Software Datapaths for Multi-Tenant Packet Processing

Paul Chaignon

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Software Datapaths for Multi-Tenant Packet Processing

Ph.D. Thesis

to obtain the degree of Doctor of Philosophy issued by

University of Lorraine

(Specialization in Computer Science)

publicly presented and discussed May 7th, 2019

by

Paul Chaignon

Committee in charge

President:	Marine Minier	Professor, University of Lorraine
Reporters:	Pierre Sens	Professor, Pierre and Marie Curie University
	Laurent Mathy	Professor, University of Liège
Examiners:	Fulvio Risso	Associate Professor, Polytechnic University of Turin
	Filip De Turck	Professor, Ghent University
Advisors:	Kahina Lazri	Research Engineer, Orange Labs
	Jérôme François	Researcher, Inria
	Olivier Festor	Professor, University of Lorraine

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Formatted with a modified version of the thesul class.

Acknowledgments

First of all, I would like to thank my academic advisors, Jérôme François and Olivier Festor, for their support and availability throughout my Ph.D. studies despite my limited stays at Inria Nancy.

I would also like to acknowledge Fulvio Risso, Pierre Sens, Laurent Mathy, Filip De Turck, and Marine Minier for accepting to join my dissertation committee. I'm particularly thankful to Fulvio Risso who drove a great distance to attend my (inconsequential) defense.

I remember the first time I met most people who ended up having a significant contribution to my thesis. I met Kahina Lazri in her office, on the first day of my Master internship, and remembered attending her presentation at SSTIC'14. Little did I know that I would pursue my studies, just six months later, as a Ph.D. student under her guidance. Thank you Kahina for your trust and unwavering support. I know I often was able to focus on technical aspects of my work *grâce à toi*.

On the first day of my internship, I also met Xiao. He was sharing a small and dark office with Aurélien, surrounded by computer parts, development boards, and books. I cannot thank Xiao enough for the many times he helped me understand weird code behaviors, debug my programs, or rewrite an introduction on a Friday evening. Passing from Xiao's office to Maxime's, next door, was quite a change; Maxime had a large, bright, and tidy office. Maxime's self-discipline has been an inspiration to me. Three years later, I find myself having (somewhat) successfully copied several of Maxime's good habits[1]. Weirdly enough, I don't remember when I first met Alex. Yet, Alex probably had the most influence on my work. Having read enough papers to write several theses, Alex often gave me short, inspirational summaries of research trends in systems, which guided me in my research. I'm also grateful for the many talks we had on the way to the subway (after he went back for the forgotten keys).

I also want to thank my interns, Thibault and Diane, who I met respectively for his interview and on the first day of her internship. Their contagious motivation arrived just when I needed it.

I unfortunately cannot thank every single past and present colleague at Orange Labs, but I am really grateful to all of them for the shared times working (a little) and laughing (a lot). If there is one last colleague I ought to thank, it is Pascal. Pascal equally shares his time between work, laughter, and miscellaneous concerns[2], but at Orange Labs, he accounts for a large part of the team's good atmosphere.

Of course, I remember seeing Céline for the first time[3]. Thank you Céline for your love and our discussions, which gave me another point of view on many aspects of my thesis and life.

My thanks also go to my family, especially Lou, who proof-read my first papers before morning deadlines, while the rest of us, mere diurnal creatures, slept.

Before concluding these acknowledgments, I wanted to thank the many writers who posted advice and accounts of Ph.D. studies online, including Philip Guo's *The PhD Grind*. These texts often helped me look past my own experiences.

[1] Yes Maxime, I even ditched the 5pm biscuits for fruits.

[2] Based on real-life measurements.

[3] Students, do go to summer school, you never know...

À mes grands-parents,

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Cloud computing realizes the dream of utility computing by virtualizing and sharing hardware resources across tenants, thereby allowing crucial economies of scale. The illusion of infinite resources and the elimination of up-front, infrastructure investments convinced customers to outsource their processing workloads to the cloud [7]. While system virtualization enables sharing the CPU, the memory, and I/O devices across tenants, properly sharing the network itself requires additional care: tenants should be properly isolated, both in terms of performance and physical access while each having their own view of the network. Cloud computing therefore calls for *multi-tenant networks*: networks capable of delivering packets to multiple, isolated tenants, from the core network to tenant domains on end hosts.

Whereas public and private clouds initially hosted few network IO-bound workloads, there is now considerable pressure on multi-tenant networks, and in particular on end hosts, to deliver packets with high throughput and low latency. The cause of this change is two fold. On one hand, virtual machine density on hosts is increasing, pushing cloud providers to adopt 10Gbps and 40Gbps network interface cards (NIC). On the other hand, due to the rise of Network Functions Virtualization (NFV) and recent advances in software packet processing, network IO-bound workloads are becoming more common in virtual machines. Initially proposed by ISPs, the NFV paradigm advocates a move away from specialized hardware appliances, towards software implementations running on commodity hardware.

Offering high performance network interfaces to tenants over commodity hardware is however challenging for cloud providers. Cloud providers must process packets in and out of virtual machines with as few resources as possible, always in an effort to increase virtual machine density on hosts. In addition, commodity hardware achieves high performance through the use of numerous caches (e.g., the instruction and data caches and the TLB) and pipelining optimizations (e.g., out-of-order execution and branch target prediction). These components and optimizations, as well as the contention for shared hardware resources, introduce variability in packet processing performance. Finally, the need to share the network across tenants on end hosts adds to the high performance challenge: taking packets across memory isolation boundaries has a cost, which we investigate in Section 2.1.

As discussed in Chapter 2, cloud providers increasingly rely on specialized, high performance hardware to provide network accesses to tenants. While some implement the entire datapath on dedicate, special hardware [66], most achieve high performance through partial offloads of network functions [89, 33, 50]. As the contrasts between cloud providers' reliance on hardware highlights, hardware implementations are not without their drawbacks. First, hardware devices are rather inflexible: they have limited resources, are difficult to upgrade, and require expertise in dedicated, low-level languages to program. Emerging programmable devices, although easier to upgrade, retain the resource limitations of fixed-function hardware. Worse, these devices may simply not be available to smaller cloud providers before several years.

In this thesis, we argue that significant performance improvements are attainable in end-host multi-tenant networks, without depending on specialized hardware. While we do not expect software implementations to meet the performance of hardware devices, as we will show, software datapaths can be designed to process packets several times more efficiently than current solutions, thereby reducing the performance gap. In particular, we advocate for a consolidation of cloud provider services on the host to reduce the cost of packet processing between the NIC and the virtual machines. Such consolidated datapaths also enable tenants to offload network services from their virtual machines to the host.

1.1.1 Network Functions in Cloud Platforms

Cloud platforms run a large and diverse set of network functions, to support a variety of needs.

- *Connectivity and Isolation.* At the very least, the host ends tunnels, forwards packets according to L2 or L3 lookups, and communicates with (or emulates) the network drivers of virtual machines.
- *Traffic Engineering.* Cloud networks rely on a number of traffic engineering functions at the end hosts, both for their own needs (e.g., accounting) and for tenants (e.g., load balancing services).
- *Security.* In addition to the common firewalling services exposed to tenants, cloud platforms may run Intrusion Detection/Prevention Systems (IDPSes), rate limiters, and VPNs. These functions protect both the cloud against malicious users and tenants against external attackers.
- *Tenant Services.* The most diverse set of functions likely runs in virtual machines. Tenants may create virtual machines to run their own network functions, from enterprise middleboxes [130] to Evolved Packet Core (EPC) components [76].

These functions are diverse in goals, but also in terms of processing overhead—rate limiters have a small cost compared to IPSec VPNs—and responsibility; some are the responsibility of cloud providers, others of tenants. Conversely, these functions have in common that they all run inline, on the path of packets to virtual machines, making their efficiency critical to the end to end performance.

1.1.2 Performance Challenges

The datapaths of cloud platforms must provide the aforementioned network functions to tenants with high throughput and low latency. They however face a number of challenges, which hinder their efficiency.

Long Datapaths. In addition to the already costly reception and transmission of packets, cloud datapaths must execute a large number of operations on packets before sending them to the virtual machines or to the NIC. Nevertheless, datapaths have a budget of only few CPU cycles per packet: on a 2.5GHz CPU core, minimum sized, 64B packets must be processed in an average of less than 168 cycles to support line rate 10Gbps processing.

Inefficient Detours. Many services are implemented as inefficient detours from the existing datapath. For example, a load balancer may run as a separate process to which the software switch redirects packets before sending them to virtual machines. These detours encode organizational boundaries rather than technical constraints: in the previous example, different entities or companies may develop the load balancer and the software switch.

Performance Gap. As we have seen with the examples of network functions, datapaths execute a number of operations with very different costs. One operation in particular, common to all multi-tenant datapaths, has a much higher cost than others: the exchange of packets between the host and the virtual machines. This higher cost induces a performance gap in the datapath operations: on the receive path, performance gains on operations executed after the gap may have

a lesser effect on the overall datapath performance than if those operations had been executed before the gap. For example, aggregating packets or dropping illegitimate packets likely reduces the processing cost for subsequent network functions and will have a much higher impact if executed before the copy to virtual machines. Admittedly, these optimizations are best executed as soon as possible during datapath processing.

1.1.3 Opportunities at the Infrastructure Layer

We argue that these challenges can be overcome by designing key components of the datapath to follow two high level principles.

Consolidate network functions in the datapath. Several network functions can be consolidated into a single component to reduce the cost of packet transmissions between functions. Such consolidated functions eliminate the inefficient datapath detours, but must preserve the logical separations between network functions. Indeed, maintaining a separation of concerns is important as different network functions are likely to be operated and maintained by different groups of people.

Execute network functions before the performance gap. The datapath should offer solutions for tenants to offload some of their network functions to the host. In particular, both the tenant and the cloud provider benefit from running security services before the copy of packets to the virtual machines, since these services are more likely to drop packets and connections.

In applying these principles to existing multi-tenant datapaths, we face a number of obstacles which we address in this thesis' contributions.

Prevent failure propagation in consolidated components. Complex network functions are unlikely to be free of bugs. To prevent a failure inside a network function to propagate to other, consolidated functions, we need to maintain a form of isolation between functions. To isolate functions without hindering their exchanges of packets, we can rely on a large number of software memory isolation techniques, which we survey in Section 2.1.3.

Extend caching optimizations. Despite recurrent concerns on the performance variations caused packet processing caches, as we highlight in Section 2.2, several large-scale production systems rely on caches, sometimes with several layers, to improve performance. Because these caching mechanisms make assumptions on the processing steps, they are however difficult to extend when consolidating network functions.

Isolate offloaded functions. Lightweight isolation techniques are also necessary to isolate the host from the offloaded network functions and to prevent one network function from accessing other tenants' data. In particular, although all offloaded functions run inside the same consolidated component, they should not be able to receive other tenants' packets or to access data structures owned by other offloaded functions.

Maintain resource fairness despite consolidation. In addition, hardware resources must be fairly shared among offloaded programs, as if the latter were running inside virtual machines or containers. However, because all offloaded functions are consolidated into a single program to improve performance, we cannot rely on the usual schedulers to enforce fairness.

1.2 Contributions and results

This thesis presents the state of the art of end-host multi-tenant networking as well as two novel systems to apply the aforementioned principles to concrete state-of-the-art systems and improve their end-to-end performance.

State of the art. Chapter 2 discusses the state of the art of end-host multi-tenant networking with a focus on packet processing performance bottlenecks. We review the literature on three aspects of end-host networking architectures that are key to the overall performance:

- The demultiplexing and delivery of packets to tenant domains accounts for large portion of processing cost in multi-tenant networking architectures. We take an in-depth look at processing costs and improvements to multi-tenant datapaths over the last 15 years, since the first paravirtualized drivers in Xen. We highlight the cost of hardware memory isolation and survey alternative isolation techniques.
- The demultiplexing logic, implemented at the software switch before the actual delivery of packets to tenant domain, can also have a severe impact on performance. We survey the literature on packet classification algorithms and their necessary optimizations. We contrast contributions from the literature with reports on experiences with large-scale production systems.
- Processing offloads delegate tasks to faster components, often implemented on specialized hardware, in order to improve the overall performance or to reduce the burden on the multi-tenant system itself. We identify and review three generations of hardware offloads and draw a parallel with offloads of tenants' workloads to the host.

Extensible software switch. We design the Oko software switch to execute stateful filtering and monitoring programs as part of its packet classification pipeline. Oko enables the consolidation at the software switch of network services running on the host. Oko achieves the following improvements over state-of-the-art systems:

- The extension of highly optimized packet classification algorithms with a limited impact on performance. Compared to the software switch on which Oko is based, our modifications add no measurable overhead (number of packets processed each second) with all caches enabled and a 2% overhead with the first-level cache disabled.
- The isolation of faults from extensions with negligible runtime overhead. Oko relies on recent advances in software memory isolation to enforce isolation with a static analysis and few runtime bounds checks.
- Significant performance improvements compared to the virtual machines and separate processes that usually host extensions to the datapath. When evaluated with a set of network functions, Oko outperforms the same functions running inside virtual machines by 2–3x, and by 1.7–1.9x when running as processes separate from the switch.

We present Oko in Chapter 3.

Tenant offloads to the host. We design a framework to allow tenants to offload network services from their domains to the host's datapath, thereby benefiting from increased packet processing performance. Our framework demonstrates the following:

- The feasibility of performance isolation despite all tenant services running inside the same datapath. When evaluating our fairness mechanism with a web server benchmark, our system adds a 2.6% overhead on the web server's performance, but performs significantly better than the Linux scheduler under the same constraints (2–14x depending on the number of offloaded functions).
- The practicability of software memory techniques to isolate tenants with low cost. In this framework, we extend the software memory isolation technique used in Oko to sandbox the services of each tenant.
- The benefit of host offloads in increasing the performance attainable by network services. By executing network functions before the performance gap, i.e., the copy to the virtual machines or userspace processes, our system enables a 4–6x performance improvement.

We present our host offload work in Chapter 4.

1.3 Overview of the thesis

This thesis is structured as follows. In Chapter 2, we discuss the state of the art of end-host multi-tenant networking architectures through a review of the literature. In Chapter 3, we present Oko, an extensible software switch that serves as a basis to consolidate network services on the host. In Chapter 4, we design our framework to offload tenant workloads from virtual machines and containers to the host datapath. Finally, in Chapter 5, we conclude this thesis with discussions on the increasing heterogeneity of packet processing hardware and the applicability of our work besides cloud networks.

Chapter 2

Multi-Tenant Networking Architectures

In cloud computing infrastructures, the end host and the upstream network devices must execute a number of tasks characteristics of multi-tenant networking. These tasks include demultiplexing traffic to tenants, enforcing security policies, and delivering packets to tenants' domains. In this chapter, we discuss related work on performance challenges inherent of multi-tenant networking. In each section, after providing brief background information, we review efforts to improve the state of the art.

First, we survey work on packet demultiplexing and delivery and discuss its intrinsic performance limitations. We then review advances in software switching and the algorithmic optimizations involved in fast packet classification on commodity hardware. Finally, we investigate recent trends in packet processing offloads as a mean to further improve performance.

2.1 Packet Demultiplexing and Delivery

When delivering packets to tenant's domains in multi-tenant networks, the last few processing steps are often performed at end hosts, on hardware shared with the tenant's workloads. Hence, their efficiency is critical. In this section, we describe these processing steps and survey recent works to improve their efficiency in multi-tenant setups. We focus on the receive path, but the transmit path has similar processing steps, in a reverse order.

2.1.1 Virtualization

To enforce isolation between virtual machines, virtualization platforms must intercept all I/O operations. Networking is no exception: on the receive path, packets are demultiplexed to virtual machines based on their headers; on the transmit path, the hypervisor validates packets to prevent malicious behaviors (e.g., spoofing attacks or floods).

In the Xen virtualization platform [12], a privileged virtual machine, the *host domain*, is responsible for virtualizing I/O accesses. When packets arrive at the Network Interface Card (NIC), an interrupt is first routed to the hypervisor, which notifies the host domain. The NIC then "DMAs" the packet to the host domain's memory. At that point, the host domain can inspect headers to determine the destination guest domain (virtual machine) for that packet.

In the original version of Xen, packets were delivered to virtual machines by exchanging memory pages between the host domain and the guest domain, a technique often referred to as *page flipping* or *page remapping* [12]. A. Menon *et al.* [103] showed that the cost of mapping and unmapping memory pages for the exchange equals the cost of a large packet (1500B) copy. Page flipping was therefore abandoned in subsequent versions of Xen in favor of packet copies.

In [127], J. Santos *et al.* proposed a number of implementation and architectural improvements to Xen's networking path and performed an in-depth analysis of its CPU cost. Two architectural changes had a decisive role in the performance improvements:

- The guest's CPU becomes responsible for copying packets from the host memory to the guest's memory. Before this change the host's CPU was copying packets to the guest domain. Since the guest's CPU is likely to read packets again afterward—if only to copy them to the guest's userspace—, this change improves cache hit rates.

- In Xen, the guest grants the host domain the right to write to a few of its memory pages, in order to receive packets. J. Santos *et al.* removed the need to perform a new grant request per packet; the guest can now recycle pages previously granted to the host domain.

These two design changes stood the test of time and were retained in the more recent paravirtualized virtio driver [125]. In virtio, the host requests a few memory pages to write incoming packets. The guest's driver then copies packets to its own memory when it allocates its internal packet data structure (e.g., `sk_buff` in the Linux kernel).

Network Function Virtualization. With the advent of network function virtualization, researchers focused on improving virtualization performance for network I/O-bound workloads. ClickOS [97] and NetVM [73] are two virtual platforms for network intensive workloads that come with revamped vNICs, software switches, and guest operating system designs.

ClickOS is based on Xen with many of J. Santos *et al.*'s improvements. On the host domain side, ClickOS relies on the VALE software switch [124] with two threads to poll packets from the NIC to the virtual machines and vice versa. On the guest side, they use a version of the MiniOS unikernel tailored for packet processing. In particular, they run a single thread in MiniOS to poll packets from the VALE vNIC. Since unikernel OSes have a single address space, using the MiniOS unikernel removes the need for an additional kernel to userspace copy and significantly boosts performance. To implement network functions, they rely on the Click packet processing framework [107].

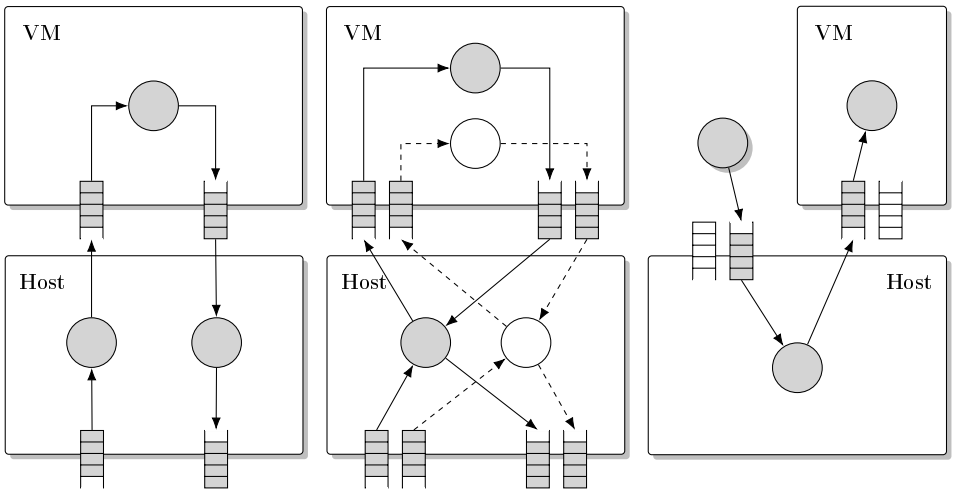
They report a 14.2 Mpps forwarding speed through a virtual machine using 3 dedicated threads, one polling from the NIC on the host, the second polling from the vNIC in the virtual machine, and the last polling from the vNIC on the host and sending packets back through the NIC. The setup used in ClickOS's evaluation is illustrated in Figure 2.1, along with NetVM's.

Published the same year, NetVM took a fairly different approach. NetVM is based on the KVM hypervisor and uses a userspace packet processing library, DPDK [37], to poll packets from the physical and virtual NICs. In addition, where ClickOS uses VALE, NetVM comes with its own demultiplexing logic. More importantly, NetVM has a zero-copy design in which packets are DMAed to hugepages on the host, and virtual machines can read packets from these hugepages without copying them. This zero-copy design comes at the cost of isolation as any virtual machine can access all packets received from the NIC. In the design of NetVM, the authors mention, but do not implement nor evaluate, the possibility to isolate several trust groups.

They report a throughput of 14.88 Mpps with four dedicated threads, two polling from the NIC on the host on the receive path and from the vNICs on the transmit path, two in the virtual machine to poll packets from the vNIC, process them, and send them back to the host. The authors doubled the number of polling threads to dedicate one to each of the two NUMA nodes of their system. Their evaluation is however limited by the 10Gbps NIC. In addition, their evaluation doesn't lend itself easily to comparison with ClickOS: they use one more polling thread and a significantly different CPU[4]. Taking the hardware differences into account, and given that NetVM requires one less copy per packet than ClickOS, it would likely still be able to saturate the 10Gbps NIC with a single NUMA node and two threads.

S. Garzarella *et al.* published ptnetmap [53], a virtualization platform for network intensive workloads similar to NetVM, but using a different packet processing library and software switch. Contrary to NetVM, they evaluated both a fast, zero-copy design without memory isolation between virtual machines and a slower design with a single packet copy to enforce isolation. They performed thorough evaluations with varying packet batch sizes and throughput measurements both from the NIC to virtual machines and between virtual machines. With two polling threads—one in the host, one in the virtual machine—and the same batch size as ClickOS, they achieve a

[4]In particular, the CPU used in ClickOS's evaluation doesn't support DCA [72], meaning that packets are received in the main memory instead of the last-level CPU cache as in NetVM's evaluation.



(a) ClickOS (b) NetVM (c) ptnetmap

Figure 2.1 – Adaptation of an illustration of NetVM's core assignment [73, Figure 2(a)] to ClickOS and ptnetmap. For NetVM, the cores on the second NUMA node are drawn in white with dashed arrows. Whereas ClickOS and NetVM evaluate a round-trip to the VM over the physical network, ptnetmap evaluates only the receive path, using a userspace process to free itself from the limitations of the physical network.

throughput of 14.88 Mpps, limited by the NIC capacity, to receive packets in an isolated virtual machine (one packet copy).

Perhaps of more interest is their evaluation of throughput between a sending userspace process and a receiving virtual machine, as illustrated in Figure 2.1. Because this evaluation is not limited by the NIC's capacity, it better highlights the cost of isolation. When copying packets once to enforce isolation, ptnetmap achieves around 17 Mpps, whereas without packet copies it achieves around 60 Mpps.

2.1.2 Operating System

The challenges of efficiently delivering packets in multi-tenant architectures is not limited to virtual machines. In a similar fashion, operating systems need to demultiplex packets to applications. If packet filters were considered in the 80s [104], nowadays operating systems generally demultiplex packets to userspace processes based on a few static transport layer fields.

In the most common operating systems, incoming packets are processed by a monolithic kernel up to the transport layer, at which point they are copied to their destination userspace process. Kernel processing involves a large number of indirections to handle the many possible protocols, security checks to reject invalid and spoofed packets, and statistic updates [115].

In recent years, several frameworks [37, 123, 22] have been proposed to remove these overheads by receiving packets directly in a userspace process. These frameworks all actively poll packets from the NIC, preallocate memory to avoid dynamic per-packet allocations, and process packets in batches to reduce cache misses. netmap [123], however, contrasts with DPDK [37] and PF_RING [22] by two of its design choices. First, whereas DPDK and PF_RING depart from the POSIX API, netmap retains the Linux kernel for the initialization and uses standard system calls to initiate packet transfers. Second, netmap does not monopolize the CPU core on

which it is running, but instead dynamically adjust the wait time between two polling operations depending on the load.

These frameworks, however, are unfit for multi-tenant architectures on their own. Because they receive packets directly into the process' memory space, they can only support a single memory space.

To overcome this limitation and allow multiple applications to receive packets directly in userspace, the Arrakis OS [115] leverages new capabilities from SmartNICs. In particular, Arrakis delegates security checks and demultiplexing to the NIC. The NIC should support complex predicates to allow for demultiplexing of packets to virtual NICs, which userspace applications control. Intel Flow Director [77], for example, enables demultiplexing of packets to queues based on transport layer fields. These features are nevertheless difficult to support in hardware, especially in the face of packet fragmentation and new, unsupported tunneling protocols.

Instead of SmartNICs, IX [13] relies on hardware virtualization to securely receive packets in userspace processes. Its security relies on an ingenious design in which a non-root ring 0, run-to-completion thread switches to ring 3 before executing the application's logic.

Packet delivery to processes has also been investigated in the context of containers, processes sandboxed by the kernel. OpenNetVM [152] adapts NetVM to containers. Packets are received into a shared memory area that all containers can read, with the same drawback as NetVM: one application can read and modify packets destined to other applications. In contrast, VIRTIO-USER [138] is a new interface, based on virtio [125], to receive packets from a host userspace application (e.g., a userspace software switch) to containers securely; it enforces isolation through one packet copy.

2.1.3 Software Memory Isolation

Figure 2.2 on the following page illustrates the different trade-offs between performance and isolation taken by vanilla Xen [12], NetVM [73], ClickOS [97], and ptnetmap [53]. Whereas ClickOS requires two copies to bring packets to tenants, NetVM, with a single tenant per NIC, requires only one copy per packet. ClickOS's second copy is necessary to bring the packet across the hardware memory boundary, from the demultiplexer's memory space into the destination application's. Although this copy can be avoided by demultiplexing packets at the NIC, most NICs still have very limited demultiplexing logic [67, 55]. They are often unable to parse encapsulated packets and, even without encapsulation, can only demultiplex packets based on a few static fields at the data link and network layers.

Another approach is possible, however: the hardware MMU can be replaced by software memory isolation techniques. These techniques rely on static and dynamic verifications to ensure a program only accesses its own memory. In part because they are implemented in software, they can more easily be adapted to allow specific data (e.g., packets) to cross memory boundaries. These techniques have been extensively studied in the context of kernel extensions [109, 99, 14, 44] and browser plugins [148, 4, 65, 93, 44].

In the following, we review the different approaches for software memory isolation. We briefly describe seminal works and their recent applications. These techniques often enable more than just memory isolation (e.g., type safety or checking arbitrary safety policies), although we focus on memory safety herein.

Binary Rewriting. Several techniques rely on *binary rewriting* to insert bounds checks on memory accesses into binaries before their execution.

Software-based Fault Isolation (SFI), proposed in 1993 [145], gives each module (e.g., kernel

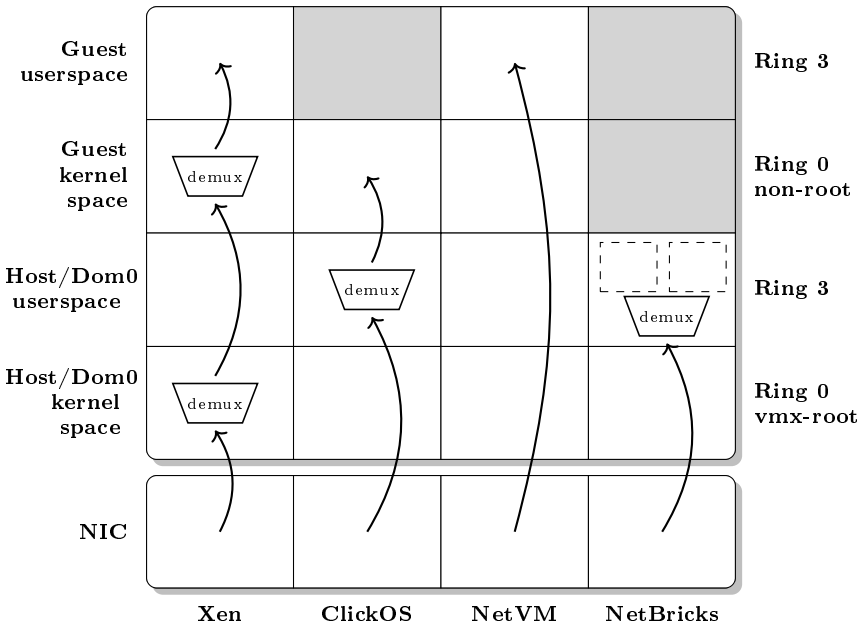


Figure 2.2 – Paths of packets through memory boundaries and demultiplexers in multi-tenant platforms. Dashes delineate software-enforced memory boundaries and arrows indicate packet copies. ClickOS' guest OS, MiniOS, has a single address space. ptnetmap implements both ClickOS and NetVM's approaches.

extensions) a portion of the main address space (e.g., the kernel address space). Because portions of the address space are aligned such that they each have a unique pattern of upper bits, SFI can enforce isolation simply by masking addresses of memory accesses in the object code. Enforcing memory isolation with SFI remains expensive since bit masking instructions are required for every memory accesses. Using CPU benchmarks, the authors measure an overhead of 18–22% to isolate all memory accesses and a much lower overhead (around 4%) to check only jumps and memory writes.

The initial implementation of SFI targeted RISC architectures. S. McCamant *et al.* [98] adapted it to the smaller number of registers and the variable-length instructions of CISC architectures. With Native Client [148], used in the Chrome web browser to sandbox native code, B. Yee *et al.* rely on x86 hardware support for memory segmentation to replace the bounds checks for memory accesses, thus reducing the overhead. Although Native Client leverages the same underlying idea as SFI, it relies on a modified compiler rather than binary rewriting.

XFI [44], proposed by Ú. Erlingsson *et al.*, allows native code to run in the contexts of the Windows kernel and the Internet Explorer web browser. XFI supports multiple memory regions with different access permissions. Its binary rewriter adds bounds checks for memory accesses that cannot be statically verified. Contrary to S. McCamant *et al.*'s work on SFI for CISC architectures, XFI requires registers for runtime checks and must therefore run a register liveness analysis to find available registers. The resulting untrusted binary rewriter is complex, but the authors strove to keep the trusted static verifier simple, including through the use of verification hints inserted by the binary rewriter.

