only once, albeit at the cost of latency. Finally, since the Linux stack is primarily optimized for long connections, mTCP has opted to accelerate short ones through two modifications: more efficient timer management and priority-based packet queuing (i.e., prioritizing control packets over data packets).

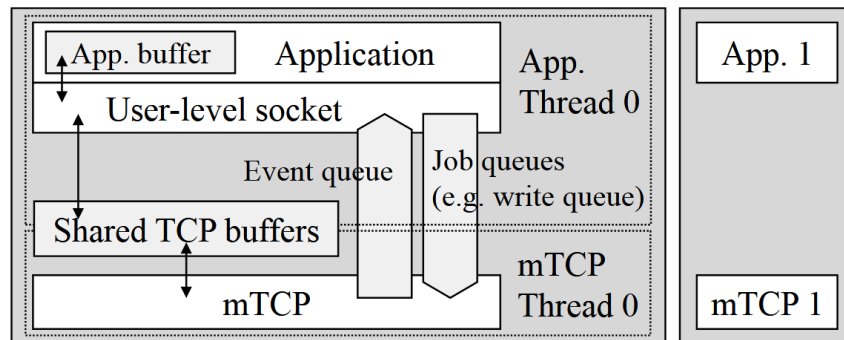


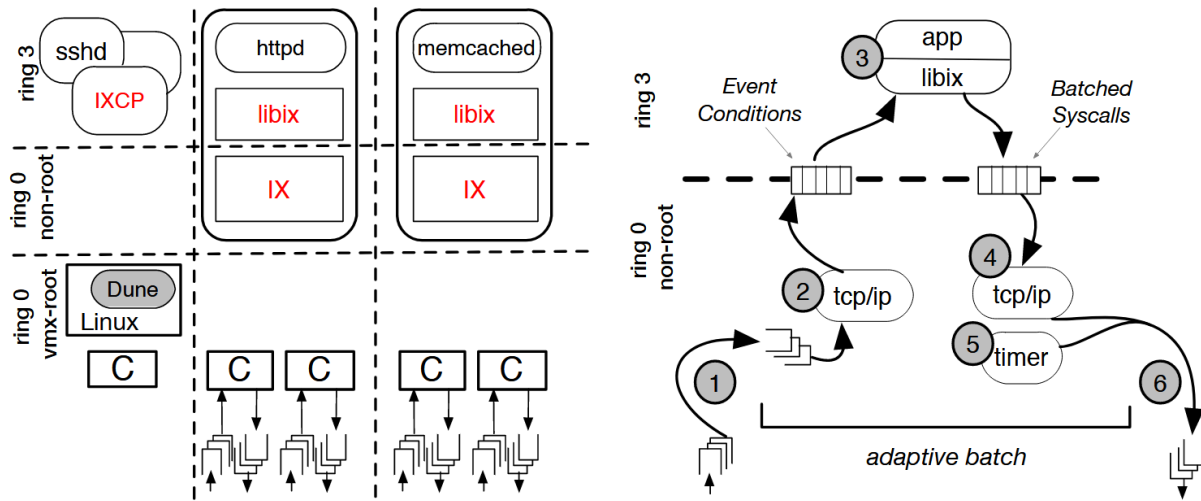
Figure 2.3: Thread model of mTCP, running entirely in user space [1]

mTCP has proven its effectiveness in multiple scenarios. The authors observed an overall 25-fold improvement in performance for small message transactions compared to the standard Linux TCP stack and a 3-fold improvement compared to MegaPipe [27], which was the highest-performing implementation at the time. Significant gains were also observed in real-world applications, with improvements of 33% for SSLShader [28] and up to 320% for lighttpd [29]. The implementation was also able to saturate a 10 Gbps link with segments as small as 1 KiB, which the standard TCP stack could not do at the time. However, mTCP's approach also poses some problems, the main one being that it is difficult to guarantee full TCP compliance. In a 2021 paper on TCP fuzzing [30], researchers demonstrated that mTCP had nine compliance issues with the TCP RFC. Additionally, every update to the TCP specification requires significant maintenance effort to port everything correctly. This type of stack is therefore bound to be discontinued, given that it requires a sufficient number of maintainers to keep it up to date.

2.3.2.2 IX

A major issue with user-space stacks is that they do not benefit from the isolation provided by the kernel. A poorly developed application can easily crash the network stack. IX [2] addresses this issue by restructuring how the OS handles networking. It does so by splitting it into two distinct planes: the control plane and the data plane. The first is a standard Linux kernel responsible for resource provisioning and system configuration. The second is a lightweight, specialized operating system that runs a TCP/IP stack along with the application logic (see Figure 2.4a).

To achieve this separation, IX uses Dune [31], a system that leverages Intel's VT-x hardware virtualization. This enables IX data planes and applications to run at different levels of hardware protection (i.e., ring 0 for the former and ring 3 for the latter): applications therefore cannot corrupt the network stack.



(a) Protection and separation of control and data plane. (b) Interleaving of protocol processing and application execution

Figure 2.4: The IX dataplane operating system [2]

Like mTCP, and as noted in Figure 2.4b, IX also batches system calls. However, it takes it a step further by batching incoming and outgoing packets before passing them to the TCP/IP stack. This aggressive batching limits cache misses in the CPU's L1/L2 caches, consequently improving throughput.

This system delivers performance far beyond that of Linux. The authors demonstrated that IX could handle over three million RPCs per second, whereas Linux was limited to around 300,000. They also ported Memcached [32], a key-value store. The application's throughput increased 3.6-fold, and tail latency (99th percentile) decreased by half compared to running it on Linux. However, since IX is not POSIX-compliant, every application wishing to use its dataplanes must be ported. It is also entirely dependent on Dune, which requires a CPU that supports VT-x. This makes IX ineffective in nested virtualization environments, which many public clouds use.

2.3.2.3 TAS

As its name suggests, TAS (TCP Acceleration as a Service) [3] is a software-based TCP stack that runs as a user-level service. It splits TCP processing into two paths: a slow and a fast path. The first is reserved for complex, stateful, or infrequent events, such as connection setups and teardowns, timeouts, error handling, and congestion control policies. It is essentially a reimplement of the TCP state machine. The second path is used for established and healthy connections. Incoming packets are distributed across multiple CPU cores via RSS, and the various payloads are placed directly into the user-space memory of the target application. TAS implements the transmission side similarly.

To completely bypass the Linux kernel, TAS uses DPDK (Data Plane Development Kit) [18], a set of libraries initially developed by Intel that intercepts incoming packets on the NIC and redirects them to user space for further processing.

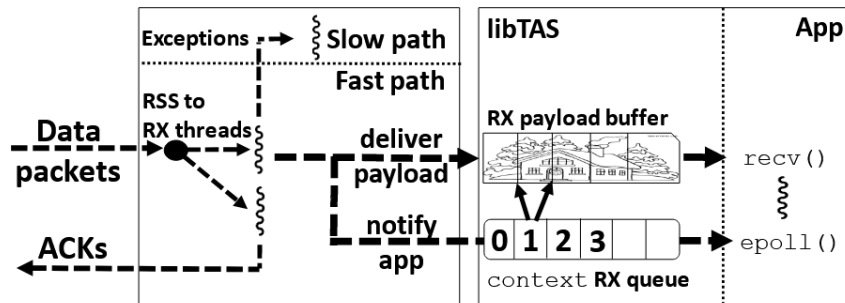


Figure 2.5: TAS receive flow [3]

The authors compared TAS's performance against IX. They observed 90% higher throughput, as well as 57% lower tail latency for RPCs. It also scales better, providing 2.2 times higher throughput than IX when handling 64,000 concurrent connections. One of TAS's main advantage over previous solutions is that it does not require application rewriting, since it exposes a fully POSIX-compliant API. Nevertheless, TAS is still a prototype, and has therefore never been deployed at scale, neither does it support certain TCP or IP options or features (e.g., there is no mechanism to defragment an IP packet).

2.3.2.4 StackMap

Of all the papers presented in this section, StackMap [4] is the only proposed acceleration technique that makes full use of the Linux stack. According to the paper's latency analysis, the native Linux stack is highly efficient for TCP/IP processing, taking less than 0.8 microseconds in each direction (ingress and egress). The real bottleneck comes from the socket API, packet I/O, buffer management, and context switching.

To address these issues, StackMap uses a hybrid architecture based on Netmap [19] (see Figure 2.6), a framework similar to DPDK. Applications still use the standard socket API for everything related to the control path (e.g., establishing connections), which allows the OS to efficiently manage these complex states. Once the connection is established, StackMap takes over. Raw packets arriving from the network are placed directly into a "NIC port": a ring buffer managed by Netmap that maps to the NIC's physical hardware rings. StackMap then passes them to the kernel's network stack, which verifies whether the packet is valid and in-order. If so, StackMap retrieves the packet and places it in a "stack port", another ring buffer shared with the application, which enables zero-copy reads of the payload without a context switch.

If the packet arrives out of order, StackMap places it in a buffer pool until the rest arrives correctly. This operation is implemented by changing an index in a data structure and is also zero-copy.

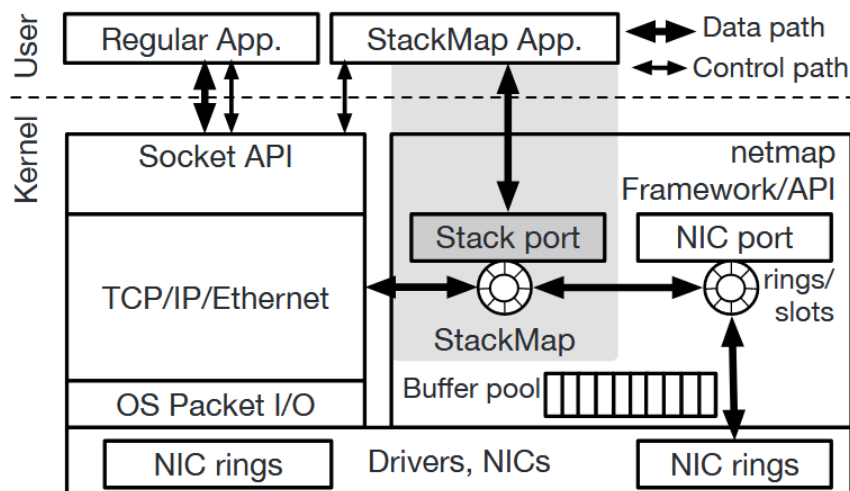


Figure 2.6: StackMap architecture overview [4]

Compared to the base Linux stack, StackMap improves in average latency by 4% to 80%. Throughput also improves, ranging from 4% to 391% depending on the payload size. Memcached has also been ported, and improvements in throughput ranging from 42% to 133% and reductions in latency ranging from 30% to 58% were observed. However, the system is less efficient in terms of scalability; other DPDK-based stacks, such as Seastar [33], outperform it starting at six CPU cores.

A clear advantage StackMap has over previous papers is that the kernel maintains a view of the various processed packets. Therefore, all kernel-based tools (e.g., `tcpdump`, `nftables`, `netstat`) remain fully functional, unlike with user-space stacks. However, applications require rewriting to use the API exposed by Netmap to read packets, and a NIC must be dedicated to a specific application (a limitation imposed by Netmap's ring buffers). Even more problematic is that, since the application and kernel share the same buffer to achieve zero-copy, a malicious or buggy application could modify data that the kernel is still holding (e.g., a packet waiting in the TCP retransmission queue), which could corrupt the TCP state machine's state.

2.3.3 Summary

As illustrated in this subsection, the current state of the art forces a trade-off between pure performance and system compatibility. User-space stacks, such as mTCP, IX, and TAS, deliver far superior throughput to Linux with reduced latency. However, they require significant maintenance, require applications to be rewritten or relinked, and completely break compatibility with kernel-based networking tools. On the other end, optimizations that attempt to preserve the entire Linux network stack, such as StackMap, offer a middle ground. While they keep kernel tooling functional and are easier to integrate into an existing system, they provide smaller performance gains, lower scalability, still require application-level API changes, and introduce security risks.

Achieving a safe fast path that accelerates traffic without losing security, requiring application rewrites, or abandoning native tooling necessitates a mechanism to execute custom logic directly within the

kernel's network datapath. The following section introduces the different technologies that make this modern architecture possible: eBPF and XDP.

2.4 The eBPF and XDP ecosystem

Introduced in 2014, the Extended Berkeley Packet Filter (eBPF) [20] is a technology that allows programs to run in a privileged environment, typically directly within the kernel. The general workings of the architecture are illustrated on Figure 2.7.

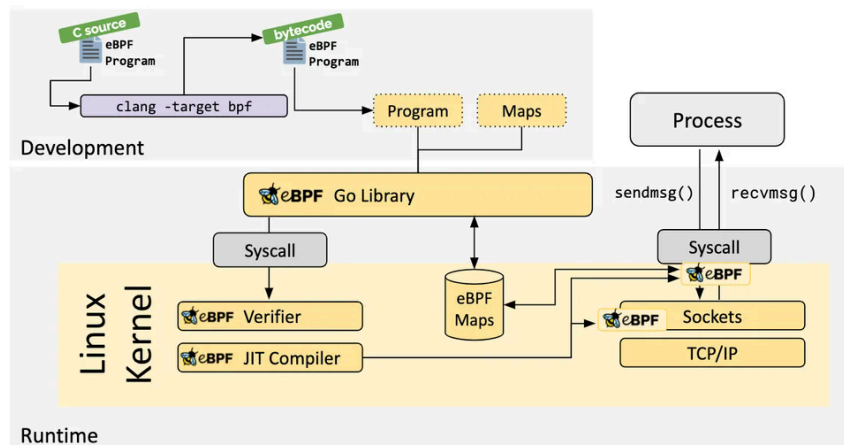


Figure 2.7: Overview of eBPF's inner workings [5]

An eBPF program is, at its core, simply a program written in a low-level language (e.g., C, Rust). It can be compiled into eBPF bytecode using LLVM/Clang, and can then be loaded into the kernel via the `bpf()` system call. After passing verification (see Section 2.4.2), it is loaded into the eBPF virtual machine, which JIT-compiles it into native machine instructions, thereby achieving near-native speeds. The execution of an eBPF program is always triggered by a hook: a point in the kernel code that calls any functions attached to it. These programs then have (restricted) access to an execution context that varies depending on the hook, allowing them to observe or modify data.

Several major projects use eBPF and related technologies; the best known are likely Meta's Katran [34] (a load balancer used to route virtually all packets entering their data centers) and Cilium [35], which provides observability, security, and networking tools for Kubernetes clusters.

2.4.1 eBPF maps

By design, an eBPF program is inherently stateless. When triggered by a hook, it executes its logic, returns a value, and terminates. Upon completion, all local variables or state created during execution are destroyed. For complex applications that need to save information (e.g., a load balancer, a firewall), this limitation quickly becomes significant. To address this issue, the eBPF architecture provides eBPF maps. Maps are generic key-value data structures residing in kernel memory that allow an eBPF program to persist state. They are created and managed by the `bpf()` syscall and can be accessed

via specific eBPF helpers. The Linux kernel provides access to many types of maps, including hash tables (`BPF_MAP_TYPE_HASH`), arrays (`BPF_MAP_TYPE_ARRAY`), ring buffers (`BPF_MAP_TYPE_RINGBUF`), and much more [36].

eBPF maps serve two primary purposes. First, they enable state sharing across the kernel; multiple instances of the same eBPF program, or entirely different programs attached to various hooks, can concurrently read and write to a shared map. Second, they are one of the primary means of communication with user-space. Unlike eBPF programs, maps are not purely restricted to the kernel. By holding file descriptors that reference these maps, user-space applications can asynchronously exchange data with the eBPF programs.

2.4.2 The eBPF verifier

At first glance, eBPF programs and LKMs may not seem very different: both allow kernel functionality to be extended and can be loaded “on demand”. The main distinction lies in the security they provide. A poorly developed LKM can easily crash the kernel or corrupt its data, as was demonstrated during the CrowdStrike incident in 2024 [37], [38]. Although this outage occurred on Microsoft Windows systems, the underlying principle remains the same. eBPF programs, on the other hand, are systematically validated by the eBPF verifier every time they are loaded to ensure that they will not have any harmful effects on the kernel.

To determine whether a program is valid, the verifier analyzes it to check all possible execution paths. To achieve this, it maintains a state for each register of the eBPF virtual machine in a `bpf_reg_state` structure. This structure stores information about the register’s contents, such as the type of value (e.g., `NOT_INIT` for uninitialized registers, `SCALAR_VALUE` for non-pointer values, and various `PTR_TO_*` for different types of pointers), along with the upper and lower bounds of the data it holds. It then uses this information to determine whether invalid or dangerous actions could be executed.