Type-Safe Languages. With type-safe languages, accesses to objects in memory are verified through their type. There exist many type-safe languages; we discuss only a few well-known

examples that pertains to the extension of kernels and web browsers.

In [14], B. N. Bershad *et al.* propose SPIN, an extensible operating system built with the Modula-3 type-safe language. Kernel extensions, written in Modula-3 as well, are compiled by a trusted compiler, representing a large trusted code base compared to the static verifiers of previous approaches.

Conversely to Modula-3 in SPIN, Java does not rely on a trusted compiler. Instead, type safety is verified at the level of the Java bytecode [93], before its execution by the JVM.

Proposed by a consortium of engineers from the four major browser vendors, WebAssembly [65] aims to provide a new low-level bytecode with explicitly typed instructions and operators that can be used as a compilation target for C/C++ programs. WebAssembly programs can use a single dynamic memory area and all memory accesses are checked to be within that area at runtime. Since WebAssembly bytecode is Just-In-Time compiled to binary, if the size of the dynamic memory area changes, the binary must be patched to change the bounds checks.

Closer to the subject of this thesis, NetBricks [114] proposed to replace virtualization with a safe runtime and language to implement and isolate network functions. NetBricks leverages the Rust language and the LLVM runtime to isolate network functions. As illustrated in Figure 2.2 on the preceding page, NetBricks isolates several tenants (network functions) using the same NIC without requiring a second packet copy. Other software memory isolation techniques may achieve the same end result. While it prevents most faults and malicious behaviors, its run-to-completion execution model could allow one network function to monopolize a CPU core.

Proof-Carrying Code. In 1996, G. Necula *et al.* designed a new approach, proof-carrying code (PCC) [109], to enforce memory isolation without requiring runtime checks as in the aforementioned approaches. To implement PCC, the kernel defines a formal safety policy, such as the authorized memory locations for reads and writes. The userspace application then loads a kernel extension in native code with its associate proof of correctness, a proof that the extension abide by the safety policy. Before executing the extension, the kernel derives its safety predicate (a predicate that returns true if the program abides by the policy) and verifies that the proof proves the predicate.

Although G. Necula *et al.*'s approach outperforms previous software memory isolation techniques, it has one main drawback that impedes adoption: the generation of extension's proofs often requires manual intervention. In addition, the authors only test their proof generator with very simple programs, of a few instructions each, used to filter packets.

Static Analysis of Untyped Bytecodes. The last memory safety approach we discuss relies on interpreters, and in that sense, is close to some of the aforementioned runtimes, such as WebAssembly or the JVM. These approaches, however, rely neither on a type-safe bytecode nor on higher-level, type-safe languages.

DTrace [24], for example, provides a safe runtime in the Solaris operating system to trace both user-level and kernel-level software. DTrace probes are written in D, a C-like language, and compiled to DIF, a small RISC instruction set. Memory safety is enforced through both load-time and runtime verifications. Because backward edges are disallowed in the DIF control flow graph, D programs cannot express loops, thereby preventing infinite execution in the context of a probe and simplifying load-time verification.

Like DTrace, the BPF runtime [99] was first designed for a specific application in the context of the BSD kernel: it allowed userspace programs to safely extend the kernel with packet filters to prevent unnecessary packet copies to userspace. BPF first relied on a basic register-based machine abstraction with 2 registers and 22 instructions. Memory accesses were bounded at

runtime and the instruction set simply didn't allow jumps to negative offsets, thereby preventing infinite execution.

In the Linux kernel, the BPF runtime was later rewritten to use a more complete instruction set [95], closer to current hardware instruction set, and more amenable to Just-In-Time compilation. In addition, runtime verifications were replaced by a static analysis, during which the BPF bytecode program is symbolically executed to ensure, among other things, that all memory accesses are correctly bounded. Bounds checks on variable-sized input data (e.g., packets) are the responsibility of the developer and are checked during the static analysis. Accesses to dynamic data structures, implemented outside the BPF virtual machine, are however verified at runtime.

Nowadays, the BPF runtime is used in the Linux kernel to safely probe the kernel as with DTrace in Solaris, but also to safely execute packet processing programs in the context of kernel drivers [70].

In [56], E. Gershuni *et al.* propose a different approach to the verification of BPF bytecode, using Abstract Interpretation [32]. They evaluate the efficiency of several abstract domains (over-approximations of possible values for registers and stack slots) in terms of speed and false positives for a large corpus of networking BPF programs. Contrary to Linux's static analyzer[5], their static analyzer is able to verify BPF programs are safe to execute in the presence of loops, but does not verify termination.

2.2 Software Switches

Besides the bare task of delivering packets to tenant's domains, end host must also *decide* whether and where to forward packets, on the receive path as well as the transmit path. In practice, because of the large number of virtual machines or containers per host and the complexity of network policies, executing the logic to decide whether and where to forward packets can be expensive.

In this section, we survey works on the software switch, the end-host component in charge of executing that logic. We begin with a brief discussion on the evolution over the last decade of the role of the software switch in multi-tenant networks. We then review the literature on *packet classification* algorithms, algorithms to execute the aforementioned logic. Finally, we discuss the challenge of extending software switches, a problem which we address in Chapter 3.

2.2.1 Packet Switching

In the first virtualization platforms [12], networking between virtual machines and the physical network was managed at the link layer (and, less frequently, at the network layer [3]). A software component of the hypervisor would therefore demultiplex packets to virtual machines based on the Ethernet addresses and VLAN tags. This component, generally referred to as the virtual switch, could also enforce policies, such as rate-limiting traffic or filtering outbound packets to prevent spoofing.

Other approaches were proposed to process packets in hardware with higher performance [62]. The NIC [78] or the upstream top-of-rack (ToR) switch can for example enforce policies at much higher speeds than the host's CPU. These approaches, however, are limited by the PCIe bandwidth, and in the case of the ToR switch, require additional tagging of packets and demultiplexing at the hypervisor.

[5] At least until Linux v5.2

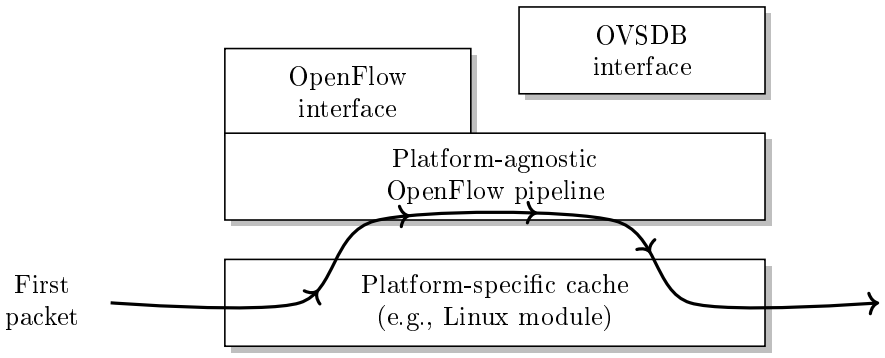


Figure 2.3 – Open vSwitch architecture, with the OpenFlow and OVSDB interfaces.

If demultiplexing was predominantly performed on the host's CPU, virtualization platforms still relied on the mature, reliable implementations of ToR switches for many tasks, including management (e.g., SNMP), accounting, monitoring and mirroring (e.g., NetFlow and SPAN), and quality-of-service [117].

In 2009, B. Pfaff *et al.* presented Open vSwitch [119], a software switch for virtualization platforms with full-fledged management interfaces and fine-grained control over forwarding rules. Its OVSDB interface [35], illustrated in Figure 2.3 enables the configuration of port mirroring, QoS policies, and NetFlow logging. The second interface implements the OpenFlow protocol [51] to give network operators (or an OpenFlow controller) control over the forwarding, filtering, and load-balancing of network flows. The OpenFlow forwarding table is generic in that it can direct packets based on their L2, L3, and L4 headers. Thus, the combination of OpenFlow and OVSDB allows for the implementation of a large number of network tasks at the software switch, in replacement of the ToR switch.

Enforcing network virtualization policies at the hypervisor level benefits virtualization platforms: because it is closer to the virtual machines, it is easier for the software switch to uniquely identify a virtual machine's packets (each virtual NIC identifies a virtual machine) and for the virtualization platform to migrate network configurations with the virtual machine when necessary.

Internally, Open vSwitch has a split architecture that both eases porting to new platforms and improves performance. The implementation of the full forwarding pipeline resides in the userspace, platform-agnostic component, whereas the in-kernel component implements a cache for the most frequently matched forwarding rules. The flow caching mechanisms of Open vSwitch, which we discuss in the next section, were detailed in [120].

2.2.2 Forwarding Pipelines

At a high-level, forwarding rules in both hardware and software switches are organized in a pipeline of match-action tables, each table requiring a lookup over a few header fields to find the appropriate action. Rules may have wildcarded fields (i.e., fields that match all values) or may even match a range of values. This high-level model can have both very limited implementations, with few tables (and rules per tables) matching on a restricted set of fields, or very general implementations, as in Open vSwitch, in which the execution on commodity resources (CPU and RAM) allows for a large number of tables and rules matching on a diverse set of fields.

Hardware forwarding pipelines are generally of the first kind: they support a limited number of tables and fields, often with each table matching on a specific protocol. In an effort to open

networks, the OpenFlow protocol [100] attempted to standardize a general forwarding model. OpenFlow defines a pipeline of cross-layer forwarding tables with priorities. All tables support the same set of matching fields and rules may have wildcarded fields to match all values. OpenFlow however received limited support in hardware switches; most products have low limits on the number of rules per table and, often, each table can match on a very narrow set of fields among the 40 OpenFlow 1.3 defines [51].

Software forwarding pipelines, on the other hand, don't have these limitations. For example, several software switches have near-complete support for OpenFlow, even sometimes the latest, extensive versions [120, 46, 75, 74]. Software switches, however, have other shortcomings: whereas hardware switches can rely on TCAMs and BCAMs to implement fast lookups over flow tables, software switches must find efficient ways to perform the same lookups using the CPU.

Packet Classification. The problem of efficiently matching packets with rules in a forwarding pipeline is known as *packet classification*. Three aspects of forwarding pipelines complicate packet classification in software. First, virtualization platforms have long pipelines [89] for which a naive implementation would require multiple, expensive table lookups. Second, each table may contain a large number of rules and receive frequent updates. Thus, the tables' data structures must enable fast lookups and updates, regardless of the number of entries. Third, whereas hardware switches have TCAMs, CPUs do not have an efficient way (i.e., $O(1)$) to perform lookups over tables with wildcards—tables with only exact-match rules can be implemented with hash tables.

In 1999, V. Srinivasan *et al.* [136] proposed one solution to this problem, namely Tuple Space Search. In a forwarding table with wildcarded rules, each tuple models a set of fields being matched on and is associated to a hash table implementing a lookup over those fields. The lookup over the forwarding table, *i.e.*, the Tuple Space Search, is then implemented as a linear search over each hash table. In their 1999's paper, V. Srinivasan *et al.* discussed an optimization to filter the number of hash table lookups by searching the longest prefix match over all IP addresses first.

Another approach to this problem relies on binary logic [91, 9]. For each packet field in the forwarding table, a set of bit vectors is defined. Each bit vector encodes the forwarding rules matching a given range of values. For example, for a table of four rules matching on the source and destination TCP ports, a bit vector of 0011 associated with source ports 1–1024 signifies that only the two last rules match on source ports 1–1024. Thus, the length of bit vectors is equal to the number of rules in the table. During lookups, for each packet field in the forwarding table, a corresponding bit vector is selected. Bit vectors are ANDed and the remaining 1-bits indicate rules that match the packet for all packet fields in the forwarding table.

Many decision tree-based packet classifiers have been proposed in the literature [137, 131, 143]. These algorithms have varying memory versus throughput trade-offs, but compared to Tuple Space Search, they are generally faster at the cost of memory consumptions polynomial in the number of rules—whereas Tuple Space Search uses memory linear in the number of rules. In addition, even though decision trees can be incrementally updated, they require complex and slower update logic than Tuple Space Search, for which a single hash table operation is needed.

Another approach, referred to as cross-producting [64], consists in doing separate lookups for each field and, using the cross-products of found fields, doing a lookup into the cross-product table of all possible field value combinations. This last cross-product table however grows exponentially with the number of rules.

[6] Complexity when a new match field is added. The complexity to add a new value to an existing match field is slightly lower.

Classifier	Complexity		Memory space
	Lookup	Update	
Tuple Space Search [136]	$O(t)$	$O(1)$	$O(n)$
Decision trees [137]	$O(w)$	-	$O(n \cdot w)$
Binary logic [91]	$O(f \cdot log(2n))$	-	$O(f \cdot n^2)$
Cross-producting [64]	$O(f)$	$\prod_i {v_i}^6$	$\prod_i v_i$

Table 2.1 – Worst-case complexity and memory space consumed by a few packet classifiers. n is the number of rules, f the number of different match fields, t the number of different tuples, w the maximum number of bits matched by rules in the table, and v_i the number of values for match field i. We consider the worst-case only in terms of table content: hash tables without collisions are assumed for Tuple Space Search and Cross-producting.

Table 2.1 summarizes the different lookup and update complexities for the packet classification algorithms presented in this section, as well as their memory space consumption. The binary logic and cross-producting methods have prohibitive memory costs. Two viable methods remain, namely Tuple Space Search classifiers and decision trees. The choice between these two last methods mostly depends on the expected forwarding rule patterns as Tuple Space Search achieves higher performance with a smaller number of different wildcard patterns. As an example, there is an ongoing discussion in the Linux kernel community to choose a general packet classification algorithm for a new Open vSwitch datapath [141], with one side arguing for decision trees to avoid making strong assumptions on rule patterns, the other arguing for a Tuple Space Search classifier with constant-time updates. Similar design choices arise with iptables implementations [16].

Flow Caching. None of these algorithms for packet classification address the first challenge, that of supporting long forwarding pipelines with multiple tables. They all would require one lookup per table in the pipeline. Flow caching has long been studied as a solution to this problem [110, 26, 88], to reduce the average cost of lookups, especially with the use of hardware caches.

Exact-match caches are one simple design for caches. After a packet has been classified using the full forwarding pipeline, a rule is installed in the cache with the action found and all field values equal to those of the packet; thus, subsequent packets with the same field values hit the cache. Nevertheless, if even a single field used in the classifier changes in a subsequent packet, a new full classification is required. This case can happen fairly frequently with short HTTP requests, but also with port scans or even route changes (due to the TTL value changes).

To overcome this issue, second-level caches are used, as an intermediate step between the exact match cache and the full pipeline. In [129], N. Shelly *et al.* study several strategies to compute rules for a second-level cache with support for wildcards but not priorities. Removing support for priorities from the second-level cache simplifies lookups as iterating over all matches to find the highest-priority is not required, but the cache can then only contain disjoint rules. Using Header Space Analysis [86], they first consider the population of a perfect cache, one that completely avoids cache misses. They show that such a cache grows polynomially with the number of rules in the full pipeline and consider an incremental population strategy as an alternative. Thus, for a given incoming packet, they need to compute the cached rule that matches the packet while having as many wildcards as possible, to increase cache hits. They find that the problem is NP-hard and turn to heuristics. They design several heuristic algorithms and evaluate their respective trade-offs in complexity versus optimality (highest number of wildcards to increase

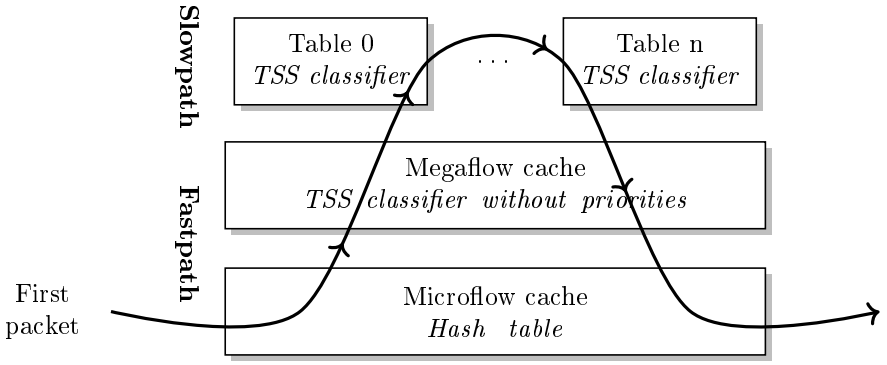


Figure 2.4 – Open vSwitch caching architecture. TSS stands for Tuple Space Search.

cache hits).

Open vSwitch. In [120], B. Pfaff *et al.* present the design of Open vSwitch and its evolution to support diverse virtualization platform workloads. Open vSwitch draws heavily from previous research on packet classification to implement the highly general OpenFlow forwarding pipeline without sacrificing performance. As illustrated in Figure 2.4, Open vSwitch consists of a *slowpath* with the full OpenFlow implementation and a *fastpath* with two cache levels. The full pipeline implementation uses Tuple Space Search classifiers for each table, with added support for priorities by ordering rules in hash tables according to their priority. As an optimization, Open vSwitch records the highest priority in each hash table and skips lookups if a rule of higher priority was already found.

The first-level cache is an exact-match cache, while the second-level cache has a single forwarding pipeline with support for wildcards but not priorities, as in N. Shelly *et al.*'s work. The single table of the second-level cache uses a Tuple Space Search classifier for which rules are constructed reactively, upon packet arrival. Rule construction does not rely on any of N. Sherry *et al.*'s heuristics, but instead tracks the fields used during the full pipeline lookup and un-wildcards them with the packet's values to construct the cache rule. Open vSwitch includes a number of other optimizations [120] to compute keys for hash tables incrementally and improve the optimality of cached rules for IP addresses, for example.

Dataplane Specialization. In [106], L. Molnár *et al.* make a case against flow caching in software switches. They show that, under certain traffic patterns, flow caching can introduce high performance variations. In addition, flow caching requires complex invalidation algorithms to keep caches consistent with rules in the full pipeline[7].

The authors propose ESWITCH, a software switch that specializes the fastpath to attain high performance. They decompose the full pipeline into four prepared code templates: a hash table, a prefix tree, a *direct template* with hardcoded rules, and a default linked list. Their rationale is that forwarding pipelines can often be decomposed into a set of common forwarding tables (e.g., hash table for ACLs or prefix tree for L3 routing).

For simple forwarding pipelines (no more than three consecutive tables) with large numbers of network flows (10K and more), ESWITCH achieves one order of magnitude better performance than Open vSwitch general packet classification algorithm. In these conditions, the efficiency

[7]Open vSwitch revalidates the whole cache for each update in the slowpath.

		VMware's NVP	GCE's Andromeda	Azure's AccelNet
Publication		2014 [89]	2018 [33]	2018 [50]
Packet classifier	Slowpath	TSS classifiers	Not reported[8]	Specialized
	2nd cache	TSS classifier	TSS classifiers	-
	1st cache	Hash table	Hash table	FPGA
Hardware offloads		Protocol offloads	Protocol offloads, packet copies, and encryption	Cache in FPGA

Table 2.2 – Design choices for packet classification in large-scale virtualization platform deployments.

of the cache degrades with the number of network flows, and Open vSwitch is unable to keep up with ESWITCH's specialized classifiers. They however do not compare their from-scratch prototype to Open vSwitch for the long pipelines characteristic of virtualization platforms [89].

VFP. In [48], D. Firestone presents his experience of the iterative design and deployment of VFP, Azure's software switch. VFP takes a somewhat intermediate position between Open vSwitch's aggressive flow caching and ESWITCH's specialization: VFP adopts a specialized slowpath with a single, exact-match cache. D. Firestone justifies the lack of a second-level cache by the lowest cost of VFP's slowpath lookups compared to Open vSwitch's; whereas Open vSwitch's second-level cache runs in userspace and requires a context switch, VFP's runs in kernel space, alongside the exact-match cache.

VFP's slowpath consists of four specialized classifiers: an interval tree, a prefix tree, a hash table, and a default linked list. Instead of decomposing the forwarding pipeline into a set of tables that can be efficiently implemented with the specialized classifiers as in ESWITCH, VFP relies on a heuristic algorithm [49] to map each rule to a classifier. When updating the forwarding pipeline, VFP first adds all new rules to all classifiers in order to compute a score for each couple of rule and classifier. The score takes into account the lookup cost and how the rule impacts the lookup cost of other rules in the same classifier (e.g., due to collisions in a hash table). VFP then assigns each rule to the classifier that gave the best score.

Table 2.2 summarizes the different design choices for packet classification in three large-scale virtualization platforms: VMware's NVP [89], GCE's Andromeda [33], and Azure's AccelNet [50]. Although NVP and Andromeda both rely on Open vSwitch for packet classification, whereas NVP uses an unmodified Open vSwitch, Andromeda uses only its slowpath component as a second-level cache and replaces its fastpath by a custom implementation of a hash table-based, first-level cache. AccelNet, on the other hand, relies heavily on an FPGA-based cache and does not implement a second-level cache. Interestingly, all three platforms leverage flow caching and diverse hardware offloads to achieve high performance. We discuss hardware offloads further in Section 2.3.1.

[8] In Andromeda, the Open vSwitch slowpath acts as a second-level cache for a physically separate software switch whose classifier is not described.

2.2.3 Extensibility

Despite its generality, OpenFlow has several drawbacks that limit its applicability. First, it is protocol-dependent; it matches packets on pre-defined fields that correspond to supported protocols. To support new protocols, the specification and its implementations must be updated and new matching fields added. As a result, the last version of OpenFlow supports over 40 fields. Second, OpenFlow remains limited to its original intents, describing forwarding pipelines in hardware devices. Many researchers have noted the benefits of applying the core principles of OpenFlow (central control and open interfaces) to a larger set of devices (e.g., middleboxes [5, 54, 23]) and use cases (e.g., fine-grained monitoring [149, 96]). The main barrier to this evolution is the stateless nature of OpenFlow, each packet being processed independently with no means to use persistent information to take forwarding decisions.

Protocol Independence. The first limitation is a consequence of OpenFlow initially targeting hardware devices with fixed ASIC-based pipelines whose parser cannot be reconfigured. Recent advances in the design of hardware switches [20, 58] enable operators to reconfigure switches at runtime. With these advances came high-level languages and compilers [21, 83] to ease switch programing. P4 [21], for example, is a popular language in the research community to describe forwarding pipeline. Contrary to OpenFlow's fixed pipeline description, P4 is protocol-independent and can describe pipelines that match packets at arbitrary offsets.

Although all widespread software switches are protocol-dependent, there wasn't any significant barrier for a protocol-independent implementation. Since software switches run on CPUs, they benefit from higher flexibility to match packets. Nevertheless, the recent advances on hardware switch programmability motivated and helped design protocol-independent software switches.

PISCES is one such design of a P4-compatible software switch based on Open vSwitch. The authors focus their efforts on compiler optimizations to retain the high performance of Open vSwitch despite the use of a higher-level programming language.

In [48], D. Firestone reports that VFP is protocol-dependent, in that matching fields are compiled in, but supports programmable actions that abstract away some of the specifics of protocols (e.g., tunneling protocols).

Switching Frameworks. Several software switches were designed to be easily extended. mSwitch [71] is such a software switch. Based on VALE [124], it runs entirely in the kernel, supports a large number of forwarding ports, and replaces the fixed learning switch logic of VALE with a pluggable *switching logic* component. This new component must be written and loaded as a Linux kernel module. The authors develop four different switching logic modules, but none of them requires more than a single lookup.

Several fast packet processing frameworks built atop the Click modular router have been proposed in the literature. With SMP Click [29], B. Chen *et al.* made one of the earliest attempt to improve the performance of Click, by leveraging multiprocessor servers. RouteBricks [36] explores the replacement of hardware routers with software routers based on Click and distributed over a cluster of commodity servers. In [11], T. Barbette *et al.* evaluate several packet I/O frameworks before designing a fast packet processing framework based on Click and running on both netmap and DPDK. ClickNF [52] extends Click with a high-performance TPC stack to enable application-layer processing.

BPFabric [85] is a switch framework built on DPDK with a switching logic expressed in BPF. It includes only two data structures which can be used as packet classifiers: a hash table and an

Destination	Action
10.0.0.1	goto table 2
*	send to port 3

(a) Table 1

Dest.	Action
*:80	send to port 1
*:53	send to port 2

(b) Table 2

Destination	Action
10.0.0.1:80	send to port 1
10.0.0.1:53	send to port 2

(c) Second-level cache

TABLES 2.3 – Simplified OpenFlow pipeline with the aggregated rules in the second-level cache.

array. The authors do not implement pipelines with several tables or even tables with wildcards. Complex packet classification logic is known to be difficult to express in BPF [141], so it is not clear whether BPFabric could support such logic.

BESS [17], formerly known as SoftNIC [67], and VPP [144] are two other switching frameworks built on DPDK. They both come with a number of modules to address specific networking scenarios, such as network address translation, L2 forwarding, or encapsulation.

None of the above switching frameworks, including mSwitch, were designed to handle long, general forwarding pipelines such as OpenFlow, and none come with built-in optimizations to reduce the cost of lookups. One could make the argument that such optimizations (e.g., flow caching) can be added to existing switching frameworks. Whether this is even feasible is not clear however, as such frameworks lack a shared abstraction (e.g., OpenFlow in Open vSwitch) for processing stages. As a result, whereas they can be easily extended to address new packet processing applications, they are unlikely to achieve high performance with the long pipelines characteristic of virtualization platforms [89, 48].

Building Extensible Software Switches. Building an extensible software switch is difficult for the same abstract reason OpenFlow is difficult to reconcile with high performance: generality comes at the expense of performance [120]. In the case of Open vSwitch for example, the OpenFlow table model serves as a common abstraction across all processing stages that enables the aggregation of rules in a single, second-level cached rule. As an example, consider Table 2.3. For a packet destined to `10.0.0.1:80`, the results of the lookups in Tables 1 and 2 can be aggregated into a single rule in the second-level cache because Tables 1 and 2 both implement the OpenFlow table model. Any extension of the forwarding model risks breaking this common abstraction and may prevent the caching optimization [118]. If OpenFlow provides a convenient abstraction, this challenge is not limited to OpenFlow switches, but applies to all software switches that aim for high performance.

Extensible Control Planes. Several researchers attempted to extend the OpenFlow forwarding model. J. Sonchack *et al.* [134] and H. Mekky *et al.* [101] execute extensions in a process separate from the OpenFlow switch. This process acts as a kind of local OpenFlow controller for the switch and manages rules that redirect traffic to its extensions. This approach, however, requires a context switch to the separate process for the extensions to handle any packet, as well as a packet copy to bring the packet in the local controller's memory space.

In a subsequent paper [102], H. Mekky *et al.* improve this design by relocating the local controller and the extensions directly into the slowpath of Open vSwitch. This new design,

Source	Action
10.0.0.1	set source to 10.0.0.2 goto table 2

(a) Table 1

Source	Action
10.0.0.2	send to port 1
*	send to port 2

(b) Table 2

Source	Action
10.0.0.1	set source to 10.0.0.2 send to port 1

(c) Second-level cache

TABLES 2.4 – Simplified OpenFlow pipeline with `set field` action.

named NEWS, still requires a context switch and a packet copy from kernel to userspace for the extensions to process any packet. In effect, going to the slowpath to process a packet is expensive and is the reason Open vSwitch aggressively caches rules in the fastpath. In addition, in NEWS, extensions are dynamically loaded in the userspace process of Open vSwitch, removing the previous isolation between the local controller and the switch. If one extension contains a fault (crashes, leaks memory, or even monopolizes the CPU), it will affect Open vSwitch's slowpath.

Extensible Fastpath. With SoftFlow, E. J. Jackson *et al.* address the performance limitation of previous Open vSwitch extensions. SoftFlow can execute arbitrary programs as actions in the OpenFlow pipeline. Because these programs are executed in the fastpath, SoftFlow preserves the run-to-completion model of Open vSwitch: except for a cache miss, packets are processed by the same thread from reception to transmission, without context switch (neither to the slowpath nor to another process).

Without additional care, SoftFlow's programs break Open vSwitch's common abstraction between tables and prevent the caching optimization. Consider the simplified OpenFlow pipeline of Table 2.4. If a packet with destination `10.0.0.1` is received and processed by the slowpath, an aggregated rule with actions `set source to 10.0.0.2` and `send to port 1` can be cached because subsequent packets with the same destination `10.0.0.1` will inevitably match the same rules and require the same actions. However, if the action in the first forwarding table is replaced with an arbitrary program, it might set the destination header to `10.0.0.2` during its first execution in the slowpath, but there is no guarantee that it will perform the same change during subsequent executions. For example, it might set the destination header to `10.0.0.3`, in which case the packet should be sent out of port 2. Even if two packets with the exact same headers are processed by a SoftFlow action, the result (*i.e.*, the packet changes) may differ because it may also depend on persistent state accessed by the action or even on the packet payload. In essence, because SoftFlow actions are arbitrary programs, short of executing them, the switch cannot know their output. Or, to take E. J. Jackson *et al.*'s words, from the perspective of the flow cache, SoftFlow actions are non-deterministic.

For this reason, by default SoftFlow does a new slowpath lookup after each SoftFlow action. Developers can avoid this additional lookup by setting a metadata flag if the program didn't change the packet headers or at least didn't affect the path through slowpath tables.

Although it improves performance, SoftFlow has the same isolation default as NEWS; a faulty SoftFlow action may crash the switch or exhaust resources. The authors present the lack of isolation as a necessary "sacrifice" to attain high performance. In our Oko work, we show that high performance does not have to come at the cost of isolation.

2.3 Packet Processing Offloads

Packet processing systems often rely on offloads to delegate tasks usually performed in a first component of the system to a faster component, thereby improving performance. In this section, we first discuss hardware offload, in which the NIC or the upstream hardware switch executes computations in place of the CPU, often before handing packets over to the CPU for further processing. Then, while drawing a parallel with hardware offloads, we review *host offloads*, *i.e.*, the offload of guest workloads to the host CPU or to the NIC. While host offloads lag behind compared to hardware offloads, we believe it may represent a promising approach to alleviate packet processing limitations of multi-tenant environments. Host offloading is the subject of our second contribution.

2.3.1 Hardware Offloads

Among hardware offloads, we identify and discuss three generations of offloading features. These evolutions are motivated by the continuous demand for high performance and the emergence of new processing workloads.

Protocol Offloads. The first offloads from the CPU to the NIC focused on specific network and transport-layer protocol computations, often targeting the widespread TCP/IP suite and encapsulations thereof.

In [28], after a discussion of the challenges associated with checksum offloading for TCP/IP, J. S. Chase *et al.* evaluate its performance benefits. Although largely taken for granted today, TCP checksum offloading is not straightforward for NICs to support. Computing TCP checksums involves parsing the IP header, handling the computation of a checksum over several fragments, and ensuring checksums are computed before sending packets on the wire (thereby preventing cut-through switching).

Several researchers and hardware vendors proposed to fully offload TCP/IP processing to the NIC, using TCP Offload Engines (TOEs). Full support for TCP/IP in NIC was never widely adopted and was notably rejected from the Linux kernel [31]. In [105], although J. Mogul argues TOE should be reconsidered with the advent of storage protocols over IP, he makes a great case against TOE by detailing the arguments against. These arguments mostly pertain to the limited performance benefit compared to partial offloads (checksumming and TCP segmentation offload), the complexity of the implementation, and more generally, the inflexibility of hardware implementations.

However, even partial TCP/IP offloads are fairly fragile in the presence of encapsulation. For example, in [89], T. Koponen *et al.* explain that IP encapsulation prevents protocol offloads because the NIC is unable to parse the inner headers. To overcome this limitation, they propose a new encapsulation scheme, STT [34], with a fake TCP header after the outer IP header to enable TCP offloads.

In [68], T. Herbert reviews recent work in the Linux community to improve support for UDP encapsulation schemes. He describes several options to enable checksum offload despite the UDP encapsulation, with and without support from hardware devices, and details the necessary NIC changes to support Large Receive Offload (LRO) and TSO.

Forwarding Pipeline Offloads. The offload of forwarding pipeline workloads, whether they be demultiplexing or filtering, to the NIC were motivated by the emergence of virtualization and the limited performance of software switches (cf. Section 2.2). The first such offload is probably

VMDq [78], in which L2 switching is delegated to the NIC, thereby allowing each virtual machine to be assigned a NIC queue. SR-IOV [79] extends this offload by allowing the NIC to be shared across virtual machines, each virtual machine owning a *virtual function*, a virtual instance of the NIC. Thanks to SR-IOV, packets can be directly copied to the virtual machines, thus bypassing the software switch. Finally, Intel Flow Director [77] enables filtering and demultiplexing in the NIC based on network and transport-layer fields.

In [67], S. Han *et al.* design SoftNIC, a software abstraction to extend the hardware NIC. In particular, SoftNIC abstracts the underlying NIC's limitations and offers a minimum set of features independently of actual hardware support. SoftNIC, however, is an all-or-nothing solution: it completely replaces the hardware feature it emulates. For example, SoftNIC uses the host memory to emulate Flow Director with a much larger number of connections than the NIC could support. In doing so, SoftNIC does not leverage the hardware Flow Director, even when the number of connections is low enough to hold on the NIC.

R. N. Mysore *et al.* propose FasTrak [111], a system to partition forwarding rules between the software switch and the hardware devices. In contrast to SoftNIC, FasTrak is able to leverage the hardware devices and the CPU at the same time. FasTrak setups two paths for each VM: one through the SR-IOV VF and one through vSwitch VIF. The two paths are merged in the VM through a modified bonding driver. Due to the limited support for forwarding rules in the NIC (e.g., limited number of rules and limited number of matching fields), their implementation relies on the Top-of-Rack switch (ToR) to enforce the rules. The NIC to ToR path uses VLAN IDs to preserve the tenant ID information.

Stateful Offloads. Two present-day circumstances drive research works on more general offloads. First, the end of Moore's Law limits the ability to scale workloads using general-purpose CPUs. Second, the availability of programmable hardware devices, whether reconfigurable ASICs [20, 133] or SmartNICs [87, 50], creates an opportunity to offload workloads from the CPU.

In [50], D. Firestone describes how VFP's stateful forwarding rules (cf. Section 2.2.2) are offloaded to FPGAs on the NIC. The FPGA uses the NIC memory to track the state of all incoming TCP connections.

In KV-Direct [94], B. Li *et al.* leverage the same FPGA-equipped NICs to offload RDMA processing in key-value stores (KVS). KV-Direct implements, in the NIC, the basic KVS operations as well as atomic and vector operations and distributes key-value couples between the on-board NIC memory and the main memory. The authors discuss several optimizations to reduce the overhead from the operations to the main memory over the PCIe bus.

2.3.2 Offloading Tenant Workloads

We now discuss host offloads, from guests to the host CPU or to the NIC. We argue that these features are following the path opened by hardware offloads. However, with regard to hardware offloads, host offloads raise new challenges. In particular, the host runtime for offloaded tasks needs to retain the isolation and fairness properties of virtualization platforms.

Protocol Offloads. The same protocol offloads offered to the host have been progressively expanded to guests through their vNIC. In [103], A. Menon *et al.* extend the virtual NIC and Xen's driver domain to support protocol offloads, namely TCP/IP checksumming and TCP segmentation offload. They emulate protocol offloads if the physical NIC doesn't support them, to offer a consistent interface to the guest. In particular, TSO allows guest to transmit packets larger than the MTU in a single transfer, reducing overheads compared to the several transfers

of MTU-sized packets required otherwise.

In [125], R. Russel retained and standardized these offload features for the virtio virtual NIC. TSO and checksum offload are now supported in most paravirtualized drivers, including Amazon's ENA [42], Cisco's ENIC [43], and VMware's vmxnet3 [121], sometimes even inside tunneling protocols.

Filtering Offloads. In Section 2.3.1, we discussed forwarding pipeline offloads as a second family of offloads. These offloads (e.g., demultiplexing) are very specific to multi-tenant systems and are therefore unlikely to extend to guests. Because guest systems tend to be fairly specialized and run a single application, there are few benefits to offer complex demultiplexing logic at the vNIC. In some cases, however, guest systems may be interested to receive a subset of incoming packets. An overloaded application may for example want to rate-limit traffic at the host to avoid unnecessary overhead (interrupt processing) for packets that would end up dropped. Similarly, an IDS virtual network function could leverage host offloading features to apply a first, coarse filter.

In [25], A. Cardigliano *et al.* present the design of vPF_RING, a framework to accelerate packet capture using virtual machines. In vPF_RING, virtual machines can specify packet filters installed in the physical NIC to prevent unnecessary cost for the guest's CPU. The authors describe lawful interception as a specific capture application in need of filtering; lawful interception systems typically receive a copy of all packets on a link, but only need to monitor specific users or applications.

Paravirtualized drivers sometimes support filtering offloads, when the underlying hardware supports it. For example, Cisco's ENIC [43] supports Intel Flow Director to filter packets based on network and transport-layer fields.

Network Service Offloads. Similarly to the hardware offload trend, and to overcome the inevitable packet copy (cf. Section 2.1.3), recent research works proposed offloading entire packet processing tasks from virtual machines or containers to their host.

In [112], Z. Niu *et al.* present NetKernel, an architecture to run the network stack of virtual machines on the host, as a service. With NetKernel, the authors run the network stack in a separate virtual machine and exchange data between the tenant virtual machine and the network stack through shared memory. Each network stack virtual machine has its own virtual NIC, using SR-IOV. By giving control on the network stack to the cloud provider, Z. Niu *et al.* do not aim to improve performance, but rather to improve flexibility for both the tenant, who can choose between diverse network stacks, and the cloud provider, who can tailor the network stacks to its physical network's constraints.

R. Nakamura *et al.* [108] proposed to offload the Linux software stack of Docker containers to the host. They implement a new Linux socket interface, `AF_GRAFT`, which directly connects the container's socket interface to the host's. This approach improves performance by avoiding duplicate processing in the container without giving complete control to the container over the host's network stack. CPU and memory consumptions, however, will be accounted in the host's quotas instead of the container's. The authors do not discuss this point.

As part of his master thesis [122], J.-E. Rediger designed XenBPF, a mechanism to offload network functions from Xen virtual machines to the host's network stack using BPF [95] as a safe runtime. Network functions are written in C and compiled to the BPF bytecode before being attached to the host's traffic classifier. Through a shared memory channel, guests send commands to the host to install BPF programs and update their persistent data structures. J.-E. Rediger implements and evaluate an anti-DDoS function with static rules, a load balancer, and a first-

level cache for memcached. While the memcached cache doesn't improve performance due to the limited memory available to BPF programs, the anti-DDoS and the load balancer do improve performance while consuming a smaller portion of the CPU's time.

XenBPF, however, does not properly isolate CPU consumption of offloaded network functions. To account for CPU consumption outside the guest, J.-E. Rediger estimates the CPU consumption for a single packet. This estimate is computed using a simple ICMP responder, but as we show in our own work on the subject (cf. Chapter 4), CPU consumption can significantly vary between BPF programs, or even for the same program and different packets.

2.3.3 Conclusion

In this chapter, we reviewed the literature on three aspects of end-host networking architectures that are key to the overall performance.

- In Section 2.1, we studied the mechanisms that enable the demultiplexing and delivery of packets to tenant's domains, whether they be virtual machines or processes. We saw that, because of the hardware memory isolation technique on which virtual machines and containers rely, the transfer of packets to tenant's domains constitutes a costly and unavoidable task.

- In Section 2.2, we described the key role of software switches to decide whether and where packets should be forwarded. In particular, we saw that flow caching optimizations are common to reduce the per-packet cost of packet classification algorithms. In addition, we detailed why this same optimization makes it hard to extend software switches to execute arbitrary packet processing tasks.

- In Section 2.3, we discussed hardware offloads and offloads from tenant's domains to the host. We saw that host offloads follow a trend towards more generic offloads, whereby entire packet processing tasks can be offload to the host datapath.

Overall, this chapter provides background to our work and highlights several challenges that we address in this thesis. In Chapter 3, we tackle the challenge of safely and efficiently extending software switches. In Chapter 4, we address the issue mentioned in Section 2.3.2, that is to retain isolation and fairness properties when offloading tenant's tasks to the host.

Chapter 3

Software Switch Extensions

In Chapter 2, we discussed two seemingly unrelated problems: software memory isolation and software switch extensibility. Related works on the first problem investigate low-overhead techniques to enforce memory isolation on CPU, without relying on the hardware MMU, often in an attempt to reduce the cost of memory boundary crossing. The second problem, building extensible software switches, stems from the relentless need to increase the reach of software switches without sacrificing their performance. In this chapter, we contribute a solution to the second problem and use an interpreter and static analysis to isolate the switch from its extensions.

3.1 Introduction

Software switches are taking an increasingly important role as the edge of datacenter networks. Not only do they switch packets between virtual machines (or containers) and physical interfaces, but they also often terminate overlay tunnels and enforce ACLs and QoS policies [89]. Their short development cycle—compared to their hardware counterparts—and their ubiquity in datacenter networks make them an ideal place to implement new network services.

There is, additionally, growing pressure on performance for software switches. Datacenters are moving to 40 Gbps and 100 Gbps physical interfaces, and network IO-bound workloads are becoming more common.

As we saw in Chapter 2, software switches often rely on a hierarchy of caches to achieve higher performance. In particular, OpenFlow offers a convenient forwarding table abstraction to aggregate several processing stages into a single equivalent stage to be cached. Such a common abstraction across processing stages is however difficult to extend without breaking the potential for aggregation.

SoftFlow [80] for example extends the match-action pipeline of Open vSwitch with arbitrary actions. To preserve the aggregation of forwarding rules in Open vSwitch, SoftFlow requires instructions from the actions' developers.

Preserving caching mechanisms is not the only challenge software switch extensions face; they should also isolate the switch from failures in its extensions. Indeed, like hypervisors, software switches are critical components whose failure disrupt any service hosted on the server. Extensible software switches [102, 101, 134] often rely on process isolation to isolate extensions, at the cost of expensive context switches between the switch and its extensions.

In this chapter, we present Oko, an extensible, stateful software switch. Oko executes *filter programs* as part of the OpenFlow pipeline. These programs can access persistent data structures to perform stateful operations such as tracking the state of TCP connections or aggregating traffic measurements. By limiting the scope of filter programs—they have a read-only access to packets,—we are able to maintain the flow caching mechanisms critical to high performance. To isolate the switch from filter programs, we rely on the Berkeley Packet Filter (BPF) interpreter [99], which removes the need for runtime bounds checks thanks to an ahead-of-time static analysis.

In summary, we make the following contributions:

- The design of Oko, an extensible software switch that supports stateful filtering and monitoring programs as part of its OpenFlow pipeline.
- An implementation of Oko[9] based on Open vSwitch-DPDK. Thanks to a careful extension, Oko preserves Open vSwitch's cache design despite its original stateless assumption (Section 3.3.3 and 3.3.3). Because we leverage the high-performance version of Open vSwitch,

[9] Oko is open source and available at `https://github.com/Orange-OpenSource/oko`.

we cannot directly reuse the Linux BPF infrastructure. Instead, our implementation relies on a userspace BPF implementation based on uBPF [92] (Section 3.3.2).

- An evaluation of our implementation and a comparison to state-of-the-art approaches to extend software switches (Section 3.5), with three security and network diagnosis programs (Section 3.4). We show that Oko provides a near 2x improvement of performance compared to a process using shared memory to exchange packets with the switch.

3.2 Design Constraints

Our extensible software switch design must address two main challenges.

1. Prevent switch failures due to faulty extensions. The pipeline may receive frequent updates in response to attacks, or to debug transient network problems. The pipeline thus needs to be updated without interruption of forwarding, as with classic OpenFlow rules. Without proper verification, however, loading unsafe programs at runtime increases the risk of failure. A fault in the new features, whether it may be an invalid operation or a subtler denial of service (e.g., a memory leak or an infinite loop), may cause the switch to crash, thereby disconnecting virtual machines from the network. The switch must therefore provide a mean to prevent these failures.

Related works on extensible software switches rely on processes to enforce isolation, but each invocation of an extension requires a costly context switch, severely limiting performance as we show in Section 3.5.3. In [114], P. Aurojit *et al.* demonstrate that safe languages and runtimes (Rust and LLVM) may provide a viable alternative to virtual machines for isolation of network services. In Oko, we adopt a similar approach, but use a different runtime to remove the need for costly runtime bounds checks [145, 114]. Indeed, by bounding the number of instructions and limiting the expressiveness of filter programs, BPF allows us to statically analyze programs ahead-of-time.

2. Preserve caching mechanisms. Open vSwitch implements several flow caching mechanisms, fundamental to its high performance, in particular for the long forwarding pipelines typical of multi-tenant networks [120, 118, 89]. These mechanisms are built on the assumption that forwarding rules change infrequently compared to the rate of packet arrival. Therefore, stateless rules stay valid for a long enough time to be worth caching. Conversely, with stateful rules, switches may have to reconstruct cache entries after each state change, thus eliminating any caching benefit.

One approach to overcome this challenge, implemented in SoftFlow [80], relies on program developers to tell the switch whether a new slowpath lookup is necessary after the execution of a program. In contrast to SoftFlow, Oko programs act as filters over packets, in the same manner as conventional OpenFlow match fields. Because its programs never modify packets, Oko always caches them without requiring further information from developers. This design choice effectively limits programs to filtering and monitoring applications. In this sense, Oko is complementary to the SoftFlow approach since programs requiring write access to packets could be implemented as actions.

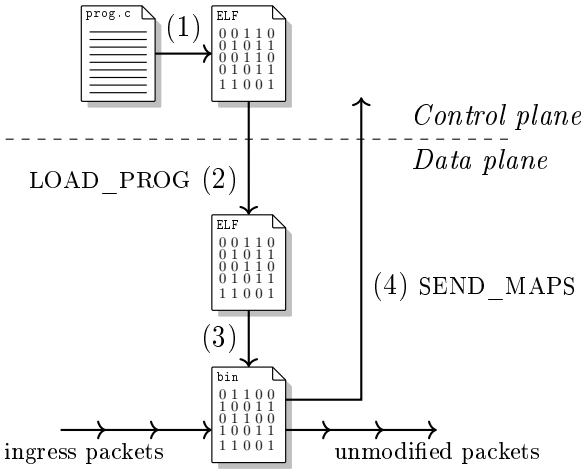


Figure 3.1 – Oko workflow from the compilation of the filter program to its execution in the switch. At step 1, the filter program is compiled to an object file containing the bytecode program, to be sent to the Oko switches at step 2. At step 3, the switch statically verifies that the bytecode program is safe to execute and compiles it. The compiled program then executes on packets according to the flow table configuration. Program may trigger the optional SEND_MAPS action, herein labelled step 4, to send content from internal data structures to the control plane.

3.3 Oko: Design

We first describe the overall workflow of Oko. We then elaborate on our specialized BPF infrastructure and our extension to Open vSwitch's caching mechanisms. We conclude this section with a brief discussion of control plane designs.

3.3.1 Oko Workflow

In this section, we describe our extensions to OpenFlow as well as the Oko workflow, from the compilation of programs at the controller to their execution in switches. Figure 3.1 illustrates said workflow.

Oko extends match-action tables of OpenFlow with an optional match field referencing a *filter program*, a stateful packet-matching program. If all other fields match the packet headers, the filter program is executed, with the packet as its sole argument. Filter programs have a binary result: if they *match*, their corresponding action is executed; if they *do not match*, the lookup continues with rules of lower priority. Table 3.4a on page 35 displays an example forwarding table in Oko.

With some restrictions detailed in Section 3.3.2, each filter program may read and write to its *maps*, persistent data structures allocated on the switch, which the controller can retrieve. Since they are embedded as match fields in flow tables, filter programs only impact whether or not actions of a rule are executed; they cannot define new actions. As we discuss in Section 3.3.3, this restriction allows us to extend flow caching mechanisms without help from developers.

Filter programs are compiled into BPF bytecode (cf. Section 3.3.2), which the switch executes. This compilation step happens once, at the controller. The controller then sends the compiled program to switches in LOAD_PROG OpenFlow messages. Upon reception of the OpenFlow message, Oko switches allocate a *filter program instance*, a memory object that contains both the program and its maps.

Name	Type	Description
LOAD_PROG	Controller to switch message	Contains an ID and a filter program as an object file. The switch loads the filter program and assigns it the given ID.
FLOW_MOD	Controller to switch message	Modified to include a new filter_prog field with a filter program ID. Used to add, remove, or modify OpenFlow rules.
SEND_MAPS	Action	Instructs the switch to send the content of data structures given in argument to the controller. The argument is a bit array where each bit corresponds to a data structure referenced in the filter program attached to the rule.
SEND_MAPS	Switch to controller message	Contains the binary content of data structures.

Table 3.1 – Oko's extensions to the OpenFlow protocol.

Match-action rules may then reference filter program instances. In particular, several rules may share the same filter program instance, allowing different rules to share and update the same maps. A filter program may be instantiated several times to dispose of several different memories. For example, when implementing a stateful firewall, each ACL rule may be associated with its own record of established connections (similar to conntrack zones). In our current implementation, to create a new filter program instance, the controller must send the program to the switch a second time.

We extended the OpenFlow protocol with new messages and fields to load filter programs, retrieve maps, and change OpenFlow rules with filter programs attached. These extensions to the OpenFlow protocol are summarized in Table 3.1. We used OpenFlow vendor extensions for new messages, but had to modify the OpenFlow protocol itself for the new match field, since such vendor extensions are not available. We added support for these extensions in both Open vSwitch and the OpenDaylight SDN controller [113].

The astute reader might notice that Oko does not provide an OpenFlow message for the controller to proactively retrieve the content of maps. To retrieve information collected by filter programs, the controller defines a SEND_MAPS action, which the filter program can then trigger (an example is given in Section 3.4). While developing Oko filter programs, we have not found a need for a *pull* message. Filter programs are often in a better position to decide when to *push* the content of their maps to the controller: they can trigger the SEND_MAPS action when their maps reach a threshold size, or after a specific network event. Nevertheless, if required, the addition of a controller to switch SEND_MAPS message would be trivial to implement.

3.3.2 Safe Execution of Filter Programs

BPF was originally designed as a minimalist instruction set and an in-kernel infrastructure to filter packets destined to a userspace capture application [99]. In the Linux kernel, it evolved into a general-purpose infrastructure [30], also referred to as extended BPF or eBPF. Its current applications include tracing [61], firewalling [19], and container networking [6]. In this section, we (i) provide the necessary background on the security model of BPF and (ii) describe the design and implementation of our userspace BPF infrastructure to execute filter programs.

Background on the BPF security model. In the Linux kernel, the BPF virtual machine consists of a stack and a set of 64-bit registers. The bytecode it recognizes was designed to closely match hardware instruction sets. Bytecode programs can call external functions, implemented outside the VM, to perform operations or retrieve information not available within the VM.

In-kernel BPF interpreters[10] impose limits on the number of instructions and the size of the stack and reject programs with out-of-bounds memory accesses, jumps to non-existing instructions, or null accesses. In addition, in-kernel interpreters expose a BPF machine abstraction with a computational power equivalent to that of a Decider [132], a type of Turing machine that always halts. In the Linux kernel, this computational power is enforced in a strict way by rejecting all jumps to previously visited instructions. This approach prohibits cycles when interpreting the bytecode and, at a higher level, makes long loops difficult to implement[11].

The jump restriction, however, does not apply to external functions; data structures are therefore implemented outside the BPF VM. For example, the Linux BPF infrastructure contains a hash table implementation with linked lists for collision resolution. It is exposed to BPF programs through three external functions: `bpf_map_lookup_elem`, `bpf_map_update_elem`, and `bpf_map_delete_elem` to lookup, update, or delete an element respectively.

Although external functions can iterate through dynamic data structures, they do not increase the computational power of the BPF machine abstraction. Indeed, external data structures have a bounded size, fixed by program developers, to ensure termination and limit memory consumption.

Compared to other bytecodes [93, 114], the BPF bytecode can be verified ahead-of-time: its limited number of instructions and the absence of loops enables a static analysis of all paths through the programs. This feature comes at the cost of expressiveness—iterations on data structures must be implemented outside the BPF VM,—but it removes the need for expensive bounds checks at runtime. As we show in Section 3.4, despite the sacrifice in expressiveness, filter programs can implement diverse network services.

Oko's userspace BPF infrastructure. We retain the BPF bytecode and its abstract machine, but we re-implement a set of external functions and a verifier tailored to Oko.

Our BPF infrastructure relies on a userspace implementation of the BPF VM [92] and extends it to support maps (allocation and external functions). When a Oko switch receives BPF programs from the control plane, it loads them in BPF VMs. To this end, it first allocates memory for maps and relocates addresses in the bytecode. A just-in-time compiler then translates the BPF bytecode into machine code.

Oko supports two data structures outside the BPF VM: a hash table and a simple array. The hash table implementation uses Bob Jenkins' *lookup3* hash function [82] and resolves hash collisions with linked lists.

We implemented five external functions filter programs can call:

- Three functions to lookup, update, and delete elements from data structures.
- `ubpf_time` to read the current time—used to implement traffic policing algorithms and to periodically send measurements to the controller.
- `ubpf_hash` to compute a 32-bit hash of a variable-length input. Together with the array data structure, `ubpf_hash` can be used to implement probabilistic data structures common

[10] Both the Linux kernel and the BSD kernel have a form of BPF in-kernel interpreter.
[11] Short loops can be unrolled during compilation to the bytecode.

Priority	Source	Destination	Action
100	*	10.0.0.1:80	drop
10	*	*:80	port 1
1	*	*	drop

Table 3.2 – OpenFlow table.

in network monitoring applications [149], such as Bloom filters and Count-Min sketches. This external function can also be used for packet sampling.

For the verifier, we implemented a depth-first traversal of the control flow graph (CFG) of BPF programs, to detect and reject programs with cycles (back-edges in the CFG). In a second traversal of the CFG, the verifier tracks the state of registers to determine if they contain, for example, a constant, a pointer to the packet, or a potential null pointer. This information is then used to reject programs with potential invalid operations such as null memory accesses or writes to the packet.

In summary, our verifier currently implements the same safety checks as the Linux[12] verifier, except for pointer leaks and unaligned memory accesses. We do not, however, implement some of the optimizations the Linux verifier performs (such as state pruning). We leave these improvements of the verifier to future works.

3.3.3 Flow Caching

As discussed in Chapter 2, Open vSwitch relies on a hierarchy of caches to achieve high performance without loss of generality [120]. In this section, we (i) give additional background on Open vSwitch's caching mechanism and (ii) elaborate on our design to preserve the performance of Open vSwitch and execute filter programs during the cache lookup.

Open vSwitch's forwarding pipeline. In addition to the userspace slowpath implementing the full match-action pipeline of the OpenFlow forwarding model, Open vSwitch includes a number of datapath implementations for different environments: kernel modules for the default Linux and Windows datapaths and an additional userspace implementation based on DPDK for Linux. Our implementation of Oko uses the DPDK userspace datapath as it achieves the highest performance.

Open vSwitch's caching mechanisms. The DPDK datapath implementation includes two cache levels illustrated in Figure 2.4 on page 18: the megaflow cache and the microflow cache.

The megaflow cache is a simplified implementation of the OpenFlow match-action pipeline: it contains a single OpenFlow table with no priority. The slowpath component can thus only install disjoint rules into this cache. These megaflow rules are built as an aggregation of rules matched during the slowpath lookup. To this end, and in order to build only disjoint megaflow rules, Open vSwitch keeps track of which fields were used during the slowpath lookup. The megaflow rule installed afterward matches only on the fields used; other match fields are wildcarded.

Consider Table 3.2. When a packet destined to `10.0.0.1:80` arrives at the switch, the first rule is identified as the highest priority match during the slowpath lookup. In this case, the slowpath uses the destination IP address and port fields to classify the packet. Therefore, a rule

[12]That is Linux 4.14. The Linux verifier is evolving at a fast pace and more recent versions include support for e.g., function calls inside a single BPF program.

Source	Destination	Action
*	10.0.0.1:80	drop
*	10.0.0.2:80	port 1

Table 3.3 – Example of megaflow cache in Oko after slowpath lookups for two packets destined to `10.0.0.1:80` and `10.0.0.2:80`. In this case, each slowpath lookup resulted in a new megaflow cache rule.

matching only these two fields is installed in the megaflow cache. When a second packet, destined to `10.0.0.2`:80, arrives at the switch, the second rule is identified as the highest priority match. This time, even though the second rule doesn't match on the destination IP address field, this field is used to rule out the first rule as a match. Therefore, a second rule matching the same two fields, albeit on different values, is installed in the cache.

At this time, there is no known algorithm for efficiently building least-specific megaflow rules. Open vSwitch therefore approximates such rules, that is, it builds megaflow rules that may match on more fields than necessary. Less specific megaflow rules match more packets, thereby improving cache hit rate and the overall performance of the switch. Open vSwitch relies on a number of heuristics to build less specific megaflow rules, including heuristics to stop the lookups earlier and a custom algorithm for IP address prefixes [120].

Because even the simplified megaflow table has a high lookup cost, Open vSwitch's second cache, the microflow cache, implements an exact-match table. As any packet header change results in a cache miss, the microflow cache does not perform well with many traffic patterns such as short lived flows, in which case Open vSwitch must fall back to the megaflow cache.

Oko's filter program chains. To maintain the performance of Open vSwitch, filter programs need to be cached and executed in the datapath. The caching mechanisms of Open vSwitch, however, are tailored for its stateless OpenFlow pipeline. Introducing stateful programs in caches raises an important challenge.

The result from the execution of a filter program depends on the full content of the packet and the memory associated with the program. Conversely, in the OpenFlow forwarding model, the actions depend only on the packet's headers. Thus, two packets with the same header will always be processed identically in an OpenFlow pipeline, whereas they may result in different actions taken when processed by Oko.

Open vSwitch relies on this determinism to aggregate a succession of OpenFlow rules into a single megaflow rule. With Oko, the path through the OpenFlow pipeline, and thus the megaflow aggregated rule, depends on the results from executed filter programs, which may change from one execution to the next.

To overcome this challenge, we introduce the concept of *filter program chains* for caches. A filter program chain consists of an ordered list of filter programs, each with an expected result. Each megaflow cache entry contains a filter program chain, assembled during the slowpath lookup as the concatenation of executed filter programs with their result. By construction, the filter program chain preserves the order in which filter programs were executed.

Table 3.5 illustrates a plausible content of the megaflow cache after Oko processed two packets through the slowpath table defined in Tables 3.4a and 3.4b. Program *a* matched the first packet—and the filter program chain *[a:1]* was installed in the cache—but not the second packet. The slowpath lookup continued for the second packet with rules of lower priority. Program *b* matched the second packet and the slowpath lookup continued with the second table. In the second table,

Priority	Source	Destination	Filter prog.	Action
100	*	10.0.0.1:80	a	drop
10	*	*:80	b	table 2
1	*	*	-	drop

(a) Table 1. Packets that do not match `a` will inevitably run against `b`.

Priority	VLAN	Filter prog.	Action
10	100	c	port 1
10	101	d	port 2
1	*	-	drop

(b) Table 2. Packets will never run through both `c` and `d`.

Tables 3.4 – Example pipeline of Oko match-action tables.

VLAN	Source	Destination	Filter program chain	Action
*	*	10.0.0.1:80	[a:1]	drop
100	*	10.0.0.1:80	[a:0, b:1, c:1]	port 1
101	*	10.0.0.2:80	[b:1, d:0]	drop

Table 3.5 – Example of megaflow cache in Oko after slowpath lookups for two packets destined to `10.0.0.1:80` with VLAN `100`, one destined to `10.0.0.2:80` with VLAN `101`. Only one of the packets for `10.0.0.1` matched `a`, the other matched `b`.