Each time the verifier encounters a branch, it pushes the current state of all registers onto a stack and continues the analysis along one of the paths. If it reaches the end of the path without detecting any issues, it pops the stack and continues with the next. This process repeats until all states have been explored. Since naively going through all paths would be computationally expensive, the verifier incorporates a state pruning system. Periodically (at the time of writing, every ten instructions), it saves the state of the registers to a cache. If the verifier later re-evaluates the same instruction with the registers in the same state, it can safely stop the evaluation, since the resulting path has already been validated.

To assist the verifier in its analysis, each eBPF helper and `kfunc` annotates the type of argument it expects to receive. The annotation for the `bpf_map_lookup_elem` helper, which performs a lookup of a value in a map, is shown in Listing 2.1. By examining the `.ret_type`, `.arg1_type`, and `.arg2_type` fields, the verifier deduces that the return value is either an element of the map or null and that the arguments are, respectively, a constant pointer to a map and a pointer to a map key. This mechanism enables it to express constraints more complex than those natively supported by the C language.

Listing 2.1: Annotation of the `bpf_map_lookup_elem` eBPF helper in the Linux kernel v6.19

```
1 const struct bpf_func_proto bpf_map_lookup_elem_proto = {
2     .func      = bpf_map_lookup_elem,
3     .gpl_only  = false,
4     .pkt_access = true,
5     .ret_type  = RET_PTR_TO_MAP_VALUE_OR_NULL,
6     .arg1_type = ARG_CONST_MAP_PTR,
7     .arg2_type = ARG_PTR_TO_MAP_KEY,
8 };
```

In addition to checking the types of the arguments, it also ensures the following:

- The helpers being called can be invoked from the hook used by the eBPF program.
- The program's license is compatible with the helpers it uses.
- Memory accesses are valid.
- No potentially null pointers are dereferenced.
- The eBPF program does not read or modify parts of the context to which it does not have access.
- The program returns a value at the end of its execution.
- No invalid or unreachable instructions are present.
- The program executes in a finite amount of time.

While most of these elements are easy to verify, some, such as valid memory accesses, the absence of null dereferences, and running to completion, are more difficult to confirm.

2.4.2.1 Validating memory accesses

Invalid memory accesses are the cause of many bugs, and eBPF is unique in that it has the potential to access kernel memory [39], [40]. Therefore, an invalid read or write can lead to serious instability, corruption, or data leaks. The verifier ensures that each access is valid (i.e., it occurs within a memory region to which the eBPF program actually has access). In the case of XDP (a specialized network hook for eBPF programs, see Section 2.4.3), the program's context defines the limits of the packet it can inspect and modify. Every access to the packet must occur within these limits. Otherwise, the program fails validation.

2.4.2.2 Validating pointer dereferences

As seen earlier in Listing 2.1, certain types may have the suffix `_OR_NULL`, which indicates that the value could be null. When the verifier encounters this status for a register, it ensures that all dereferences of that register's value are covered by a null check (i.e., `if (variable != null) {}`). If not, validation fails.

2.4.2.3 Validating run to completion

Determining whether a program will eventually terminate is a well-known problem in computational theory. In 1936, Alan Turing concluded that it is undecidable [41], meaning that no general algorithm

can be designed to solve it. Rather than attempting to determine through various heuristics whether an eBPF program will terminate, the verifier enforces strict constraints to guarantee finite execution. First, it processes up to one million instructions during its evaluation. If this limit is reached, validation fails and the program is rejected. Second, to prevent infinite execution, the verifier historically prohibited loops entirely. Modern versions of the Linux kernel permit loops, but only if they are strictly bounded, meaning the verifier must be able to mathematically prove at load time that the loop will terminate after a known maximum number of iterations.

It should be easy to see why eBPF programs are not suitable candidates for a full TCP stack implementation, as mTCP might have done in user space. The limitations of the verifier (particularly the hard limit of one million instructions per program) make complex implementations challenging. For instance, parsing variable-length TCP options or managing unpredictable connection states becomes significantly harder when unbounded iterations are strictly forbidden.

2.4.3 XDP

The eXpress Data Path (XDP) [42] is a high-performance network framework based on eBPF. First included in the 4.8 release of the Linux kernel in 2016 [43], it adds a hook directly into the NIC driver (provided the driver supports it), enabling an eBPF program to process a packet before it enters the kernel's network stack. Once processing is complete, the program returns a value called the verdict, which indicates the packet's fate:

- XDP_PASS: the packet must be sent to the standard network stack.
- XDP_DROP: the packet must be discarded immediately.
- XDP_TX: the packet must be transmitted over the same interface it arrived on.
- XDP_REDIRECT: the packet must be transmitted over a different network interface.
- XDP_ABORTED: the packet must be discarded due to an unexpected error.

XDP defines three operating modes that dictate when the hook is executed in the processing. The most common mode is the “native” mode. This mode allows the hook to execute in the soft IRQ context, within the `poll()` function (see Section 2.2.1). Its main advantage is that at the moment of execution, the kernel has not yet allocated any structures or resources to handle the packet (specifically, the `sk_buff` has not yet been initialized). However, since the `poll()` function is defined in the NIC driver, the network card manufacturer must include the hook call in their code; otherwise, the NIC will not be compatible. A variant of this mode is the “offloaded” mode. As the name suggests, it runs the eBPF program directly on the network card's NPU. Hardware-offloaded XDP is limited to a small subset of NICs, as the manufacturer must design the silicon specifically to support programmable packet processing at the hardware level. Nevertheless, it has the advantage of fully delegating XDP processing, thereby freeing the CPU to work on other tasks. The final mode is “generic” mode. It enables any network card (including those that do not support XDP) to run a program in a similar execution context. To achieve this, the kernel processes the packet using the standard stack, attaches an `sk_buff` to it, and passes pointers to the packet from

the `sk_buff` to the eBPF program. By design, this mode does not deliver any performance gains, since the entire network stack is traversed before the hook is executed. It is therefore primarily used for development and testing.

Since XDP is based on eBPF, it inherits all of its security features. Programs using this network hook are therefore guaranteed to be safe by the verifier. As discussed in Section 2.4.2.1, when an eBPF program manipulates a packet, it must explicitly verify that accesses are made to valid locations. To this end, any program attached to the XDP hook receives as an argument a `struct xdp_md` context containing pointers to the start and end of the packet. Listing 2.2 illustrates this point by presenting a program that drops all IPv4 packets.

Listing 2.2: An XDP program that drops IPv4 packets

```
1 SEC("xdp")
2 int xdp_drop_ipv4(struct xdp_md *ctx) {
3     void *data = (void *) (long) ctx->data;
4     void *data_end = (void *) (long) ctx->data_end;
5
6     struct ethhdr *eth = data;
7
8     if ((void *) (eth + 1) > data_end)
9         return XDP_PASS;
10
11    if (eth->h_proto == bpf_htons(ETH_P_IP))
12        return XDP_DROP;
13
14    return XDP_PASS;
15 }
```

The first two lines of the function extract the packet boundaries, which the verifier tracks throughout the program. If a single instruction modifies these boundaries, validation fails. The packet is then cast into an Ethernet header. Before any memory access can be performed (in this case, reading the type of payload carried in the Ethernet frame), the verifier requires the program to “prove” that the access will be valid. Line 8 verifies that the packet size announced by the context is large enough to contain a complete Ethernet frame. If so, reading a field within this structure will certainly occur between the `data` and `data_end` boundaries, which satisfies the verifier.

This security feature, coupled with the ability to drop up to 24 million packets per second on a single CPU core (performing approximately 20 times faster than traditional `iptables`) [42] makes XDP an ideal choice for high-performance network inspection and DDoS attack mitigation [44].

2.4.4 Traffic control hook

Traffic Control (TC) is one of the network hooks provided by eBPF. Similar to XDP, it allows packets to be inspected and modified while using special return values to dictate their fate. The core difference is that the TC hook is triggered immediately after the kernel allocates the `sk_buff` structure, but before standard

network stack processing starts. Since the eBPF program receives and manipulates the `sk_buff` directly, it can interact with extensive kernel metadata, providing much greater flexibility than XDP's raw-data limitations.

A notable distinction is that TC supports an egress hook, allowing packets to be intercepted and modified just before they are transmitted via the network interface. This is a capability that XDP's design currently does not support. As for return values/verdicts, TC currently defines the following:

- `TC_ACT_OK`: similar to `XDP_PASS`, instructs the stack to continue processing the packet normally.
- `TC_ACT_SHOT`: similar to `XDP_DROP`, terminates processing and immediately frees the packet.
- `TC_ACT_REDIRECT`: similar to `XDP_REDIRECT`, allows the packet to be sent to a different network interface or redirected to the ingress/egress path of another device.
- `TC_ACT_RECLASSIFY`: forces the packet to be re-processed by the TC subsystem from the beginning. This is particularly useful if the eBPF program has modified the packet headers in a way that would change its classification.
- `TC_ACT_PIPE`: passes the packet to the next action in the TC action chain, if one exists.

The subsystem also defines several auxiliary codes that are yet to be implemented for eBPF programs:

- `TC_ACT_STOLEN`: indicates that the packet has been "taken over" by another subsystem. The TC hook stops processing, but unlike `TC_ACT_SHOT`, the memory is not freed as ownership has been transferred.
- `TC_ACT_QUEUED`: indicates the packet has been placed into a queue.
- `TC_ACT_REPEAT`: instructs the system to repeat the current classification filter.

3

Implementation

Contents

3.1 Introduction	20
3.2 Architectural overview	20
3.3 Kernel module and functions	21
3.4 Stateless eBPF bypass	27
3.5 Stateful flow offloading	28

3.1 Introduction

This chapter details the design and implementation of the proposed eBPF-based TCP fast paths. It begins with an architectural overview of the hybrid system, explaining the interaction between the kernel and the eBPF context. It then explores the internal mechanics of the Loadable Kernel Module (LKM), specifically the implementation of kernel functions (kfuncs) that handle memory management, socket synchronization, and packet injection. Finally, we demonstrate the versatility of this architecture by presenting two distinct fast path implementations: a stateless bypass designed for maximum CPU efficiency and a stateful flow-offloading program that integrates with the Linux Netfilter subsystem.

3.2 Architectural overview

The system consists of two distinct parts: a LKM that exposes a kfunc per hook, and eBPF programs that interface with these kfuncs. The latter are entirely user-supplied and can therefore be programmed to meet specific needs. In this thesis, we present two distinct use cases: an aggressive bypass that sends all packets that can be accelerated to the fast path (see Section 3.4) and a software implementation of flow offloading (see Section 3.5)

When a packet arrives at the NIC, the eBPF program performs checks (notably on the nature of the packet, whether it is fragmented, etc.) and decides whether or not to send it to the fast path by calling a kfunc. The LKM then takes over, allocating the necessary minimal structures and feeding the packet into the TCP state machine. If the state machine indicates that the packet has been processed and consumed correctly, the eBPF program drops the pending packet to prevent processing it twice. The general structure of the implementation is shown in Figure 3.1.

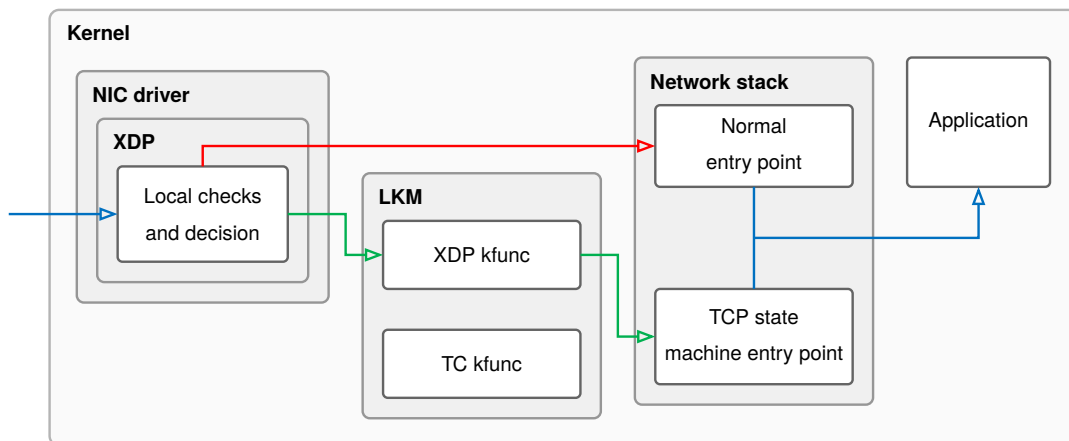


Figure 3.1: Structure of the implementation. Red arrow indicates normal packets, green arrow indicates accelerated ones.

3.3 Kernel module and functions

Due to its limited nature, eBPF cannot be used to interface directly with the Linux networking stack. Notably, the verifier would be unable to guarantee that no kernel data corruption occurs. For more complex tasks requiring extensive interaction with internal structures, eBPF relies on the use of helpers or `kfuncs`. These are specially annotated functions, as shown in the Listing 2.1, which are not subject to the same restrictions as standard eBPF programs since they run in the kernel and thus in a privileged, non-sandboxed context.

However, using `kfuncs` involves significant trade-offs. First, because they are not sandboxed, the verifier cannot guarantee that a poorly written `kfunc` will not crash the system. In practice, the verifier only ensures that the arguments passed to the function are compatible with its annotations; it does not inspect the actual executed code. This makes it possible to introduce critical bugs, such as null pointer dereferences, which are normally impossible within a standard eBPF context. Furthermore, `kfuncs` are inherently less portable as they are typically tied to specific kernel versions. This issue is partially mitigated by BPF Type Format (BTF), which describes the various fields used by the function and allows the kernel to adapt memory offsets dynamically (a mechanism known as CO-RE, Compile Once - Run Everywhere). Finally, invoking a `kfunc` from an eBPF program incurs a performance overhead, a topic that will be explored in greater detail in Section 4.5.

The system described in this thesis exposes two `kfuncs` (one for XDP and one for TC), which are registered in a LKM. Any system administrator can compile this module and load it in their own kernel without having to reboot the host. It can also be disabled at any time, making it non-invasive, unlike some existing kernel bypasses.

3.3.1 Finding an entry point in the TCP stack

Before examining how `kfuncs` are implemented, it is worth first examining the entry point into the TCP stack. The selection of this entry point involves a significant design trade-off: it must be situated deep enough within the processing pipeline to maximize the performance benefits of a bypass, yet remain sufficiently shallow to preserve protocol correctness. For instance, while it is technically feasible to deliver a packet directly to the application's receive queue, doing so would bypass critical mechanisms such as acknowledgment generation and retransmission logic. Using "Packet, Where Are You?" [45], a tool made by Cilium that tracks packets through the network stack, we can identify the functions traversed by a typical packet. This sequence is detailed in Figure 3.2.

The entry point into the TCP state machine for established flows is the `tcp_rcv_established` function. Provided the packet arrives at this point or earlier, the correctness of the protocol (excluding optional TCP extensions) is guaranteed. One notable feature of the call flow is the extent to which the stack relies on the `sk_buff` structure. All the functions in the list take it as an argument, as does Layer 3 (IP, not shown here). As we do not wish to modify the TCP implementation in the kernel, our system must construct a minimal `sk_buff`. This will be detailed in the following section.

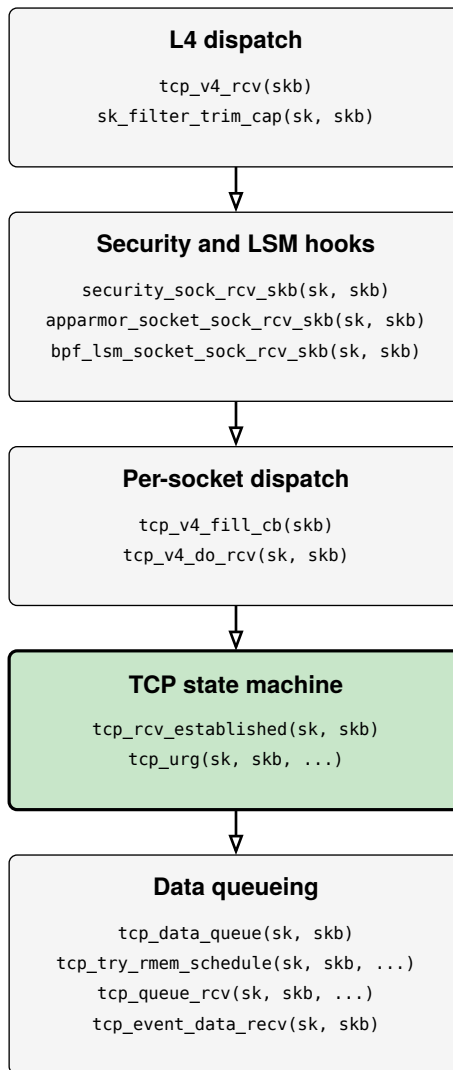


Figure 3.2: Path of an established-flow TCP packet through the kernel, grouped by responsibility. The TCP state machine is highlighted in green.

3.3.2 XDP kernel function

Of the two `kfuncs`, the one serving as the entry point for XDP-intercepted packets is significantly more complex. Its signature is shown in Listing 3.1.

Listing 3.1: Kernel function signature for the XDP fast path

```
1 __bpf_kfunc int xdp_fastpath_rcv(struct xdp_buff *buf, struct sock* sk, __u16 ip_hdr_len)
```

One significant constraint is the lack of an `sk_buff` in the signature. This is to be expected, as XDP's performance advantage stems from its execution before the allocation of heavy data structure, as explained in Section 2.4.3. Therefore, the `kfunc` will need to allocate and initialize a socket buffer before the packet can be processed by the TCP entry point.

3.3.2.1 Verifying socket state

Since the bypass relies on `tcp_rcv_established`, the `socket` must be in the `ESTABLISHED` state before attempting an injection. Fortunately, efficiently retrieving a pointer to the `socket` from the XDP context is relatively easy, which is why it is passed as an argument to the `kfunc`. The implementation must subsequently verify that no errors are present in the `sk_err` field and that the `sk_state` is indeed `TCP_ESTABLISHED`.

3.3.2.2 Allocating the socket buffer

As previously noted, the XDP program does not pass a `socket buffer` structure as an argument when calling the `kfunc`. Instead, it provides an `xdp_buff`, which serves as the kernel's internal representation of the `xdp_md` context provided to eBPF programs. Like its counterpart, this structure provides access to the packet's memory buffers while offering additional metadata. This will be further discussed in Section 3.4.

The `kfunc` is therefore responsible for allocating the `socket buffer` itself and mapping it onto the XDP memory page, as illustrated in Figure 3.3. First, the function allocates an empty `sk_buff` by calling `napi_build_skb`. This variant is an optimized version of the standard `build_skb` function and is restricted to execution within a `soft IRQ` context. It uses a special per-CPU cache named `napi_alloc_cache` containing several pre-allocated `sk_buff` structures. When called, the function retrieves a free `socket buffer` from this pool and returns it to the caller. As soon as the structure is no longer needed, it is “recycled” by the system and placed back into the cache. This design enables efficient, high-performance allocation by eliminating the need for locks, as each core is responsible for managing its own local memory pool. Additionally, `napi_build_skb` configures the newly retrieved `sk_buff` to point to `buf->data_hard_start`, which represents the start of the memory page containing the packet, and defines its size (via `buf->frame_sz`). This corresponds to the first step of Figure 3.3, where the head and end pointers are set around the XDP page while `data` and `tail` remain at the page start.

A critical challenge with this implementation is ownership management. As the newly allocated `socket buffer` references a page originally allocated by XDP, two problematic scenarios may arise depending on the relative processing speeds of the TCP stack and the XDP program. In the first and most likely case, the XDP program may attempt to free the packet's memory immediately after it is injected in the TCP stack. If the packet is still being processed by the network stack (e.g., during reordering), the `sk_buff` would then point to a freed area, resulting in a use-after-free vulnerability. Conversely, if the TCP stack releases the packet first, XDP may remain unaware and attempt a second deallocation, causing a double-free error. Therefore, the system must explicitly indicate that the page is shared by both subsystems. This is achieved via the `ref_xdp_page` function.

The Linux kernel uses a well-known memory management mechanism known as reference counting, where each object maintains a counter indicating how many entities are currently utilizing it. A subsystem increments this counter upon accessing the object and decrements it once its task is complete. The object is only deallocated once the reference count reaches zero. Since pages allocated by XDP follow this exact principle, ownership can be safely shared with the network stack simply by incrementing

the reference count. Once XDP completes its processing, it decrements the counter (leaving it at 1). Subsequently, when the TCP stack finishes, the count drops to zero and the page is safely freed.

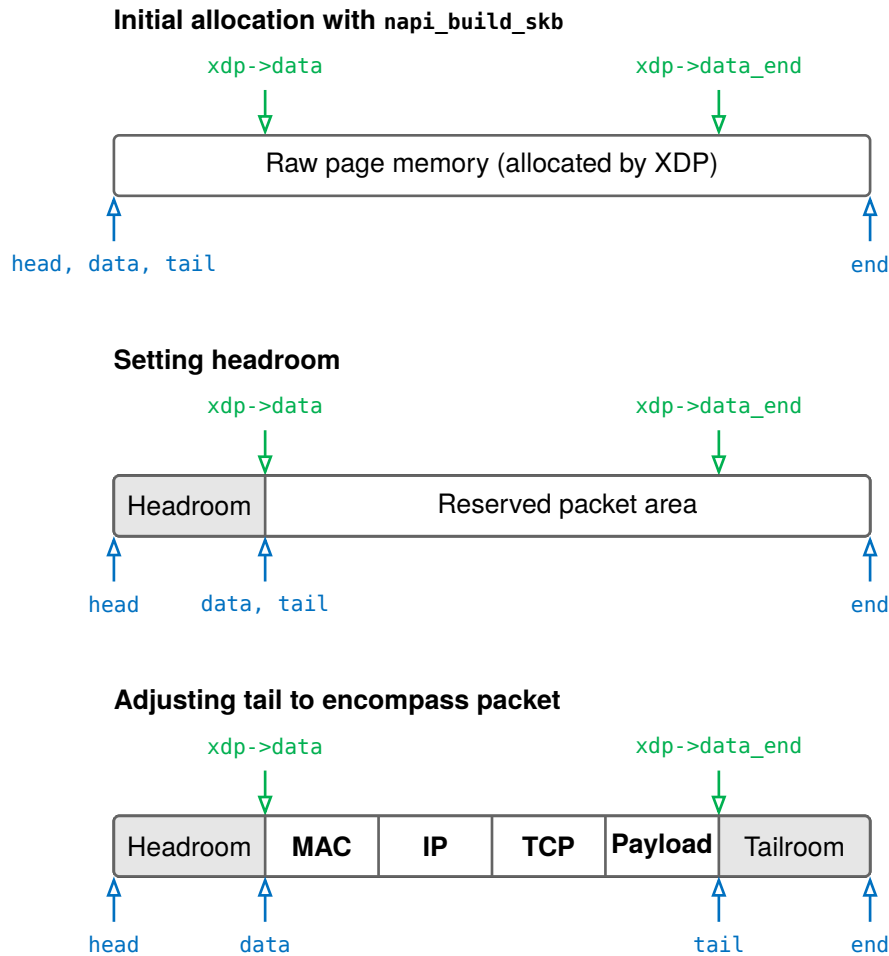


Figure 3.3: Step-by-step `sk_buff` pointer adjustments to encapsulate raw XDP page memory.

Memory allocated by XDP originates from two distinct sources (indicated by the `buf->rxq->mem.type` field): the page pool (`MEM_TYPE_PAGE_POOL`) or the standard page allocator (`MEM_TYPE_PAGE_SHARED`). In the former case, the page is retrieved from a cache similar to the one used by the `napi_build_skb` function. In the latter, the page is allocated normally through a standard call to `alloc_page`. Distinguishing these two different types is necessary, as the allocation source dictates how the page must be freed. Specifically, pages from the page pool must be returned to the pool rather than being released normally. To ensure that the `sk_buff` handle this correctly, the `pp_recycle` flag indicates whether the payload should be recycled (i.e., returned to a page pool) or processed by the standard allocator.

The remaining two steps, illustrated in the second and third panels of Figure 3.3, involve adjusting the `sk_buff` pointers to the correct locations. While `napi_build_skb` initializes the `head` and `end` pointers based on the provided values (namely the beginning and size of the page), the `data` and `tail` pointers must be manually aligned to the beginning and end of the packet, respectively. This is achieved using the `skb_reserve` and `skb_put` functions. The former advances both `data` and `tail` simultaneously by a given offset (in this case, the distance between the page start and the actual packet start, see the second step

of Figure 3.3). `skb_put`, on the other hand, is used to extend the `tail` pointer to the end of the packet (third step of Figure 3.3). At this point, an empty `sk_buff` has been successfully initialized and linked to the XDP page.

3.3.2.3 Populating the socket buffer

Once the socket buffer is allocated, it must be populated with the metadata required by the TCP state machine. Interestingly, the `skb->cb` field is the only strictly mandatory component for the stack's operation. Most other fields are optional.

The `skb->cb` field (also known as the control block) is a 48-byte memory area that serves as a scratchpad for the various layers of the network stack. Each layer can store temporary data within this space to facilitate or accelerate processing. In the case of TCP, this area is used to store sequence numbers, acknowledgment numbers, and protocol flags, alongside specific IP-layer information. Given its role, it is essential that the fast path initializes this area. Failure to do so would cause the stack to read uninitialized or incorrect data, inevitably corrupting the TCP state machine. To handle this, the kernel provides the `tcp_v4_fill_cb` function. However, as this function is not exported by default, the kernel headers must be modified to make it visible to the module. This represents the only modification required within the standard network stack. An alternative approach would be to replicate the function's logic directly within the LKM, which would render the module entirely self-contained.

A second field critical for performance is `skb->ip_summed`. This field indicates whether the TCP stack needs to perform checksum validation or if the packet has already been verified by a lower layer. By setting this field to `CHECKSUM_UNNECESSARY`, we instruct the stack to bypass the recalculation. It is important to note that this optimization assumes the checksum has been verified at the NIC level via hardware checksum offloading. Without hardware verification, a corrupted packet could potentially reach the application layer undetected.

3.3.3 TC kernel function

Before detailing the injection into the TCP stack, which is identical for both implementations, we will examine the `kfunc` associated with the TC hook. This `kfunc` is significantly simpler than its XDP counterpart because the TC hook provides direct access to the `sk_buff` structure, allowing it to be passed directly as an argument.

After ensuring the header data is contiguous in memory via `pskb_may_pull`, the socket buffer metadata is cloned, and all subsequent operations are performed on this new copy. While cloning may appear counterintuitive from a performance perspective, it aligns with kernel best practices [46], as the TC hook requires the `sk_buff` to remain in its pre-call state upon returning from the `kfunc`. Specifically, the hook stores internal processing metadata within the control block (`skb->cb`). If the TCP stack were to overwrite this area during processing, the TC hook would encounter corrupted state data upon resuming execution, resulting in a crash.

3.3.4 Injection into the TCP state machine

As established in Section 3.3.1, `tcp_rcv_established` serves as the entry point for packet injection. The implementation of this injection logic is detailed in Listing 3.2.

Because the bypass skips `tcp_v4_rcv`, which is the standard function responsible for managing `socket` buffer pointers, these pointers must be manually adjusted. Unlike `skb_put` (see Section 3.3.2.2), `skb_pull` is used to move the data pointer forward to decapsulate the Ethernet and IP headers. Since the length of the latter can vary based on options, its value is calculated within the eBPF context and passed to the `kfunc` as an argument. Once these adjustments are complete, the `sk_buff` is correctly aligned (as shown in Figure 3.4), allowing the previously discussed `tcp_v4_fill_cb` to correctly populate the control block.

Before proceeding with the injection, the code must acquire the `socket`'s spinlock. This synchronization is critical because the `socket` structure is a shared resource accessible by hardware interrupts, kernel timers, and user-space applications. Locking thus ensures atomic access and prevents race conditions. Once secured, `tcp_segs_in` is called to update internal kernel counters. This ensures that the injected traffic remains visible to system monitoring tools, such as `netstat`, which rely on these metrics to track processed TCP segments.

Listing 3.2: Implementation of the `tcp_inject_packet` function

```
1 // Parse L3/L4 and fill control buffer
2 // [...]
3
4 bh_lock_sock_nested(sk);
5 tcp_segs_in(tcp_sk(sk), skb);
6
7 if (likely(!sock_owned_by_user(sk))) {
8     tcp_rcv_established(sk, skb);
9 } else {
10     if (tcp_add_backlog(sk, skb, &drop_reason)) {
11         kfree_skb(skb);
12         ret = -EIO;
13     }
14 }
15
16 bh_unlock_sock(sk);
```

A key consideration when invoking `tcp_rcv_established` is the current state of the `socket` relative to the user-space application. If the application is not currently performing a read operation, the packet can be safely passed to the TCP stack for immediate processing and delivery to the receive queue. However, if the application is actively accessing the `socket`, the stack cannot modify the receive queue without risking data corruption. In this case, the packet is instead diverted to the backlog queue (a FIFO buffer for packets awaiting deferred processing) through `tcp_add_backlog`. Should this step fail, the `sk_buff` is freed and an error is returned to the eBPF program, signalling it to transmit the packet to the standard network

stack. Once the application releases the `socket`, the TCP stack drains this backlog and processes the packets sequentially.

3.3.4.1 A note on socket ownership semantics

It might seem counterintuitive that a user-space application can maintain access to a `socket` even after the `kfunc` has acquired its spinlock. This behavior is the result of a deliberate design choice in the Linux networking stack. When a process executes a `read()` or `recv()` system call, it briefly acquires the spinlock to set the `socket`'s `owned` flag to 1, then releases the lock. This design prevents a long-running system call from holding a spinlock indefinitely, which would degrade system performance. Because the receive queue is not protected by the spinlock while the `socket` is “owned” by a process, checking the `owned` field is mandatory before any direct injection. The backlog queue, on the other hand, remains strictly protected by the spinlock. Consequently, the `kfunc` can safely append packets to it regardless of the application's current state.

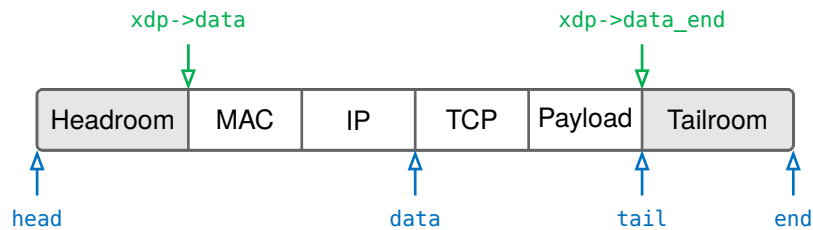


Figure 3.4: A standard TCP/IP packet and its associated `sk_buff` pointers before injection.

3.4 Stateless eBPF bypass

With the LKM now introduced, we can examine how its `kfuncs` can be used by standard XDP programs. The eBPF program shown in Listing 3.3 bypasses all data packets regardless of their source. It is entirely stateless, meaning that the processing of each packet is independent. After parsing the headers (as introduced in Listing 2.2), the program selects which packets to bypass. To maintain the underlying connection state, all control packets (e.g., SYN, ACK, and RST) are strictly forwarded to the standard kernel stack for full processing. This ensures protocol correctness, following the same principle implemented by TAS [3]. If the injection succeeds, XDP drops the packet as it has already been consumed by the TCP stack. Otherwise, it is passed to the stack to avoid packet loss.

The types of the arguments passed to the `kfunc` may initially appear to mismatch those defined in its signature. For example, `ctx` is provided as a `struct xdp_md`, whereas the `kfunc` expects a `struct xdp_buff`. Similarly, `sk` transitions from a `struct bpf_sk` to a `struct sk`. This is a deliberate mechanism of the eBPF verifier. As noted in Section 2.4.2, the verifier ensures that the eBPF program does not access restricted context fields. While the actual context passed to an XDP program is a `struct xdp_buff`, it is cast to the more restricted `struct xdp_md` before the program executes. If the program attempts to access a field that exists in `xdp_buff` but not in `xdp_md`, the verifier will reject the program. However, when these

Listing 3.3: Implementation of the stateless XDP program

```
1  int xdp_tcp_fastpath(struct xdp_md *ctx) {
2      // Header parsing
3      // [...]
4
5      __u16 ip_hlen = iph->ihl * 4;
6
7      if (th->syn || th->fin || th->rst)
8          return XDP_PASS;
9
10     struct bpf_sock_tuple tuple = {
11         .ipv4.saddr = iph->saddr, .ipv4.daddr = iph->daddr,
12         .ipv4.sport = th->source, .ipv4.dport = th->dest
13     };
14
15     struct bpf_sock *sk = bpf_sk_lookup_tcp(ctx, &tuple, sizeof(tuple.ipv4),
16         BPF_F_CURRENT_NETNS, 0);
17
18     if (!sk)
19         return XDP_PASS;
20
21     int ret = xdp_fastpath_rcv((void *) ctx, (void *) sk, ip_hlen);
22     bpf_sk_release(sk);
23
24     return ret == 0 ? XDP_DROP : XDP_PASS;
25 }
```

arguments are passed to a `kfunc`, the function automatically gains access to the complete underlying structure. The relationship between `bpf_sk` and `sk` follows a similar pattern.