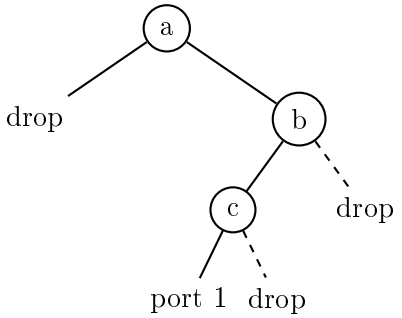
program *c* matched the packet, resulting in the installation of the filter program chain *[a:0, b:1, c:1]* in the cache.

The datapath executes filter program chains by iterating through them and executing each program until it finds one that does not return the expected value. A filter program chain matches a packet if all its filter programs return their expected results. In the caches' classifiers, several rules with the same stateless match fields but different filter program chains are referenced under the same hash. Oko iterates through them until it finds a matching filter program chain. If none of the cached rules match, the traditional behavior of Open vSwitch is preserved and a slowpath lookup is performed.

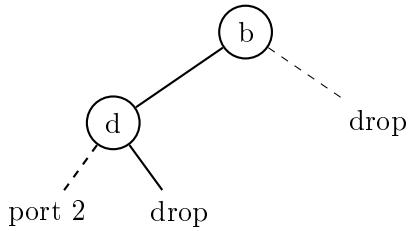
For a given set of packet headers, the possible paths through the OpenFlow pipeline can be viewed as a binary tree whose nodes are filter programs. Each slowpath lookup with this same set of headers results in a new path through the tree being cached. Filter program chains encode these paths. Figures 3.2a and 3.2b on the following page represent the filter program tree for flows `vlan=100,ip_dst=10.0.0.1,port=80` and `vlan=101,ip_dst=10.0.0.2,port=80` respectively, according to Tables 3.4a and 3.4b, with solid lines for the paths already cached (listed in Table 3.5).

Of course, as the trees illustrate, the number of possible filter program chains, and therefore the number of potential cached rules, grows with the number of filter programs. In the worst case, the number of cached rules grows exponentially with the number of filter programs: if n is the number of filter programs, the worst case number of cached rules is 2^n. In Appendix A, example tables and filter program trees illustrate a best-case linear and a worst-case exponential growths of the number of cached rules.

The reasons we don't expect this worst-case scenario to be an issue in practice are twofold. First, we expect neither the worst case scenario to be a common scenario (discussed in Ap-



(a) Tree of filter programs for flow vlan=100,ip_dst=10.0.0.1,port=80. Only branches `a->b->drop` and `a->b->c->drop` have not been cached yet.



(b) Tree of filter programs for flow vlan=101,ip_dst=10.0.0.2,port=80. Only branch `b->d->drop` has been cached.

FIGURES 3.2 – Trees of filter programs from Tables 3.4a and 3.4b. Cached rules from Table 3.5 are represented with solid lines. For each node, the left branch represents the case when the node's filter program returns 1.

pendix A) nor the filter program chains to be much longer than a few programs. Second, Oko, like Open vSwitch, installs rules in the cache reactively, in response to cache-miss events. In addition, for some programs, one result is likely to be more frequent than the other; for example, some programs only *match* a packet when they need to send aggregated statistics to the controller (cf. Section 3.4.3), others *don't match* only in case of error. Therefore, some tree traversals are more likely than others, and some paths through the tree may never be cached.

Read-only access to packets. If filter programs had write access to packets, our filter program chain extension would be impracticable. In this case, the path through the OpenFlow pipeline would also depend on the packet modifications, which would have to be encoded in cache entries as well. Encoding all packet modifications, in addition to the program results, is prohibitively expensive.

In the filter program chain, in addition to the binary result from the program's execution, each program would have a list of packet modifications, including the modified field and the new value. Comparing the results from a program's execution to an item in the list would have a higher cost, due to the additional iteration on packet modifications. Furthermore, enabling write access to packets would also increase the number of cached rules. In essence, in the filter program trees, each node would have an unbounded[13] number of children, resulting in an explosion of the number of paths through the tree, *i.e.* the number of rules.

History of executed filter programs. Filter program chains, however, do not guarantee on their own that a program is not executed twice for the same packet. Two filter program chains may share some filter programs that we do not want executed twice for a same packet. For example, in Table 3.5, if the first cached rule did not match the packet because program a did not, we don't want to run program a again for the second cached rule. To prevent this, a history of executed filter programs is saved along with their results for each processed packet. This history is then checked before executing any filter program.

We investigated several approaches for the implementation of this history. We first used

[13] The number of children would be bounded only by the total number of combinations of packet modifications.

a linked list to store an unlimited number of executed filter programs for each packet. This approach, however, has a high cost when the number of filter programs is small. Our current implementation therefore relies on an array to store executed filter programs, a significantly faster approach with the downside of limiting the total number of filter programs in the switch. We fixed the size of this array to 256 in our implementation. The need to zero out this history when packets are received can add some overhead to Open vSwitch (evaluated in Section 3.5.2 on page 42).

Cache Invalidation. Because cached rules depend on their aggregated OpenFlow rules and on higher priority rules, their invalidation is difficult to compute. Therefore, rather than trying to track which cache entries need to be updated after an OpenFlow update, Open vSwitch adopts a brute-force approach and proactively revalidates the whole cache.

To this end, it runs the set of packet headers that generated each cached rule through the slowpath again. The new rule generated and its actions are compared against the installed cached rule. Depending on the result, cached rules may be updated, removed from the cache, or kept as they are.

Preserving this cache revalidation mechanism is challenging for filter programs because it presumes that a rule can be matched against several times and give the same result. Unfortunately, executing filter programs during cache revalidation would update their internal state and so may change the expected result of a filter program for future incoming packets. To avoid any execution of filter programs during cache revalidation, we use the filter program chain attached to cached rules. During the slowpath lookup for cache revalidation, we compare the identifier of selected filter programs with those from the cached filter program chain. A cached rule is deemed valid only if the same filter programs are selected, in the same order.

3.3.4 Control Plane

We envision two different ways in which applications in a centralized control plane may define and load filter programs in switches. These two approaches may coexist in the same network.

Program Libraries. In a simple scenario, the control plane exposes pre-defined filter programs to be used in applications or directly by network operators. This scenario is well-suited for classical network functions, such as firewalls, load balancers, and performance monitors. It may also be the most appropriate for network operators trying to debug outages and performance bottlenecks. In such situations, they will not have the time to develop their own filter programs and may thus benefit from a library of readily-available programs.

Applications and operators may however need slightly different variations of these network functions. For example, two firewalls may have different expiration duration for TCP connections. To accommodate this need, the filter programs can either be re-compiled for each variation or the bytecode can be patched. While the latter may lead to lower performance, due to the lack of dead code elimination and constant folding for the patched code, the former requires the control plane software to include a compiler toolchain.

Our two last example filter programs in Section 3.4 consider a control plane with a library of programs.

On-the-Fly Generation. Pre-defined filter programs are however ill-suited to applications that require frequent or extensive changes to the programs' logic. Denial-of-service mitigation is one example of such an application. In this case, the entire filter program may be assembled by

a control plane application in reaction to network attacks.

Again, two approaches are possible to construct the filter programs, with the same drawbacks and advantages as before: generation of C code and compilation to bytecode or generation of bytecode directly.

Our first example filter program in Section 3.4 is generated on the fly. We extended the OpenDaylight controller [113] to support our new OpenFlow messages. An application at the controller generates an attacker signature based on sampled TCP packets received from the switch. The signature is converted into C code matching the attacker's TCP packets, which is then compiled to BPF bytecode. If the attackers' signatures were expected to change often, we may have to generate the BPF bytecode directly, to remove the slower compilation step from the process.

Runtime Control. Once the filter programs are loaded in switches, interactions with the control plane have to occur through maps. As detailed in Section 3.3.1, we have implemented a new OpenFlow message (and an associated action) to send the content of maps to the controller. A switch may use this OpenFlow message to send statistics to the controller at regular interval or simply to send notifications (e.g., with arrays containing a single element).

Our OpenFlow extension does not include an OpenFlow message for the controller to update the content of maps, but that is a matter of implementation[14]. Such a message would enable the controller to change a filter program's behavior without reloading it, for example to add items to a whitelist or to enable new options (e.g., with options conditioned on the value of a single-element array).

3.4 Filter Program Examples

In this section, we describe three example use cases for Oko: a stateful firewall, a TCP performance diagnosis system, and a stateless filtering application. Each use case requires a single filter program written in C and compiled to BPF bytecode. We evaluate these filter programs in Section 3.5.

3.4.1 Stateless Signature Filtering

We describe a simple stateless filtering application of Oko, to emphasize its extended matching capabilities. We implemented a TCP SYN denial-of-service mitigation application based on p0f signatures [150]. p0f is a passive fingerprinting tool that can identify the system from which a packet originated based on pre-established signatures. p0f can also generate signatures to discriminate SYN flood packets from legitimate traffic [15]. These signatures encode expected values for several fields in the IP and TCP headers (cf. example signature in Listing 3.1).

Matching p0f signatures against packets requires a few arithmetic and bitwise operations (e.g., subtractions and bit shifts) on TCP options, and as such, could not be expressed with OpenFlow rules, even if new fields were added. Conversely, with Oko, the filter program extends the OpenFlow table and performs the operations required to match p0f signatures against packets.

The first rule in Table 3.6a illustrates the use of p0f signatures to filter packets at the beginning of the pipeline (with the highest priority). The filter program matches and drops packets that

[14]Since this thesis was first submitted, a contributor to our open source implementation of Oko implemented such a message.

Prio.	Source	Dest.	Filter prog.	Actions
12	*	10.0.0.1	anti-ddos	drop
11	*	10.0.0.1	firewall	table 2
10	*	10.0.0.1	-	table 2
10	10.0.0.1	*	firewall	table 2
1	*	*	-	drop

(a) Table 1

Prio.	Source	Dest.	Filter prog.	Actions
10	10.0.0.1	*	Dapper	send_maps, port 1
10	*	10.0.0.1	Dapper	send_maps, port 2
1	10.0.0.1	*	-	port 1
1	*	10.0.0.1	-	port 2

(b) Table 2

Tables 3.6 – Pipeline of Oko tables to illustrate example filter programs.

match its p0f signatures. Listing 3.1 gives an example of filter program matching a p0f signature, with the corresponding BPF bytecode in Appendix B. Without lines 6–8 which check the packet bytes loaded by the program are within the packet range, the verifier would reject the filter program to prevent any potential out-of-bounds loads. Note that we could teach the verifier that packets are always at least 64 bytes long (the minimum length of Ethernet packets) and remove the need for this bounds check. Lines 10–22 check the p0f signature itself.

```
1  uint64_t entry(void *pkt, u64 pkt_len) {
2    struct ether_header *eh = (void *)pkt;
3    struct iphdr *ih = (void *)(eh + 1);
4    struct tcphdr *th = (void *)(ih + 1);
5
6    int len = sizeof(*eh) + sizeof(*ih) + sizeof(*th);
7    if (pkt_len < len)
8      return 0;
9
10   if (ih->ttl > 128 || ih->ttl <= 93 || ih->ihl != 5)
11     return 0;
12   int df = (ntohs(ih->frag_off) & (1 << 14)) >> 14;
13   if (df == 0)
14     return 0;
15   int zero = (ih->tos & (1 << 3)) >> 3;
16   if (zero != 0)
17     return 0;
18   int ip_tot_len = ih->ihl * 4 - th->doff * 4;
19   if (th->window != 8192
20       || ntohs(ih->tot_len) != ip_tot_len)
21     return 0;
22   return 1;
23 }
```

Listing 3.1 – Filter program matching the p0f signature `4:128+0:0:2:8192,8::df,id+,seq-:0`.

Our application relies on the OpenDaylight REST API to control the switch. It first generates a p0f signature from sampled TCP packets (mirrored to the controller). The p0f signature is then compiled to BPF and loaded in the switch to block attacks directly in the dataplane.

3.4.2 Stateful Firewall

As a descriptive example of Oko's workflow, we implemented a stateful firewall using a filter program as the connection tracker and OpenFlow rules for the ACL rules. As illustrated in Table 3.6a, the same program, denoted as *firewall*, tracks connections in both directions in order to protect a web server hosted at `10.0.0.1`.

The stateful firewall allows clients to establish connections with `10.0.0.1`, but only packets from established connections can leave `10.0.0.1`. Packets match either the second or the fourth rule from Table 3.6a, such that the `firewall` filter program can track all connections. The filter program returns 1 only if the packet is part of an established connection or an ongoing handshake. Therefore, packets destined to `10.0.0.1` pass the firewall (and continue to the second table), even if they are not part of an established connection (third rule), whereas packets from `10.0.0.1` are dropped if they are not part of an established connection (fifth rule).

The stateful firewall maintains the state of each connection, identified by their 5-tuple (protocol, source and destination IP addresses and ports), in a hash table. The filter program only needs to implement the TCP state machine and to access the hash table through the three external functions defined in Section 3.3.2.

```
1 struct conn_info info = {0};
2 info.state = SYNSENT;
3 info.server = iphdr->daddr;
4 ubpf_map_update(&sessions, &conn, &info);
```

Listing 3.2 – Snippet of the filter program for the stateful firewall. The snippet shows the update of the TCP state in the hash table upon reception of a TCP SYN packet. `&sessions` points to the hash table. `conn` contains the key for the packet's TCP connection (4-tuple).

```
 1 r1 = SYNSENT
 2 *(u64 *)(r10 - 24) = r1
 3 r1 = *(u32 *)(r7 + 30)
 4 *(u32 *)(r10 - 20) = r1
 5 r2 = r10
 6 r2 += -16
 7 r3 = r10
 8 r3 += -24
 9 r1 = sessions ll
10 call ubpf_map_update
```

Listing 3.3 – BPF bytecode for the filter program in Listing 3.2. `r7` points to the packet and `r10` to the stack.

Listing 3.2 shows the update of the TCP state in the hash table upon reception of a TCP SYN packet, with the corresponding bytecode in Listing 3.3. The BPF calling convention [95] defines registers r1–r5 to store arguments of function calls. In the above bytecode, register `r1` (line 9) contains a placeholder for `sessions` that will be replaced with the address to the allocated data structure when loaded in the switch; registers `r2` and `r3` point to the stack (lines 5–8).

3.4.3 Dapper: TCP Performance Analysis

Dapper [57] is a system to analyze TCP connections in real-time near end hosts. Dapper collects a set of information on each TCP connection (e.g., flight size, MSS, sender's reaction time) at the edge switch. A controller can then retrieve this information to determine the limiting factor of a connection among the sender, the receiver, and the network.

An example integration of Dapper in the OpenFlow pipeline is given in Table 3.6b. Our Dapper implementation analyzes all traffic to and from the server and stores information on TCP connections in a hash table. By default, the Dapper filter program returns false and packets therefore match the last two rules from the OpenFlow table. The program returning true triggers the SEND_MAPS action: statistics on all connections are sent to the controller. This action allows the program to send its collected statistics in batches to the controller when it reaches a given number of connections, after a given amount of time, or when it detects a troubled connection.

3.5 Evaluations

Our evaluations aim to answer the following questions:

- How does Oko compare to other methods to extend software switches?
- How much overhead do Oko's extensions to Open vSwitch introduce? In particular, how does it impact traditional stateless forwarding pipelines?
- How efficient is our filter program chain extension for Open vSwitch's caching mechanisms?

First, we describe our evaluation environment. Next, through a set of microbenchmarks, we measure the overhead and efficiency of our extensions to Open vSwitch. Finally, we compare, for the three use cases described in Section 3.4, the performance of Open vSwitch to that of two alternate solutions to extend software switches: (i) a KVM virtual machine using a vhost-user interface and running a DPDK application, and (ii) a zero-copy DPDK application running as a process.

3.5.1 Evaluation Environment

Our testbed consists of two servers directly connected with Mellanox 40 Gbps NICs. The two NICs use firmware 2.40.700 and are configured with a single queue. The device under test hosting the switch (Oko or vanilla Open vSwitch) has an Intel Xeon E5-2640 2.6 Ghz with 20 MB of L3 cache and 16 GB of DDR4 memory at 2133 MHz and runs Linux 4.4.0. Example filter programs were compiled to BPF bytecode with Clang 3.8.

To avoid Linux's I/O interfaces being the main bottleneck, and to get a clear view of the performance limitations of our own modifications, we based our prototype on the high-performance, userspace datapath of Open vSwitch.

In all experiments, the switch runs on a single core, isolated from the Linux scheduler, with hyper threading disabled. All cores used in experiments are on the same NUMA node, to which the NIC is connected. Unless stated otherwise, we use Open vSwitch 2.5.0, on which Oko is based, with DPDK 2.2 for all comparisons. The switch is configured with a single poll-mode thread and receive checksum offload disabled. The second server runs MoonGen [41] to send minimum-size 64 bytes frames and replay packet captures.

In several of the following evaluations, the packet generator replays a CAIDA packet trace [140] of 34 minutes. The trace contains 87 million packets forming 7 million L4 conversations. On average, L4 conversations last 5.6 seconds and contain 11.7 packets. The maximum number of packets in a single conversation is 11k and the median is 4 packets.

Each experiment lasts 5 minutes and we report the mean and the standard deviation over 10 runs.

3.5.2 Microbenchmarks

We first measure the overhead introduced in Open vSwitch by Oko's extensions through a set of microbenchmarks.

Setup. Because Oko extends only the packet classification algorithms of Open vSwitch and does not impact the OpenFlow actions, we configure the switch to drop received packets after classification. The switch therefore acts as a packet classifier in all comparisons to vanilla Open vSwitch. We perform end-to-end evaluations in Section 3.5.3.

Because of the two-level caching architecture, the performance of Open vSwitch greatly depends on the traffic patterns and the cache hit rate of both its microflow and megaflow caches. Therefore, in all experiments, we provide results under three switch setups: the first setup is the default switch configuration with all caches; in the second setup, the microflow cache is disabled and packets directly hit the megaflow cache with its single OpenFlow table; in the last setup, both caches are disabled and all packets go through the full OpenFlow pipeline in the slowpath.

Scenarios. In the following microbenchmarks, we configure the switch and the MoonGen packet generator according to two scenarios.

In the **synthetic scenario**, a single OpenFlow rule matching all incoming packets is installed while the packet generator sends a single flow of minimum sized (64 bytes) UDP packets. The rule has a *baseline filter program* attached, with a single instruction to return 1, and therefore matches all packets. This scenario is an ideal case for Open vSwitch's performance. Once the OpenFlow rule is cached, Open vSwitch requires few cycles to process subsequent packets, and any overhead added by Oko is therefore accentuated.

Conversely, the **realistic scenario** is designed to stress the switch datapath. It consists of a two-stage pipeline, with an L3 table over 170 IP prefixes and an ACL table consisting of 500 rules over destination ports.

We choose the L3 and ACL tables so as to produce a high number of cached rules at runtime. The L3 table matches the 101 `/8` IP prefixes from the trace, as well as the 69 most-used `/16` IP prefixes, whereas the ACL table matches the 500 most-used destination ports. As shown in Figure 3.3 on the facing page, thanks to this setup, packets are well distributed over all OpenFlow rules. The two ACL rules matching a larger number of packets are the default deny rule and the port 80 rule. After a brief ramp up period, this pipeline results in an average of 170k distinct rules installed in the datapath.

The short duration of conversations and the high distribution of packets over OpenFlow

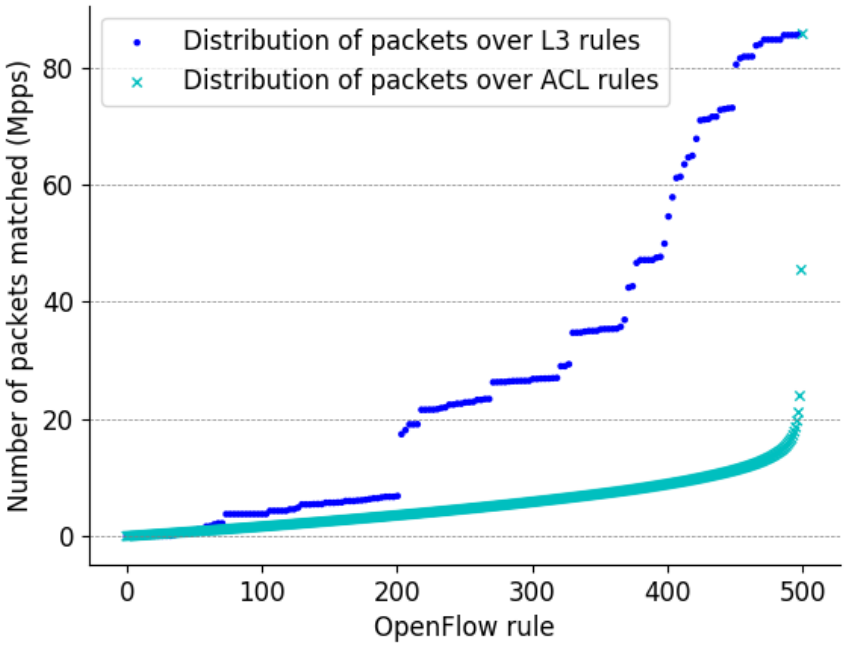


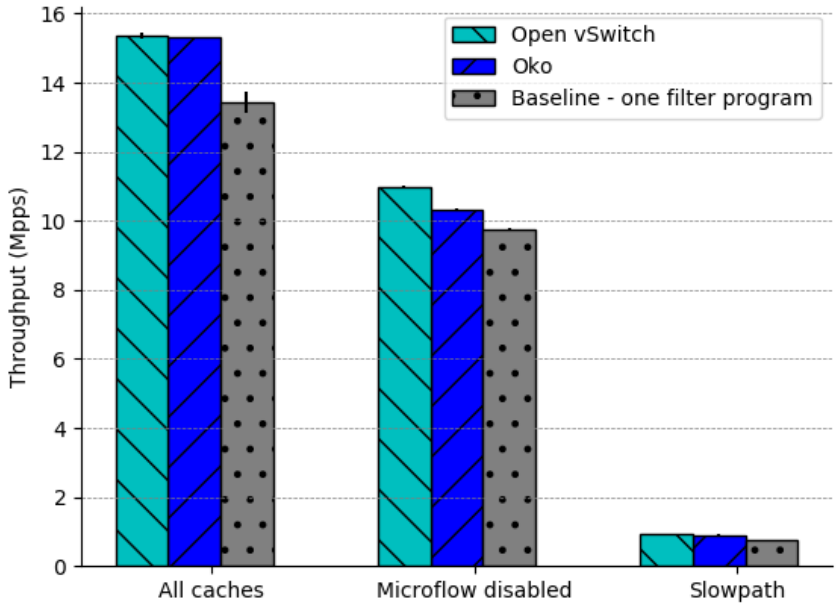
Figure 3.3 – Cumulative distribution of packets over OpenFlow rules.

rules significantly stresses the switch caches. Indeed, as can be observed when comparing Figures 3.4a and 3.4b on the next page (*Open vSwitch* bars), the efficiency of the two datapath caches is greatly reduced in the realistic scenario; with all caches enabled, performance degrades by 5x under the realistic scenario. In particular, because of the short duration of conversations, the microflow cache brings little improvement (3%) over the megaflow cache under the realistic scenario.

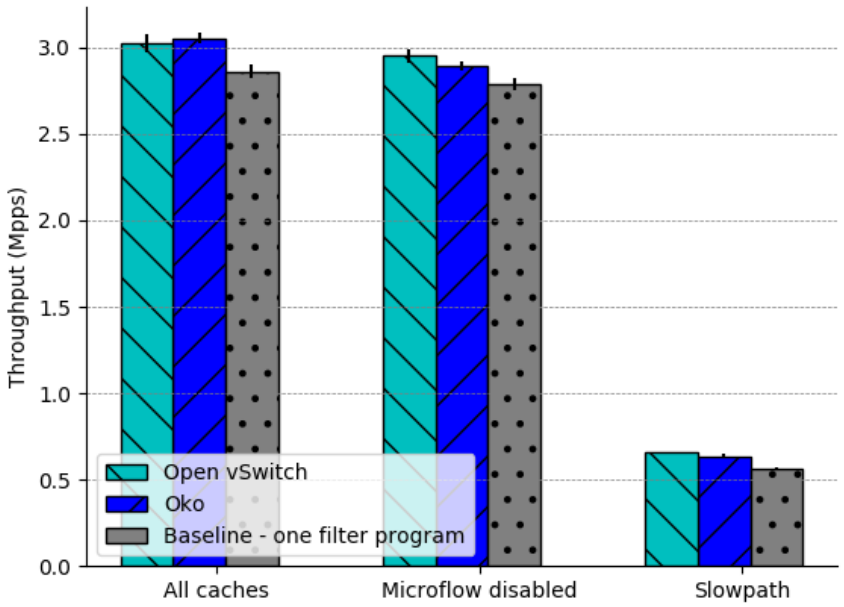
Overhead Evaluation. We next measure the performance of Open vSwitch, Oko without filter programs, and Oko with a single baseline filter program. Figure 3.4a on the following page shows the packet classification performance for each cache level under the synthetic scenario.

Without filter programs in the forwarding pipeline, Oko has a small performance overhead over Open vSwitch (no noticeable overhead with all caches, 6% with the microflow cache disabled) largely due to the initialization of the additional packet field to store the history of executed filter programs. With a baseline filter program, the overhead grows to 12.6% when all cache levels are enabled, and 11.3% when the microflow cache is disabled. This evaluation provides us with an estimated upper bound of the performance overhead. Under the synthetic scenario, because of the single OpenFlow rule, the two caches require very few instructions to classify each packet. Therefore any additional processing (fields initialization, filter program chain iteration, etc.) appears accentuated. Under the realistic scenario, however, Oko adds very little overhead (no noticeable overhead with all caches, 2% with the microflow cache disabled) to Open vSwitch and the overhead added by a baseline filter program decreases to 5.5% with all caches (5.6% with the microflow cache disabled).

Figures 3.4a and 3.4b on the next page also highlight the need for our filter program chain extension to Open vSwitch's caching mechanisms. Without filter program chains, the performance



(a) Synthetic scenario, with a standard deviation below 0.30 for all measurements.



(b) Realistic scenario, with a standard deviation below 0.06 for all measurements.

FIGURES 3.4 – Comparison of packet classification performance between Open vSwitch and Oko.

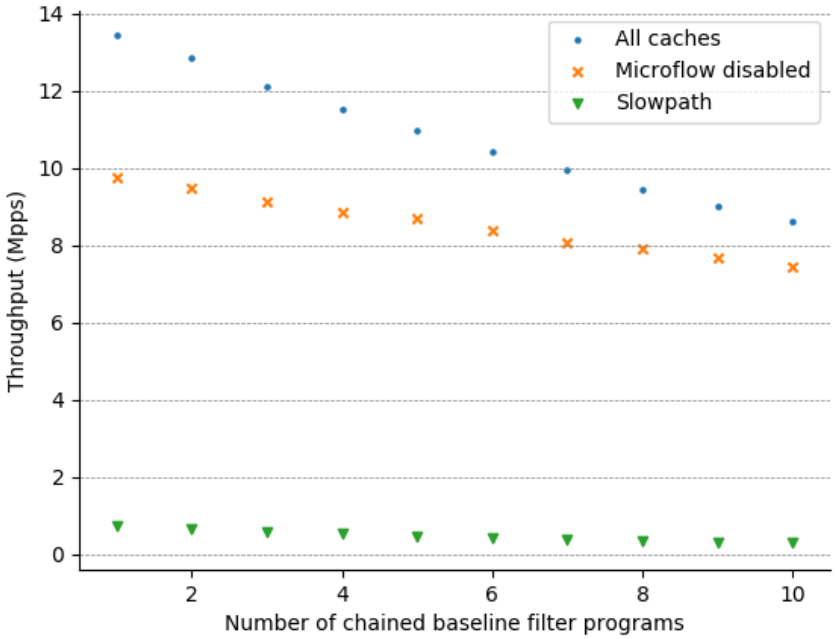


Figure 3.5 – Packet classification performance for different filter program chain lengths, with a standard deviation below 0.30 for all measurements.

of Oko with a filter program would be that of the slowpath since the caches would be stateless and all packets would go through the slowpath. Filter program chains therefore provide an estimated 5x improvement of performance under the realistic scenario (18x for the synthetic scenario).

Filter program chains. OpenFlow rules are installed in the datapath caches after a packet passed through the full OpenFlow pipeline in the slowpath. If filter programs were executed during the slowpath lookup, the cached rule has a non-empty filter program chain attached. The size of the filter program chain is equal to the number of filter programs executed during the slowpath lookup.

To obtain a chain of n filter programs in the datapath, we modify the synthetic scenario and create a pipeline of n tables in the slowpath. Each table contains a single OpenFlow rule with a baseline filter program attached and forwards packets to the next table. Baseline filter programs match all packets. Figure 3.5 shows the packet classification performance for these pipelines. Each new filter program results in about 4% additional overhead. The packet classification performance is almost halved (-43.8%) with 10 filter programs in the pipeline. These numbers are in line with what others have reported for similar chains of programs executed in Open vSwitch's datapath [80].

Filter Programs Evaluation. To provide an estimate of the cost of running filter programs as part of the forwarding pipeline, we integrate and evaluate each filter program example from Section 3.4 under the realistic scenario. The three pipelines for the three filter programs are illustrated in Figure 3.7 on page 47.