3.5 Stateful flow offloading

To demonstrate the versatility of XDP, we can consider an example of a stateful flow offloading implementation, similar to the functionality provided by `nftables` [47].

Flow offloading is a common optimization technique that evaluates firewall rules only on the initial packet of a flow. If that packet passes the checks, the evaluation is bypassed for all subsequent packets in the same flow. Since these rules are defined from a flow perspective (e.g., blocking traffic on port 80 or rate-limiting a specific IP) rather than for individual packets, this optimization achieves the same result as if each packet had been evaluated independently. The architecture and logic for this bypass are illustrated in Figure 3.5 and Algorithm 3.1, respectively.

Most of the necessary components have already been introduced in the previous sections. The remaining requirement is the link between Netfilter and eBPF, which is facilitated by various kernel helpers. Among these, `bpf_xdp_ct_lookup` allows the program to query the current state of a flow. The cache is implemented using eBPF maps, as discussed in Section 2.4.1. While the kernel offers several

suitable types, such as `BPF_MAP_TYPE_HASH` and `BPF_MAP_TYPE_LRU_HASH`, this implementation utilizes the former. This choice provides lower overhead, as the cache is naturally flushed upon the arrival of control packets (as seen in line 3 of Algorithm 3.1).

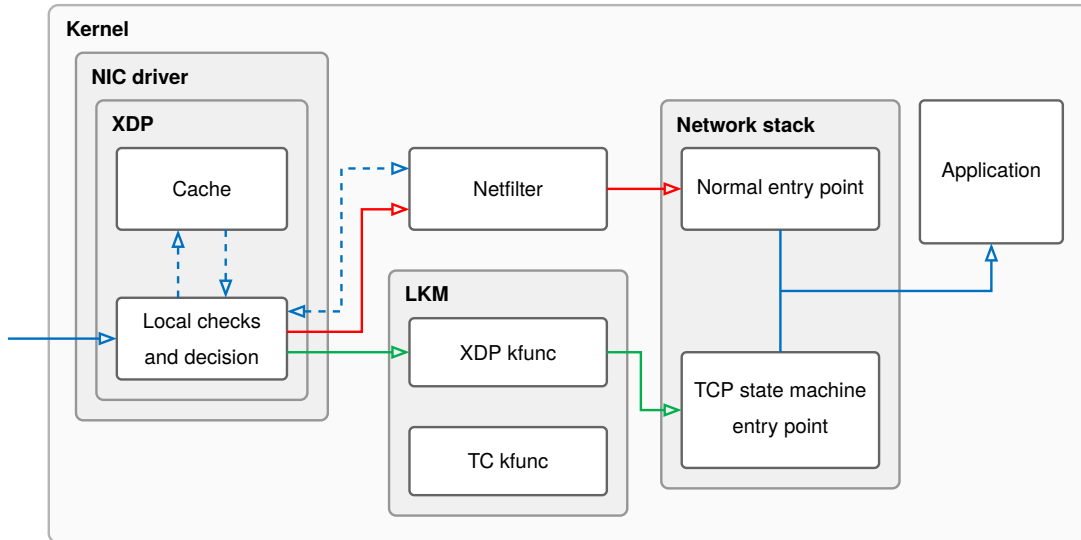


Figure 3.5: Structure of the flow offloading implementation. Red arrow indicates normal packets, green arrow indicates accelerated ones, dotted arrows indicate cache flow.

The cache itself is indexed by the connection four-tuple, which uniquely identifies a TCP flow. The associated value is intentionally minimal and consists of a boolean indicating whether the flow has been validated by Netfilter. This keeps each cache entry small, thereby maximizing the number of concurrent flows that can be stored within a given size. On a cache hit, the program avoids incurring two distinct costs. The first is the call to `bpf_xdp_ct_lookup`, which, while efficient, requires the data stored by conntrack to be traversed. The second cost avoided is the re-traversal and re-evaluation of the firewall rule chain.

Algorithm 3.1: Basic flow offloading bypass

```

1 Parse headers
2 if packet is a control packet then
3   | Delete corresponding flow from cache
4   | return XDP_PASS
5 end
6 if flow not present in cache then
7   | Lookup flow state from Netfilter
8   | if lookup fails or flow status is not confirmed then
9     | | return XDP_PASS
10  | end
11  | Add corresponding flow to cache
12 end
13 Bypass the packet

```

It is important to note that this implementation is deliberately simplistic and does not account for the full range of Netfilter features. However, this is not a fundamental limitation of the architecture, and the implementation could be extended to support a wider variety of configurations.

4

Performance evaluation

Contents

4.1 Introduction	32
4.2 Experimental setup and methodology	32
4.3 Throughput analysis	32
4.4 Latency analysis	34
4.5 CPU utilization and efficiency	35
4.6 Scalability and connection dynamics	39
4.7 Summary	42

4.1 Introduction

This chapter evaluates the performance of the proposed eBPF-based TCP fast paths against the baseline Linux network stack. It begins by describing the experimental setup, including the hardware and software configuration, and the workload generators. It then presents the results along four axes: throughput, latency, CPU efficiency, and scalability. Each section compares the four bypass variants introduced in Section 3 against the baseline across a range of configurations, varying the availability of hardware offloading, the presence of Netfilter rules, and the number of concurrent flows. The chapter concludes with a summary that highlights the configurations in which the bypasses are most and least effective.

4.2 Experimental setup and methodology

The various performance tests presented in this chapter were conducted on a server running Ubuntu Server 22.04 equipped with an Intel Xeon E-2378 processor (8 cores, 16 threads, 2.60 GHz) and 32 GB of 3200 MHz DDR4 RAM. Networking hardware consisted of an Nvidia BlueField-2 100GbE NIC supporting native XDP. To ensure that packets traverse the complete network stack, a 40 Gb/s cable was set up in a loopback configuration across two ports on the NIC, and the two corresponding interfaces were isolated in distinct network namespaces. Furthermore, all experimental scenarios were evaluated using Netperf [48] and orchestrated through Network Performance Framework (NPF) scripts, making all results easily reproducible.

Unless otherwise stated, all tests were performed with the server interface's IRQs (traffic receiver) pinned to core 0, the server (`netserver`) pinned to core 1, the client interface's IRQs (traffic generator) pinned to core 2, and the client (`netperf`) pinned to core 3. The sent message size was set to 64 kilobytes and all tests were performed five times each. Both throughput and latency tests ran for ten seconds.

Two additional conventions are used throughout this chapter. The term `offload` collectively refers to GRO, LRO, GSO, and TSO. The Netfilter rule set, when enabled, consists of a hundred ports in the ACCEPT state with `conntrack` active.

4.3 Throughput analysis

Throughput, or the number of bytes processed and delivered by the network stack to the application in a given amount of time, is the most direct measure of performance improvement provided by various bypasses. Figure 4.1 shows the throughput measured across the full configuration matrix for the different variants, and is organized from the configuration most advantageous for the base network stack (on the left) to the least advantageous (on the right).

The two configurations on the left (with offloading enabled) reveal a key aspect of the Linux network stack: its performance primarily depends on the presence of stack optimizations, such as GRO, GSO, LRO, and TSO (see Section 2.3.1). Disabling these optimizations reduces the baseline throughput by more than half, from 47 Gbps to 20 Gbps. This dependence on offloading explains the poor performance

of XDP (25 Gbps for the stateless version and 32 Gbps for the stateful version). Since it runs before the kernel has had a chance to perform coalescing, XDP bypasses are forced to process many smaller packets. The TC bypasses, which run after coalescing, are not subject to this penalty and match the baseline's throughput.

When stack optimizations are disabled, or when the stack processes many small flows for which coalescing is impossible, all variants drop to around 20 Gbps. In this configuration, the stateless versions slightly exceed the baseline's throughput, reaching 23 Gbps. This minimal difference demonstrates the extensive optimization that has gone into the TCP stack over the years: the baseline achieves comparable throughput while executing substantially more instructions per packet than the bypasses do.

When Netfilter rules are loaded, the baseline's throughput drops from 47 Gbps to 40 Gbps for the offloaded version, and from 20 Gbps to 6 Gbps for the non-offloaded version. The kernel must evaluate a chain of hooks for each packet, and this cost becomes dominant as soon as the offloads can no longer amortize it away. Stateless bypasses show no change compared to the configuration without rules, since they unconditionally bypass packets. The stateful variants reach 22 Gbps in the non-offloaded configuration, making them faster than the same variants in the configuration without Netfilter rules. This is easily explained by the fact that the presence of these rules allows flows to be placed in the cache in the first place. Without rules, each packet is sent to the slow path after inevitably failing the Netfilter lookup (see Algorithm 3.1).

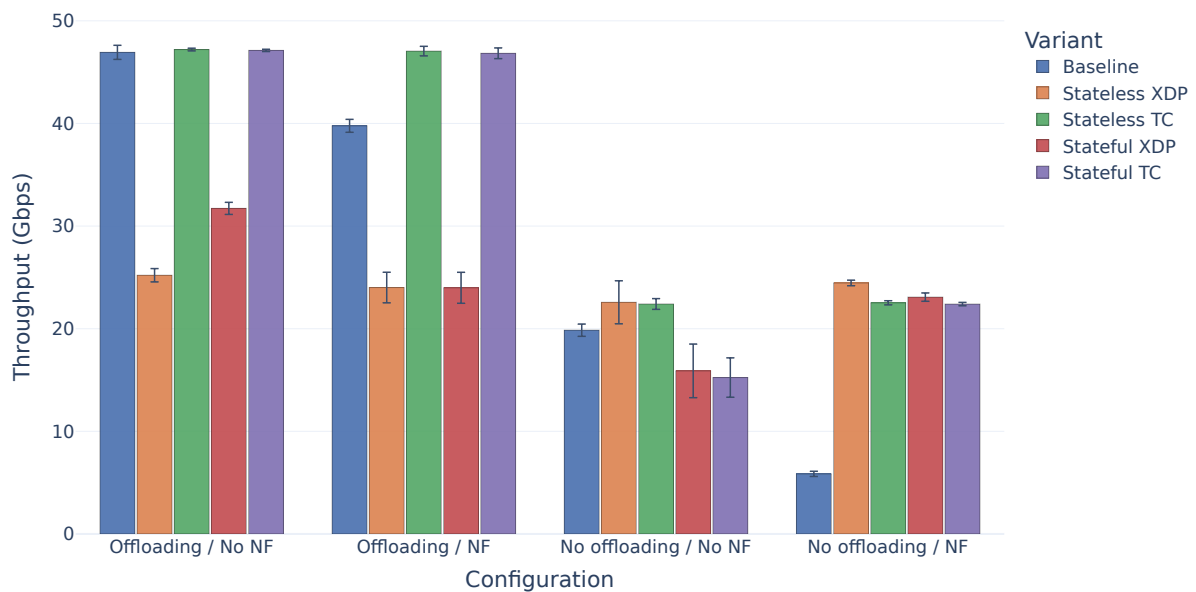


Figure 4.1: Throughput per variant across different configurations (higher is better)

The previously observed trend remains the same in a worst-case scenario designed to stress-test the TCP stack. In this test, the Maximum Segment Size (MSS) was capped at 80 bytes, instructing the sender to encapsulate its payload in segments of this exact size. This results in an abnormally high number of TCP segments that the stack must process for a given volume of data. Consequently, the costs per packet (e.g., evaluation of Netfilter rules and allocation of socket buffers) are amplified relative to the

costs per byte (e.g., copying the payload). Figure 4.2 shows the various throughputs observed, which follow a pattern similar to Figure 4.1.

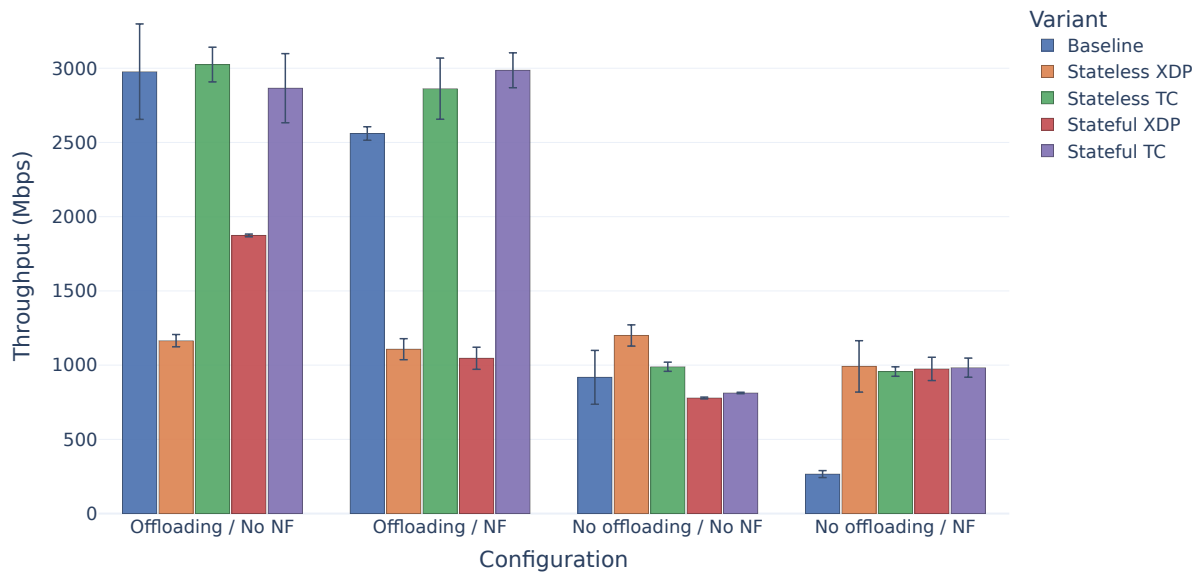


Figure 4.2: Throughput per variant across different configurations during a worst-case scenario (MSS set to 80 bytes, higher is better)

4.4 Latency analysis

Latency, or the round-trip time between sending a packet and receiving its response, is the second key performance indicator for a bypass. While throughput measures the amount of data the stack can process under sustained load, latency measures the speed at which an individual packet traverses the stack. It is therefore the metric of interest for interactivity. Figure 4.3 displays both the average latency and the 99th percentile for each configuration and variant.

The average latency remains remarkably consistent. Across all variants, the mean round-trip time is 16 microseconds, with one exception: the baseline jumps to 17.9 microseconds when Netfilter rules are loaded, regardless of whether offloading is enabled. This penalty is the per-packet cost of evaluating the various Netfilter hooks, and is the logical counterpart to the decrease in throughput discussed in the previous section. The bypass variants, on the other hand, remain stable at 16 microseconds. They are essentially unaffected by the presence of Netfilter rules and show no measurable penalty.

The 99th percentile graph, however, is extremely noisy, and the error bars are large enough that it is difficult to draw statistically meaningful conclusions from it. The high variance (particularly among the stateful variants) can be explained by the fact that the 99th percentile latency is primarily determined by the first packet of each flow: that packet traverses the slow path and triggers the insertion of the flow into the cache, while all subsequent packets skip this part. Any jitter in that “setup” phase directly increases the tail latency. This behavior is less noticeable in the baseline because the Netfilter cost is incurred on every packet, making the results more stable.

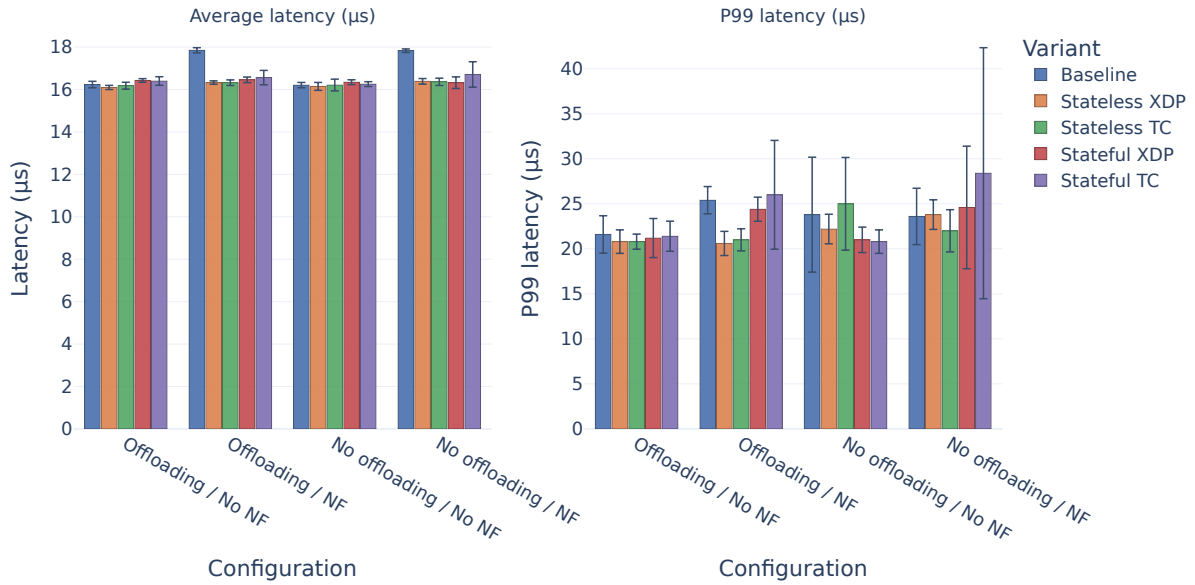


Figure 4.3: Average and P99 latency per variant across different configurations (lower is better)

4.5 CPU utilization and efficiency

On the CPU side, bypass variants reduce both branch misses and cycles per megabyte of throughput. Simplifying the processing path enables the CPU to predict branches more effectively, resulting in more stable pipelining. This is most noticeable when Netfilter hooks are involved, as shown in Figure 4.4. Its rule-traversal logic is dominated by indirect jumps (i.e., following pointers), which branch predictors are notoriously less likely to learn properly. The worst-case configuration (offloading disabled and Netfilter enabled) shows the baseline suffers a number of branch misses an order of magnitude higher than any bypass variant. The only source of unpredictability in the latter arises during the initial evaluation of a flow. Once the flow is added to the cache, the fast path is short and trivially handled by the branch predictor.

This effect translates directly into cycles per megabyte of throughput, as shown in Figure 4.5. By reducing the number of functions executed per packet and keeping the pipeline full, the bypass variants cut the cycle cost by a factor of four in the worst-case configuration. When offloading is enabled, however, the difference shrinks drastically.

Cache misses, in contrast, are not significantly reduced by the presence of a bypass, and the various metrics show high variance between runs (see Figure 4.6). This suggests that Layer 3 and Layer 4 processing does not exert a significant load on the CPU caches in the first place. Removing Netfilter therefore yields only a small benefit, which is to be expected given that the bypass accelerates processing by eliminating branch-heavy code, rather than by improving cache locality.

4.5.1 Profiling

Figure 4.7 shows the CPU cycle breakdown for the three configurations in the worst-case scenario for the TCP stack (i.e., no offloading and Netfilter rules enabled). The leftmost section represents the

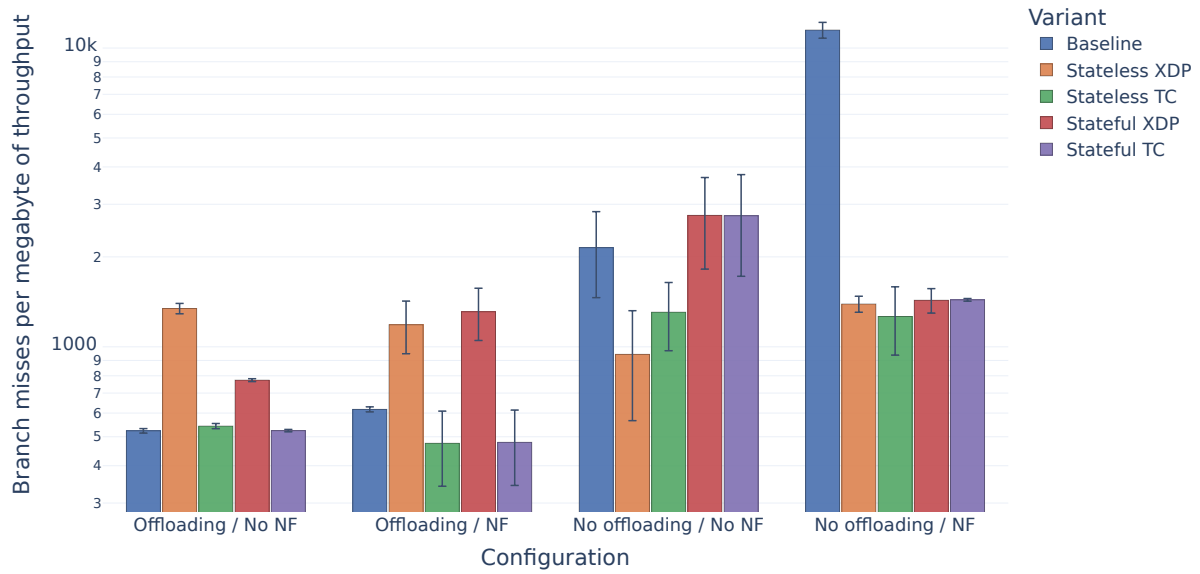


Figure 4.4: Branch misses per megabyte of throughput per variant (log scale, lower is better)

baseline. The most striking visual feature is the filtering segment, which accounts for 26.8% of CPU cycles alone. This corresponds to the time spent in `nf_hook_slow` (the Netfilter entry point) as well as its various subsystems such as `conntrack`. A further 18.5% of cycles are spent idling, which indicates that the CPU is not saturated and that the system processes packets only as quickly as the Netfilter pipeline allows. The remaining cycles are distributed as expected across the TCP stack's receive path: 17.1% are spent on combined TCP and IP processing (with `tcp_v4_rcv` being a notable contributor on the TCP side), 9.2% on `sk_buff` management, 8.3% on packet I/O (NAPI, the NIC driver), and 7.1% on synchronization primitives.

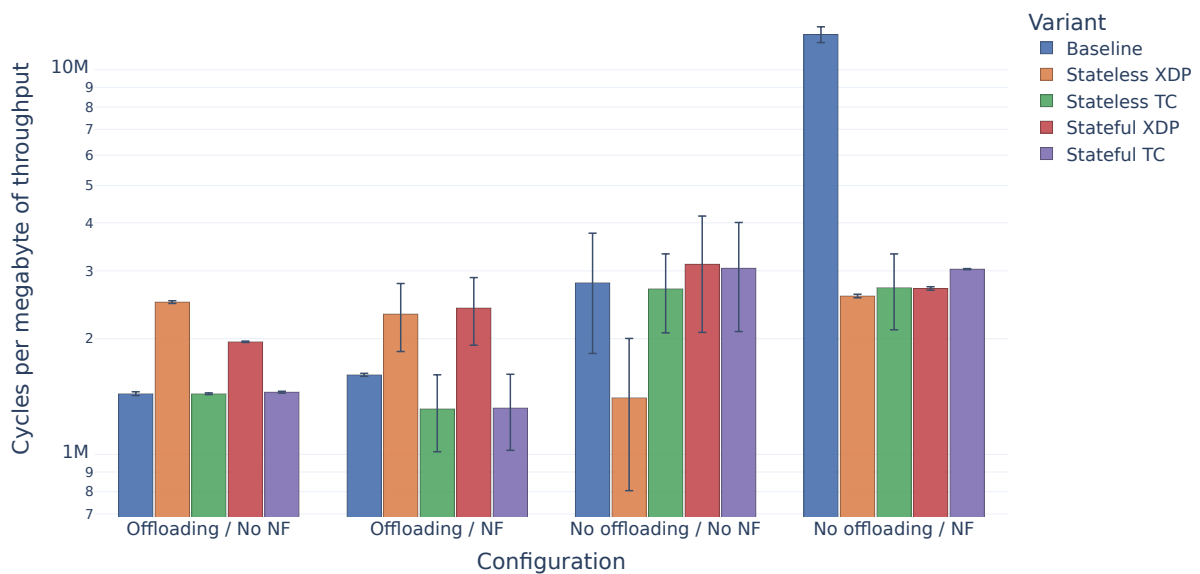


Figure 4.5: Cycles per megabyte of throughput per variant (log scale, lower is better)

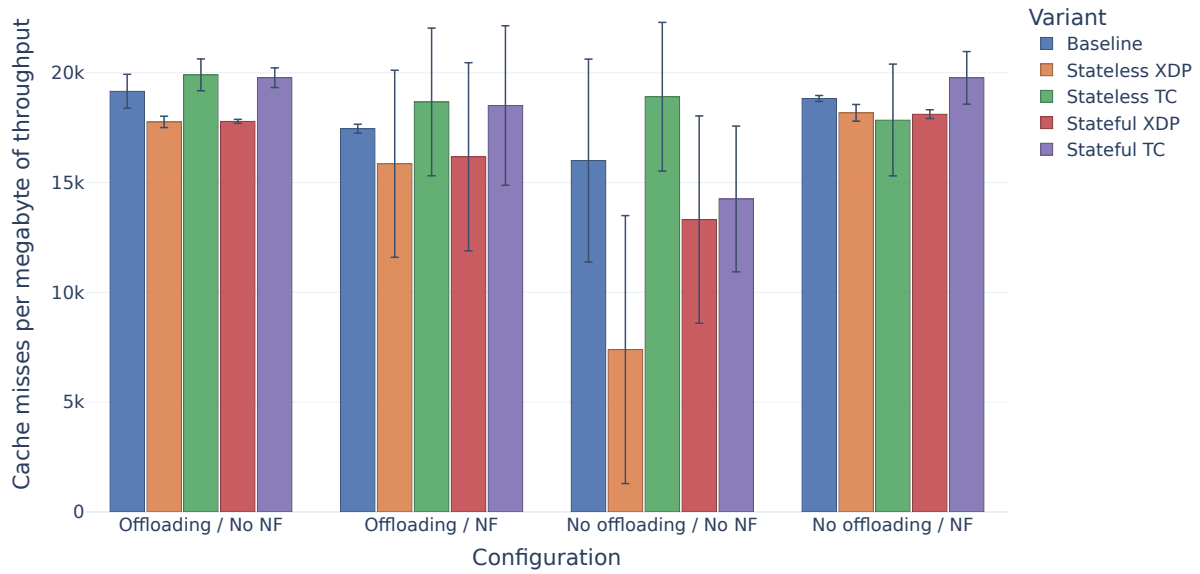


Figure 4.6: Cache misses per megabyte of throughput per variant (lower is better)

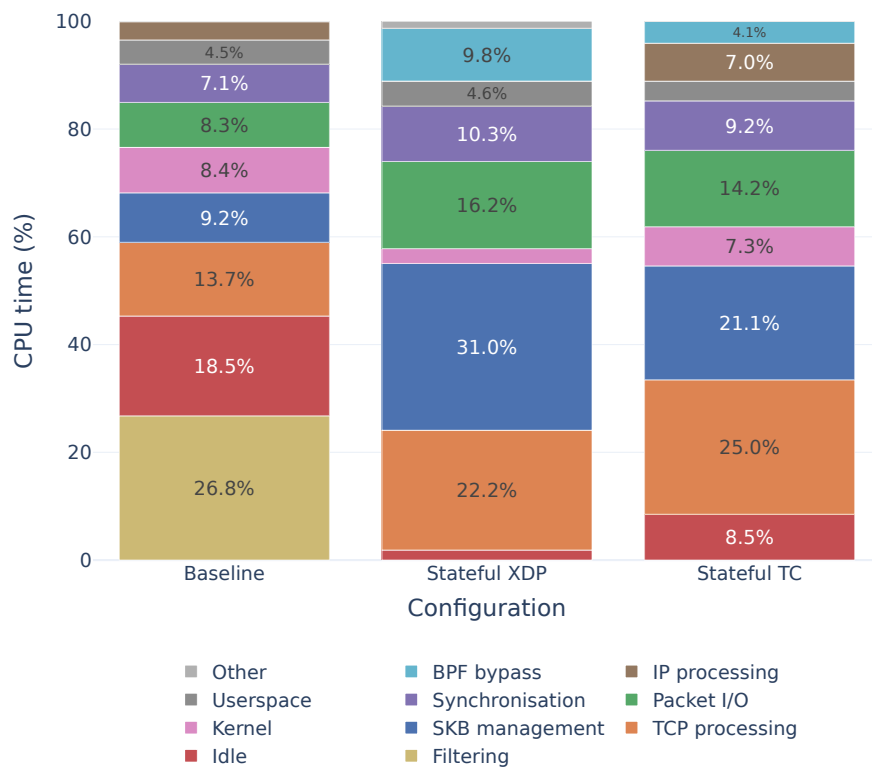


Figure 4.7: CPU cycle breakdown by functional category for configurations (offloading disabled, Netfilter enabled)

The middle bar, corresponding to the stateful XDP bypass, is completely different. The filtering segment drops to 0.4%, which directly explains the throughput gain highlighted in Section 4.3. The remaining cycles do not disappear, however: the idle segment plummets from 18.5% to 1.9%, indicating that the CPU is effectively saturated and that the cycles freed up by removing Netfilter are immediately reinvested

in processing a higher packet rate. This shift is also visible in the flamegraph (see Figure A.2), where the bypass performs only the minimum necessary work to inject the packet into the stack. It retrieves the socket using `bpf_xdp_sk_lookup`, calls the `kfunc` that allocates the socket buffer using `napi_build_skb`, and finally calls the TCP stack using `tcp_add_backlog` and `tcp_rcv_established`. The stacked bar makes the cost of this allocation explicit: the segment representing `sk_buff` management accounts for 31.0% of cycles, becoming the new dominant cost of the receive path. Removing this allocation is therefore a promising avenue for improvement. The eBPF infrastructure itself (dispatcher, program execution, and map lookup) accounts for 9.8% of cycles, representing the inherent overhead of the bypass mechanism.

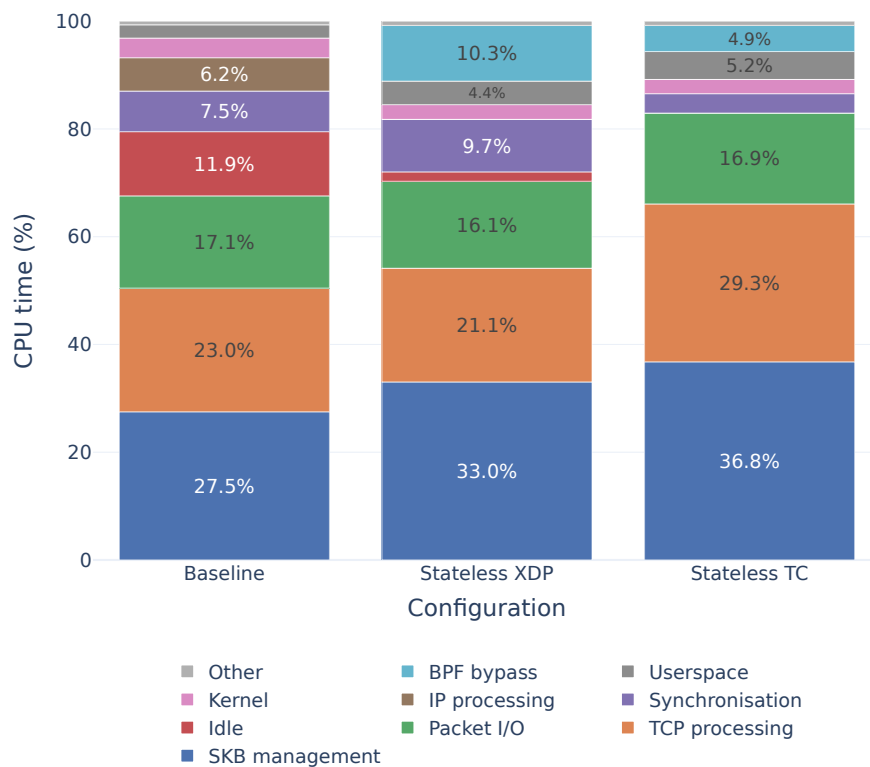


Figure 4.8: CPU cycle breakdown by functional category for configurations (offloading disabled, Netfilter disabled)

The bar on the far right, corresponding to the TC bypass, occupies an intermediate position. Like XDP, it eliminates filtering entirely, and the corresponding flamegraph (see Figure A.3) is shorter than the baseline's. However, as explained in previous sections, the TC bypass runs higher up in the network stack. It has the same entry point as the classic stack, namely `netif_receive_skb_list_internal`. This function ultimately calls `tc_run`, the entry point for traffic control in the kernel, which invokes the various eBPF hooks during its execution. The stacked bar reveals two interesting observations. First, SKB management accounts for 21.1% of the total CPU time, reflecting the high cost of cloning the `sk_buff` metadata (a necessary step in the processing, as explained in the Section 3.3.3), but lower than XDP's 31%, since the structure already exists at the time the hook is executed and only the metadata needs to be copied. Second, the idle segment drops to only 8.5%, indicating that the system retains some

headroom and is not fully saturating the core. This is consistent with TC achieving lower throughput than XDP on the same configuration.