To evaluate the **p0f signature filtering** program, an additional table is added at the beginning of the realistic scenario's pipeline. This first table has two rules to (i) drop all packets that

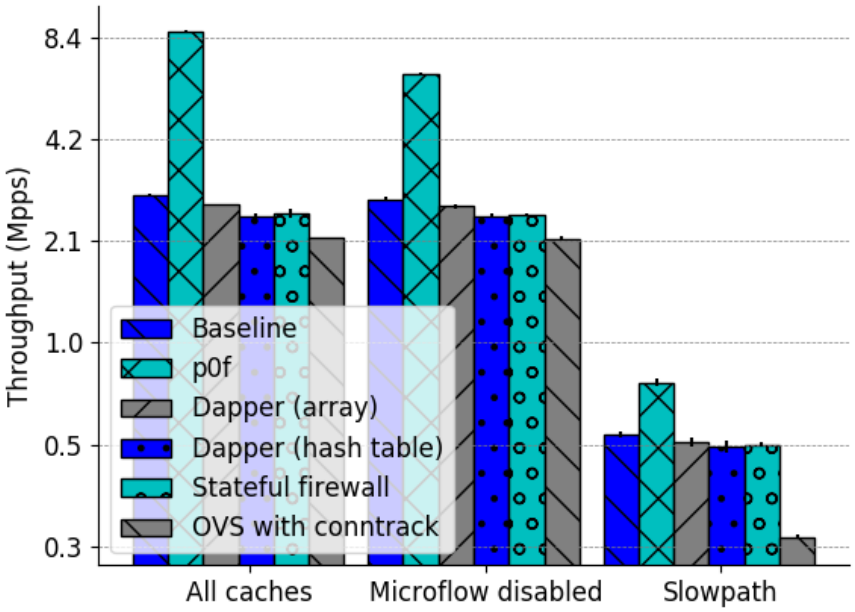


Figure 3.6 – Performance evaluation (on a log scale) for the three Oko use cases, with a standard deviation below 0.08 for all measurements. Performance numbers for p0f are higher because in this case most packets are dropped in the first table of the pipeline (cf. p0f pipeline in Figure 3.7 on the next page) before the more expensive classifications.

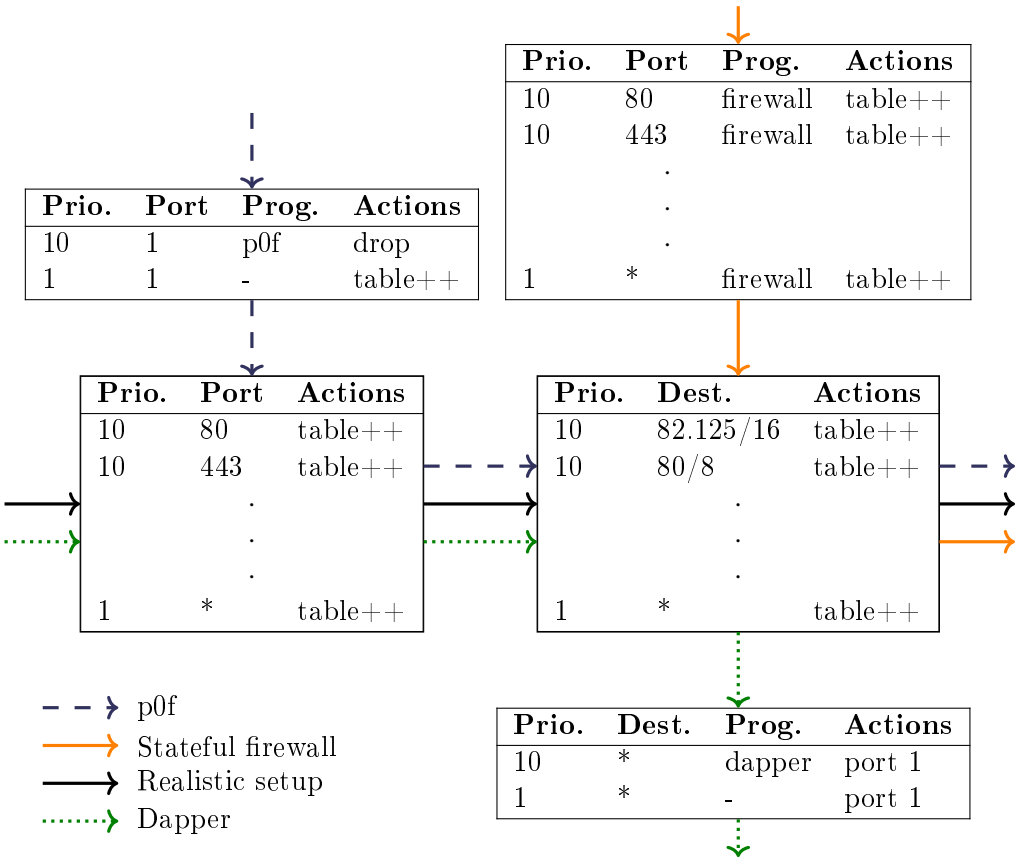


Figure 3.7 – Illustration of the forwarding pipelines for the three example filter programs. Each forwarding pipeline traverses a subset of the tables. The realistic setup's pipeline follows the black path, the p0f pipeline the dashed, blue path, the Dapper pipeline the dotted, green path, and the stateful firewall pipeline the orange path with only two tables.

match the p0f signature, and (ii) forward other packets to the second table. In addition, at the packet generator, we inject both legitimate traffic (from the realistic packet trace) and a flood of TCP SYN packets matching the p0f signature. The TCP SYN flood was generated by the hping3 tool [126], but sent to the device under test with MoonGen like the legitimate traffic[15]. As in previous tests, the packet trace is replayed with a single core; 3 additional cores participate in the TCP SYN flood, resulting in approximately one on four packets being legitimate.

The results in Figure 3.6 at page 46 count all packets successfully classified by Oko. The *Baseline* bar reports the performance of Oko when there are no filter programs in the pipeline, for comparison. With the p0f filter program, Oko achieves better performance than the baseline case because it drops illegitimate packets at the beginning of the pipeline, before the L3 and ACL lookups. As highlighted by this p0f example, Oko can help filter illegitimate or malformed packets at the beginning of the pipeline to avoid unnecessary processing steps.

To evaluate our **Dapper** implementation, tables are added at the end of the realistic pipeline with corresponding filter programs. We use the same packet trace as in the realistic scenario.

The initial implementation of Dapper [57], written in P4 [21] and designed for hardware switches, relies on a fixed-size data structure to store statistics on TCP connections. For this reason, it can miss some connections if the number of TCP connections grows larger than the size of the data structure. With Oko we have the choice to implement a similar array-based version of Dapper or rely on a hash table to store statistics. We implement both options and compare their performance in Figure 3.6 on page 46. Under our test scenario, the hash table implementation has a relatively low cost compared to the array implementation. Oko can therefore benefit from the higher flexibility offered by CPUs to implement algorithms impractical with specialized hardware.

Since its 2.6.1 version, Open vSwitch can act as a **stateful firewall** thanks to a connection tracking module (conntrack) implemented in the DPDK datapath. We compare this implementation against our Oko stateful firewall under the realistic scenario, with additional rules to distinguish established TCP connections from new connections. The Open vSwitch firewall uses Open vSwitch 2.6.1 and DPDK 16.07.

As shown in Figure 3.6 on page 46, the two stateful firewalls perform comparably. Our implementation achieves higher packet classification performance (18% higher with all caches), but contrary to Open vSwitch's connection tracking module, it does not track UDP conversations. With Oko, however, network operators can customize the connection tracking program at runtime to fit a particular application's needs.

In summary, each filter program adds only a small overhead to the baseline pipeline, with the exception of the p0f filtering program that improves performance when the switch is under attack. Notice that Dapper and our stateful firewall implementations achieve very close performance results, despite being completely different applications. An in-depth analysis of the CPU consumption reveals that, in both cases, the hash table lookup dominates the running time with a majority of CPU cycles spent on the 5-tuple hash computation.

3.5.3 End-to-End Comparisons

Setups. For each of the three use case examples, we compare Oko to two existing solutions to extend software switches.

- A DPDK application running inside a KVM virtual machine. The VM is connected to Open vSwitch using the vhost-user virtual NIC. Packets are copied from the switch memory space

[15] MoonGen allows us to send TCP SYN packets faster than hping could.

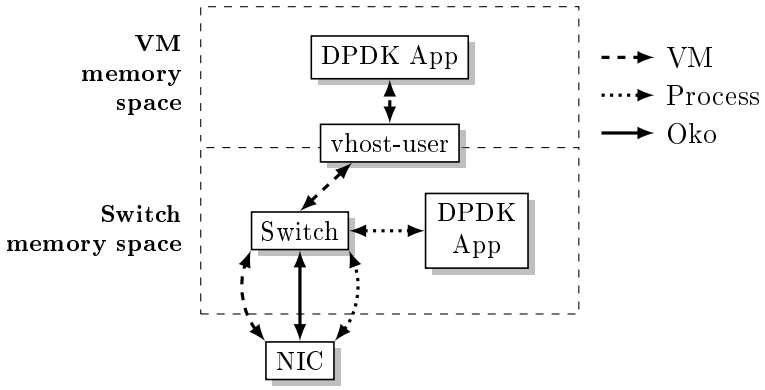


Figure 3.8 – The three evaluation setups for the end-to-end performance comparison. Packet copies are only necessary when crossing memory-space boundaries.

to the VM memory space through a shared memory. vhost-user is the recommended virtual NIC for use with Open vSwitch-DPDK [47] because it achieves the highest performance by opening a direct channel between the host userspace and the guest memory space.

- A DPDK application running as a single process on the host. By using the DPDK Ring Port, the process shares the memory space of Open vSwitch, enabling zero-copy packet processing.

The switch (both Oko and Open vSwitch), the VM, and the process each have their own dedicated core, isolated from the Linux scheduler. In each setup, the switch has only two rules to send packets through the application.

Even though the *VM* and the *Process* setups each have two dedicated cores, they use CPU resources comparable to the Oko setup. In both cases, the core dedicated to the switch does minimal work: it only forwards packets from the NIC to the virtual port and vice versa.

Figure 3.8 illustrates our three test setups. When using Oko, a single core receives packets from the NIC, executes the filter program on them and sends them back to the NIC, as per the run-to-completion model of Open vSwitch. Under the *Process* setup, however, the addition of a second process to run the application breaks the run-to-completion model. Packets are first forwarded by the switch, then processed by the DPDK application, and finally forwarded back to the NIC by the switch. Finally, the *VM* setup entails two additional packet copies to cross the memory boundary: from the switch to the VM and back.

For the purpose of this comparison, we implemented the three use cases from Section 3.4 as DPDK applications. The two filtering applications, the stateful firewall and the p0f application, drop filtered packets in the VM and in the process respectively, without requiring an additional transfer to the switch. The Dapper application only analyzes packets without dropping any.

The packet generator replays the same CAIDA trace as previously. For the evaluation of the p0f application only, we configure 3 additional cores at the packet generator to send a continuous stream of TCP SYN packets.

Results. Figure 3.9 on the next page presents the results of our comparison evaluation. When compared to the *Process* setup, Oko provides a 1.7–1.9x improvement of performance. Thanks to the run-to-completion model of Open vSwitch, Oko benefits from fewer cache misses, consolidated processing steps (e.g., packet classification is performed once), and the lack of IPC.

However, since packets are received in a shared memory—shared between the process and

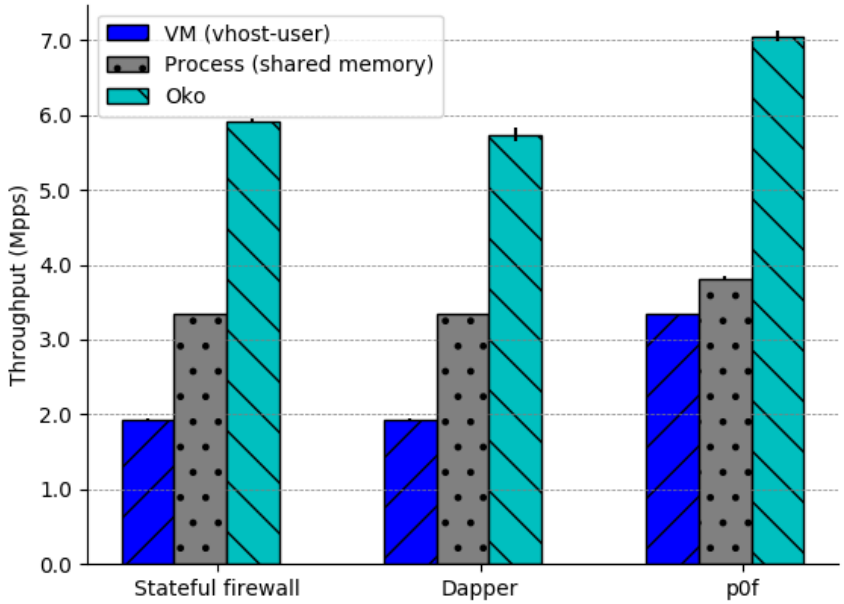


Figure 3.9 – Comparison of performance for the three use cases, with Oko, a vhost-user KVM virtual machine, and a DPDK Ring Port process. The standard deviation is below 0.10 for all measurements.

the switch,—the *Process* setup does not enforce packet isolation [114]: the process may read and write to all packets going through the switch. Conversely, both Oko and the *VM* setup enforce packet isolation. The *VM* setup achieves this isolation at the cost of an additional packet copy across memory boundaries. In contrast, Oko does not leverage the hardware MMU, and instead verifies all memory accesses by the filter programs, ahead of their execution. This approach avoids the need for an additional packet copy, but comes at the expense of flexibility. For example, all accesses to persistent memory from filter programs must go through static implementations of data structures.

As expected, Oko outperforms the VM applications by 2–3x. The difference is less pronounced for the p0f filtering application because illegitimate packets (3 out of 4 packets) are dropped at the VM, requiring one less copy.

3.6 Related Work

In Chapter 2, we presented related work on software switch extensions (Section 2.2.3) and alternatives to BPF for the isolation of packet processing programs (Section 2.1.3). In this section, we focus on emerging usages of BPF in software switches.

In [142], C.-C. Tu *et al.* detail the design of a BPF-based datapath for Open vSwitch, to replace the Linux kernel module datapath. Their work focuses on implementing the two caches in BPF and does not expose BPF capabilities through the switch API.

In [116], J. Pettit *et al.* investigate the use of BPF to offer a common programming interface for the three software switches supported by VMware's virtualization platform, including Open

vSwitch. Their extension to Open vSwitch supports BPF programs as actions in the pipeline, but they do not discuss the integration with Open vSwitch's caching mechanisms.

S. Jouet et al. propose in [84] to use stateless BPF programs as OpenFlow Extensible Match (OXM) fields to build a protocol independent switch. In [85], they propose a packet processing framework based on BPF, upon which a software switch can be developed. In both [84] and [85], the prototypes do not implement any flow caching mechanisms, which eases the design but severely limits performance for long forwarding pipelines.

Finally, it is also possible to place BPF hooks in the Linux kernel network stack (e.g., at the traffic classifier or in the driver). Such programs, however, cannot benefit from the ease and performance of Open vSwitch's forwarding pipeline; they would have to re-implement flow tables and their caching optimizations from scratch. To benefit from both Open vSwitch and the Linux BPF infrastructure, a new hook point and an extended Open vSwitch kernel module would be required. We leave this in-kernel implementation of Oko to our future work.

3.7 Conclusion

Because of their stateless match-action abstractions, software switches do not leverage the flexibility offered by the commodity hardware on which they run. Due to their unique position at the edge of datacenter networks, software switches are critical pieces of software, difficult to extend.

In this chapter, we introduced Oko, a software switch based on Open vSwitch that can be extended at runtime with stateful programs. Oko isolates programs using the BPF interpreter to avoid runtime bounds checks. As our evaluations demonstrate, Oko can preserve the high packet-classification performance of Open vSwitch, while providing a near 2x improvement over existing approaches to extend software switches. Thanks to Oko, network applications can load programs in the dataplane to mitigate attacks, diagnose performance drops, or customize network services at runtime. In the next chapter, we show how tenants can offload network services to the host dataplane, including the Oko software switch.

Chapter 4

Offloads to the Host

In Chapter 2, we saw that virtualization is ill-suited for high-performance packet processing because of the additional costs to bring packets to each tenant's domain. In the previous chapter, we presented a new datapath for network functions, in the form of an extensible software switch. In this chapter, we enable the offload of tenant's network functions to the host's datapath, including our extensible software switch. In particular, we propose to offload security services from virtual machines and containers to the host.

4.1 Introduction

To defend against network threats, cloud applications rely on a diverse set of security services from application-layer rate limiting to TCP SYN cookies and application firewalls. Many of these services are closely tied to the applications they protect; for example, anti-DDoS services often rely on deep knowledge of targeted applications to filter malformed packets. These security services, however, share a common characteristic: they all implement a form of filtering, and as such, they are best executed closer to the wire.

In the past few years, new and faster packet processing frameworks have been developed at the infrastructure layer. Several software switches can now execute arbitrary programs [17, 144, 80], the Linux kernel allows userspace processes to execute programs at the lowest point of the network stack [70], and some SmartNICs can execute offloaded programs on their embedded CPUs [87]. These frameworks have strong execution constraints inherited from their environment or required to achieve higher performance. For example, several of the above cited examples impose a bounded execution time to ensure programs running in the kernel or in the NIC terminate. In addition, they have a run-to-completion execution model: a single thread processes each packet from reception to transmission, often without interruptions (e.g., context switches).

These high-performance packet processing frameworks are already leveraged to offer security services to cloud tenants [48]. Nevertheless, because they are tightly coupled to the services they protect, many security services remain difficult to abstract and expose to tenants.

In this chapter, we discuss and propose a framework to offload security services from cloud applications to the cloud infrastructure. The benefits of filtering packets at the infrastructure layer, before sending them to VMs or containers, have been noted in previous work [25]. We propose to go a step further and enable tenants to offload near-arbitrary programs to the infrastructure, significantly extending the range of candidate programs for offload compared to stateless filters. As concrete examples, in Section 4.5, we describe and evaluate two programs we offloaded using our framework: a SYN cookie mechanism similar to that of the Linux kernel and a DNS rate-limiter.

The main challenge to enable the offload of these programs resides in the high diversity of their resource consumptions. Current security services for tenants (firewalling, stateless filtering, etc.) executed at the infrastructure layer have fairly stable and well-understood resource consumptions. For this reason, fairness can be achieved by limiting the number of processing steps taken for each tenant (e.g., the number of rules for firewalls). With our proposal, however, tenants can offload programs with dissimilar and variable resource consumptions to the infrastructure, while retaining the same performance isolation guarantees as with virtual machines. Without a mechanism to ensure fairness, a tenant with a resource-hungry program could starve other tenants.

To address this issue, state-of-the-art systems often dedicate a core to each tenant. For example, M. Dalton *et al.* [33] report that Google's Andromeda performs CPU-intensive processing in per-VM hardware threads whose CPU time is attributed to the tenant. This approach, however,

wastes CPU resources that could otherwise be offered to clients.

To achieve fairness among tenants independently of their own custom offloaded programs, we monitor the CPU time consumed by each packet's processing. We then ensure each tenant does not consume more than its fair share of CPU time, while retaining the run-to-completion model typical of these execution environments.

We first describe the design of our framework, including the algorithm for our CPU fairness mechanism, in Section 4.3. In Sections 4.4 and 4.5 respectively, we detail our in-kernel and userspace implementations and two example security services. Section 4.6 presents our initial evaluation results and shows that our framework provides a 4–6x performance improvement for containerized applications while enforcing fairness with a 14% worst-case overhead. Finally, we discuss related works and conclude this chapter in Sections 4.7 and 4.8 respectively.

4.2 Background on High-Performance Datapaths

This section provides some background on common high-performance datapaths and presents their shared characteristics, with a focus on their run-to-completion execution model.

The last decade has seen the emergence of several frameworks to accelerate packet processing on CPUs, including the netmap framework [123], the userspace DPDK library [37], and the recent XDP hook in Linux drivers [70]. These frameworks may also be indirectly available on virtual machine hosts, either through extensible software switches built on DPDK [80, 27, 17, 144, 71] or through the NIC driver in the case of XDP.

Applications built on top of these frameworks usually have performance goals in the millions of packets per second per core, that is a per-packet budget of only a few hundreds CPU cycles. To meet this constraint, the frameworks share several key characteristics:

- They specialize the network stack: common packet processing tasks and data structures are defined in libraries and only the strict necessity is compiled in the application. The application retrieves a raw packet and performs any additional parsing itself, to avoid the many indirections of a general-purpose network stack.

- They avoid memory allocations: since allocating and deallocating an object on the heap can cost several hundreds of CPU cycles, it is critical to avoid any memory allocation while processing packets. These frameworks rely on preallocation of packet buffers, although in the case of XDP, the actual implementation depends on the driver used.

- They avoid context switches: context switches are costly due to the need to save and restore the processes' states and, in some cases, to switch address spaces. In particular, to avoid context switches, high-performance datapaths follow a run-to-completion execution model, in which each packet is processed by a single thread, on a single core, from its reception to its transmission.

4.3 Design

In this section, we first describe the different steps executed when tenants decide to offload a security service to the underlying infrastructure. We then elaborate on program isolation and our mechanism to ensure CPU fairness among tenants while retaining the high-performance run-to-completion model. Finally, we briefly describe the per-packet tracing of CPU consumption.

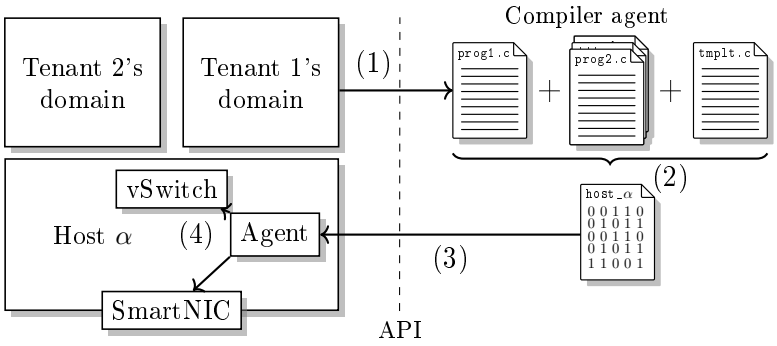


Figure 4.1 – Offload workflow from the request to the API to the execution on the host or NIC.

4.3.1 Offload Workflow

As illustrated in Figure 4.1, from the implementation of security programs to their actual execution at the infrastructure layer, a number of steps must be executed to ensure a fair and secure offload.

As a first step to any offload, cloud tenants must port their security programs to the infrastructure environment. In our prototype we only support C programs, but support for other languages is possible with the appropriate compiler. Porting programs to the infrastructure environment usually involves changing a few API calls, such as the initial reception of the packet pointer, and replacing data structure to use those exposed by the infrastructure environment.

Once programs are written, tenants can request (step 1 in Figure 4.1) an offload to the infrastructure through their usual cloud API. They send the source code of the program to offload and the list of containers or virtual machines they want to protect.

New tenant programs are sent to the *Compiler agent*. For each host, the Compiler agent relies on a template to combine all tenant programs into a single *infrastructure program*. The template, whose outline is given in Algorithm 1, also contains (i) a demultiplexer (line 1) to execute the code from the appropriate tenant when a packet arrives on the system and (ii) a few operations (lines 2–7) to enforce fairness among tenants. Since programs are attached to a tenant and not to specific applications, the demultiplexer only needs to select the appropriate tenant, not the exact container or virtual machine; for example, demultiplexing can rely on the VLAN identifier or the GRE key. This infrastructure program is then compiled (step 2) into a bytecode the infrastructure environment can understand (in our case, BPF bytecode to install in the kernel or in the NIC) and sent (step 3) to all hosts that require an update, *i.e.*, all hosts where the new tenant program should be executed.

On each host, an agent receives the infrastructure program and replaces (step 4) the currently running program with the new one. We expect the replacement of a program to be an atomic operation, as is the case in the Linux kernel.

Our fairness mechanism relies on an assignment of tokens to tenants. Tenants consume tokens as they process packets and the host agent produces new tokens at regular intervals. The *CPU share* of each tenant on the host (or in the embedded CPU of the NIC) is a direct function of the number of tokens produced by the host agent for that tenant. By default, all tenants get the same CPU fair share, but the fairness mechanism allows for different CPU shares among tenants.

Algorithm 1 Template algorithm

▷ Each tenant i has a bucket b_i. The following algorithm is executed upon reception of a packet p.

```
1: i ← demultiplex(p)
2: if b_i > 0 then
3:     t_1 ← get_current_time()
4:     execute(i, p)
5: else
6:     drop(p)
7: end
```

▷ The bucket is decremented with the number of tokens consumed, using t_1, once the packet is fully processed, as described in Section 4.3.4.

4.3.2 Program Safety

When run on the host, an offloaded program should be properly isolated, to protect the host and the other offloaded programs. A program should only be able to access its own restricted memory region and packets destined to its tenant. The offload system should also prevent faults that could lead to a denial of service, such as a program that never halts or unaligned memory accesses on some architectures [38].

Of course, both hypervisors and operating system kernels offer these safety properties to the programs they execute, but because they rely on the hardware MMU for memory isolation, they would require costly context switches between processes (respectively virtual machines) and break the run-to-completion execution model.

As discussed in Section 2.1.3, several techniques exist to enforce memory isolation in software. Among these techniques, we choose to rely on the BPF virtual machine for several reasons. First, the BPF instruction set and computational power are well-suited for packet processing, as it was the initial application [99]. Second, its limited computational power (BPF programs have a bounded execution time) enables the ahead-of-time verification of programs through static analysis. The lack of runtime verifications is likely to make BPF faster than its alternatives[16].

Because we run all offloaded programs for a host as a single BPF program (using our template), we cannot rely on the BPF virtual machine to enforce isolation between offloaded programs. Fortunately, we only need to care for accesses to persistent data structures and packets: the only other memory region BPF programs can access is a bounded, per-packet stack.

The demultiplexing logic of the template ensures each offloaded program can only access packets destined to its tenant. Note that the template only need to perform a partial demultiplexing of packets. Indeed, contrary to the software switch, the template only needs to select the appropriate program to execute, not the specific virtual machine. For example, if a tenant wants to protect all of its virtual machines with the same security service, the template may demultiplex packets based only on the tenant's ID (e.g., the VLAN ID).

Offloaded programs refer to persistent data structure via a developer-chosen identifier. To ensure one program cannot access another program's data structure, the Compiler agent prepends all identifiers with a per-tenant identifier, before compilation.

[16] Compared to Software Fault Isolation [145] which runs native code, the bytecode used by BPF may introduce some inefficiencies, but they are unlikely to be as costly as runtime verifications.

4.3.3 Run-to-Completion CPU Fairness

Many high-performance execution environments have a run-to-completion model in which packets are processed from reception to transmission (or end of processing) by a single thread, with as few interruptions as possible. Retaining this model with several programs competing for CPU resources calls for a non-preemptive CPU scheduler as, with a preemptive scheduling policy, programs (or, to be exact, their processes) would be regularly interrupted by the scheduler.

Fine-grained CPU allocation. The usual approach to allocate CPU resources to multiple programs in a run-to-completion environment consists in giving each program its own core [114, 13]. This approach has several drawbacks.

First, it requires a prior demultiplexing step, at the hardware NIC or in a dedicated core, to assign packets to the appropriate core. However, not all NICs support programmable demultiplexing policies to cores and, even when they do, they are often limited to a few specific fields (e.g., L4 ports or MAC addresses) [67, 55]. The alternative, dedicating a core to the demultiplexing, consumes additional CPU resources and may become a bottleneck.

Second, this approach doesn't allow for fine-grained allocation of CPU resources. Contrary to preemptive scheduling policies, it may not, for example, assign half a core to a first application and the remaining cores to a second application.

Token consumption. As described in Section 4.3.1, for each host, all tenants' programs are incorporated into a single *infrastructure program*. We therefore execute all programs as part of the same process and apply our fairness mechanism on each tenant program. Programs consume tokens in proportion to the time they spend on the CPU.

Fairness is enforced indirectly: instead of limiting the CPU time for each program—which would require a preemptive scheduler,—we limit the packet rate depending on the current CPU consumption (number of tokens remaining).

The fairness mechanism is part of the infrastructure program; it runs after the demultiplexer and before the tenants' programs. It rejects the packet if its tenant's bucket contains a null or negative number of tokens (we explain the negative case below). If the bucket contains a strictly positive number of tokens, the tenant's program is executed.

Once the CPU consumption for the packet has been fully accounted for (see Section 4.3.4 below), an equivalent number of tokens is removed from the bucket. At this point, the bucket may contain a negative number of tokens since we don't know before executing the program if there are enough tokens to process the packet.

Token generation. Whereas the rate-limiting and consumption of tokens is straightforward, the generation of tokens requires more care. Our token generation algorithm, summarized in Algorithm 2, allows for different generation rates among tenants (i.e., different CPU shares). In addition, it implements a work-conserving allocation of CPU time: a tenant that already consumed its fair share may use the fair share of idle tenants, if any.

The generation algorithm runs at Δt intervals, in two steps. In the first step, each tenant receives a number of tokens proportional to their fair share (lines 1 and 5), expressed as a token generation speed, v_i. Each tenant's bucket may contain a maximum of B tokens and surplus tokens are kept for the second step (line 4).

The second step runs as many times as necessary to distribute the surplus tokens fairly among tenants. At each iteration, surplus tokens are distributed among tenants who do not yet have full buckets, as a ratio of their speed compared to other tenants with non-full buckets (lines 11 and

Algorithm 2 Token generation algorithm

▷ Each tenant i has a bucket b_i and a token generation speed v_i. Tokens are generated at Δt intervals. Buckets contain a maximum of B tokens.

▷ Distribute tokens to buckets and count surplus tokens:

1: $r_i \leftarrow v_i \cdot \Delta t$
2: $r' \leftarrow 0$
3: **for all** $i \in I$ **do**
4: $\quad r' \leftarrow r' + r_i - min(B - b_i, r_i)$
5: $\quad b_i \leftarrow min(B, b_i + r_i)$
6: **end**

▷ Distribute surplus tokens fairly until none are left or all buckets are full:

7: **while** $r' > 0$ **and** $\exists i \in I : b_i < B$ **do**
8: $\quad J \leftarrow \{i \in S : b_i < B\}$
9: $\quad r'' \leftarrow 0$
10: $\quad$ **for all** $j \in J$ **do**
11: $\quad\quad r_j \leftarrow r' \cdot \frac{v_j}{\sum_{i \in J} v_i}$
12: $\quad\quad r'' \leftarrow r'' + r_i - min(B - b_i, r_i)$
13: $\quad\quad b_j \leftarrow min(B, b_j + r_j)$
14: $\quad$ **end**
15: $\quad r' \leftarrow r''$
16: **end**

13). This second step stops when there are no more surplus tokens or when all tenants have full buckets. This step makes the allocation of CPU time work-conserving, as it ensures that tokens will not be wasted as long as there are busy tenants.

4.3.4 Per-Packet Tracing of CPU Shares

To properly apply our fairness mechanism, we need to monitor all CPU consumption on the infrastructure, including for tenants that didn't offload security services.

As a toy example, consider a server with only two tenants, the first with an offloaded program that consumes few CPU cycles per packet, the second without any offloaded program. If the first tenant receives and drops a large number of illegitimate packets, she will consume less CPU cycles per packet, on average, than the second tenant. However, if we only accounted for and limited the CPU consumption of tenants with offloaded programs, the second tenant would always receive the same—or a larger—number of packets than the first tenant, who may be rate-limited by our fairness mechanism. The second tenant would therefore receive a larger, undue share of the CPU time.

We therefore need to trace, for each packet and for all tenants, the CPU time consumed on the host, *i.e.*, all CPU time that is not already allocated through traditional preemptive schedulers (be it in operating systems or in hypervisors).

After an offloaded program has been run on a packet, there are three possible actions: the packet may either be dropped, sent up the network stack to the tenant's domain, or retransmitted back on the interface (as is the case with the TCP SYN proxy for example). We therefore need to account for (i) the CPU time spent to execute the offloaded program and (ii) the CPU time spent for any subsequent action. We install probes to monitor the CPU time spent for the three

External function	Auth.	Description
perf_event_output	No	Send data to userspace through perf buffer
get_smp_processor_id	No	Get the SMP processor id
map_lookup_elem	Yes	Lookup an entry in a data structure
map_update_elem	Yes	Update an entry in a data structure
map_delete_elem	Yes	Delete an entry in a data structure
get_prandom_u32	Yes	Get a pseudo-random number
tail_call	No	Jump to another BPF program
ktime_get_ns	Yes	Get the time elapsed since system boot
trace_printk	No	Print message to debug file

Table 4.1 – External functions available to BPF programs at the XDP hook in Linux 4.9. The second column indicates functions that are authorized in our prototype.

possible actions. We detail their implementation in Section 4.4. In each case, once the packet is fully processed, we remove the CPU time spent on that packet from its tenant's bucket.

4.4 Implementation

As described in this section, we implemented our offload mechanism over two different high-performance host datapaths. The first implementation offload programs to eXpress Data Path (XDP), the recent Linux high-performance datapath for Linux [70], whereas the second runs in userspace, using the DPDK library.

4.4.1 In-Driver Datapath

eXpress Data Path prototype. XDP allows userspace programs to load BPF bytecode programs in the kernel, to be executed at the reception of packets in the driver, right after packets are DMAed from the NIC to the main memory.

In our prototype, the template BPF program includes the definition of BPF array maps for buckets and parses packets to select the appropriate tenant program to execute based on the VLAN id only. The template program has special placeholders that are replaced with the tenant programs by the Compiler agent. Buckets are updated by a separate process, running in userspace.

The XDP hook in the Linux kernel currently requires administrative privileges to load BPF programs. Since in our case, any tenant may offload program, we had to modify the Linux kernel BPF verifier to further restrict the set of external functions programs can call from inside the BPF VM. For example, our modified BPF verifier forbids jumps to other BPF programs and debugging functions. The list of functions, both authorized and forbidden, is shown in Table 4.1.

CPU consumption tracing. We also use BPF programs attached to kprobes, dynamic probes in the Linux kernel, to trace the CPU consumption at the infrastructure for each packet. Using BPF for tracing allows us to share metadata on packets (e.g., start time of processing and tenant identifier) directly with the XDP program. In this way, we measure the CPU time spent from the reception of the packet, with a first timestamp retrieved in the template (see Algorithm 1), to the last instruction of the network stack, with a second timestamp retrieved using kprobes.

To measure the CPU time for retransmitted packets, dropped packets, and packets sent to the tenant's domain, we trace two functions in the network device driver. These two functions are specific to the driver we use, but we expect finding equivalent functions for other drivers to be straightforward; these functions only need to mark the point in time when the packet is retransmitted (respectively, sent to the tenant's domain or dropped).

In the Linux kernel's mlx4 driver we use, we trace the return points of functions `mlx4_en_xmit_frame` and `mlx4_en_free_frag`. The first function is called when packets are retransmitted, whereas the second is called after packets are sent to the tenant's domain or when packets are dropped.

4.4.2 Userspace Datapath

Our second implementation offloads programs to a DPDK-based, userspace datapath on the host. We use it in Section 4.6 to compare our fairness mechanism with the Linux CPU scheduler.

To that end, we rely on our userspace implementation of the BPF virtual machine (cf. Section 3.3.2) to run our template BPF program, including the fairness mechanism and the demultiplexing logic. A native implementation of the template is possible, but we would still need the BPF virtual machine to isolate offloaded programs inside a single process. Our userspace implementation of the BPF verifier enables the same external functions as our kernel implementation with fewer data structures available for programs to use. Nevertheless, implementing new data structures is straightforward thanks to the availability of many userspace implementations of common data structures as libraries.

To collect timestamps and update the token buckets, we use the `rte_rdtsc_precise` function provided by DPDK instead of the `bpf_ktime_get_ns` external function available to BPF programs, which we used in our XDP prototype.

4.5 Offload Examples

This section describes two example offloads, the first of a TCP proxy with support for SYN cookies, the second of a DNS rate-limiter. We implemented these two security services and use them for our evaluations in Section 4.6.

4.5.1 TCP Proxy

The TCP proxy implements the SYN cookie protection of the Linux kernel to protect against TCP SYN floods. It answers TCP SYN packets with a TCP SYN+ACK packet containing a SYN cookie in place of the sequence number. The SYN cookie is generated using the same algorithm as the Linux kernel, including the SipHash hash algorithm [8]. TCP ACK packets received in response to the TCP SYN+ACK packet are either dropped or sent to the tenant's domain depending on whether the received SYN cookie is valid.

Therefore, once offloaded from a virtual machine, the TCP proxy runs the three-way handshake for the virtual machine's kernel. When the connection is established, it forwards the TCP ACK packet to the virtual machine. The TCP Maximum Segment Size (MSS) value negotiated during the handshake is encoded in that last packet. Because the expected encoding for the MSS value depends on the kernel, the TCP proxy is a good example of a security service that would be difficult to offer in a common manner for all tenants.

The implementation of the TCP proxy in BPF overcomes two limitations of the current BPF design: BPF programs cannot craft new packets and loops are forbidden. To overcome the

first limitation and send TCP SYN+ACK packets, the BPF program craft the TCP SYN+ACK packet from the received TCP SYN packet. To that end, the program swaps the Ethernet and IP addresses and updates the TCP checksum[17]. It then updates the TCP flags and the sequence number with the SYN cookie.

The second limitation forces us to unroll all loops in the implementation of the SipHash algorithm. Unrolling loops is likely to reduce performance, but we expect to be able to use bounded loops with future versions of BPF [45] and, as shown in Section 4.6, the current, inefficient implementation still provides a significant performance boost thanks to the offload.

In addition to running the SipHash algorithm and swapping addresses, the BPF program performs a hash table lookup to check whether the packet belongs to an established connection.

4.5.2 DNS rate-limiter

The DNS rate-limiter protects a downstream DNS server against DNS query floods. It parses UDP DNS queries packets to retrieve the queried domain name and compares it to a hardcoded value. Packets with the expected domain name run through a rate-limiter implemented with a token bucket. If a packet contains an unexpected domain name or if the bucket is empty, the packet is dropped.

The DNS rate-limiter program is less CPU-hungry than the TCP proxy, a difference we use in the evaluations to illustrate the fairness of our mechanism. Indeed, without the fairness mechanism, each program would receive the same number of packets, and the DNS rate-limiter, having a smaller per-packet CPU consumption than the TCP proxy, would receive a smaller share of the CPU. However, we expect practical implementations of this security service to be more complex.

This program only needs to parse the DNS question and perform a few memory accesses for each packet. The token bucket is implemented with an array BPF data structure containing a single item. The generation of tokens runs inline, at regular intervals, before parsing the DNS question.

4.6 Evaluations

This section aims to evaluate three aspects of our design and prototype: (i) the overhead from our fairness mechanism, (ii) how this mechanism compares to traditional fairness mechanisms, and (iii) the performance gain from the offload itself.

4.6.1 Evaluation Setup

Our testbed consists of two servers connected with Mellanox 40 Gbps NICs. The Device Under Test (DUT) has an Intel Xeon E5-2640 2.6 Ghz with hyper threading disabled, 20 MB of L3 cache, and 16 GB of DDR4 memory at 2133 MHz. The DUT runs Linux 4.9 without the KPTI patch or the retpoline compilation option. Offloaded programs were compiled to BPF bytecode with Clang 3.8.

In all experiments, we use a single core at the DUT. To do so, we ensure all packets arrive on the same queue at the NIC, and therefore on the same CPU on the host. Packet processing performance with several cores strongly depends on the efficiency of packet demultiplexing to queues at the NIC (e.g., using receive-side scaling).

[17] The IP checksum doesn't need to be updated as it is computed with a commutative operation over the 16-bit words of the header and we only swapped these 16-bit words.

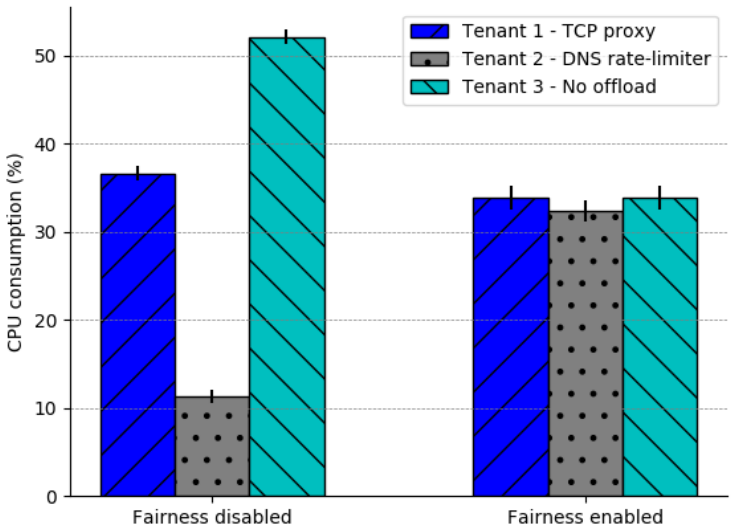


Figure 4.2 – CPU consumption as a percentage of the total time for a single core, for each tenant, with and without fairness mechanism.

We configure three tenants on the DUT, each with a single Docker container and their own VLAN. Tenant 1 and 2 offloaded the TCP proxy and the DNS rate-limiter in the host's in-kernel datapath respectively, whereas tenant 3 doesn't have any security service offloaded. All tenants receive the same number of tokens at each generation interval and, therefore, have the same CPU share on the host. In the template program, packets are demultiplexed based on the VLAN IDs only. The second server uses MoonGen [41] to launch a TCP SYN flood against the first tenant, a DNS flood against the second, and replay a CAIDA packet capture against the third.

Each experiment lasts 5 minutes and we report the mean and the standard deviation over 10 runs.

4.6.2 Fairness Mechanism

Effectiveness of CPU Fairness Mechanism. First, using our probes in the driver, we measure the CPU consumption of each tenant as a percentage of the total CPU time. Figure 4.2 depicts the results, both with and without our fairness mechanism.

While they all receive the same number of packets, because tenant 3 receives all packets in userspace and does not drop any at the infrastructure, she consumes the majority of the CPU time on the host. In contrast, due to her offloaded program, tenant 2 requires few CPU cycles to drop packets. Nevertheless, without the fairness mechanism, tenant 2 is allowed a smaller share of the CPU time. This result illustrates the need for our fairness mechanism to ensure that one tenant will not starve other colocated tenants.

Because it takes a few tens of CPU cycles to read a packet in memory, identify its tenant, and free its memory buffer [2], a tenant that does not have any tokens left but still receives a large number of packets may impact the overall performance. Since this weakness affects all CPU-based systems, one approach to mitigate it could leverage upstream hardware devices (e.g.,

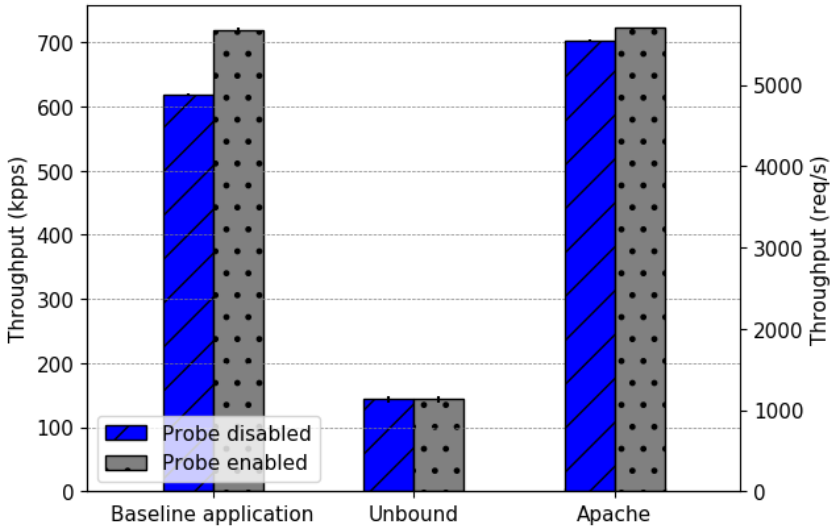


Figure 4.3 – Packet processing performance with and without the tracing probes. The throughput is measured in requests per seconds for Apache only.

NICs and switches) to install a first, coarse-grained rate-limiter and filter oversized flows before they reach the host's CPU. We leave the design of such a defense to future work.

Overhead from CPU Consumption Probes. We next measure the overhead caused by the installation of our probes in the driver. On the DUT, we configure three applications, in Docker containers: the first simply drops packets and serves as a baseline to measure the worst possible overhead, the second is an Unbound DNS recursive server, and the last an Apache HTTP server. On the second server, we configure MoonGen to send DNS requests to the first two applications. For the Apache server, we use Apache Benchmark [1] to send a flood of HTTP GET requests. We measure the maximum number of packets (respectively, requests for the Apache server) the application is able to process on a single core, both with and without our probes.

As shown in Figure 4.3, our probes in the driver only add a slight overhead on the end-to-end performance of applications. Because our probes require a constant number of cycles per packet, the overhead is larger for application that require few cycles to process each packet, as with the baseline application. While for the Apache application we measure a 2.6% overhead, we could not measure any perceived overhead for the Unbound DNS server. Because it performs the minimum operations possible on each packet (receive and drop), the baseline application provides a good estimate of the worst possible overhead (13.9%).

Comparison with Preemptive Scheduler. To compare our fairness mechanism to the Linux preemptive scheduler, we ported it to DPDK. Our DPDK application polls packets directly from the NIC, enforces fairness, and sends packets back to the NIC; the offloaded programs are empty for this evaluation. Tokens are generated in a second thread running on the same core as the DPDK process. We compare this setup with a second DPDK-based setup using the Linux schedulers and the `cgroups` feature: each program runs as a DPDK task and the Linux scheduler preempts tasks to enforce fairness. In both setups, programs are assigned an equal share of the CPU and run on a single core. We vary the number of concurrent programs and measure the throughput at the sink.

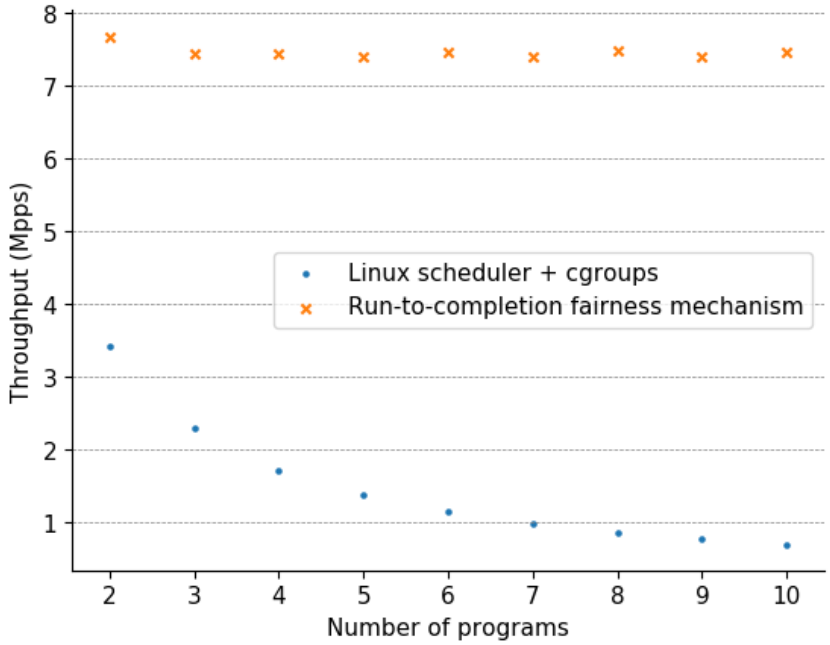


Figure 4.4 – Packet processing performance for different fairness mechanisms.

As shown in Figure 4.4, because it breaks the run-to-completion model of DPDK, the Linux scheduler has a severe impact on performance: the frequent context switches between tasks increasingly impede performance as the number of concurrent tasks grows. Conversely, our approach has a lower overhead and its performance only slightly decreases with the number of tasks; in our system, only the complexity of the token generation algorithm depends on the number of programs.

4.6.3 Performance Gain

Finally, we evaluate the performance gain our prototype can bring to applications by allowing them to offload security services to the infrastructure. We port our TCP SYN cookie proxy and our DNS rate-limiter to userspace, on the Linux API, and execute them both in containers. We compare the performance of these two applications to their offloaded counterparts.

Figure 4.5 on the following page shows the packet processing performance in each case for the two applications. The offload to the infrastructure—in this case the host's kernel—brings a 4–6x performance improvement by avoiding unecessary packet copies to userspace and leveraging the higher performance of the infrastructure's datapath. Whereas the rate-limiter drops most packets, the TCP proxy replies to received flood packets with a TCP SYN+ACK packet (with a SYN cookie) and therefore requires more cycles per packet. For this reason, the performance gain is lower for the TCP proxy.

Note that this performance improvement is only possible because we can ensure each tenant still receives a fair share of CPU time. We also note that the performance improvement largely depends on the offload capabilities of the underlying infrastructure. For this reason, we expect a slightly better performance improvement if the infrastructure relies on kernel bypass technologies (in which case, solutions such as [80, 27] could be used to execute offloaded programs) and a much stronger improvement if programs can be offloaded to a SmartNIC.

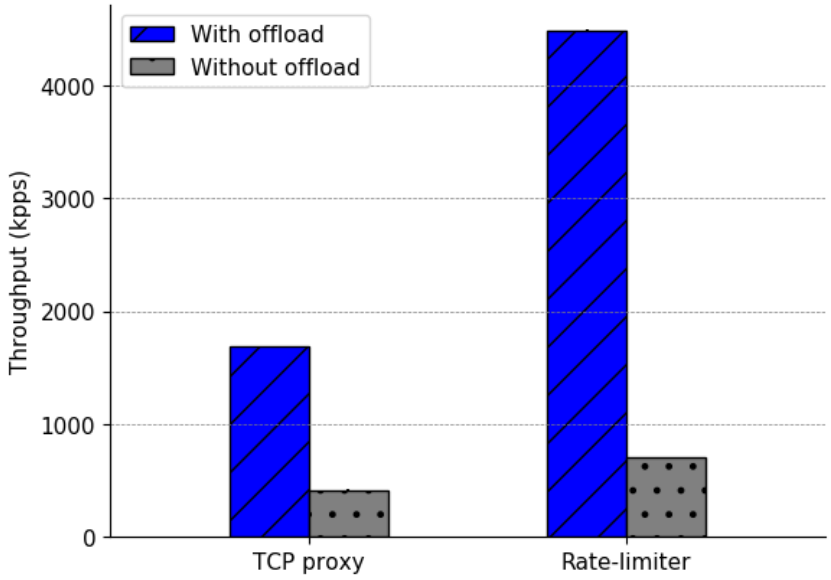


Figure 4.5 – Packet processing performance gain from offload.

4.7 Related Work

In Chapter 2, we discussed related works to offload tenant workloads to the host and alternate software memory isolation techniques to BPF. In this section, we discuss works on performance isolation on the host, in the context of multi-tenant environments.

In [63], D. Gupta *et al.* extend Xen to account for work done in the host domain on behalf of tenants. They use a heuristic to estimate each tenant's CPU consumption in the host domain and propose a new scheduler to take this additional consumption into account when scheduling virtual machines. The estimated CPU consumption is based on measurements of the average per-packet CPU overhead on their system. This approach would be inadequate in our system because, with the offloaded programs, two packets can result in very different processing steps and CPU consumptions.

Network I/O resource isolation has also been explored in the context of multi-applications operating systems. With Lazy Receiver Processing (LSR) [39], packet processing is performed lazily, once the userspace process requests new data. This lazy processing, combined with per-application receive queues, ensures that (i) unnecessary I/O processing is avoided if the receiving application is already overloaded, (ii) packet processing done on behalf of a userspace process can be charged to that process. LSR however requires the NIC to be able to deliver packets directly to the appropriate application's receive queue.

In [10], G. Banga *et al* define a new operating system abstraction, namely resource containers. A resource container logically contains all system resources used by an application, including CPU time, sockets, and packet buffers. Resource containers allow the kernel to make a clear distinction between protection domains (e.g., Linux tasks) and the resources they use (e.g., CPU time). The CPU scheduler can then use information from resource containers to schedule the associated processes; resource containers, however, rely on a preemptive scheduler to enforce performance isolation.

In a concurrent work [2], Addanki *et al.* designed a *fair dropping* mechanism close to ours, though for a different use case. Their system aims to enforce max-min fairness among different

flows inside software switches. Compared to our system, theirs processes packets in batches to improve performance, but it approximates the per-packet CPU consumption from the exact CPU consumption for a whole batch.

4.8 Conclusion

In this chapter, we presented a new framework to enable the offload of security services from virtual machines and containers to the host. Because these security services run inside a single process, in the Linux kernel or in userspace, we cannot rely on the traditional isolation and fairness mechanisms of virtualization and containers. Instead, we rely on an hardened BPF verifier to enforce isolation. In addition, we address the fairness concerns with a token-based mechanism that ensures each tenant gets their fair share of CPU time, regardless of the program they offloaded. We prototyped our design on Linux's high-performance datapath and measured a 4–6x performance improvement for offloaded security services.

Chapter 5

Conclusion

In this thesis, we have argued that significant performance improvements are attainable at the end host, in multi-tenant networks, without relying on specialized hardware devices.

To that end, in Chapter 2, we have discussed the state of the art of multi-tenant software datapaths and reviewed the literature on each subsystem that constitutes them, from the reception and transmission of packets at the NIC to the exchange of those packets with the tenant domains, be they virtual machines or userspace processes. We first analyzed the delivery of packets to tenant domains and surveyed related works to deliver packets more efficiently. This first section allowed us to identify the irreducible cost of hardware memory isolation and brought us to discuss software memory isolation alternatives. We next turned to an important component of multi-tenant datapaths, the software switch, which we also used as the base of our consolidation efforts with Oko. In particular, we focused on the packet classification algorithms used in forwarding pipelines and stressed the importance of caching mechanisms to achieve high performance. We then discussed the challenge of building efficient extensible software switches and recent efforts to address this challenge. We concluded our review of the state of the art with a discussion of packet processing offloads, by drawing a parallel between the increasing reliance on hardware offloads and the opportunity to offload tenant workloads to the host.

In Chapter 3, we detailed the design of Oko, an extensible software switch that executes stateful filtering and monitoring programs, called filter programs, as part of its packet classification pipeline. In designing Oko, we had to overcome two main challenges related to packet classification algorithms and safety.

The software switches of multi-tenant platforms usually consist of a pipeline of packet processing stages, with a number of different caching strategies. We built our Oko prototype on Open vSwitch, which has the most aggressive caching mechanisms described in the literature. Extending these caching mechanisms is challenging because they rely on the simple, stateless Match-Action abstraction provided by OpenFlow to construct cache entries. In Oko, we addressed this challenge with a concept of filter program chains, which encode the execution path through stateful filter programs during packet classifications.

The second challenge, providing a safe runtime to execute filter programs as part of the switch's pipeline, was already the subject of a large body of works in the context of kernel and web browser extensions. We chose to rely on the BPF runtime for its minimal instruction set, well-suited to packet processing, and its limited runtime overhead. We implemented the BPF virtual machine, with its static analyzer and external data structures, in the userspace version of Open vSwitch.

Having addressed these challenges, Oko enables the consolidation of network functions at the software switch, to avoid redirections of packets to separate processes and virtual machines. Oko also facilitates the development of new switch features, which can be prototyped as filter programs and loaded in running switches without risking outages.

In Chapter 4, we proposed to offload network services from tenants' domains, be they virtual machines or containers, to the host's datapath. We designed a framework to allow tenants to perform this offload of their network services to an isolated execution environment on the host. As in Oko, we used the BPF runtime to provide memory and fault isolation, but had to devise a new mechanism to enforce performance isolation. In particular, the offload framework needs to enforce CPU fairness among several offloaded network services running in the same process, e.g., in the software switch's process. We addressed this challenge with per-tenant rate-limiters whose rates depend on the tenant's allocated CPU share and the CPU consumption from network services, as reported by a set of probes across the system.

Our offload framework enables tenant to run filtering services, such as security services, closer to the wire, to avoid unnecessary packet processing. By offloading a TCP proxy with a TCP SYN

cookies protection mechanism, we also demonstrated that network services may be offloaded to the host to provide faster responses to client's requests (in our case, TCP SYN packets).

We conclude this thesis with a discussion of future works, including open challenges and new opportunities for improvement.

5.1 Beyond Datacenter Networks

Although the challenges discussed in Chapters 3 and 4 apply to packet processing in any multi-tenant networks, we have focused our work on datacenter networks. Since cloud computing workloads are primarily hosted in datacenters, datacenter networks host the most common cases of multi-tenancy in networking today. Datacenter networks have also fostered innovations in networking in the last decade. Because cloud providers often have an end-to-end control over their datacenter networks, they can more easily experiment and deploy recent advances in networking, including Software-Defined Networking [81, 90], packet processing on general-purpose hardware [40], and hardware offloads [50].

Nevertheless, cloud networks are growing beyond datacenter networks. Edge computing extends the blueprint of cloud computing to Points of Presence (PoPs), small facilities hosting compute resources close to the customers. ISPs are also working on multi-tenant solutions for their on-premise devices, to host network functions from multiple vendors. Finally, 5G promises to enable service providers to rent "slices" of ISPs' networks, thereby expanding multi-tenancy to the WANs.

Several of the solutions to multi-tenancy challenges we designed or discussed in this thesis could apply equally well to multi-tenant networks beside datacenter networks.

Multi-tenant PoPs are likely to run the same software switches as found in datacenter networks. At the same time, hardware resources are scarcer at the edge of the network than in datacenters. Both our contributions may help bring much needed efficiency gains in this context to consolidate network functions at the software switch and offload select network functions from virtual machines and containers to the host. Several content delivery network providers have already started using software isolation techniques at the edge of their networks: Cloudflare relies on V8, a JavaScript engine, to isolate client's code running in their PoPs [18], whereas Fastly is experimenting with WebAssembly [69] for the same usage.

In addition, these edge servers are unlikely to have as many cores as datacenter servers and may therefore not afford to dedicate cores to specific tasks (e.g., to the datapath in datacenters [33]). Our fairness mechanism for the offload framework may help run isolated programs concurrently on the same cores without relying on expensive preemptive schedulers. However, edge servers, and in particular smaller on-premise devices, often rely on fixed-function hardware to achieve high performance, which prevents the use of more flexible software isolation techniques. Whether recent software techniques to accelerate packet processing could satisfy the performance needs of edge servers and devices remains unclear.

5.2 Runtime Software Switch Specialization

In Chapter 2, we surveyed several algorithms for packet classifications on CPUs. In particular, we opposed general designs, such as the Tuple Space Search classifiers implemented in Open vSwitch [120], and specialized designs, such as those implemented in VFP [48] and ESWITCH [106]. The former store forwarding rules in a single, general-purpose data structure, whereas the latter

rely on a set of efficient, specialized data structures to store rules with known patterns (e.g., IP prefix lookups).

As we have seen, production systems rely heavily on flow caching to classify packets with high throughput. Because the general designs are better suited to build aggregated cache entries from multiple forwarding stages—the common data structure across stages eases aggregation of rules,—we can expect them to perform better for longer pipelines. Conversely, specialized designs are likely to achieve higher performance for shorter pipelines with a few well-known stages. Although these differences have been noted before [118], they have yet to be evaluated, i.e., the performance of general-purpose and specialized packet classifiers have not been publicly compared for long forwarding pipelines.

With ESWITCH [106], L. Molnár *et al.* propose to replace the general-purpose data structure and caching mechanisms of Open vSwitch with a set of specialized data structures to which rules are mapped. Their new switch however doesn't support runtime specialization: the switch must be recompiled every time new types of rules are added or removed from the pipeline. In addition, L. Molnár *et al.* discard general-purpose data structures completely, even though they could be useful as a fallback strategy when no known efficient data structure is found for a given rule type—in this case, ESWITCH implements a linear search through the set of rules.

We believe Oko, our extensible software switch, could be adapted to enable runtime switch specialization, in response to changes in the structure of the forwarding pipelines. Instead of filtering and monitoring functions, switch extensions could implement a specialized shortcut for some flows, before the flow caches. The software switch would recognize rule patterns, much as in ESWITCH, and load extensions for fast lookups into specialized data structures. Experiments are needed, however, to assess whether such specialized data structures can be implemented efficiently with BPF.

5.3 Low-Overhead Software Isolation for Datapath Extensions

In both our contributions, we relied on software memory isolation techniques to isolate components while allowing transmission of packets at a low cost. In particular, we relied on BPF, the same technique used in the Linux kernel to isolate extensions [30].

As Chapter 2 illustrates, BPF is by far not the only software technique available. The main difference lies in the targeted applications: the new (e)BPF bytecode in the Linux kernel targets high-throughput packet processing applications, such as the mitigation of denial of service attacks. In such a context, runtime checks on memory access bounds may add a noticeable overhead. The BPF verifier tries to reduce such checks to a minimum by verifying most memory accesses ahead-of-time. In both the Linux kernel and in Oko, it implements a sort of symbolic execution of all possible execution paths through the code. This exhaustive search is what allows it to avoid many runtime checks.

A bug in the Linux BPF verifier can have dire consequences as it can be used to run BPF programs from unprivileged users in the kernel context. As we use BPF to isolate programs from cloud tenants in Chapter 4, we also need to establish a high trust in BPF's isolation mechanisms. Nevertheless, the current verifier does not have any formal foundation; it is therefore easier to extend for kernel developers, but harder to trust for anyone else.

Finding alternatives that scale better than the current exhaustive search without adding to the runtime overhead is a known open research problem. If E. Gershuni *et al.* [56] propose a first[18] approach to this problem. Based on abstract interpretation [32], their prototype remains

[18] Previous works focused on the reduced instruction set and limited capabilities of cBPF [109, 147].

several orders of magnitude slower than Linux's exhaustive search for small BPF programs.

There is also a pressing need for new test generation tools suitable to BPF's context (its execution in the kernel) and interfaces (through the usual system calls and via the VM's external functions). Existing fuzzing tools for the Linux system call API, such as syzkaller [60], are ill-suited to find complex bugs in the BPF verifier. The BPF programs they generate often fail to pass the very first verifier checks. Attempts to port the BPF verifier to userspace [135] and fuzz it with userspace tools such as AFL [151] and libfuzzer [59] are faced with the difficulty of maintaining such a userspace port. In any case, none of the current tools are able to generate test cases for kernel functions that are accessible only through BPF programs, which leaves a large surface of the BPF code untested.

5.4 The Heterogeneity of Packet Processing Devices

With the end of Moore's law, service providers will need to find new ways to scale their packet processing workloads. Although we have argued in this thesis that performance improvements are still achievable in software datapaths, there is no doubt that multi-tenant datapaths and packet processing tasks in general will increasingly rely on hardware offloads to achieve high performance. In Chapter 2, we reviewed reports on experience with packet-classification systems in production virtualization platforms. The three platforms we discussed all rely on some form of hardware offload (cf. Table 2.2 on page 19), with the more recent reports describing a trend to offload more work [33, 50].

There is, therefore, an increasing reliance on specialized hardware devices, to which packet processing tasks are offloaded. Packet processing tasks are often only partially offloaded, to retain the benefits of both hardware and software solutions.

Specialized hardware devices are, by design, of very different natures; they each have their own capabilities (e.g., cryptographic accelerators), memories of diverse sizes and speeds, and can more or less easily be updated and programmed. The key challenge to network programmers is perhaps that these devices expose very different machine and programming models. Reconfigurable switches [20] are programmed through Match-Action abstractions, FPGAs through Hardware Description Languages (HDLs), and NPUs through specialized low-level instruction sets.

Ideally, a single language would enable network programmers to write packet processing logic for their network without worrying about the execution target. However, because of the disparate programming model exposed by specialized hardware devices, designing such a common abstraction is challenging.

There are, nevertheless, a few recent improvements to this situation. As a first example, the BPF bytecode, originally designed to be easily JIT compiled to common CPU architectures, is used to program NPUs in SmartNICs [87], through a special JIT compiler. Another example may be found in the P4 language [21], a domain-specific language (DSL) to describe packet forwarding pipelines. Although P4 is well-suited to program the Match-Action pipeline of reconfigurable switches [83], it may also be used to describe similar forwarding pipelines running on FPGAs [146], NPUs [87], or CPUs [128].

Although a general, common language for these diverse hardware devices is probably out of reach, as P4 illustrates, domain-specific languages may be better suited to program different hardware devices. P4 provides a common abstraction to specialized hardware devices for packet forwarding; could similar abstractions be devised for e.g., network monitoring or denial-of-service mitigation?

Appendix A

Example Filter Program Trees

This appendix provides two examples, a best-case and a worst-case, of the evolution of the number of potential cached rules with the number of filter programs in the slowpath pipeline of Oko.

The first example, illustrated in Table A.1 and Figure A.1, has five potential cached rules (number of paths through the filter program tree) for four filter programs. In this case, the number of potential cached rules grows linearly with the number of filter programs. We expect this case to be frequent for programs acting as filters (as opposed to monitoring programs). Both our stateless signature filtering (cf. Section 3.4.1) and our stateful firewall (cf. Section 3.4.2) programs have this behavior.

The second example, illustrated in Table A.2 and Figure A.2 on the following page, has 16 potential cached rules with the same number of filter programs as the first example. In this case, the path through the pipeline is the same regardless of the filter programs' execution results. The number of potential cached rules therefore grows exponentially with the number of filter programs: if n is the number of filter programs, there are 2^n potential cached rules. We expect this case to be more common with monitoring programs as they are less likely to alter the path through subsequent tables.

To conclude, we expect neither the best case configuration nor the worst case to be very common for long pipelines of filter programs. If a pipeline contains several filter programs, there are likely to be both filtering and monitoring programs, resulting in a growth of the number of potential cached rules in between the two above, extreme scenarios. In practice, we have found programs acting as filters to be more common, perhaps because packet analyses tend to be combined into a single monitoring program.

Prio.	**Dest.**	**Prog.**	**Action**
1000	*	a	in_port
100	*	b	drop
10	*	c	drop
1	*	d	drop
0	*	-	port 1

Table A.1 – Example Oko table with four filter programs.

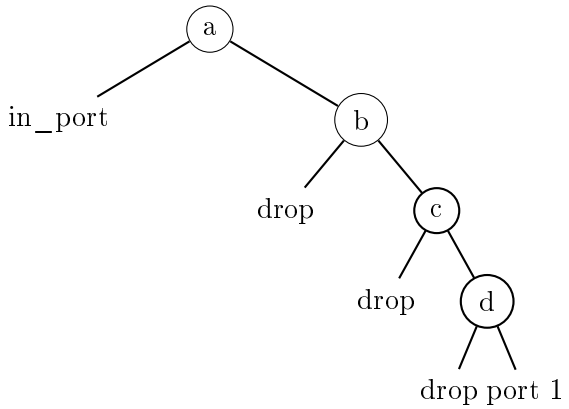


Figure A.1 – Tree of filter programs for the Oko pipeline from Table A.1. This table has five potential cached rules for four filter programs.

Prio.	**Dest.**	**Prog.**	**Action**
10	*	a	table 2
1	*	-	table 2

(a) Table 1

Prio.	**Dest.**	**Prog.**	**Action**
10	*	b	table 3
1	*	-	table 3

(b) Table 2

Prio.	**Dest.**	**Prog.**	**Action**
10	*	c	table 4
1	*	-	table 4

(c) Table 3

Prio.	**Dest.**	**Prog.**	**Action**
10	*	d	drop
1	*	-	drop

(d) Table 4

Table A.2 – Example Oko pipeline with four filter programs. The path through the pipeline does not depend on the filter programs.

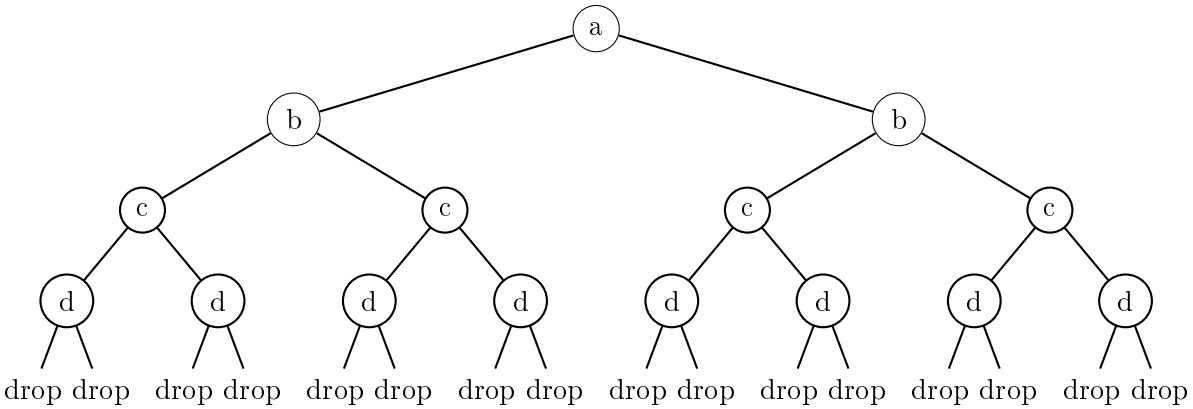


Figure A.2 – Tree of filter programs for the Oko pipeline from Tables A.2. This pipeline has 16 potential cached rules for four filter programs.

Appendix B

Example BPF Bytecode

The following bytecode program was compiled from the stateless signature filtering program given in Listing 3.1 with Clang 6.0.0.

Lines 2–3 correspond to the bounds check on the packet. Thanks to this bounds check, the verifier knows that if the jump at instruction 3 is not taken, the packet is at least 54 bytes long. Lines 4–29 check the p0f signature itself.

```
1  entry:
2        r3 = 54
3        if r3 > r2 goto <LBB0_9>
4        r2 = *(u8 *)(r1 + 22)
5        r2 += -94
6        r2 &= 255
7        if r2 > 34 goto <LBB0_9>
8        r2 = *(u8 *)(r1 + 14)
9        r3 = r2
10       r3 &= 15
11       if r3 != 5 goto <LBB0_9>
12       r3 = *(u8 *)(r1 + 20)
13       r3 &= 64
14       if r3 == 0 goto <LBB0_9>
15       r3 = *(u8 *)(r1 + 15)
16       r3 &= 8
17       if r3 != 0 goto <LBB0_9>
18       r3 = *(u16 *)(r1 + 48)
19       if r3 != 8192 goto <LBB0_9>
20       r2 <<= 2
21       r2 &= 60
22       r3 = *(u16 *)(r1 + 46)
23       r3 >>= 2
24       r3 &= 60
25       r2 -= r3
26       r1 = *(u16 *)(r1 + 16)
27       r1 = be16 r1
28       r0 = 1
```

```
29        if r2 == r1 goto <LBB0_8>
30        r0 = 0
31
32  LBB0_8:
33        exit
34
35  LBB0_9:
36        r0 = 0
37        exit
```

Listing B.1 – BPF bytecode from the p0f filter program in Listing 3.1.

Annexe C

Résumé en français

French Summary

Introduction

L'informatique en nuage requiert un partage des ressources entre les différents clients hébergés sur une même infrastructure physique. Ce partage doit s'effectuer de manière équitable face à des clients potentiellement malveillants. Ce partage est d'autant plus compliqué pour les accès réseaux : non seulement les ressources réseaux doivent être équitablement partagées mais chaque client doit disposer de sa propre "vue du réseau".

Ces dernières années, la nécessité de hautes performances réseaux s'est faite plus pressante pour l'informatique en nuage. D'une part, afin de réduire les coûts, de plus en plus de machines virtuelles sont installées sur chaque machine hôte. La pile réseau des machines hôtes doit donc supporter des débits plus importants, sans pour autant consommer plus de ressources, ceci afin de ne pas consommer des ressources qui pourraient être vendues aux clients. De plus, les traitements réseaux en environnement virtuel se sont fait plus fréquents avec l'arrivée des premières fonctions réseaux virtualisées et la délocalisation de fonctions de réseaux d'entreprise vers les plateformes de l'informatique en nuage [130].

Dans un contexte d'intensification de la demande en haute performance sur les réseaux, les acteurs de l'informatique en nuage ont souvent recours à des équipements matériels spécialisés mais inflexibles, leur permettant d'atteindre les performances requises. Néanmoins, dans cette thèse, nous défendons la possibilité d'améliorer les performances significativement sans avoir recours à de tels équipements.

Performance de la pile réseau

Plus particulièrement, nous avons identifié plusieurs challenges posés par les piles réseaux des plateformes multi-tenants.

Premièrement, les logiciels de la pile réseau effectuent un nombre important d'opérations plus ou moins coûteuses, en plus de la simple réception et transmission des paquets réseaux. Ces opérations incluent par exemple l'exécution des algorithmes de classification des paquets [139] et l'application des politiques de sécurité et de qualité de service.

De plus, les différents services de la pile réseau sont souvent la responsabilité d'équipes différentes et sont donc implémentés comme autant de composants séparés (sous la forme de processus ou même de machines virtuelles). De nombreuses redirections sont donc nécessaires entre ces composants, au prix de dégradations importantes des performances.

Finalement, l'isolation réseau elle-même à un coût important : les paquets réseaux doivent être copiés deux fois pour les démultiplexer vers les machines virtuelles et containers destinations. La première copie s'effectue de la carte réseau vers l'espace mémoire de la machine hôte ; la seconde vers l'espace mémoire du client (ex. celui de la machine virtuelle). Sans assistance matérielle,[19] les paquets ne peuvent être copiés directement dans l'espace mémoire du client puisque le commutateur logiciel de la machine hôte doit les lire pour déterminer la destination. Ainsi, même une opération très simple (ex. filtrage) effectuée sur un paquet dans une machine virtuelle aura un coût important dû aux deux copies.

Approche de la thèse

Dans cette thèse, nous proposons deux approches pour adresser la problématique de performance ci-dessus.

Nous proposons tout d'abord une solution technique permettant la consolidation des fonctions réseaux de la pile réseau afin d'éviter les nombreuses redirections. Cette consolidation se fait sans pour autant perdre les séparations logiques entre les diverses fonctions.

Notre seconde contribution consiste en une plateforme logicielle permettant la délocalisation de traitements réseaux de machines virtuelles et conteneurs vers la pile réseau de la machine hôte. En particulier, notre plateforme logicielle assure l'isolation, notamment en terme de consommation des ressources.

Contributions

Dans le Chapitre 2, nous présentons un état de l'art des traitements réseaux propres aux plateformes multi-tenants. Nous concentrons notre attention sur les goulets d'étranglement tels que le demultiplexing et la classification des paquets [139].

Le Chapitre 3 présente Oko, un commutateur logiciel dont les fonctionnalités peuvent être étendues à chaud. Oko permet de consolider les services de la pile réseau au niveau des commutateurs logiciels déjà présents dans la plupart des plateformes multi-tenants.

Au Chapitre 4, nous présentons un mécanisme permettant aux clients de délocaliser certains de leurs traitements réseaux vers une pile réseau optimisée de la machine hôte. Nous proposons un algorithme pour assurer l'isolation, notamment en terme de consommation des ressources physiques.

Extensions pour commutateurs logiciels

Nous résumons dans cette première partie notre première contribution, Oko.

Introduction

Les commutateurs logiciels jouent un rôle essentiel au niveau de la pile réseau des plateformes multi-tenants : ils connectent les ports réseaux physiques aux ports virtuels des machines virtuelles et conteneurs. Ils ont aussi généralement la charge d'appliquer les politiques d'accès et de qualité de service.

Plusieurs travaux de l'état de l'art [101, 134, 80, 102] s'appliquent à permettre l'extension des commutateurs logiciels. La plupart exécutent cependant les extensions dans des composants séparés, au détriment des performances, et ne permettent donc pas la consolidation que nous

[19] Pour éviter cette seconde copie, la carte réseau doit pouvoir démultiplexer les paquets elle-même.

cherchons. SoftFlow [80] fait exception : les extensions écrites en C sont exécutées au niveau du cache d'Open vSwitch, le commutateur logiciel sur lequel SoftFlow est basé. Les extensions de SoftFlow ne sont cependant pas isolées du reste du commutateur ; une erreur de développement dans une extension peut entraîner l'arrêt du commutateur. De plus, SoftFlow requiert des instructions des développeurs afin de préserver les mécanismes de mise en cache existant ; les développeurs doivent donc comprendre les mécanismes en question.

Notre contribution, Oko, est basée sur Open vSwitch comme SoftFlow mais isole les extensions du reste du commutateur grâce à un mécanisme basé sur le bytecode BPF [99]. Nous avons de plus fait le choix de limiter les privilèges des extensions afin de faciliter leur mise en cache. Grâce à ces limitations, Oko ne nécessite aucune instructions des développeurs et atteint des performances similaires à celles de SoftFlow.

Contraintes

Oko doit adresser deux contraintes.

Parce que le commutateur connecte les machines virtuelles au réseau physique, il est un composant critique de la pile réseau ; un arrêt même court du commutateur peut entraîner des pertes de revenus pour les clients. Les chargements d'extensions doivent donc se faire sans aucune interruption du fonctionnement du commutateur. De plus, le commutateur doit empêcher le chargement d'extensions incorrectes, celles qui pourraient mener à des bogues.

La plupart des travaux de l'état de l'art adressent cette première contrainte en exécutant les extensions sous forme de processus séparés. Cette approche a cependant un coup élevé. NetBricks [114], de P. Aurojit *et al.*, montre qu'une autre approche est possible : les techniques d'isolation logicielle de la mémoire [145, 44, 109, 99] peuvent être utilisées pour isoler différent programmes tout en permettant l'échange d'informations entre ces programmes (dans notre cas, les paquets réseaux) à moindre coût.

La seconde contrainte concerne les performances des algorithmes de classification des paquets des commutateurs logiciels. En effet, afin d'atteindre de hautes performances, ceux-ci implémentent des mécanismes de mise en cache des règles de transmission [120]. Ces mécanismes prennent pour hypothèse que les règles de transmission changent infréquemment comparé à la fréquence d'arrivé des paquets réseaux ; les règles sont donc valides suffisamment longtemps pour valoir le coup d'être mise en cache. Cette hypothèse ne tient plus avec nos extensions. Celles-ci ajoutent un état au commutateur duquel les règles de transmission à exécuter pour un paquet donné peuvent dépendre. Avec une approche naïve, le cache nécessiterait donc d'être reconstruit après chaque changement de l'état, potentiellement à chaque exécution d'une extension.

Conception et implémentation

Oko étend le modèle de la table OpenFlow [100] avec un nouveau champs optionnel de type *match*. Ce champs contient une référence vers l'extension à exécuter pour une règle. L'extension consiste en un programme en bytecode BPF [99] qui retourne 1, pour indiquer un *match*, ou 0, pour indiquer l'absence de *match*. Ces programmes peuvent lire et écrire dans des structures de données qui persistent d'un paquet aux suivants, de manière à effectuer des actions qui prennent en compte les paquets précédemment reçus.

***Verifier* BPF.** Pour isoler les extensions entre elles et du reste du commutateur, nous utilisons le même mécanisme de vérification du bytecode BPF que le noyau Linux. Un logiciel, le *verifier*, s'assure tout d'abord que le programme BPF est assuré de se terminer, en vérifiant que son graphe

de flot de contrôle ne contient pas de retour en arrière au niveau des arrêtes. Le *verifier* exécute ensuite symboliquement l'ensemble des chemins possibles dans le graphe de flot de contrôle en inférant le type (ex. pointeur vers une valeur de structure de donnée, nombre entier ou pointer vers la pile) et les bornes des registres. Les types des registres permettent notamment de vérifier que le programme ne contient pas d'opérations invalides telles que des accès à des zones mémoires invalides ou des divisions par zéro.

Le *verifier* d'Oko ne contient pas la plupart des optimisations implémentées dans le *verifier* du noyau Linux (ex. élagage des branches à exécuter symboliquement). Il ne protège pas non plus le commutateur logiciel contre les accès mémoires non-alignés et les fuites de pointeurs.

Mise en cache des extensions. Open vSwitch utilise deux niveaux de caches pour atteindre de hautes performances sans sacrifier la généralité du modèle OpenFlow [120]. L'implémentation complète du modèle OpenFlow implique l'exécution d'un nombre variable de recherches dans une pipeline de tables de transmission. Dans Open vSwitch, les règles mises en cache consistent ensuite en l'agrégat de l'ensemble des règles trouvées dans chaque table de la pipeline (si plusieurs tables sont effectivement parcourues).

Avec Oko, les règles de transmission exécutées pour un paquet arrivant sur le commutateur dépendent du résultat de l'exécution des programmes BPF. Cette dépendance doit donc être encodées dans les règles mise en cache. Pour ce faire, nous définissons une notion de chaîne de programmes BPF. Cette chaîne contient, pour une règle mise en cache, les références vers l'ensemble des programmes BPF exécutés avec le résultat de leur exécution. De cette manière, nous savons que la règle est valide pour un paquet suivant arrivant sur le commutateur si les mêmes programmes, exécutés dans le même ordre, produisent les mêmes résultats.

Cas d'application

Nous avons implémenté trois exemples de cas d'application de Oko, trois programmes BPF qui répondent à des problématiques différentes. Les programmes sont écrits en C et compilés en bytecode BPF.

Le premier programme implémente un simple mécanisme de filtrage sans état des paquets. Ce mécanisme se base sur des signatures p0f [150] et peut être utilisé pour détecter et bloquer des attaques par déni de service. Ces signatures encodent des informations sur les options TCP des paquets qui dépendent souvent des logiciels et systèmes d'exploitation utilisés pour envoyer les dits paquets. Un logiciel converti les signatures p0f en programmes C qui sont ensuite compilés en bytecode BPF qu'Oko peut charger et exécuter.

Le deuxième programme peut être utilisé, combiné avec les règles OpenFlow, pour implémenter un parefeu à état. Il suit l'état des connexions TCP afin de distinguer les connexions établies des nouvelles connexions. Pour cela il implémente la machine à état de TCP et effectue des accès vers une table de hachage contenant les informations sur chaque connexion.

Le dernier programme permet de récolter des statistiques sur les connexions TCP afin de déterminer le facteur limitant dans chaque connexion entre le réseau, l'émetteur et le récepteur [57]. Ce programme effectue aussi des accès vers une table de hachage contenant les statistiques sur chaque connexion, de la même manière que le programme du parefeu à état.

Évaluations

Nous avons effectué des évaluations pour répondre à trois questions :

- Quel est le coût des modifications apportées par Oko à Open vSwitch ?

- Est-ce que notre mécanisme de mise en cache des extensions est efficace ?
- Comment Oko se situe en terme de performance par rapport aux autres approches d'extension du commutateur logiciel ?

Les évaluations sont réalisées sur deux serveurs directement connectés. La configuration du serveur qui exécute le commutateur logiciel est détaillée en Section 3.5.1. Les évaluations suivent deux scénarios : un premier scénario très simple pour lequel Open vSwitch atteint les meilleures performances et un second scénario plus réaliste qui constitue un pire cas pour les mécanismes de cache du commutateur.

Nous montrons tout d'abord que, sous le scénario réaliste, les performances d'Open vSwitch ne sont pas dégradées par nos modifications (Figure 3.4b). Nous observons par contre une légère dégradation, sous le scénario simple, lorsque le premier niveau de cache est désactivé (Figure 3.4a). Cette dégradation peut s'expliquer par le fait que, sous ce scénario, Open vSwitch a très peu de traitements à effectuer pour chaque paquet ; le moindre coût supplémentaire est donc accentué.

Nous évaluons ensuite le coût de longue chaînes de programmes BPF dans le cache (Figure 3.5). Avec 10 programmes chaînés, nous observons une forte dégradation des performances, de l'ordre de -44%. Cette dégradation est en accord avec une expérimentation proche mené pour SoftFlow [80]. Néanmoins, sans les chaînes de programmes, les performances seraient moins bonnes (voir *Slowpath* sur la Figure 3.6).

Enfin, nous comparons le débit atteint par Oko pour nos trois programmes exemples avec les mêmes programmes s'exécutant dans (i) un processus séparé ayant une mémoire partagée avec le commutateur ; (ii) une machine virtuelle. Oko obtient un débit de traitement deux à trois fois supérieur à celui des machines virtuelles et 1.7–1.9 fois supérieur à celui du processus séparé.

Délocalisation vers l'hyperviseur

Nous résumons maintenant notre seconde contribution, permettant la délocalisation de traitements réseaux clients vers les machines hôtes de l'informatique en nuage.

Introduction

Pour ce défendre contre les attaques réseaux, les applications de l'informatique en nuage s'appuient sur un ensemble de services sécurités, allant de la simple limite du débit de requêtes aux proxy TCP SYN et parefeux applicatifs. Ces services de sécurité encodent souvent des informations spécifiques aux applications qu'ils protègent. Ils se ressemblent cependant en cela qu'ils implémentent tous une forme de filtrage. Pour cette raison, il est préférable de les exécuter au plus tôt, le plus prêt possible du réseau physique, de manière à économiser des ressources sur les serveurs.

Si les bénéfices d'un filtrage au plus tôt des paquets destinés aux machines virtuelles ont déjà été relevés dans l'état de l'art [25], dans cette seconde contribution de la thèse, nous proposons d'aller plus loin en permettant aux clients de délocaliser des programmes (presque arbitraires) de traitement des paquets vers la machine hôte.

Contraintes

La principale difficulté dans la délocalisation des traitements clients vers la machine hôte réside dans l'isolation des performances. Pour l'isolation mémoire, nous pouvons utiliser les différentes

techniques d'isolation discutées précédemment (isolation matérielle avec les processus et les machines virtuelles ou isolation logicielle avec BPF par exemple). L'isolation des performances est plus difficile à mettre en œuvre sans dégrader fortement ces mêmes performances ; elle implique de partager équitablement les ressources consommées entre les différents programmes délocalisés.

Il y a deux raisons à cette difficulté. D'une part, les services de sécurité que nous souhaiterions délocaliser vers la machine hôte ont des consommations de ressources très variées. Il n'est donc pas possible de simplement limiter le nombre de programmes délocalisés par client[20]. De plus, les piles réseaux vers lesquelles n'ont souhaitons délocaliser les services de sécurité sont fortement optimisées pour les hautes performances. Pour cela, ces piles réseaux évitent toute interruption entre le début du traitement d'un paquet et sa transmission (un modèle d'exécution dit "*run-to-completion*"). Pour cette raison, le partage du temps "consommé" sur le CPU ne peut effectué par un ordonnanceur préemptif classique.

Conception

Le processus de délocalisation des services de sécurité des clients est illustré en Figure 4.1. Les clients envoient d'abord leur programme à délocaliser sur la machine hôte vers l'API du service d'informatique en nuage. Pour chaque machine hôte, le service d'informatique en nuage intègre l'ensemble des programmes à délocaliser à un programme maître. Le programme maître inclut des opérations supplémentaires pour démultiplexer les paquets reçus vers les programmes délocalisés appropriés et assurer l'équité des ressources consommées. Le programme maître est ensuite compilé en bytecode BPF à installer sur la machine hôte.

L'utilisation de BPF nous permet, comme pour notre première contribution Oko, d'isoler les différents programmes. Le programme maître compilé peut de plus être installé dans le noyau Linux, grâce à l'implémentation Linux de BPF, ou en espace utilisateur, grâce à notre implémentation pour Oko.

Le partage des ressources consommées entre les différents programmes délocalisés repose sur un mécanisme de seaux à jetons. Chaque programme dispose d'un seau à jeton. Lorsqu'un paquet est traité par un programme, autant de jetons sont retirés du seau qu'il a fallut de temps pour traiter le paquet. À intervalles réguliers, de nouveaux jetons sont générés et distribués entre les seaux des différents programmes. Si un seau est déjà plein, les jetons restant à distribuer sont répartis équitablement entre les seaux restants, jusqu'à ce qu'il ne reste plus de jetons ou que tous les seaux soient pleins.

La redistribution des jetons nous permet d'implémenter une politique dite "*work-conserving*", c'est-à-dire qu'aucune ressources ne sont perdues s'il reste des paquets à traiter. Parce que les jetons sont retirés des seaux après que les paquets ont été traités, il est possible d'obtenir un seau avec un compte négatif de jetons. Dans ce cas, le programme associé ne pourra plus traiter de paquets jusqu'à ce que le seau ait à nouveau un compte positif de jetons.

Pour compléter ce mécanisme, il reste cependant à mesurer le temps CPU utilisé pour traiter chaque paquet réseau reçu. Cette mesure n'est pas triviale car nous devons tenir compte de l'ensemble du temps passé sur la machine hôte pour chaque paquet, de manière à pouvoir allouer un temps CPU à chaque client, y compris ceux qui n'ont pas de programme délocalisé sur la machine hôte. Pour ce faire, nous installons des sondes dans la pile réseau utilisée, à chaque point de l'exécution marquant la fin du traitement d'un paquet (paquet jeté, transmis sur le réseau ou envoyé à une machine virtuelle par exemple). Ces sondes sont complétées par une mesure du temps courant dans le programme maître, à la réception du paquet. La différence du temps

[20] Cette approche est actuellement utilisée pour des mécanismes de filtrage plus simples telle que les parefeux à états en amont des machines virtuelles.

relevé par les sondes et par cette seconde mesure nous donne le temps CPU passé à traiter le paquet dans la pile réseau.

Cas d'application

Nous avons implémenté deux programmes pouvant être délocalisés vers la machine hôte. Ces deux programmes ont des consommations CPU très différentes qui permettent d'illustrer l'efficacité de notre mécanisme à jetons.

Le premier programme implémente un proxy TCP avec un mécanisme de protection TCP SYN *cookie*. Un tel mécanisme permet de se protéger contre des attaques par déni de service TCP SYN. Le proxy répond aux paquets TCP SYN avec une signature calculée à l'aide de la fonction de hachage SipHash[21] [8]. Ce mécanisme consomme cependant beaucoup de ressources CPU pour le calcul de la signature.

Le second programme implémente un simple limiteur sur les requêtes DNS. Il extrait d'abord le nom de domaine des requêtes DNS et n'autorise les requêtes que pour un seul nom de domaine. Les requêtes avec le nom de domaine correct sont ensuite limités par un mécanisme de seau à jetons.

Évaluations

Nous avons effectué des évaluations pour répondre à trois questions :

- Quel est le coût des sondes exécutées pour chaque paquet pour mesurer le temps CPU consommé sur la machine hôte ?
- Comment notre mécanisme non-préemptif se situe en terme de performance par rapport à un ordonnanceur préemptif comme celui de Linux ?
- Quel gain en débit de traitement des paquets la délocalisation offre-t-elle ?

Pour répondre à la première question, nous avons configuré trois applications en espace utilisateur. La première se contente de recevoir les paquets UDP et de les jeter ; elle représente l'application la plus simple pouvant être écrite. La seconde est un serveur récursif DNS Unbound et la troisième un serveur HTTP Apache. Pour chaque application nous mesurons le débit de traitement des requêtes reçues avec et sans sondes (voir Figure 4.3). Pour les deux applications réalistes (serveurs DNS et HTTP), les sondes occasionnent une réduction de moins de 3% du débit de traitement.

Pour adresser la seconde question, nous avons implémenté notre mécanisme en espace utilisateur : une application DPDK reçoit les paquets en espace utilisateur, exécute le mécanisme de partage du temps CPU et renvoie ensuite directement les paquets à la carte réseau. Cette application est comparée avec une seconde application DPDK dans laquelle chaque programme délocalisé s'exécute dans un processus séparé ; l'ordonnanceur de Linux est alors utilisé pour partager le temps CPU entre les différents processus. Nous observons (voir Figure 4.4) dans ce cas que les performances restent relativement stables avec notre mécanisme lorsque le nombre de programmes délocalisés augmente. À l'inverse, l'ordonnanceur de Linux impacte fortement les performances de l'application DPDK.

Enfin, pour répondre à la dernière question, nous avons comparé les performances atteintes par nos deux programmes précédemment présentées (le proxy TCP et le limiteur DNS) lorsque

[21] Les TCP SYN cookies permettent d'éviter de conserver des informations sur le serveur pour chaque TCP SYN reçu.

ceux-ci sont exécutés en espace utilisateur (sous forme de conteneurs Docker) ou dans le noyau Linux (au point d'exécution de XDP) grâce à notre mécanisme de délocalisation. Comme attendu, les débits de traitement atteints dans le second cas sont quatre à six fois plus élevés (voir Figure 4.5).

Conclusion

Cette thèse ouvre la voie à de nombreuses autres problématiques de recherche. Tout d'abord, parce qu'ils représentent aujourd'hui la principale implémentation des réseaux multi-tenants, nous nous sommes intéressés aux réseaux de datacenters typiques de l'informatique en nuage. L'informatique en nuage a cependant tendance à s'étendre vers les Point de Présence ("Points of Presence" ou PoPs en anglais) et équipements de plus faibles capacités à la limite des réseaux. De part leurs ressources plus contraintes, ces équipements pourraient bénéficier des optimisations proposées dans cette thèse. Cela nécessiterait cependant de remplacer leurs mécanismes d'accélération matérielle par des optimisations logicielles. Il n'existe aujourd'hui pas d'études de l'applicabilité de ces optimisations logicielles à un contexte à fortes contraintes en ressources, tels que les routeurs domestiques.

Les deux prototypes développés dans le cadre de cette thèse reposent sur le bytecode BPF et un programme d'analyse statique, le *verifier*, pour assurer l'isolation. Le *verifier* a été développé pour répondre à une contrainte particulière : celle de réduire au minimum le coût à l'exécution de l'isolation. Cette analyse statique passe par un parcours exhaustif de toutes les possibilités d'exécution des programmes vérifiés, une approche qui passe difficilement à l'échelle et ne repose pas sur des fondations formelles. Il nous faut donc trouver de nouvelles approches pour la vérification pour permettre le développement de programme BPF de tailles plus conséquentes. De plus, les techniques et outils de vérification et de génération automatique de tests (*fuzzing*) existants sont inadaptés aux interfaces et au contexte d'exécution de BPF. L'amélioration de ces techniques est donc nécessaire pour permettre de tester l'ensemble du code permettant l'exécution des programmes BPF.

Finalement, nous avons débattu dans cette thèse la possibilité d'améliorer les performances de traitement des paquets sans avoir recours à des accélérateurs matériels spécialisés et inflexibles. Nous avons cependant observé au Chapitre 2 que l'utilisation de ces équipements est, à terme, inévitable. L'utilisation de ces équipements posent cependant de nouvelles questions quant à leurs méthodes de programmation : chaque type d'équipement (RMT [20], FPGA, NPU, CPU, etc.) a aujourd'hui sa propre interface ou son propre langage permettant de le programmer. Si quelques langages dédiés émergent pour permettre de programmer divers types d'équipements (ex. BPF pour CPU et NPU ; P4 pour FPGA et RMT), il n'existe pas encore d'abstraction commune qui éviterait à un développeur réseau d'avoir à maîtriser autant de langages et d'interfaces qu'il n'y a d'accélérateurs matériels dans son réseau.

Bibliography

[1] *ab - Apache HTTP server benchmarking tool*. URL: `http://httpd.apache.org/docs/2.4/programs/ab.html`.

[2] V. Addanki et al. "Controlling software router resource sharing by fair packet dropping". In: *Proc. IFIP Networking*. 2018.

[3] F. Anhalt and P. Primet. *Analysis and evaluation of a XEN based virtual router*. Research Report RR-6658. INRIA, 2008.

[4] J. Ansel et al. "Language-independent sandboxing of just-in-time compilation and self-modifying code". In: *Proc. ACM SIGPLAN PLDI*. 2011.

[5] B. Anwer et al. "A slick control plane for network middleboxes". In: *Proc. ACM SIGCOMM HotSDN*. 2013.

[6] *API-aware networking and security using BPF and XDP*. Dec. 2015. URL: `https://github.com/cilium/cilium`.

[7] M. Armbrust et al. "A View of Cloud Computing". In: *Commun. ACM* (2010).

[8] J.-P. Aumasson and D. J. Bernstein . "SipHash: a fast short-input PRF". In: *International Conference on Cryptology in India*. Springer. 2012.

[9] F. Baboescu and G. Varghese. "Scalable Packet Classification". In: *Proc. ACM SIGCOMM*. 2001.

[10] G. Banga, P. Druschel, and J. C. Mogul. "Resource containers: A new facility for resource management in server systems". In: *Proc. USENIX OSDI*. 1999.

[11] T. Barbette, C. Soldani, and L. Mathy. "Fast userspace packet processing". In: *Proc. IEEE ANCS*. 2015.

[12] P. Barham et al. "Xen and the art of virtualization". In: *Proc. ACM SOSP*. 2003.

[13] A. Belay et al. "IX: A protected dataplane operating system for high throughput and low latency". In: *Proc. USENIX OSDI*. 2014.

[14] B. N. Bershad et al. "Extensibility, safety, and performance in the SPIN operating system". In: *Proc. ACM SOSP*. 1995.

[15] G. Bertin. *Introducing the p0f BPF compiler*. Aug. 2016. URL: `https://blog.cloudflare.com/introducing-the-p0f-bpf-compiler`.

[16] M. Bertrone et al. "Accelerating Linux security with eBPF iptables". In: *Proc. ACM SIGCOMM Posters and Demos*. 2018.

[17] *BESS: Berkeley extensible software switch*. URL: `https://github.com/NetSys/bess`.

[18] Z. Bloom. *Cloud computing without containers*. 2018. URL: `https://blog.cloudflare.com/cloud-computing-without-containers`.

[19] D. Borkmann. *net: add bpfilter*. Feb. 2018. URL: `https://lwn.net/Articles/747504`.

[20] P. Bosshart et al. "Forwarding Metamorphosis: Fast programmable match-action processing in hardware for SDN". In: *Proc. ACM SIGCOMM*. 2013.

[21] P. Bosshart et al. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM Comput. Commun. Rev.* (2014).

[22] L. Braun et al. "Comparing and improving current packet capturing solutions based on commodity hardware". In: *Proc. ACM IMC*. 2010.

[23] A. Bremler-Barr, Y. Harchol, and D. Hay. "OpenBox: A software-defined framework for developing, deploying, and managing network functions". In: *Proc. ACM SIGCOMM*. 2016.

[24] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. "Dynamic instrumentation of production systems". In: *Proc. USENIX ATC*. 2004.

[25] A. Cardigliano et al. "vPF_RING: Towards wire-speed network monitoring using virtual machines". In: *Proc. ACM IMC*. 2011.

[26] M. Casado et al. "Rethinking packet forwarding hardware". In: *Proc. ACM HotNets*. 2008.

[27] P. Chaignon et al. "Oko: Extending Open vSwitch with stateful filters". In: *Proc. ACM SOSR*. 2018.

[28] J. S. Chase, A. J. Gallatin, and K. G. Yocum. "End system optimizations for high-speed TCP". In: *Comm. Mag.* (2001).

[29] B. Chen and R. Morris. "Flexible control of parallelism in a multiprocessor PC router". In: *Proc. USENIX ATC*. 2001.

[30] J. Corbet. *BPF: The universal in-kernel virtual machine*. May 2014. URL: `https://lwn.net/Articles/599755`.

[31] J. Corbet. *Linux and TCP offload engines*. 2005. URL: `https://lwn.net/Articles/148697/`.

[32] P. Cousot and R. Cousot. "Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proc. ACM SIGPLAN POPL*. 1977.

[33] M. Dalton et al. "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization". In: *USENIX NSDI*. 2018.

[34] B. Davie and J. Gross. *A stateless transport tunneling protocol for network virtualization (STT)*. 2012. URL: `https://tools.ietf.org/html/draft-davie-stt-01`.

[35] B. Davie et al. "A database approach to SDN control plane design". In: *SIGCOMM Comput. Commun. Rev.* (2017).

[36] M. Dobrescu et al. "RouteBricks: Exploiting parallelism to scale software routers". In: *Proc. ACM SOSP*. 2009.

[37] *DPDK*. 2011. URL: `http://dpdk.org`.

[38] D. Drake and J. Berg. *Unaligned Memory Access - Kernel.org*. 2008. URL: `https://www.kernel.org/doc/Documentation/unaligned-memory-access.txt`.

[39] P. Druschel and G. Banga. "Lazy receiver processing (LRP): A network subsystem architecture for server systems". In: *Proc. USENIX OSDI*. 1996.

[40] D. E. Eisenbud et al. "Maglev: A fast and reliable software network load balancer". In: *Proc. USENIX NSDI*. 2016.

[41] P. Emmerich et al. "MoonGen: A scriptable high-speed packet generator". In: *Proc. ACM IMC*. 2015.

[42] *ENA poll mode driver - Documentation - DPDK*. URL: `https://doc.dpdk.org/guides/nics/ena.html`.

[43] *ENIC poll mode driver - Documentation - DPDK*. URL: https://doc.dpdk.org/guides/nics/enic.html.

[44] Ú. Erlingsson et al. "XFI: Software guards for system address spaces". In: *Proc. USENIX OSDI*. 2006.

[45] J. Fastabend and J. Wang. *bpf, bounded loop support work in progress*. 2018. URL: https://lwn.net/Articles/756284/.

[46] E. L. Fernandes. *OpenFlow 1.3 switch*. URL: http://cpqd.github.io/ofsoftswitch13/.

[47] S. Finucane. *DPDK vHost User ports*. 2019. URL: http://docs.openvswitch.org/en/latest/topics/dpdk/vhost-user.

[48] D. Firestone. "VFP: A virtual switch platform for host SDN in the public cloud". In: *Proc. USENIX NSDI*. 2017.

[49] D. Firestone, H. Katebi, and G. Varghese. *Virtual switch packet classification*. Tech. rep. MSR-TR-2016-66. 2016. URL: https://www.microsoft.com/en-us/research/publication/virtual-switch-packet-classification-2/.

[50] D. Firestone et al. "Azure accelerated networking: SmartNICs in the public cloud". In: *Proc. USENIX NSDI*. 2018.

[51] Open Networking Foundation. *OpenFlow Switch Specification Version 1.3.1 (Wire Protocol 0x04)*. 2012. URL: https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf.

[52] M. Gallo and R. Laufer. "ClickNF: A modular stack for custom network functions". In: *Proc. USENIX ATC*. 2018.

[53] S. Garzarella, G. Lettieri, and L. Rizzo. "Virtual device passthrough for high speed VM networking". In: *Proc. ACM/IEEE ANCS*. 2015.

[54] A. Gember-Jacobson et al. "OpenNF: Enabling innovation in network function control". In: *Proc. ACM SIGCOMM*. 2014.

[55] G. P. Georgios P. Katsikas et al. "Metron: NFV service chains at the true speed of the underlying hardware". In: *USENIX NSDI*. 2018.

[56] E. Gershuni et al. "Simple and precise static analysis of untrusted Linux kernel extensions". In: *Proc. ACM SIGPLAN PLDI*. 2019.

[57] M. Ghasemi, T. Benson, and J. Rexford. "Dapper: Data plane performance diagnosis of TCP". In: *Proc. ACM SOSR*. 2017.

[58] G. Gibb et al. "Design principles for packet parsers". In: *Proc. ACM/IEEE ANCS*. 2013.

[59] Google. *libFuzzer - a library for coverage-guided fuzz testing*. 2014. URL: https://llvm.org/docs/LibFuzzer.html.

[60] Google. *syzkaller: an unsupervised, coverage-guided kernel fuzzer*. 2015. URL: https://github.com/google/syzkaller.

[61] B. Gregg. "Linux 4.X tracing tools: Using BPF superpowers". In: USENIX LISA, 2016.

[62] J. Gross et al. *The Rise of Soft Switching*. 2011. URL: https://networkheresy.com/2011/06/14/the-rise-of-soft-switching-part-i-introduction-and-background.

[63] D. Gupta et al. "Enforcing performance isolation across virtual machines in Xen". In: *Proc. ACM/IFIP/USENIX Middleware*. 2006.

[64] P. Gupta and N. McKeown. "Packet classification on multiple fields". In: *Proc. ACM SIGCOMM*. 1999.

[65] A. Haas et al. "Bringing the web up to speed with WebAssembly". In: *Proc. ACM SIGPLAN PLDI*. 2017.

[66] J. Hamilton. *AWS Nitro system*. 2019. URL: https://perspectives.mvdirona.com/2019/02/aws-nitro-system.

[67] S. Han et al. *SoftNIC: A software NIC to augment hardware*. Tech. rep. UCB/EECS-2015-155. EECS Department, University of California, Berkeley, May 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html.

[68] T. Herbert. "UDP encapsulation in Linux". In: NetDev 0.1, 2015.

[69] P. Hickey. *Announcing Lucet: Fastly's native WebAssembly compiler and runtime*. 2019. URL: https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime.

[70] T. Høiland-Jørgensen et al. "The eXpress Data Path: Fast programmable packet processing in the operating system kernel". In: *Proc. ACM CoNEXT*. 2018.

[71] M. Honda et al. "mSwitch: A highly-scalable, modular software switch". In: *Proc. ACM SOSR*. 2015.

[72] R. Huggahalli, R. Iyer, and S. Tetrick. "Direct cache access for high bandwidth network I/O". In: *Proc. ACM ISCA*. 2005.

[73] J. Hwang, K. K. Ramakrishnan, and T. Wood. "NetVM: High performance and flexible networking using virtualization on commodity platforms". In: *Proc. USENIX NSDI*. 2014.

[74] Jupyter Networks Inc. *Contrail Virtual Router*. URL: https://github.com/Juniper/contrail-vrouter.

[75] Pica8 Inc. *Pica8 XorPlus*. URL: https://sourceforge.net/p/xorplus/home/Pica8%20Xorplus/.

[76] European Telecommunications Standards Institute. *NFV Whitepaper*. 2012. URL: https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[77] Intel. *How to set up Intel Ethernet Flow Director*. 2017. URL: https://software.intel.com/en-us/articles/setting-up-intel-ethernet-flow-director.

[78] Intel. *Intel VMDq technology overview*. 2008. URL: https://www.intel.com/content/www/us/en/virtualization/vmdq-technology-paper.html.

[79] Intel. *PCI-SIG SR-IOV primer*. 2011. URL: https://www.intel.sg/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf.

[80] E. J. Jackson et al. "SoftFlow: A middlebox architecture for Open vSwitch". In: *Proc. USENIX ATC*. 2016.

[81] S. Jain et al. "B4: Experience with a globally-deployed software defined WAN". In: *Proc. ACM SIGCOMM*. 2013.

[82] B. Jenkins. *A hash function for hash table lookup*. 2016. URL: http://burtleburtle.net/bob/hash/doobs.html.

[83] L. Jose et al. "Compiling packet programs to reconfigurable switches". In: *Proc. USENIX NSDI*. 2015.

[84] S. Jouet, R. Cziva, and D. Pezaros. *Programmable dataplane for next generation networks*. Mar. 2016. URL: https://netlab.dcs.gla.ac.uk/uploads/files/d99abd5bbadbed8c0f29808ee812bd26.pdf.

[85] S. Jouet and D. P. Pezaros. "BPFabric: Data plane programmability for software defined networks". In: *Proc. IEEE ANCS*. 2017.

[86] P. Kazemian, G. Varghese, and N. McKeown. "Header Space Analysis: Static checking for networks". In: *Proc. USENIX NSDI*. 2012.

[87] J. Kicinski and N. Viljoen. "eBPF/XDP hardware offload to SmartNICs". In: NetDev 1.2, 2016.

[88] C. Kim et al. "Revisiting route caching: The world should be flat". In: *Proc. PAM*. 2009.

[89] T. Koponen et al. "Network virtualization in multi-tenant datacenters". In: *Proc. USENIX NSDI*. 2014.

[90] T. Koponen et al. "Onix: A distributed control platform for large-scale production networks". In: *Proc. USENIX OSDI*. 2010.

[91] T. V. Lakshman and D. Stiliadis. "High-speed policy-based packet forwarding using efficient multi-dimensional range matching". In: *Proc. ACM SIGCOMM*. 1998.

[92] R. Lane. *Userspace eBPF VM*. Aug. 2015. URL: https://github.com/iovisor/ubpf.

[93] X. Leroy. "Java bytecode verification: Algorithms and formalizations". In: *J. Autom. Reason.* (2003).

[94] B. Li et al. "KV-Direct: High-performance in-memory key-value store with programmable NIC". In: *Proc. SOSP*. 2017.

[95] *Linux Socket Filtering aka Berkeley Packet Filter (BPF)*. 2005. URL: https://www.kernel.org/doc/Documentation/networking/filter.txt.

[96] Z. Liu et al. "One sketch to rule them all: Rethinking network flow monitoring with UnivMon". In: *Proc. ACM SIGCOMM*. 2016.

[97] J. Martins et al. "ClickOS and the art of network function virtualization". In: *Proc. USENIX NSDI*. 2014.

[98] S. McCamant and G. Morrisett. "Evaluating SFI for a CISC architecture". In: *Proc. USENIX Security*. 2006.

[99] S. Mccanne and V. Jacobson. "The BSD packet filter: A new architecture for user-level packet capture". In: *Proc. USENIX Winter Conf.* 1993.

[100] N. McKeown et al. "OpenFlow: Enabling innovation in campus networks". In: *SIGCOMM Comput. Commun. Rev.* (2008).

[101] H. Mekky et al. "Application-aware data plane processing in SDN". In: *Proc. ACM SIGCOMM HotSDN*. 2014.

[102] H. Mekky et al. "Network function virtualization enablement within SDN data plane". In: *IEEE INFOCOM*. 2017.

[103] A. Menon, A. L. Cox, and W. Zwaenepoel. "Optimizing network virtualization in Xen". In: *Proc. ATC*. 2006.

[104] J. Mogul, R. Rashid, and M. Accetta. "The Packer Filter: An efficient mechanism for user-level network code". In: *Proc. ACM Symp. Oper. Syst. Principles*. 1987.

[105] J. C. Mogul. "TCP offload is a dumb idea whose time has come". In: *Proc. HotOS*. 2003.

[106] L. Molnár et al. "Dataplane specialization for high-performance OpenFlow software switching". In: *Proc. ACM SIGCOMM*. 2016.

[107] R. Morris et al. "The Click modular router". In: *Proc. ACM SOSP*. 1999.

[108] R. Nakamura, Y. Sekiya, and H. Tazaki. "Grafting sockets for fast container networking". In: *Proc. ANCS*. 2018.

[109] G. C. Necula and P. Lee. "Safe kernel extensions without run-time checking". In: *Proc. USENIX OSDI*. 1996.

[110] P. Newman, G. Minshall, and T. L. Lyon. "IP switching—ATM under IP". In: *IEEE/ACM Trans. Netw.* (1998).

[111] R. Niranjan Mysore, G. Porter, and A. Vahdat. "FasTrak: Enabling express lanes in multi-tenant data centers". In: *Proc. ACM CoNEXT*. 2013.

[112] Z. Niu et al. "Network stack as a service in the cloud". In: *Proc. ACM HotNets*. 2017.

[113] *OpenDaylight project*. Feb. 2013. URL: `https://www.opendaylight.org`.

[114] A. Panda et al. "NetBricks: Taking the V out of NFV". In: *Proc. USENIX OSDI*. 2016.

[115] S. Peter et al. "Arrakis: The operating system is the control plane". In: *Proc. USENIX OSDI*. 2014.

[116] J. Pettit et al. "Bringing Platform Harmony to VMware NSX". In: *SIGOPS Oper. Syst. Rev.* (2018).

[117] J. Pettit et al. "Virtual switching in an era of advanced edges". In: *Proc. DCCAVES*. 2010.

[118] B. Pfaff. "Converging approaches in software switches". In: ACM APSys, 2016.

[119] B. Pfaff et al. "Extending networking into the virtualization layer". In: *Proc. ACM HotNets*. 2009.

[120] B. Pfaff et al. "The design and implementation of Open vSwitch". In: *Proc. USENIX NSDI*. 2015.

[121] *Poll mode driver for paravirtual VMXNET3 NIC - Documentation - DPDK*. URL: `https://doc.dpdk.org/guides/nics/vmxnet3.html`.

[122] J.-E. Rediger. "Network function offloading in virtualized environments". MA thesis. RWTH Aachen University, 2017.

[123] L. Rizzo. "Netmap: A novel framework for fast packet I/O". In: *Proc. USENIX ATC*. 2012.

[124] L. Rizzo and G. Lettieri. "VALE, a switched Ethernet for virtual machines". In: *Proc. CoNEXT*. 2012.

[125] R. Russell. "Virtio: Towards a de-facto standard for virtual I/O devices". In: *SIGOPS Oper. Syst. Rev.* (2008).

[126] S. Sanfilippo. *Hping - Active network security tool*. 2006. URL: `http://www.hping.org/`.

[127] J. R. Santos et al. "Bridging the gap between software and hardware techniques for I/O virtualization". In: *Proc. USENIX ATC*. 2008.

[128] M. Shahbaz et al. "PISCES: A programmable, protocol-independent software switch". In: *Proc. ACM SIGCOMM*. 2016.

[129] N. Shelly et al. "Flow caching for high entropy packet fields". In: *Proc. ACM SIGCOMM HotSDN*. 2014.

[130] J. Sherry. "Middleboxes as a Cloud Service". PhD thesis. University of California at Berkeley, 2016.

[131] S. Singh et al. "Packet classification using multidimensional cutting". In: *Proc. ACM SIGCOMM*. 2003.

[132] M. Sipser. *Introduction to the Theory of Computation*. 1st. International Thomson Publishing, 1996. ISBN: 053494728X.

[133] A. Sivaraman et al. "Packet Transactions: High-level programming for line-rate switches". In: *Proc. ACM SIGCOMM*. 2016.

[134] J. Sonchack et al. "Enabling practical software-defined networking security applications with OFX". In: *NDSS*. 2016.

[135] Y. Song. *fuzzing framework based on libfuzzer and clang sanitizer*. 2015. URL: https://github.com/iovisor/bpf-fuzzer.

[136] V. Srinivasan, S. Suri, and G. Varghese. "Packet classification using Tuple Space Search". In: *Proc. ACM SIGCOMM*. 1999.

[137] V. Srinivasan et al. "Fast and scalable layer four switching". In: *Proc. ACM SIGCOMM*. 1998.

[138] J. Tan et al. "VIRTIO-USER: A new versatile channel for kernel-bypass networks". In: *Proc. KBNets*. 2017.

[139] D. E. Taylor. "Survey and taxonomy of packet classification techniques". In: *ACM Comput. Surv.* (2005).

[140] *The CAIDA anonymized OC48 Internet traces 2002-2003 dataset*. 2012. URL: http://data.caida.org/datasets/passive/passive-oc48.

[141] C.-C. Tu and A. Starovoitov. *Re: [RFC PATCH 00/11] OVS eBPF datapath*. 2018. URL: https://lists.iovisor.org/g/iovisor-dev/message/1359.

[142] C.-C. Tu, J. Stringer, and J. Pettit. "Building an extensible Open vSwitch datapath". In: *ACM SIGOPS Oper. Syst. Rev.* (2017).

[143] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. "EffiCuts: Optimizing packet classification for memory and throughput". In: *Proc. ACM SIGCOMM*. 2010.

[144] *Vector packet processing (VPP)*. 2017. URL: https://fd.io/technology/#vpp.

[145] R. Wahbe et al. "Efficient software-based fault isolation". In: *Proc. ACM SOSP*. 1993.

[146] H. Wang et al. "P4FPGA: A rapid prototyping framework for P4". In: *Proc. ACM SOSR*. 2017.

[147] X. Wang et al. "Jitk: A trustworthy in-kernel interpreter infrastructure". In: *Proc. USENIX OSDI*. 2014.

[148] B. Yee et al. "Native Client: A sandbox for portable, untrusted x86 native code". In: *Proc. IEEE S&P*. 2009.

[149] M. Yu, L. Jose, and R. Miao. "Software defined traffic measurement with OpenSketch". In: *Proc. USENIX NSDI*. 2013.

[150] M. Zalewski. *p0f v3*. 2012. URL: http://lcamtuf.coredump.cx/p0f3.

[151] Michał Zalewski. *american fuzzy lop*. 2013. URL: http://lcamtuf.coredump.cx/afl.

[152] W. Zhang et al. "OpenNetVM: A platform for high performance network service chains". In: *Proc. HotMiddlebox*. 2016.

Résumé

En environnement multi-tenant, les réseaux s'appuient sur un ensemble de ressources matérielles partagées pour permettre à des applications isolés de communiquer avec leurs clients. Cette isolation est garantie par un ensemble de mécanismes à la bordure des réseaux : les mêmes serveurs hébergeant les machines virtuelles doivent notamment déterminer le destinataire approprié pour chaque paquet réseau, copier ces derniers entre zones mémoires isolées et supporter les tunnels permettant l'isolation du trafic lors de son transit sur le cœur de réseau. Ces différentes tâches doivent être accomplies avec aussi peu de ressources matérielles que possible, ces dernières étant tout d'abord destinées aux machines virtuelles. Dans un contexte d'intensification de la demande en haute performance sur les réseaux, les acteurs de l'informatique en nuage ont souvent recours à des équipements matériels spécialisés mais inflexibles, leur permettant d'atteindre les performances requises. Néanmoins, dans cette thèse, nous défendons la possibilité d'améliorer les performances significativement sans avoir recours à de tels équipements.

Nous prônons, d'une part, une consolidation des fonctions réseaux au niveau de la couche de virtualisation et, d'autre part, une relocalisation de certaines fonctions réseaux hors des machines virtuelles. À cette fin, nous proposons Oko, un commutateur logiciel extensible qui facilite la consolidation des fonctions réseaux dans la couche de virtualisation. Oko étend les mécanismes de l'état de l'art permettant une mise en cache des règles de commutateurs, ceci afin de permettre une exécution des fonctions réseaux sous forme d'extensions au commutateur. De plus, les extensions sont isolées du cœur du commutateur afin d'empêcher des fautes dans les extensions d'impacter le reste du réseau et de faciliter une mise en place rapide et sûre de nouvelles fonctions réseaux. En permettant aux fonctions réseaux de s'exécuter au sein du commutateur logiciel, sans redirections vers des processus distincts, Oko diminue de moitié le coût lié à l'exécution des fonctions réseaux en moyenne.

Notre seconde contribution vise à permettre une exécution de certaines fonctions réseaux en amont des machines virtuelles, au sein de la couche de virtualisation. L'exécution de ces fonctions réseaux hors des machines virtuelles permet d'importants gains de performance, mais lèvent des problématiques d'isolation. Nous réutilisons et améliorons la technique utilisé dans Oko pour isoler les fonctions réseaux et l'étendons avec un mécanisme de partage équitable du temps CPU entre les différentes fonctions réseaux relocalisées.

Mots-clés : réseau programmable, informatique des nuages, traitement des paquets, NFV, SDN

Abstract

Multi-tenant networks enable applications from multiple, isolated tenants to communicate over a shared set of underlying hardware resources. The isolation provided by these networks is enforced at the edge: end hosts demultiplex packets to the appropriate virtual machine, copy data across memory isolation boundaries, and encapsulate packets in tunnels to isolate traffic over the datacenter's physical network. Over the last few years, the growing demand for high performance network interfaces has pressured cloud providers to build more efficient multi-tenant networks.

While many turn to specialized, hard-to-upgrade hardware devices to achieve high performance, in this thesis, we argue that significant performance improvements are attainable in end-host multi-tenant networks, using commodity hardware. We advocate for a consolidation of network functions on the host and an offload of specific tenant network functions to the host. To that end, we design Oko, an extensible software switch that eases the consolidation

of network functions. Oko includes an extended flow caching algorithm to support its runtime extension with limited overhead. Extensions are isolated from the software switch to prevent failures on the path of packets. By avoiding costly redirections to separate processes and virtual machines, Oko halves the running cost of network functions on average.

We then design a framework to enable tenants to offload network functions to the host. Executing tenant network functions on the host promises large performance improvements, but raises evident isolation concerns. We extend the technique used in Oko to provide memory isolation and devise a mechanism to fairly share the CPU among offloaded network functions with limited interruptions.

Keywords: programmable network, cloud, packet processing, NFV, SDN