To complete the picture, Figure 4.8 shows the stateless configurations with no Netfilter rules loaded. The overall shape of the baseline column changes: the filtering segment disappears, and the cycles it previously consumed are redistributed to TCP processing (23.0%) and socket buffer management (27.5%), while idle time settles at 11.9%. In other words, even without Netfilter, the baseline spends most of its time moving socket buffers through the receive path. By contrast, the XDP and TC bars are virtually identical to their Netfilter-enabled counterparts: since their filtering segment is already negligible, removing it has little impact on the graph. SKB management remains dominant, reaching 33.0% for XDP and 36.8% for TC.

4.6 Scalability and connection dynamics

Now that we showed that bypasses are effective on a single flow, it is important to consider scalability, specifically how throughput changes as the number of concurrent flows increases.

Figure 4.9 reports the total throughput across all concurrent 100-megabyte flows. This confirms the trend observed in previous sections: with Netfilter, the baseline drops to 5 Gbps, whereas the bypass variants remain between 20 and 25 Gbps, a four- to fivefold gap. Without Netfilter rules, however, the gap narrows considerably and the stateful variants fall behind the baseline, which is consistent with observations made in the case of a single flow.

Figure 4.10 shows the ratio of each bypass's throughput relative to the baseline as the number of concurrent flows increases. A clear trend emerges: with one flow, each bypass delivers four times the baseline's throughput, which confirms the previous results. With four flows, the ratio drops below two, and with eight flows it essentially becomes one. Starting at sixteen flows, the bypasses are marginally slower than the baseline. Part of the performance loss can be explained by the fact that, for the stateful versions, handling more flows increases pressure on the flow cache, and the resulting misses force more packets to fall back to the slower Netfilter path. For the stateless versions, this decline is more difficult to explain. A plausible direction is that the stateless bypass pays a fixed per-packet parsing and classification cost that does not amortize across a flow, whereas the baseline benefits from kernel-internal caches (e.g., route cache, conntrack, socket lookup) that become more effective as concurrent flows become established. However, in the most extreme case of 64 concurrent flows with 1 GB per flow, the baseline could only handle an average of 51.8 flows, whereas the bypasses successfully processed all flows.

Figure 4.11 shows the various results obtained in the form of a heat map. This confirms the earlier observations: the baseline primarily struggles with small individual flows using Netfilter rules, and the various bypasses are therefore more effective in these configurations. Figure 4.11b shows a drop in throughput with 64 concurrent flows at one gigabyte per flow, as explained in the previous paragraph. In this situation, the XDP bypasses regain the advantage.

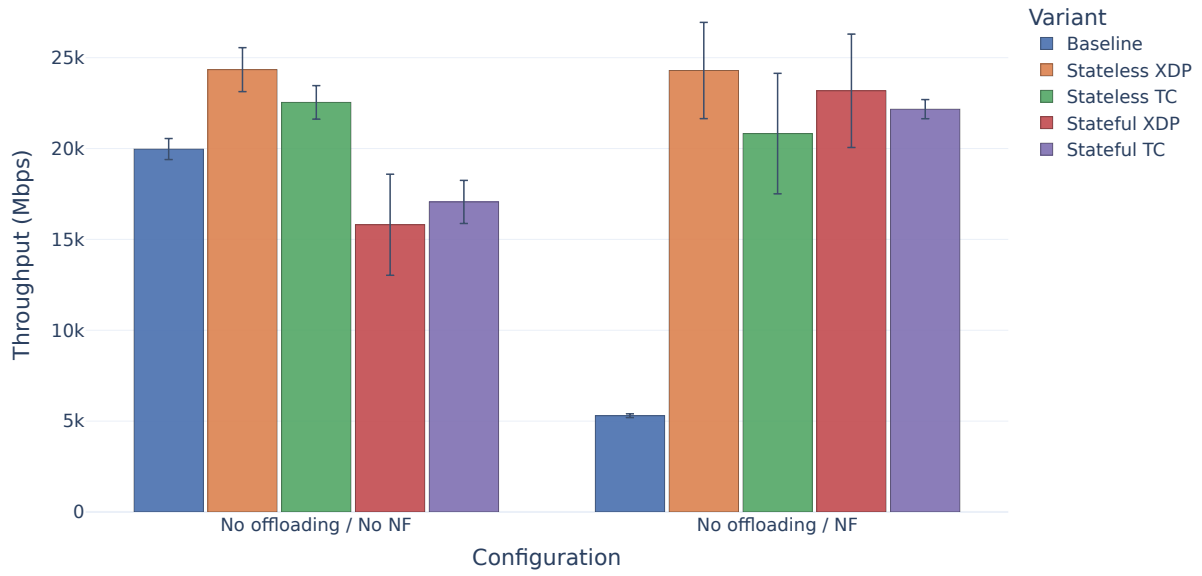


Figure 4.9: Mean throughput per variant across all flows of 100 MB with hardware offloading disabled (higher is better)

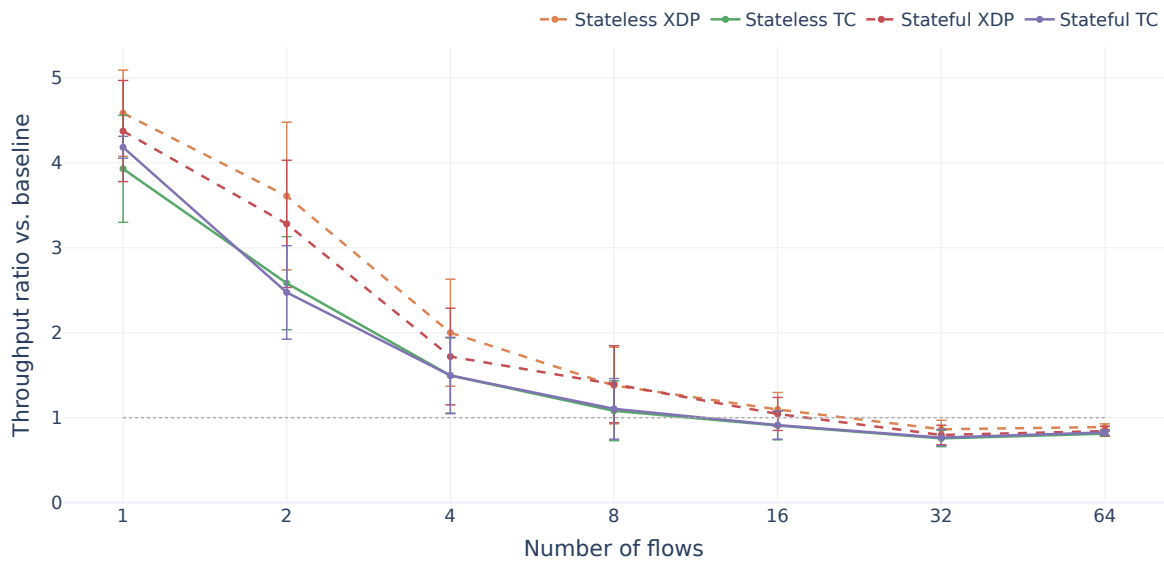
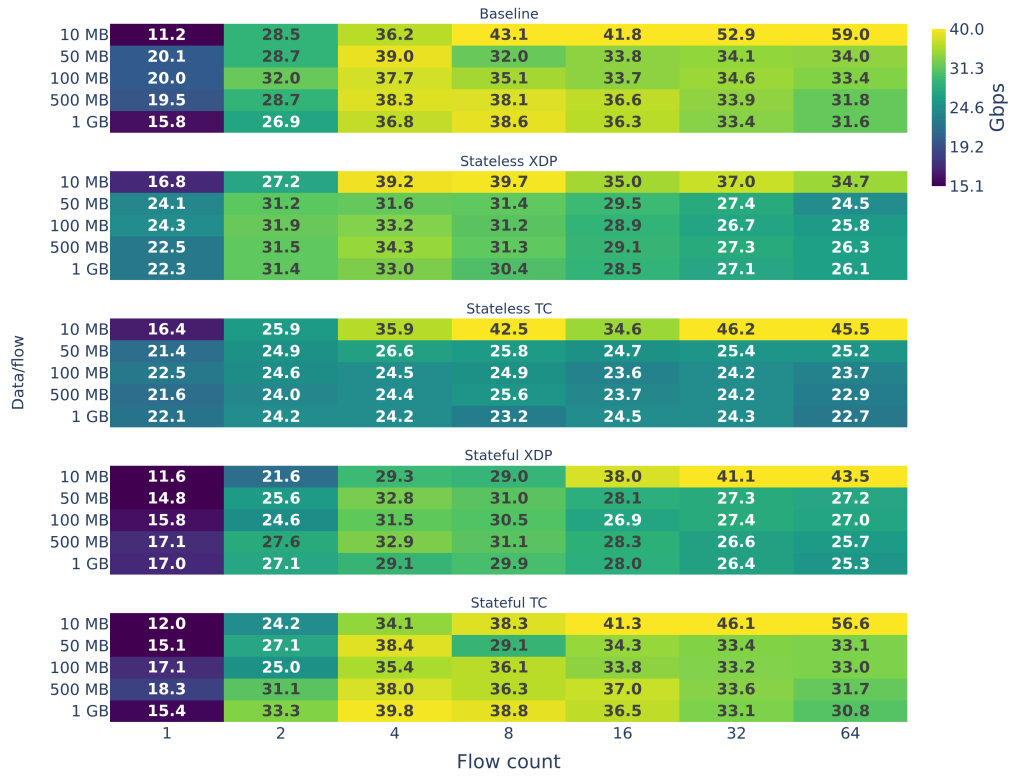
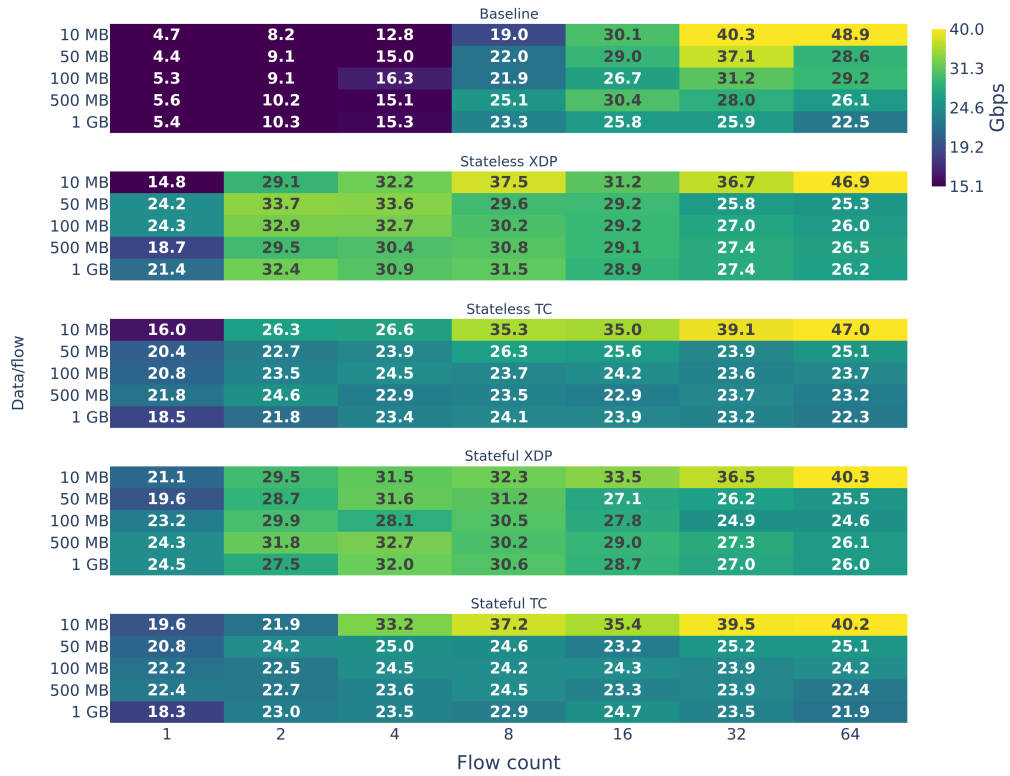


Figure 4.10: Mean throughput of each bypass variant relative to the baseline (offloading disabled, Netfilter enabled) versus concurrent flow count of 100 MB (higher is better)



(a) Netfilter disabled



(b) Netfilter enabled

Figure 4.11: Per-flow throughput across variants with offloading disabled (higher is better)

4.7 Summary

Overall, the various data points collected are consistent with one another. The bypasses are most effective in configurations that are least favorable to the kernel (i.e., when offloading is unavailable and when Netfilter rules are loaded). In these configurations, they deliver approximately four times the baseline throughput, reduce cycles per megabyte by a similar factor, and cut branch misses by an order of magnitude, all while maintaining a stable latency of 16 microseconds. The bypasses are also more robust under heavy concurrency, successfully completing all flows in scenarios where the baseline fails to do so. Conversely, they offer little benefit when the stack is not under pressure, i.e., when offloading is enabled, or when concurrency is high enough for the kernel's internal caches to be effective without causing flows to fail.

Overall, bypasses act as targeted accelerators for configurations where the Linux stack struggles the most, and remain essentially neutral or marginally behind otherwise.

5

Conclusion

In this master's thesis, we investigated whether bypassing part of the Linux kernel's network stack using eBPF is feasible while retaining the security, correctness, and application compatibility that the stack provides. The answer is a qualified yes: the approach works, but its effectiveness depends on the configuration in which the kernel is running.

The bypass performs best when the per-packet cost of the standard stack is high relative to the work performed by the application itself, and cannot be offset by hardware or software offloading. This is most evident in two scenarios. First, when stack-level optimizations such as GRO, GSO, and TSO are disabled or ineffective, the bypass delivers up to a fourfold throughput increase over the baseline. Second, when subsystems that can be safely bypassed (such as Netfilter) burden the standard stack, the bypass retains a comparable advantage. In both cases, the gain is accompanied by an average latency reduction of 1.9 microseconds.

Conversely, the bypass performs less well (and sometimes underperforms) when the standard stack is already operating in its efficient regime. When stack optimizations are active and no Netfilter rules are loaded, the baseline benefits from a heavily amortized per-packet cost that the XDP bypass cannot match, since it incurs a fixed evaluation cost on every packet.

In both favorable and unfavorable configurations, the underlying cause is the same: the bypass trades the various optimizations of the Linux TCP stack for a shorter, more predictable path. As soon as those optimizations counterbalance the fast path's gain, the baseline becomes more efficient. The presented architecture is therefore a targeted accelerator rather than a universal one. Fortunately, the versatility of eBPF makes this an asset rather than a limitation, since a system administrator can dynamically choose when, and to which flows, the bypass is applied.

Future work on this subject could explore several directions to further optimize bypass performance. First, one could explore the elimination of the socket buffer entirely. As seen throughout this thesis, the network stack heavily depends on this structure, which consumes a significant portion of CPU cycles during bypasses, even though its modification and allocation are already kept to a minimum. Removing it would therefore represent a significant potential performance gain. Second, future implementations could bypass a larger number of subsystems: in particular, we currently do not modify the TCP state machine, and it might be worth exploring whether injecting the packet further along the path would free up additional CPU cycles. Finally, one could explore the possibility of executing part of the application's processing directly within the eBPF context, for example by letting the operator define a callback that runs in the kernel. This approach would come with its own set of challenges, particularly with regard to the eBPF verifier, but it could make the bypass extremely flexible while further boosting its performance.

More broadly, we hope this work contributes a small data point to ongoing research on kernel bypasses, by showing that eBPF is a viable option for implementing such systems, and one that brings the safety guarantees of the verifier to a class of designs that traditionally lack them.

Bibliography

- [1] E. Jeong *et al.*, “mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems,” in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA: USENIX Association, Apr. 2014, pp. 489–502. [Online]. Available: <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong>
- [2] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, “IX: A Protected Dataplane Operating System for High Throughput and Low Latency,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 49–65. [Online]. Available: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay>
- [3] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson, “TAS: TCP Acceleration as an OS Service,” in *Proceedings of the Fourteenth EuroSys Conference 2019*, Dresden Germany: ACM, Mar. 2019, pp. 1–16. doi: 10.1145/3302424.3303985.
- [4] K. Yasukata, M. Honda, D. Santry, and L. Eggert, “StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs,” in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO: USENIX Association, June 2016, pp. 43–56. [Online]. Available: <https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata>
- [5] “What is eBPF? An Introduction and Deep Dive into the eBPF Technology.” Accessed: Apr. 25, 2026. [Online]. Available: <https://ebpf.io/what-is-ebpf>
- [6] “Specification of Internet Transmission Control Program,” Request for Comments RFC675, Dec. 1974. doi: 10.17487/RFC0675.
- [7] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport,” Request for Comments RFC9000, May 2021. doi: 10.17487/RFC9000.
- [8] “Worldwide Overview | Cloudflare Radar.” Accessed: May 19, 2026. [Online]. Available: <https://radar.cloudflare.com/>
- [9] S. Ha, I. Rhee, and L. Xu, “CUBIC: a new TCP-friendly high-speed TCP variant,” *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, July 2008, doi: 10.1145/1400097.1400105.
- [10] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, “BBR: congestion-based congestion control,” *Communications of the ACM*, vol. 60, no. 2, pp. 58–66, Jan. 2017, doi: 10.1145/3009824.
- [11] S. Floyd, J. Mahdavi, M. Mathis, and A. Romanow, “TCP Selective Acknowledgment Options,” Request for Comments RFC2018, Oct. 1996. doi: 10.17487/RFC2018.
- [12] D. Borman, R. T. Braden, V. Jacobson, and R. Scheffenegger, “TCP Extensions for High Performance,” Request for Comments RFC7323, Sept. 2014. doi: 10.17487/RFC7323.
- [13] “Segmentation Offloads — The Linux Kernel documentation.” Accessed: May 19, 2026. [Online]. Available: <https://docs.kernel.org/networking/segmentation-offloads.html>

- [14] J. Corbet, “Zero-copy TCP receive,” *LWN.net*, Apr. 2018, Accessed: May 19, 2026. [Online]. Available: <https://lwn.net/Articles/752188/>
- [15] “net - kernel/git/torvalds/linux.git - Linux kernel source tree.” Accessed: Apr. 28, 2026. [Online]. Available: <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net?h=v6.19>
- [16] A. Foong, T. Huff, H. Hum, J. Patwardhan, and G. Regnier, “TCP performance re-visited,” in *2003 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS 2003.*, Mar. 2003, pp. 70–79. doi: 10.1109/ISPASS.2003.1190234.
- [17] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, “Understanding host network stack overheads,” in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, in SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 65–77. doi: 10.1145/3452296.3472888.
- [18] “DPDK/dpdk.” Accessed: Apr. 28, 2026. [Online]. Available: <https://github.com/DPDK/dpdk>
- [19] L. Rizzo, “Netmap: a novel framework for fast packet I/O,” in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, in USENIX ATC'12. USA: USENIX Association, June 2012, p. 9.
- [20] B. Gbadamosi *et al.*, “The eBPF Runtime in the Linux Kernel.” Accessed: May 19, 2026. [Online]. Available: <https://arxiv.org/abs/2410.00026>
- [21] J. H. Salim, R. Olsson, and A. Kuznetsov, “Beyond softnet,” in *Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5*, in ALS '01. USA: USENIX Association, 2001, p. 18.
- [22] “skbuff.h « linux « include - kernel/git/torvalds/linux.git - Linux kernel source tree.” Accessed: May 21, 2026. [Online]. Available: <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/skbuff.h?h=v6.19>
- [23] A. Stephan, “The Path of a Packet Through the Linux Kernel,” 2023, doi: 10.2313/NET-2024-04-1_16.
- [24] “Jiri Benc - The Network Packet's Diary: A Kernel Journey.” Accessed: Apr. 21, 2026. [Online]. Available: <https://www.youtube.com/watch?v=T5TvPRQFNoM>
- [25] “IEEE P802.3dj 200 Gb/s, 400 Gb/s, 800 Gb/s, and 1.6 Tb/s Ethernet Task Force.” Accessed: May 19, 2026. [Online]. Available: <https://www.ieee802.org/3/dj/index.html>
- [26] J. Corbet, “JLS2009: Generic receive offload,” *LWN.net*, Oct. 2009, Accessed: May 21, 2026. [Online]. Available: <https://lwn.net/Articles/358910/>
- [27] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, “MegaPipe: A New Programming Interface for Scalable Network I/O.”
- [28] K. Jang, S. Han, S. Han, S. Moon, and K. Park, “SSLShader: cheap SSL acceleration with commodity processors,” in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, in NSDI'11. USA: USENIX Association, Mar. 2011, pp. 1–14.
- [29] “lighttpd/lighttpd1.4.” Accessed: May 21, 2026. [Online]. Available: <https://github.com/lighttpd/lighttpd1.4>
- [30] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, “TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing,” presented at the 2021 USENIX Annual Technical

- Conference (USENIX ATC 21), 2021, pp. 489–502. Accessed: Apr. 25, 2026. [Online]. Available: <https://www.usenix.org/conference/atc21/presentation/zou>
- [31] “project-dune/dune.” Accessed: Apr. 28, 2026. [Online]. Available: <https://github.com/project-dune/dune>
 - [32] “memcached/memcached.” Accessed: May 21, 2026. [Online]. Available: <https://github.com/memcached/memcached>
 - [33] “scylladb/seastar.” Accessed: Apr. 28, 2026. [Online]. Available: <https://github.com/scylladb/seastar>
 - [34] “facebookincubator/katran.” Accessed: Apr. 25, 2026. [Online]. Available: <https://github.com/facebookincubator/katran>
 - [35] “cilium/cilium.” Accessed: Apr. 25, 2026. [Online]. Available: <https://github.com/cilium/cilium>
 - [36] “Map types (Linux) - eBPF Docs.” Accessed: Apr. 25, 2026. [Online]. Available: <https://docs.ebpf.io/linux/map-type/>
 - [37] “2024 CrowdStrike-related IT outages.” Apr. 22, 2026. Accessed: Apr. 26, 2026. [Online]. Available: https://en.wikipedia.org/w/index.php?title=2024_CrowdStrike-related_IT_outages&oldid=1350585601
 - [38] “Falcon Content Update Remediation and Guidance Hub | CrowdStrike.” Accessed: Apr. 26, 2026. [Online]. Available: <https://www.crowdstrike.com/falcon-content-update-remediation-and-guidance-hub/>
 - [39] M. Miller, “Trends and Challenges in the Vulnerability Mitigation Landscape,” presented at the BlueHat IL, Tel Aviv, Israel: Microsoft Security Response Center, Aug. 2019.
 - [40] K. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan, and T. Jaeger, “SoK: Challenges and Paths Toward Memory Safety for eBPF,” in *2025 IEEE Symposium on Security and Privacy (SP)*, May 2025, pp. 848–866. doi: 10.1109/SP61157.2025.00134.
 - [41] A. M. Turing, “On Computable Numbers, with an Application to the Entscheidungsproblem,” *Proceedings of the London Mathematical Society*, vol. 42, no. 1, pp. 230–265, Jan. 1937, doi: 10.1112/plms/s2-42.1.230.
 - [42] T. Høiland-Jørgensen *et al.*, “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*, 2018, pp. 54–66.
 - [43] “[GIT] Networking - David Miller.” Accessed: Apr. 27, 2026. [Online]. Available: <https://lore.kernel.org/lkml/20160727.010753.2221383279830501569.davem@davemloft.net/>
 - [44] “L4Drop: XDP DDoS Mitigations.” Accessed: Apr. 27, 2026. [Online]. Available: <https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/>
 - [45] “Packet, Where Are You?.” [Online]. Available: <https://github.com/cilium/pwru>
 - [46] “TC Actions - Environmental Rules — The Linux Kernel documentation.” Accessed: May 02, 2026. [Online]. Available: <https://www.kernel.org/doc/html/v6.19/networking/tc-actions-env-rules.html>
 - [47] “Netfilter’s flowtable infrastructure — The Linux Kernel documentation.” Accessed: May 02, 2026. [Online]. Available: https://www.kernel.org/doc/html/v5.15/networking/nf_flowtable.html

[48] "HewlettPackard/netperf." Accessed: May 16, 2026. [Online]. Available: <https://github.com/HewlettPackard/netperf>

Appendix A

Partial flamegraphs

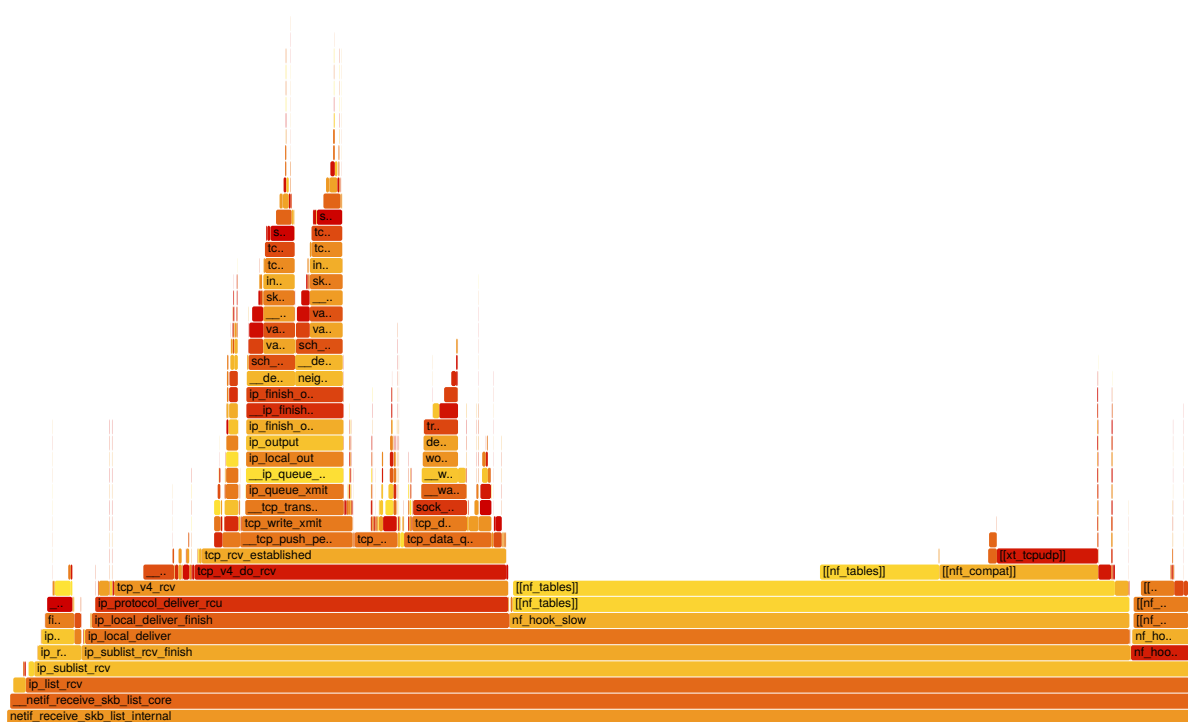


Figure A.1: Flamegraph of the baseline during a throughput test zoomed on the stack entry point (offload disabled, Netfilter enabled)

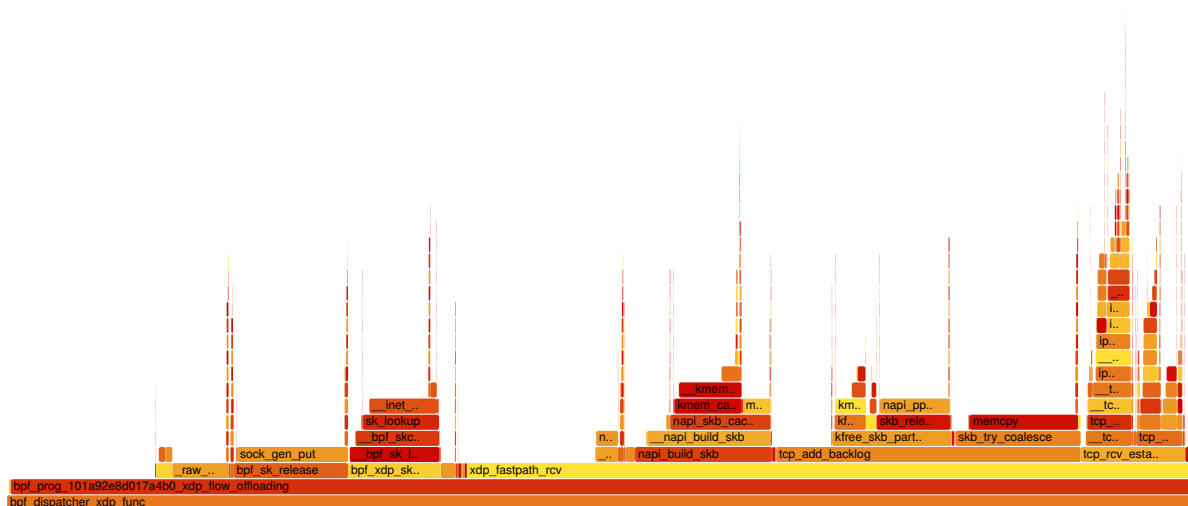


Figure A.2: Flamegraph of the XDP bypass during a throughput test zoomed on the BPF entry point (offload disabled, Netfilter disabled)

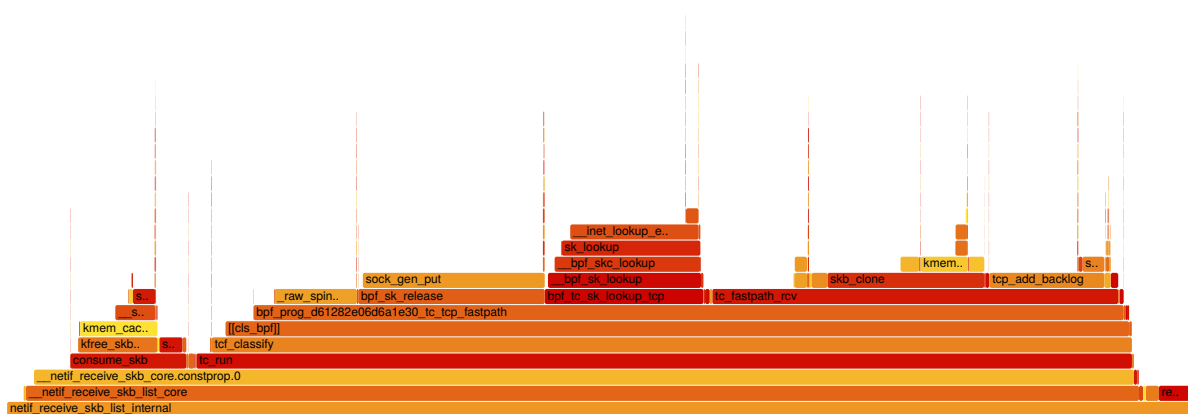


Figure A.3: Flamegraph of the TC bypass during a throughput test zoomed on the stack entry point (offload disabled, Netfilter disabled)

Appendix B

Complete flamegraphs

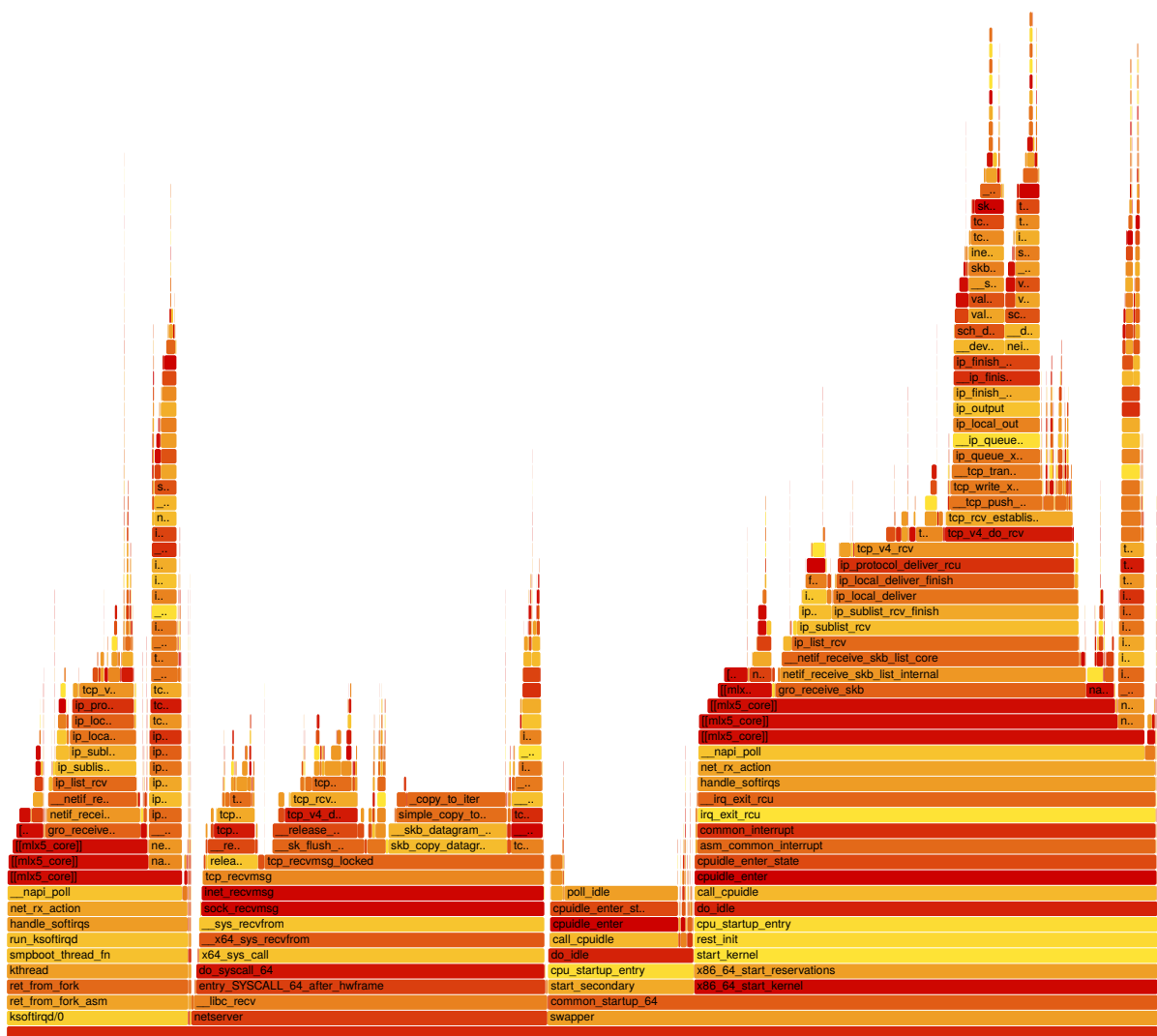


Figure B.1: Flamegraph of the baseline during a throughput test (offloading disabled, Netfilter disabled)

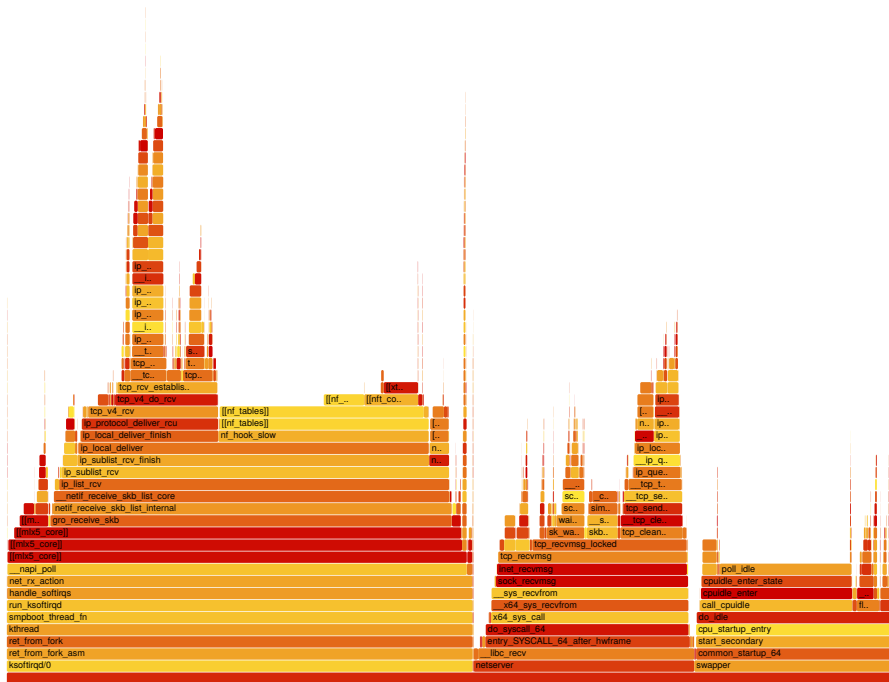


Figure B.2: Full flamegraph of the baseline during a throughput test (offloading disabled, Netfilter enabled)

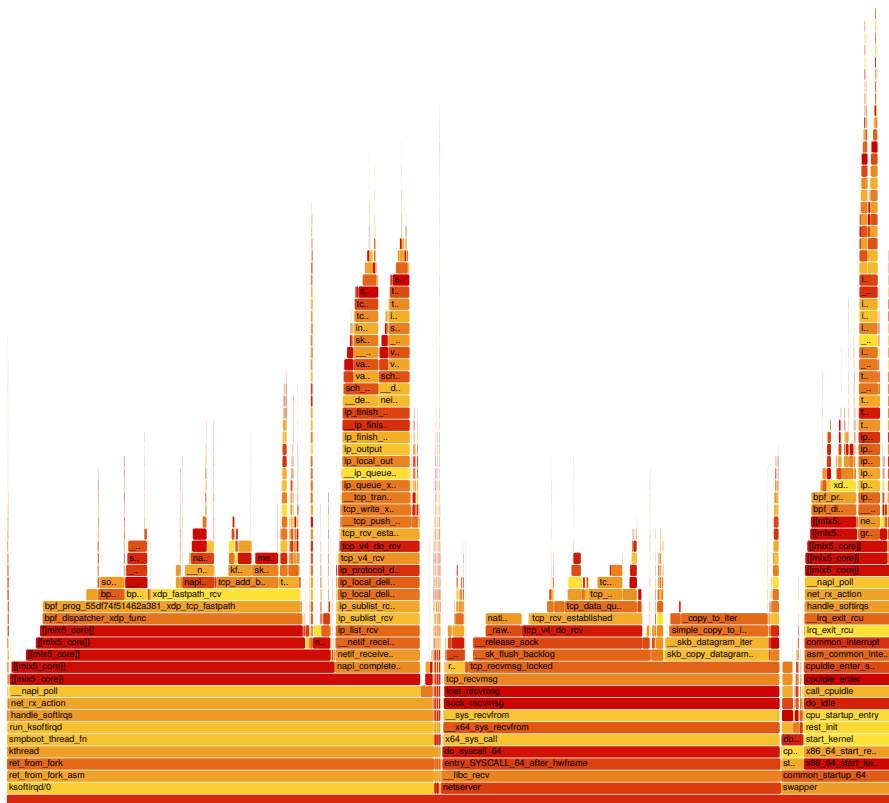
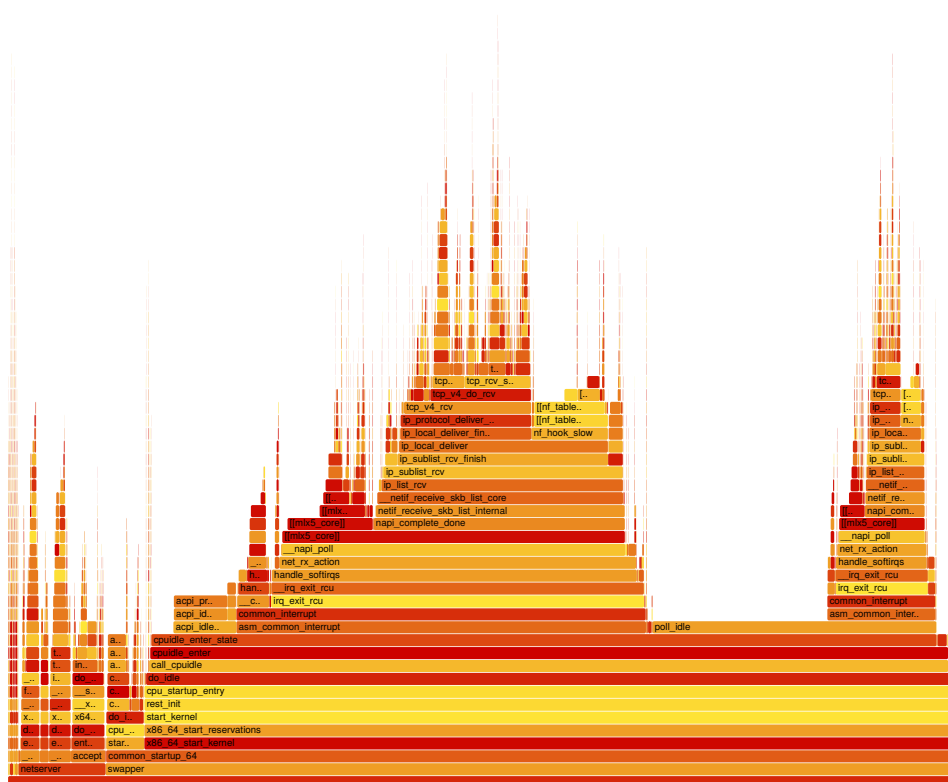
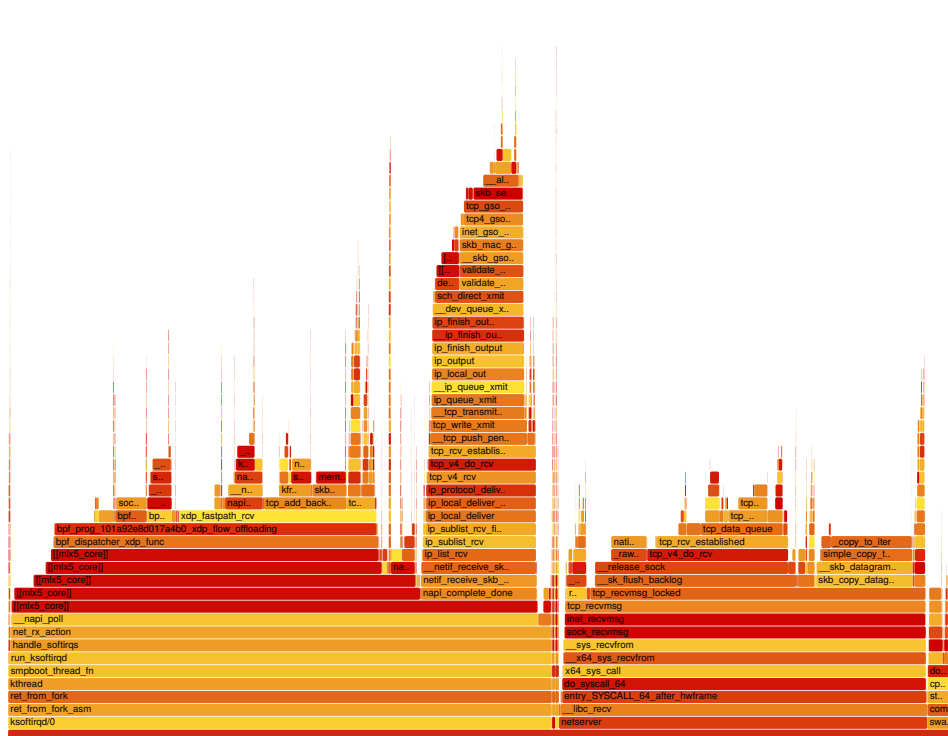


Figure B.3: Full flamegraph of the XDP bypass during a throughput test (offloading disabled, Netfilter disabled)



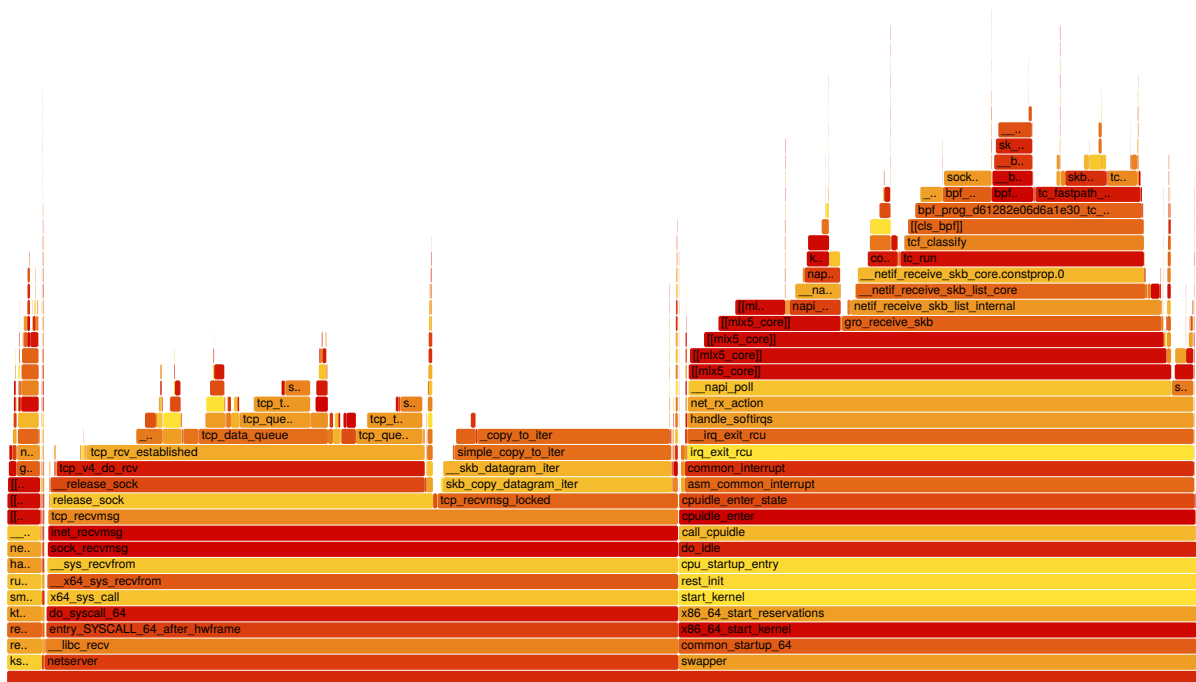


Figure B.6: Full flamegraph of the TC bypass during a throughput test (offloading disabled, Netfilter disabled)

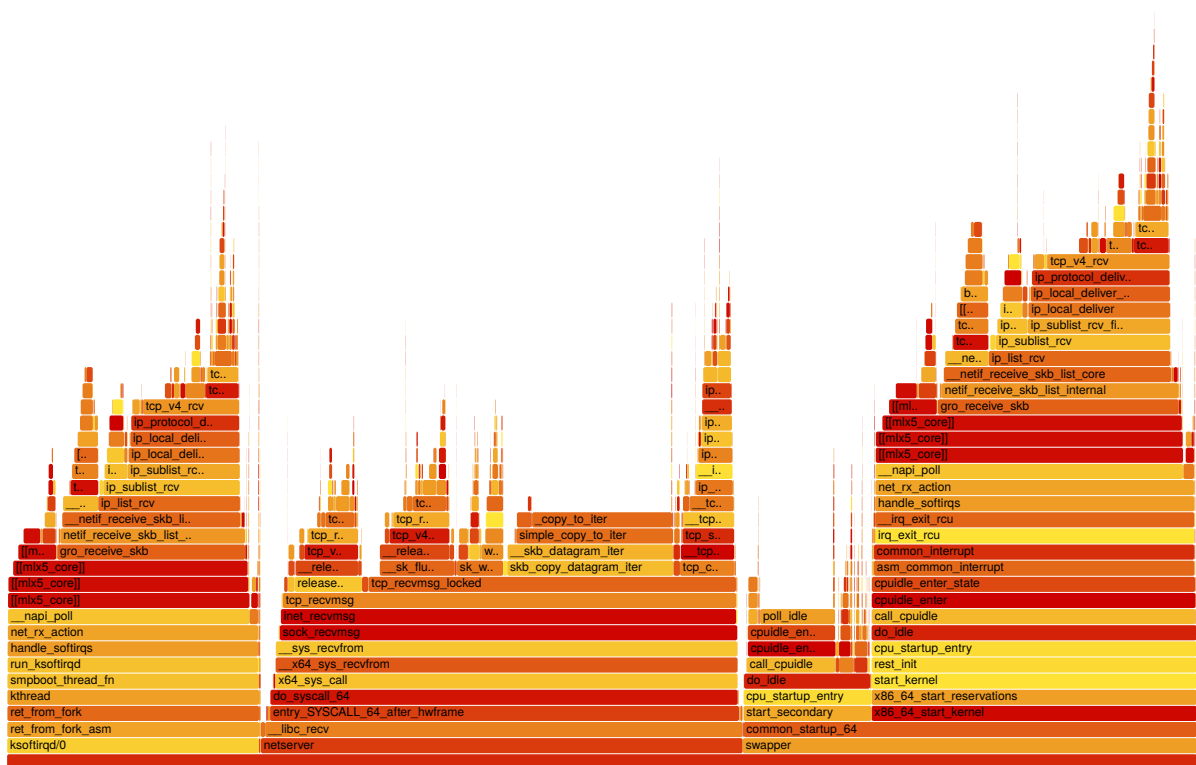


Figure B.7: Full flamegraph of the TC bypass during a throughput test (offloading disabled, Netfilter enabled)

UNIVERSITÉ CATHOLIQUE DE LOUVAIN

École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl