Network Function Offloading in Virtualized Environments

Master Thesis

Jan-Erik Rediger

This work was submitted to the

Chair of Communication and Distributed Systems

RWTH Aachen University, Germany

Advisers:

Johannes Krude, M. Sc.
Jan Rüth, M. Sc.

Examiners:

Prof. Dr.-Ing. Klaus Wehrle
Prof. Dr.-Ing. Ulrike Meyer

Registration date: 2017-01-05
Submission date: 2017-07-05

Kurzfassung

Die Forschung von Netzwerkfunktionsvirtualisierung erforscht die Möglichkeit der Platzierung von Code in Netzwerken. Solche Netzwerkfunktionen können dedizierte Hardwaregeräte, wie Firewalls, Loadbalancer oder Video-Transcoder, ersetzen. Software-Implementierung von Netzwerkfunktionen sollte existierende Rechenressourcen verwenden können, welche oft als virtualisierte Systeme angeboten werden, bei denen sich mehrere Gäste eine physische Maschine teilen. Der derzeitige Netzwerk-Stack solcher virtualisierten Systeme benötigt mehrere teure Operationen um Gastsystemen Netzwerkfunktionalität bereitzustellen. Durch Umgehung des Stacks und dem Ausladen von Netzwerkfunktionen direkt in die Hostmaschine kann die Performance deutlich verbessert werden.

In dieser Arbeit präsentieren wir `XenBPF`, ein Framework, welches es virtuellen Gastmaschinen in Xen erlaubt, Netzwerkfunktionen in die Hostmaschine auszulagern. Spezifische Aufgaben der Netzwerkfunktionen können in BPF implementiert werden und mittels `XenBPF` können diese Programme als Netzwerkfilter an der Netzwerkkarte des Hostsystems genutzt werden. Wir evaluieren die Performance dieses Ansatzes in mehreren Benchmarks, wobei wir die Latenz sowie den Netzwerkdurchsatz messen, wenn Netzwerkpakete bereits im Hostsystem behandelt werden. Darüber hinaus führen wir zwei Fallstudien existierender Programme durch und zeigen wie `XenBPF` in existierende Applikationen integriert werden kann und welche Auswirkungen es dort hat.

Abstract

Research in network function virtualization explores the opportunities of placing code into the network. Such network functions may replace special purpose hardware appliances such as firewall, load balancers, and video transcoders. Network functions should be able to reuse existing computing resources, often offered as virtualized systems sharing a single physical hardware machine with other guests. The current network stack in these virtualized environments involves costly operations to provide networking to all guest systems. By bypassing this network stack and offloading network functions into the host, performance can be improved significantly.

In this thesis we present `XenBPF`, a framework that allows virtual guest machines to offload network functions into the host machine. Specific tasks of network functions can be implemented as small applications and can be attached directly to the host's Network Interface Card through `XenBPF`. We evaluate the performance of our implementation in several benchmarks, measuring latency and effect on network throughput when network packets are filtered early. We also conduct two case studies of existing applications to show the integration of `XenBPF` into these applications and the impact it can have.

Acknowledgments

First, I greatly thank my advisers Johannes Krude and Jan Rüth for the productive discussions and great assistance during the entire time.

I wish to thank Professor Klaus Wehrle and Professor Ulrike Meyer for their supervision.

I also want to thank my family and friends for their support and the interest they have shown, even though they did not really understand the topic of this thesis.

Contents

1

Introduction

Research in network function virtualization explores the opportunities of placing code into the network. Such network functions may replace special purpose hardware appliances such as firewall, load balancers, and video transcoders. When these network functions are implemented as regular network applications, they can be deployed on to already existing computing resources. Cloud providers offer computing resources and allow for easy scaling and on-demand deployment. Most of these computing resources are available as virtual machines, sharing a single physical hardware machine between multiple guest systems. Scheduling and resource assignment is handled by a hypervisor below. Virtual machines running on top of a hypervisor often cannot achieve the same performance for network operations as an application running on a bare-metal machine. The isolation properties of the virtualization platform require costly operations such as copying in order to transfer network packets to virtual machines. In order to use existing hypervisors as the platform for network functions, the networking performance must be increased. Instead of improving the network stack of a hypervisor, the network stack can be bypassed and network functions can be offloaded into the host and executed there directly. Offloading network functions from virtual machines into the networking stack of the host machine may reduce the amount of copying through filtering out unwanted packets or responding to frequently occurring requests. The Linux kernel already provides an execution environment for memory-safe sandboxed programs. The existing runtime environment for extended Berkeley Packet Filters (eBPF) allows user-space processes to inject code into the Linux kernel. When network access to virtual machines is provided by a privileged virtual machine running Linux, eBPF is a suitable candidate as a runtime environment for offloaded network functions.

1.1 Approach

Cloud providers already provide vast amounts of computing resources. Most of today's general-purpose computing needs can be fulfilled by the available platforms.

However, deploying network functions on such systems is often not possible, due to the higher requirements for networking performance.

If the networking performance in these platforms can be improved, network functions could more easily be deployed there and scaled on-demand. Improvements to the way networking works for guest systems is necessary. Instead of directly tackling the performance issues of the current networking stack, we want to bypass as much of the abstraction layers as possible and apply network functions as early as possible. The idea is to attach network functions early in the lifetime of network packets coming in from the network device. To achieve this, we need to provide an environment where network functions can be safely deployed and executed.

We are focusing on Xen as the hypervisor platform. It is an open-source project and widely used as the basis for cloud hosting platforms. Extensive research on Xen has been conducted to improve all layers of the platform, but certain bottlenecks remain. We identify these bottlenecks and provide a way to bypass the guest domain networking by offloading guest-supplied programs to the host. These programs use the existing runtime environment for BPF in the Linux kernel to guarantee memory-safety and sandboxing.

1.2 Challenges

Offloading user-provided functionality from guest machines to the host machine imposes a few challenges on the design and implementation of the framework. Running guest-supplied code in a privileged environment always poses a risk to the integrity of the running system. Virtual Machines (VMs) in a shared environment need to be considered as potential attackers. One compromised VM could at worst compromise the security of other VMs running and expose secret data or user data available on the host. A less dangerous, but still unwanted scenario is if an attacker deliberately or a user application unintentionally occupies most or all of the available resources, e.g. CPU time or network bandwidth, leading to an unfair situation where resource demands cannot be met and other applications come to a halt. Therefore, we need to ensure code provided by guests can safely be executed in the host system. By bypassing the existing abstractions provided by the system and passing arbitrary data from guest to hosts, we open up new potential attack vectors. As the host has no detailed control over the guest systems, besides restricting their resource usage, communication with guest domains must be robust against misuse.

The execution of programs on every incoming packet needs to be efficient to not reduce packet processing performance. We also need to avoid additional data copying as best as possible. The communication with guest systems and installation of guest-supplied programs as network functions into the network processing pipeline is not in the hot path and its performance should not impact the actual processing.

The full design of the Network Function Offloading Framework needs to take the potential security issues and concerns regarding fairness and performance into account.

1.3 Goals

The execution of network functions in a virtual machine needs to deliver performance comparable to the existing dedicated hardware appliances. The current network stack of virtualization platforms cannot fully deliver that at the moment. We therefore focus on performance improvements for network functions running in virtual machines. We implemented a framework consisting of an application running with privileged access on a host machine and a user-space API to handle the installation and usage of network functions from guest machines. With the requirements and challenges in mind, we can formulate the following goals:

Improve performance of packet handling: In general, the processing time for network traffic needs to be reduced significantly to make this a worthwhile approach.

Reduction of CPU usage: The overall used CPU time to handle network packets should be reduced.

Security in mind: With our newly implemented framework we bypass existing security layers of the virtualization platform and pass arbitrary data between guest machines and a privileged application with direct access to the network card. We must ensure that malicious guests are unable to crash any part of the system or get access to data not directed to their own system.

Ensure fairness of packet handling: We need to ensure that the offloaded packet processing is correctly accounted for the initiating guest with regards to its assigned resources. Scheduling and CPU times should be fairly distributed across running guest systems, even when a single guest has a larger amount of network traffic to be processed.

Minimal API changes: The provided user-space API should be similar enough to the existing kernel interface of BPF to allow applications, which use BPF programs to offload packet handling into the kernel, to be adopted to the new mechanism with minimal changes.

1.4 Structure

This thesis is structured as follows. First, we cover the necessary background for this thesis in Chapter 2. In Chapter 3, we present the problem of Network Functions Virtualization in a virtualized environment, identify the current bottlenecks of the networking in this environment and discuss the security issues an offloading approach has. Chapter 4 describes the design of the network function offloading solution and how the security issues discussed previously can be prevented. Chapter 5 will then present the implementation in more detail. Chapter 6 evaluates the performance of the approach using several different benchmarks. It also describes the integration into existing projects. Chapter 7 will briefly discuss related work. Finally, Chapter 8 summarizes the results of this thesis and provides an outlook on future work.

2

Background

This chapter explains the necessary background for this thesis. In this thesis several different technologies are combined in order to build one large framework that can be used to offload network functions in a virtualized operating system. We first describe Network Functions Virtualization, the underlying idea of what we want to achieve. In Section 2.2, we then describe BPF, the in-kernel virtual machine, used to execute guest-supplied programs. This technology can be used to implement network functions. Finally, we introduce the concept of Virtual Machine Monitoring. We first give some background on general Virtual Machine Monitoring solutions and follow with more details of the used Virtual Machine Monitor called Xen.

2.1 Network Functions Virtualization

Research in network function virtualization explores the opportunities of placing code into the network. The general availability of general purpose computing resources make it desirable to reuse these resources for network functions as well.

Each network operator runs hardware and software to power their network. Hardware appliances in the network perform a dedicated task. A single block of network infrastructure with a specific task, well-defined boundaries and functional behavior is described as a Network Function [15]. While some tasks are implemented in software, up to today most network functions still run on dedicated hardware appliances. Higher level applications already run on general purpose systems. The ubiquitous availability of virtualized environments make deployment, scaling and resource usage for these types of applications easy. Network functions however remain a task for dedicated machines. Deploying new functionality or updated implementations requires not only new hardware, but also power and space in the data center to deploy this hardware.

General software implementations running on commodity servers could replace network functions running on dedicated hardware appliances. In combination with

virtualization technology it would be possible to rapidly deploy new functionality on existing hardware [10]. Support for different operating systems, existing software packages and APIs would ensure that existing application code and tooling could be reused [21].

The idea was pushed in 2012, when multiple Telecommunications Service Providers (TSP) collaborated to outline benefits and challenges for Network Functions Virtualization (NFV) [10]. Members of the European Telecommunications Standards Institute (ETSI) drive the development of standards and share experiences developing, implementing and deploying NFV [15].

While software implementations of network functions do not necessarily require to be run on virtualized platforms, the widespread availability of virtualization platforms makes it an interesting target to run on. The ongoing development in that area, the availability of virtualized resources and the increase in flexibility for deployment, resource allocation and energy efficiency are desirable goals for network function deployment as well [24]. When network functions are built and optimized to run on platforms offering virtualized systems, applications performing tasks of network functions could be deployed in public clouds and instantiated on demand, instead of requiring expensive hardware and space availability in the network operator's own data centers. Using general software implementations would also be a step in a direction where network functions are less coupled to proprietary providers of network equipment and would allow for more freely sharing functionality between network providers. It would be possible to open up parts of the network functionality to customers.

Our work builds on the ideas of Network Functions Virtualization and provides a framework to develop small network function applications and deploy them on regular Linux systems in a virtualized environment. We rely on available technologies in Linux intended for low-level network traffic handling, and a virtualization platform that is widely used. These are explained in the following sections.

2.2 Implementation of Network Functions

To write fast and efficient applications that perform the task of network functions, different approaches exist. One way is to write small applications that are executed on each received network packet and which can then read, modify, retransmit or drop these packets. This can be achieved with a technology available in the Linux kernel since the 1990s, which is now known as classic Berkeley Packet Filter (cBPF).

The classic Berkeley Packet Filter originated as an in-kernel packet filter on Unix systems [22]. It was introduced as a performant way to dynamically filter network packets in the kernel by inspecting the content and making decisions to let packets pass or to drop them. This allowed for efficient low-level filtering, so that only a subset of matching packets need to be handled in user-space for inspection or logging. Tools like *tcpdump* and *Wireshark* use BPF to filter packets sent and received by the system. Filtered packets can then be further analyzed and inspected in user-space.

In recent years, Berkeley Packet Filter (BPF) support in the Linux kernel was extended to be more powerful and useful in more scenarios. It is now also known as the

Extended Berkeley Packet Filter (eBPF). Initial support for a user-facing API to make use of extended BPF was merged into the Linux kernel in September 2014 [32] and was available in the 3.18 release the following December. The newly introduced `bpf` syscall can be used to perform a range of operations related to BPF. Since the 3.19 release of Linux it is possible to attach BPF programs to raw sockets. More functionality and attach points were added over time. Classic BPF is still supported in Linux, but is now internally translated into extended BPF, in order to only have one implementation in the kernel [33].

In the next section we explain what exactly the BPF environment provides in more detail.

2.2.1 The BPF Virtual Machine

BPF provides a memory-safe execution environment for program code with a guarantee to terminate quickly. These guarantees allow it to run natively in the kernel. In the following we will explain BPF in more detail.

The BPF environment consists of the specification of an instruction set, based on a register machine abstraction, the execution environment in the kernel, callable helper functions and BPF maps, which are explained later on. The BPF virtual machine has 10 usable registers as well as instructions to access memory. Instructions are modeled close to available hardware instructions, with the idea that BPF can be translated into machine instructions easily. Special instructions allow calls into a set of external helper functions to enable functionality that is not possible to write in BPF directly or otherwise not accessible. The kernel restricts which functions are available. Examples for available helper functions include getting time information, the recalculation of a network frame's checksum, or force retransmission of a network packet. Every BPF program always ends with a numeric return value. The meaning of this value depends on the use case.

BPF programs need to be loaded into the kernel using the `bpf` syscall, passing a sequence of bytecode instructions. From the user-space side a loaded BPF program is identified by a file descriptor. At load time, a type for the program needs to be specified. This type determines into which subsystems the program can be loaded and which external function calls in BPF are allowed. When loaded into the kernel, the kernel first does a static analysis run of the BPF program. This analysis ensures that BPF programs will eventually terminate and are memory safe. The analysis involves several checks. First, the size of the program is checked. Programs with more than 4096 instructions are rejected. This already limits the amount of work that can be performed in a program. The kernel then runs a verifier on the code. The verifier first builds a control flow graph of the code and uses it to check that the program does not contain loops or unreachable statements. Otherwise the program is rejected and not loaded. This ensures that the program will eventually terminate. In a second step the verifier runs a more sophisticated check on the code by simulating execution of the code and tracing all state changes of the stack and registers. During this simulation, it tracks that only initialized registers are read and operations are done with values of the correct type. It ensures that data is read only from verified memory locations and pointers are handled accordingly. The previous

checks ensure that the compiled program is safe to run in a kernel context. If the verifier finishes and finds no violations, the BPF code is compiled into the host architecture's machine code immediately. Currently, the kernel has Just-in-Time (JIT) compiler support for commonly used architectures such as `x86_64`, `ARM64` or `PowerPC`. For architectures, for which no JIT compiler inside the kernel is implemented yet, or when JIT compilation is turned off by the administrator, the BPF bytecode is stored as is and later run by a BPF interpreter. The compilation to native machine code is essential to get the best performance for BPF applications. Once loaded, a program needs to be attached to a subsystem to be executed. Different subsystems have different ways to do that. Whenever an event occurs, such as a network frame coming in from the NIC, the kernel runs an attached BPF program by executing the previously compiled code or running the interpreter on the stored BPF bytecode. In case of a network filter, the BPF program will get memory access to a buffer holding the network frame.

BPF bytecode can be produced in different ways. Writing the raw bytecode manually is possible, but for all but minimal examples unpractical. This is used to test the verifier or other BPF tooling. A more approachable way is to make use of a compiler. The LLVM compiler toolchain [20] comes with a backend for BPF. This allows a developer to write programs in a restricted set of C and compile it into an object file containing the BPF bytecode and bundle it into an ELF file. Tools for working with BPF as well as our own implementation can read ELF files and extract the BPF bytecode from it. The final ELF file for a program will also contain metadata about BPF maps, which can be used to construct those maps. BPF maps are explained in more detail in the next paragraph.

While cBPF programs were purely self-contained and computed a single result from parsing the passed packet, extended BPF gained the ability to keep state and share it with user-space applications. State can be stored in dedicated data structures. These data structures are named *BPF maps*. In the beginning only a hash map type was available. Additional data structures for BPF maps were implemented and can be used, including simple data structures like arrays or more sophisticated ones like prefix trees.

BPF maps are created using the `bpf` syscall. This must happen prior to loading the BPF program that wants to use them. There exist map types to store arbitrary data and other map types with special use cases. It is up to the user to choose the format of the keys and values in data maps. For maps that hold data the kernel will allocate memory to store a fixed number of elements. On the user-space side maps are identified by a file descriptor just as BPF programs. Operations on maps include adding, updating and deleting contained data. The BPF program itself and user-space applications in possession of a file descriptor to the map can read and write to it.

One special type of map in use is a *program map*. The underlying data structure is still a hash map, but they fulfill a special purpose and are specially handled in the kernel. As mentioned before, BPF programs cannot use loops or call arbitrary external functions and are limited in size. Using program maps it is possible to jump into other existing BPF programs. This is called a *tail-call*. When doing a tail-call, the kernel looks up another BPF program by its internal pointer stored in a program map and jumps into the associated code, leaving the passed context object

and the current stack intact. Using program maps it is thus possible to dynamically expand beyond the fixed limit of instructions for a single BPF program. However, as a security mechanism and to ensure eventual termination, the kernel imposes a limit of 32 nested tail-calls. Jumps can happen into BPF programs of the same type as the current executing one, as long as they are already loaded in the kernel.

With the knowledge about what the BPF environment provides and which functionality is available to developers, we now explain in detail how BPF programs can be used as network function implementations.

2.2.2 BPF Programs as Network Functions

While initially BPF was intended for low-level packet filtering and forwarding to user-space, its usage was extended heavily over the course of the last three years and continues to be improved with each Linux release.

Using a recent kernel, BPF programs can be used to inspect and probe low-level kernel functions as well as user-space applications. Additionally, it can be used for fine-grained and programmatic filtering, tracking, modification and forwarding of network packets as part of Linux tools such as the traffic control layer and firewall implementations such as `iptables`, or even offloading BPF programs into network device drivers or the hardware directly using the Express Data Path (XDP) [18]. XDP especially allows packet processing before the kernel needs to allocate memory to handle the incoming network frames. Support for XDP needs to be implemented in the driver directly and is already implemented for some of the most used network drivers in Linux. Some proprietary smart NICs can execute BPF programs directly [19].

The kernel's traffic control layer is fully programmable and allows all kinds of different filters and actions. It also has support for BPF programs [7, 8]. Integration into the network stack of Linux allows powerful programs implemented in BPF to perform tasks on each incoming network packet. The traffic control system provides a large amount of options to control network traffic. It can be used to shape, classify, police and schedule incoming and outgoing traffic. Scheduling is handled by queuing disciplines (*qdisc*), which are responsible for rearranging packets between input and output. By default the Linux kernel uses a FIFO (first-in first-out) scheduler. Packets are passed on unmodified in the order they arrive. With different queuing disciplines it is possible to further classify or filter network packets and run actions on them. Actions can then drop, forward or retransmit packets. In 2015, the Linux kernel's traffic control layer gained support to attach BPF programs as classifiers and actions [5, 6]. This functionality enables offloading implementations of network functions into the kernel. When executed, a BPF program gets access to a buffer containing the received network packet. Multiple helper functions for parsing and modifying a network packet are available when BPF programs are used as network actions. The program's return value determines further actions and can signal that the packet should be dropped immediately, was retransmitted to one of the available network interfaces, or should be passed on in the network stack.

In our implementation we will use the traffic control layer for its general availability in stable Linux releases. With XDP becoming available for more drivers in newer releases, it would be possible to adopt the approach to it as well.

2.3 Virtual Machine Monitoring

Implementing network functions in software is only half of the story. By using virtualized environments, network functions can be deployed on demand onto existing machines and scale usage as needed. In order to run multiple independent operating system instances on a single hardware machine, one can leverage virtualization [3]. The software providing the virtualization is called *hypervisor*, sometimes also called Virtual Machine Monitor (VMM).

A *bare-metal hypervisor* is the software layer on top of the hardware, which takes control of creating and scheduling Virtual Machines (VM). It runs with a higher privilege level than the guest operating systems it manages. Access to hardware, such as the CPU, Network Interface Cards (NIC) or other devices, as well as memory access is handled through the hypervisor. Virtual machines running on top of the hypervisor get access to virtualized devices. Hardware can be represented to guest machines in two different ways. In *full virtualization* a real device is simulated in software provided by the hypervisor. Operating systems can run unmodified and reuse existing hardware drivers. Simulation of a real hardware device is complex and significantly slower than access to the real device. Additionally, every new device needs an accompanying simulation implementation. The performance loss and complex implementation is obviously not optimal, especially for use cases with high performance requirements. Some hypervisors instead use a *paravirtualization* approach [36]. The machine and hardware available inside a virtual machine is not identical to the underlying machine, but similar enough to avoid larger changes to software interacting with these devices. Modifications to the guest operating system are required, especially for hardware drivers, but in turn offer improved performance. The hypervisor can expose necessary functionality to guests through the virtual device and handle privileged actions with the hardware directly. Depending on the hypervisor, it is possible to mix both approaches and use *paravirtualization* for devices, where drivers can be modified, and *full virtualization* for other devices in the guest machine.

One available implementation of a bare-metal hypervisor is *Xen*. It allows multiple operating systems to run on a single hardware machine and provides strict isolation between guest VMs, called *domains* in Xen. A special domain acts as the control and administration domain. It is often referred to as *dom0* or *host domain*. Hardware access is coordinated through a privileged *driver domain*. In default setups of Xen, *dom0* is responsible for this as well. Since Version 3.0 the Linux kernel has full support for all functionality provided by Xen. A Linux distribution can be used to act as the host domain running on top of Xen. A variety of operating systems can be run as guest domains, including Linux, multiple BSD variants and Windows. Both Linux and BSD systems can run in paravirtualized mode, but Windows requires full virtualization, as it cannot be modified. It is possible to use paravirtualized devices with Windows using special paravirtualized drivers. Figure 2.1 shows the structure of a machine running Xen, a host domain and two guest domains.

Xen is freely available under an open-source license. Some commercial users rely on Xen for their virtualization platforms. Commercial cloud providers such as Amazon's Elastic Compute Cloud (EC2) or Rackspace Cloud use it to provide virtualized

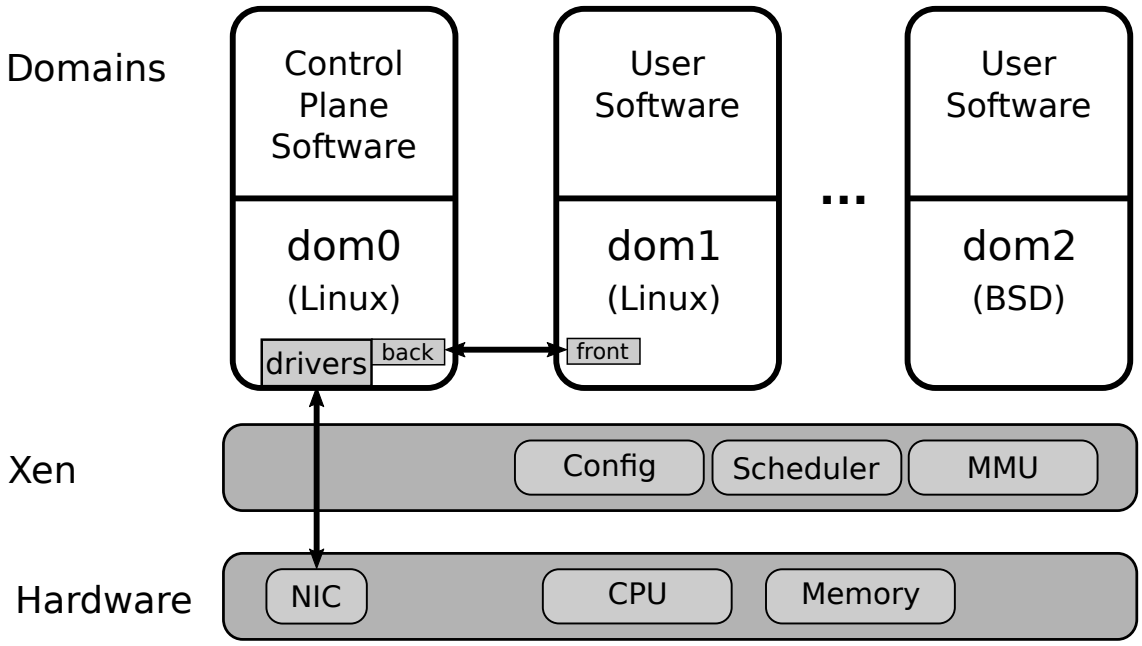


Figure 2.1 Structure of a machine running Xen. A small layer handles direct access to the CPU and memory. `dom0` runs control software and acts as the driver domain with direct access to devices. Other domains host virtualized operating systems. Device access is established by a paravirtualized driver connected to the driver domain.

machines to customers, though they run customized versions of Xen to support additional functionality [2].

As guest domains run in a less privileged mode than the host, they cannot perform every action on their own. To execute a privileged operation from user-space an application usually has to invoke a system call, short *syscall*. This causes a software trap that is handled by the kernel, which executes the operation and returns to the user-space application with a result. A similar mechanism exists in Xen to perform a synchronous operation in the hypervisor. It is executed through a *hypercall*, forcing a software trap handled by the hypervisor. Operations include things like scheduler operations or updating page tables [11]. Hypercalls are used in paravirtualized drivers to enable communication with the driver domain.

Sometimes communication between domains running on a host is necessary. Xen provides a mechanism to transfer or share memory pages between domains. This provides the basis for *inter-domain communication* and sharing larger amounts of data. Page mappings and access rights are stored in a *grant table*, handled by the hypervisor. Guest domains can invoke a hypercall to modify this grant table and ask the hypervisor to transfer or share one or more memory pages to another domain.

For communication between two domains, most often a ring buffer abstraction is used over shared memory. First, some memory pages are shared between the communicating domains using the above mentioned method. These memory pages are then used as a ring buffer. One domain becomes the producer, pointed to one end of the ring buffer, and the other domain will be the consumer, waiting for data. The producer copies data into the ring and the consumer moves it out again. Most of the implementation details are already provided by Xen. This functionality works between any two domains, even the host domain and a guest.

With a ring buffer over shared memory pages, data can be exchanged, but the receiving domain has no knowledge when new data is available. To avoid polling for changes, Xen provides another mechanism to inform domains about changes. This low-level mechanism of asynchronous notifications is provided by *event channels*. Event channels are similar to interrupts and do not carry any additional data besides the information that an event happened. Whenever the producer writes data into the ring, it schedules an event notification on the channel. When the receiving domain is scheduled and runs, it receives the notification and can read from the shared memory page. This allows for low-overhead, efficient event communication. The combination of these three mechanisms provides inter-domain communication.

To store configuration and status information and to communicate this information across domain boundaries, Xen offers a data store called *XenStore* [38]. Xen itself stores information about created and running guest domains in the *XenStore*, including configuration and status of devices associated with a domain, such as availability of virtual network interfaces and the configured MAC addresses. Configuration information for event channels is stored there as well and used by domains to setup the initial connection. The *XenStore* is exposed as a special virtual device to domains and offers a hierarchical view into the data. Access permissions for every layer of the XenStore can be configured. By default, *dom0* has full access to all data stored in XenStore, whereas unprivileged guest domains can only read their own entries. The permissions can be changed by the host domain to give read and write permissions to guest domains for specific parts. The default Xen package comes with tools and libraries to modify data stored in the XenStore. In our implementation we use this data store to discover guest domains and their configured network devices and provide configuration for the communication channel between a guest domain and the host domain.

Our approach for offloading network functions in virtualized environments applies to general VMM platforms. For the Proof-of-Concept implementation of this thesis, we use Xen as the virtualization platform. In our implementation we use the inter-domain communication features that are specific to Xen. For hypervisors that offer similar features, changes to our implementation would be required.

3

Problem Statement

In this chapter, we discuss the larger problem that is addressed with our work. We discuss how networking in Xen works and why the current way is a bottleneck when deploying network functions on Xen. We also discuss the impact on the security provided by a virtualization layer, how an offloading approach opens potential new attack vectors and how it needs to ensure the security of the offloading approach.

3.1 Problem Statement

The amount of network traffic increases every year. With it the number and size of attacks to networks, services and resources increases as well. Network operators need to dedicate more and more resources to mitigate these attacks [27].

Currently, tasks like DDoS mitigation are mostly performed by dedicated hardware appliances in the networks of larger providers. Hardware appliances provide efficient ways to handle network traffic, but requiring dedicated hardware appliances for every network function in a network makes it harder to change existing functionality, adopt new technology or scale the available resources. On the other hand, applications powering other services and websites use general-purpose processing resources. Most of the resources these days are provided by large cloud providers on their platforms [12]. However, simply deploying network functions on the existing resources in the cloud will not yield satisfiable results.

General-purpose computing resources in the cloud are most of the time offered on virtualization platforms. Multiple customers share resources on a single hardware system to operate their services and applications. Noisy neighbors, unequal distribution of workloads and different requirements with regards to processing resources make it hard to provide a consistent level of resources to every guest. In addition, the computing resources, such as the CPU, and hardware devices, such as Network Interface Cards, are shared between all guests and thus not directly accessible to the virtual machine. This often comes with a performance loss. Especially the network

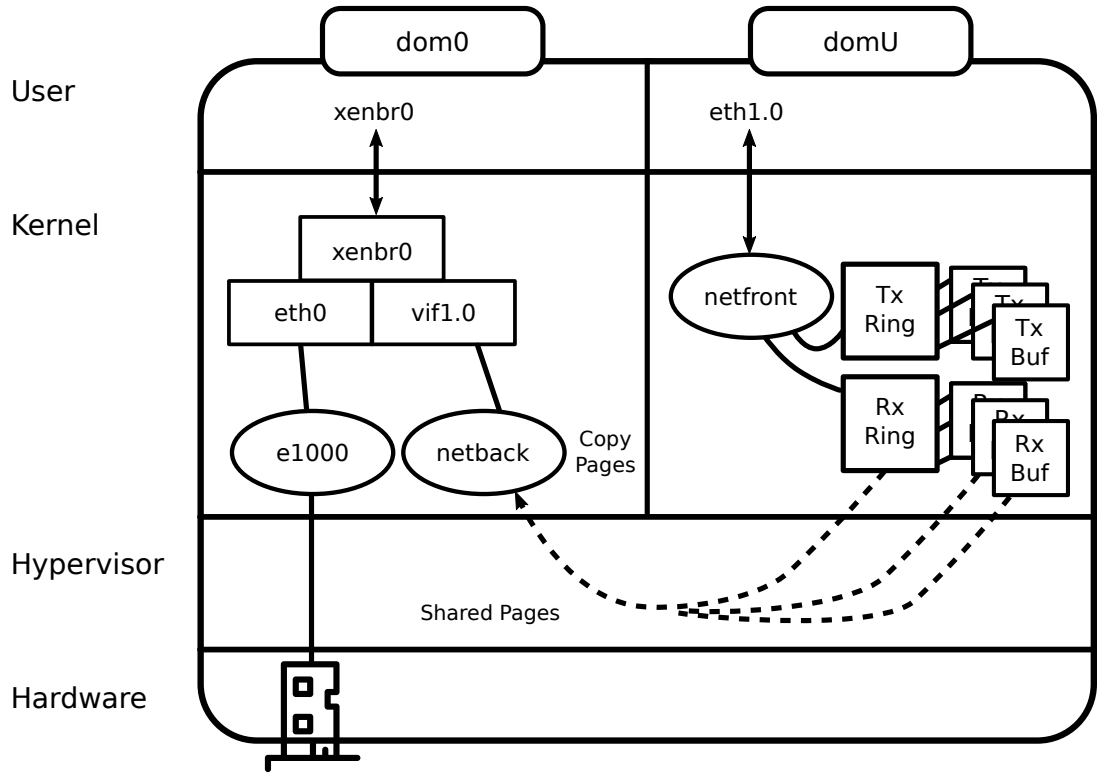


Figure 3.1 Incoming network packets are first handled by the privileged domain's kernel, passed over the software bridge and copied into shared pages with the unprivileged domain. From there the guest kernel processes the packets (adopted from "Network Throughput and Performance Guide" [37]).

performance is slower than with direct device access. To make virtualized platforms a target to run network functions on, these bottlenecks have to be reduced.

In this thesis we are working with the Xen hypervisor project as the platform for Network Functions. In the next section we take a closer look at how networking for guest domains works in Xen and identify the bottlenecks that make Xen currently an unsuitable platform for network function implementations with high performance requirements for the network communication. Rather than tackling the bottleneck of guest networking in Xen directly, we instead explore ideas to bypass most of the networking stack to improve the efficiency of network functions and offload small applications from guest machines into the host. This has potential security issues, due to bypassing already existing security mechanism of Xen. These security issues are discussed in Section 3.3.

Chapter 4 will then explain our offloading solution and how we mitigate the potential security issues to turn Xen into a platform for efficiently running implementations of network functions.

3.2 Guest Domain Networking under Xen

One of the main bottlenecks for network functions running inside virtual machines is the networking. Typically a host machine is equipped with at least one Network Interface Card (NIC) over which network traffic is sent out and received on. With

multiple guest systems running on that machine, access to the device needs to be coordinated. Direct access from guest machines to a NIC would block usage for other guest machines and thus direct access to the NIC from guest domains is prohibited. A single privileged domain handles incoming and outgoing network packets and distributes the traffic to the targeted guest domain. In case of Xen, in the default setup this is the task of `dom0`, but a driver domain could be used to separate privileges even further.

At creation time of guest domains, a virtual interface is created in the privileged domain. One part of the virtual interface is available in the guest domain. This interface is configured with an assigned MAC address. The most common way for Xen deployments to connect the virtual interface with the real interface is a software bridge. The bridge acts as a switch between the interfaces and will forward incoming traffic to virtual interfaces.

In order for guest domains to use this interface, a paravirtualized network driver is used. The device is represented by a special software interface similar to the real hardware interface. In Xen, this driver consists of two functional parts. The *backend* runs in the host domain and handles the virtual interface there, whereas the *frontend* driver is part of the guest domain and handles the virtual device in the guest. Both parts are connected and can communicate with each other through the Xen event channel mechanism and shared memory pages. This driver is natively available in Linux.

The process of handling incoming traffic is fairly complex and requires at least one copy of the data packet. Figure 3.1 visualizes this process. Whenever a network frame is received on the NIC, the kernel of `dom0` processes the frame and forwards it over the attached bridge, where it is received by the *backend* driver. The *backend* driver copies the packet data into a shared page and notifies the guest domain. At this point, the `dom0` yields the CPU. The Xen scheduler will wake up the next guest machine to run on the CPU. If the guest domain with waiting network frames has enough scheduler credit left, it is scheduled right away, otherwise it has to wait for its turn and other guest domains are scheduled first. When the guest domain is scheduled, it receives the notification and reads the new packet from its virtual interface. The packet is processed and handled in the kernel of the guest domain and eventually passed on to user-space. Considering the amount of network packets, this data copy is in part responsible for the reduced efficiency compared to direct access to the hardware.

All processing of network frames in guest domains only happens after parts of the frame are already parsed by the host's kernel in order to forward it to the right virtual interface. With our approach we will be able to apply packet processing and make decisions to drop or forward packets before a packet is passed over the network bridge and copied into the guest domain.

The current approach provides a clear separation of network packets directed to different guest domains on the same host. Packets are forwarded to the right virtual interface based on the target MAC address as given in the Ethernet header of the frame. When applying our network functions we use the same mechanism and execute guest-provided network functions based on the MAC address parsed from incoming packets.

In Xen all network I/O is primarily handled in `dom0` or another driver domain. For processing incoming or outgoing packets the driver domain is woken up and its on-CPU time is used to send packets or read them from the network card. This happens for all packets, including those directed to guest domains. The current scheduler in Xen has no knowledge about what the CPU time is used for and thus accounts this time to `dom0`'s assigned CPU share. Domains with a huge amount of network traffic can therefore effectively use more than their fair share of CPU time. Approaches to account for the additional CPU time for I/O to mitigate this effect to some extent were researched, but the implementation was not merged into the Xen project [14]. It was based on tracing page copies used for network packets and approximating the used CPU time for this.

When offloading programs to the host domain to process network packets, CPU time spent on behalf of guest domains will increase. Therefore, we want to measure the overhead and account the additional time spent in `dom0` for the guest domain receiving or sending network packets. In the next chapter we present a way to measure the overhead and discuss how this can be used to inform the scheduler.

3.3 Impact on Security

Virtualization is not only used to provide available resources to multiple guest systems, but also as a layer of security between these guests. It provides a clear boundary between virtual machines. Individual virtual machines on the same host should not interfere with each other. If one virtual machine is compromised, the controlling attacker should have no way to disrupt or exploit other machines on the same host by merely having control over a guest. This includes, but is not limited to, accessing, interrupting or controlling network traffic of other virtual machines or the whole host machine.

When offloading programs from guest machines into the host machine we open up new potential attack vectors that completely bypass the already existing security features of Xen. We need to carefully design the network function offloading to mitigate additional security issues when handling communication between the host and guest domains as well as providing a safe environment to execute guest-supplied programs.

The offloading mechanism uses inter-domain communication to send program code to an application running in the host domain. Guest domains are out of control of the host system. All data received from a guest needs to be treated as potentially harmful. Parsing the untrusted data must not crash the host or be exploitable using malformed data. Programs are given as BPF bytecode. The in-kernel checks already ensure that BPF programs terminate quickly and are memory-safe. However, these checks are not sufficient to stop malicious behavior of user-supplied applications in our offloading solution, as they might expose previously inaccessible data to guest domains. Additional checks in the offloading framework need to verify the BPF programs prior to loading them into the kernel and reject potentially problematic applications.

When using a software bridge to distribute incoming traffic to guest domains, the Linux kernel of the host domain already handles separating traffic for different do-

mains. Traffic is transmitted to the virtual interface it is directed to. Domains with different interfaces should never see foreign traffic and can therefore neither inspect nor modify it. BPF programs running in the host kernel get direct access to the network interface, which is not otherwise accessible by guest domains. Additional restrictions have to be applied to user-supplied programs to forbid the direct access. The host NIC receives traffic sent to the host machine and its guest machines. If a BPF program is attached as a network action directly on the NIC, it is executed on every frame received. Programs offloaded from guest domains should only be executed on traffic directed to the domain, from which it was offloaded. An earlier filter is necessary to execute the right BPF program per frame. Whether or not a guest domain offloads programs into the host should not be detectable from other guest domains through the offloading mechanism itself or other means.

Every potential security issue opened up by a framework for network function offloading needs to be mitigated through additional checks, restrictions or sandboxing. In the next chapter we present solutions to mitigate these issues.

4

Design

After providing the motivation to use a VMM as a platform to run network function implementations and showing how the current networking is a bottleneck for this use case, we now describe the approach to bypass the guest networking stack by offloading small applications from guest domains to the host domain. We start with an overview of the functionality our framework provides and follow with a more detailed look at available functions and how we load and attach BPF applications. We also describe how we prevent the potential security issues that were presented in Section 3.3. Last, we discuss the provided user-space library and how to integrate offloading into user-space applications.

4.1 Overview

In the previous chapter we described why the deployment of programs implementing network functions onto virtual machine infrastructure is desirable. We also identified problems with the way networking under Xen works for guest domains. The network abstraction for guests involves the network stacks of the host domain and the guest domain and requires expensive copy operations to transfer network frames into and out of the guest domain. To bypass this bottleneck, we want to deploy programs acting as network functions before the expensive network handling is even involved. To do this we provide a mechanism for guest domains to offload small programs into the host domain through an inter-domain communication channel. The host domain will install the offloaded programs on behalf of the guest domain as network filters in its own network stack, where the program is executed on incoming network traffic directed to the guest domain. By bypassing the network abstraction and executing programs in the host domain, we open up potential attack vectors. To mitigate this, offloaded programs need to be restricted in what they can do. We only allow programs acting as network filters on traffic for the offloading domain. The programs are additionally checked for potential security problems and either dynamically rewritten to avoid potential harmful behavior or rejected right away.

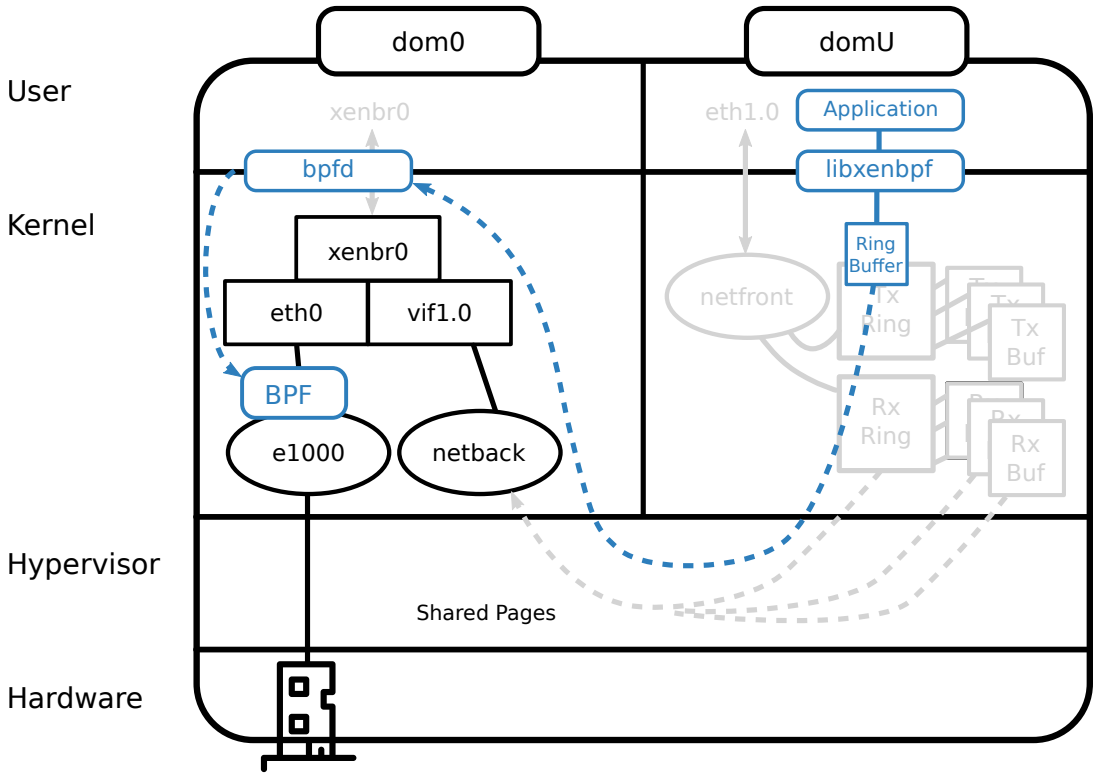


Figure 4.1 Implementation design of `XenBPF`. The `bpfd` daemon runs in user-space on the host domain and calls into the kernel to attach BPF programs directly to the host's NIC. Guest domains communicate with the host through shared memory using the provided `libxenbpf` library (original figure from "Network Throughput and Performance Guide" [37]).

This approach is implemented in `XenBPF`, a framework that allows to offload small programs from virtualized guest domains into the host domain of Xen. Figure 4.1 shows the full design of `XenBPF`. In order to move network functions from guest systems into the host system and attach them to the NIC of the host machine, we provide a privileged application, `bpfd`, which receives BPF programs from guest domains. On the guest side, a user-space application offloads BPF programs using the provided `libxenbpf` library, which communicates with the application in the host. BPF programs offloaded from guests should implement a network filter and will be attached to the host's NIC in order to perform the work of a network function. For the actual execution of the programs we rely on the traffic control subsystem in the Linux kernel. This subsystem already handles BPF and provides high-level tools to load and attach BPF programs as needed. No changes to the kernel are required for the main functionality. The framework's main task is implemented as a user-space daemon and calls into the kernel to load BPF programs and attach them as network filters.

4.2 Design Goals

The implementation of `XenBPF` is split into two parts: The privileged daemon application called `bpfd` runs in dom0 with full access to the host's kernel and subsystems. A user-space library called `libxenbpf` can be used in guest domains to communicate with the daemon, send BPF programs and access BPF maps of loaded programs.

While the offloading should make efficient use of resources, the `bpfd` daemon itself is not in the performance-critical path of execution and thus performance is less important in its implementation. However, it runs with privileged permissions on the host and receives arbitrary data from potential harmful guest domains and needs to be secure against attacks or malicious constructed data. Simply running `bpfd` should not open new exploitable attack vectors. It must never crash, corrupt state or execute unchecked code. Offloading network functionality into the host domain should reduce latency for packet processing and should increase overall throughput when packets are handled early.

In the following sections we explain the design of both parts of `XenBPF` in more detail.

4.3 bpfd: Privileged Daemon

The privileged daemon is called `bpfd` and runs in the driver domain with direct access to the NIC. In default setups of Xen this will be `dom0`. Its high-level procedure is simple:

1. On boot of the driver domain, the daemon is started by the init system.
2. Once initialized, it scans for already active guest domains and opens a communication channel to these domains.
3. It creates a watcher waiting for additional guest domains to boot up. If it detects a guest domain booting up, it opens another communication channel to this domain.
4. It waits for requests on the guest domain's communication channels. Upon receiving data, it decodes the requests and acts accordingly and responds with the result to the guest domain.

The daemon is started early in the boot process of the host domain. Guest domains might be started earlier and therefore the daemon first checks for already up and running guest domains. As mentioned before, configuration settings and state of the system is available through the `XenStore`, a hierarchical data store maintained by Xen. It includes information about virtual devices attached to domains, including their configuration and boot state. From this data store we can extract the virtual interfaces and their MAC addresses. Once a communication channel to a guest domain is set up, `bpfd` stores settings about it in the `XenStore`, where the guest domain can read it and establish a connection to the communication channel.

Once the initial set of booted domains is scanned, `bpfd` installs a watcher on the `XenStore` root level. Whenever a setting is changed the daemon is notified and can act on it. This is used to detect newly created domains and when they are ready. Again, the virtual network interface and its MAC address are extracted from the `XenStore`.

Once a guest domain is booted, `bpfd` will create a new communication channel using the event channel and shared memory mapping mechanism provided by Xen. This

will create a ring buffer in shared memory that can be accessed from the host and the guest domain for reading and writing data. After opening a communication channel to guest domains, the daemon waits for further input. Whenever new data arrives over the communication channel, the data is read and parsed according to the communication protocol. If the received data contains a full command, the requested action is executed and a response is written back to the guest domain. Each domain has their own communication channel to the host domain and therefore no communication of another domain can interfere. In the following section we describe the available commands.

4.3.1 Available Commands

Functionality to guest is exposed through commands in `bpfd`. Available commands are exposed by an identifier in the communication protocol. After receiving data, `bpfd` decodes the request. If the data is malformed or contains an invalid request, it returns an error to the requesting domain. Otherwise `bpfd` executes the requested task with the arguments received. Valid functions are wrappers around existing kernel APIs with additional checks. All functions first need to check validity of the passed arguments. If any of the arguments is in an invalid format, the execution is aborted and an error is returned to the user. After this first check, additional checks are specific to the called function. For functions working with the ID of a BPF program it checks that the requesting domain references a valid loaded BPF program and that this BPF program was loaded by the same requesting domain. Access to BPF programs of other domains is prohibited. A similar check is applied to functions accessing BPF maps. The referenced BPF maps need to be valid, loaded and the domain must have access to them. In addition, it checks that the expected size of the key and the value matches the loaded map to avoid memory corruptions.

If all checks succeed, the underlying kernel API is called with the correct arguments. Upon returning from the kernel any error is sent back to the guest domain as an error code. On a successful call, the requested data is sent back, e.g. an ID for the now loaded BPF program or the fetched BPF map data.

The set of functions available through host-guest communication is limited. Only six functions are implemented. That is enough to support all common applications, as will be discussed later. The implemented functions are:

1. `load_bpf_elf` : Loads a BPF program and all BPF maps from an ELF object file and attaches it to the NIC. It returns an ID for the loaded program as well as IDs for all loaded maps.

2. `load_bpf_bytecode` : Loads a BPF program from raw BPF bytecode and attaches it to the NIC. No BPF maps are supported this way. It returns an ID for the loaded program.

3. `unload_bpf`: Removes the BPF program from the NIC and unloads it from the kernel. It also removes associated BPF maps.

4. `map_lookup`: Looks up an element by key in the specified map and returns its value.

5. `map_update`: Updates an element in the map. If it does not exist, the element is created. Flags can alter this behavior.

6. `map_delete`: Deletes an element by key.

The arguments are similar to the Linux kernel API. Programs and maps are identified by an ID. The user-space application has to keep track of IDs and sizes of map keys and values in order to use the data.

In contrast to the kernel API we do not separate between loading a BPF program and attaching it to the right subsystem. We currently only support a single way to use BPF programs, namely as actions on incoming traffic. Handling loading and attaching in a single method keeps the API surface small.

Two additional API functions, which are available as kernel APIs for interacting with BPF are currently not exposed through `bpfd`. Namely the direct creation of BPF maps and the iteration of BPF maps are not exposed. Adding these to `bpfd` requires adding new types in the communication protocol and a wrapper around the kernel API with the appropriate checks. For the applications we tested, neither of these functions is necessary and all functionality can be satisfied with the implemented functions.

4.3.2 Inserting Network Functions

After the daemon receives the code for a BPF program from a guest domain, it needs to load it into the kernel and run it when packets are received on the NIC. As discussed in Section 2.2.2, BPF programs can run as actions on each incoming network frame. In order to do this, we use the Linux kernel's traffic control subsystem to attach BPF programs as actions on ingress traffic [30]. The traffic control subsystem can be configured through the user-space tool `tc`.

Two different ways to handle the invocation of guest domain-supplied programs are possible. Both have different trade-offs regarding feature set and complexity.

For both approaches, we load our own BPF program to dispatch to BPF programs loaded on behalf of guest-domains. For the first approach, we create a simple ingress queuing discipline, to which we add our own dispatcher action. This action loads two BPF maps. The first one is a hash map, mapping MAC addresses to some identifier. The second is a program map, mapping above identifiers to a loaded BPF program. The two-layer approach is necessary, as the key of a BPF program map is fixed to 32 bits, but MAC addresses are 48 bits. When executed, the filter function parses the passed packet as a plain Ethernet packet and extracts the destination MAC address. Using this address it looks up if a program is loaded. If it is, it tail-calls into that program. The jump table stores mappings to all loaded programs from all guest domains. Only `bpfd` and the dispatcher program have access to it. Filtering on the destination MAC address ensures separation of traffic for different domains. The downside of this approach is that only one BPF program per domain can be used. If the guest domain tries to load a new program, the old one is replaced. It is up to the guest domain's administrator to correctly set up network functions.

The traffic control subsystem allows for more sophisticated classification and filtering of traffic. This requires so called *classful queuing disciplines*, which are not available for ingress traffic by default. However, there is a workaround for this. Instead of attaching a BPF filter on ingress directly, we first mirror all incoming traffic to a virtual interface. Linux comes with an *Intermediate Functional Block device* (`ifb`). Using this kernel module we can create the required virtual interface and then mirror all traffic of a real network device to this new device. Traffic is now passing through the new device and we do have an egress port to use. We can create classification and action filters on egress of this virtual device. The full spectrum of classful queuing disciplines are available. Queuing disciplines are organized in a tree hierarchy and classifiers and actions are applied top to bottom. Classifiers determine which branch of the tree is taken. To use this for our network functions, we first add our own classifier program to the root node of the queuing discipline tree. For every domain with a loaded BPF program we add a new branch. On this branch we can attach one or more BPF programs directly. Our own classifier again extracts the destination MAC address of a packet and chooses the assigned class identifier for the associated domain from a shared BPF map. Subsequently, the traffic control system traverses into the branch identified by this class and executes attached actions. If the network packet is not targeted to a guest domain or the guest domain has no network functions loaded, no action is taken and the packet is passed on unmodified. For domains with network functions, we can now handle multiple BPF programs. BPF programs are executed in the order they were attached, just as they would be on a bare Linux system. The downside of this approach is the added complexity to handle the state of the queuing disciplines. The virtual interface incurs additional processing cost, but as it is all handled in the same kernel it should be minimal. With this approach special care needs to be taken regarding security. This setup allows traffic looping if packets are retransmitted, leading to a disconnect of the used network device for all attached machines, including the host machine. A mitigation method is explained in Section 4.3.4.

4.3.3 CPU Overhead Accounting

As explained in Section 3.2, the Xen scheduler cannot account CPU time spent in `dom0` processing network frames to the right guest domain. Guest domains with high network throughput get effectively more CPU time than they are allowed by configuration.

With network functions loaded into the host kernel, we have an entry point to measure the additional cost. It is now possible to count how many network packets are processed by BPF programs in the kernel and the amount can be reported back to administration tools, which eventually could use this information to instruct the Xen scheduler to adjust CPU time assignment. To avoid the overhead of exact time measurements on every processed network frame, an example BPF program is sampled at boot to get an approximate timing. While in theory it would be possible to get more exact measurements by timing the guest-supplied BPF programs, the testing requires a network packet to work with, which we cannot generally construct for arbitrary programs. The timing of one single BPF program under the host's control is only a rough measurement. The actual number of processed packets per

domain and the previously measured processing time for a single run provides an approximation of the time spent in `dom0` processing network packets on behalf of a guest domain. For guest domains that did not offload any network functionality into the host no additional time accounting is done.

Currently the additional processing time information is not reported to the Xen scheduler and thus not taken into account for scheduling decisions. However, based on this information, the host administrator could adjust CPU time shares for guest domains manually. This is only a temporary fix for guest domain with known traffic requirements. It will not prevent an attacker to use more than the assigned CPU time.

Details of the implementation of overhead accounting are explained in Chapter 5.

4.3.4 Plugging security holes

Virtualization is not only used to provide available resources to multiple guest systems, but also as a layer of security between these guests. This security layer is partly disabled by `XenBPF`. The communication channel bypasses the existing abstractions and thus the security layers. Our backend daemon will deal with unchecked input from guests. Proper care needs to be taken when parsing the data and acting on it. All incoming data is expected to be encoded in a defined format. The format chosen for `XenBPF` is both simple to construct and to parse. All data packets are prefixed with their size, buffers can be allocated accordingly and thoroughly checked at each stage. Whenever the parser hits an error, it immediately terminates, deallocates used buffers and returns an error to the user. Additionally, hard limits on buffer sizes are applied. These checks should ensure that an attacker cannot crash or exploit the application by constructing malformed packets or force the daemon to allocate large amounts of memory for buffers, leading to *out-of-memory* faults.

Leaking details about the environment and other running guest domains to an attacker in a guest domain is an issue. Both BPF programs and BPF maps, when loaded in the kernel, are identified by a file descriptor. If we return file descriptors as returned from the kernel to guest domains directly, we are effectively leaking information about neighboring domains as well. File descriptors are assigned incrementally, thus if an attacker domain creates two BPF programs over time, it will get two distinct file descriptors and can deduce the number of BPF programs or maps created from other domains in between these two calls. With explicit checks for access rights of used identifiers in `bpfd`, an attacker would not be able to use or access other BPF programs or maps nor know which other domain created it. The gained information is therefore of limited value. Nonetheless, it leaks information about the environment. A simple mitigation method can be applied to not leak file descriptor in the first place. Instead of returning file descriptors to guests, `bpfd` stores them per domain and a translation table assigns each BPF program or map a domain-specific ID. This ID acts as the identifier from the domain's point of view.

Some functionality available to BPF programs can be used to exploit the host system and get access to otherwise not available resources. Even though BPF programs run in a sandboxed environment with limited access to kernel functions or data other than the passed context, they still run in a privileged mode and some of the available

BPF helper functions can be potentially harmful in the shared environment of our application.

One main usage of BPF programs as network functions is the ability to rewrite and retransmit incoming network packets. When loaded as network filters, BPF programs have access to helper functions to retransmit the processed packet over one of the available network interfaces instead of passing it on. The helper functions are called with the interface to use for the retransmission and in which direction, ingress or egress, this retransmission should happen. For usage where the network cards are under full control of the administrator this poses no problems. Different network devices might be attached to different networks and retransmission through another interface is required. In the case of our virtualized environment the host might still be equipped with multiple NICs. A guest domain uses virtual devices that are connected to one of the NICs available in the host. The guest domain can configure its own virtual devices, but an offloaded BPF program will be able to use a NIC attached to the host. Previously, a guest domain would not be able to have any access to other NICs, but through the offloaded BPF program it gets access. If a BPF program retransmits a package, either one of those actual hardware devices can be used for the retransmission. A bogus program can abuse this to cause a traffic loop and with that will completely disconnect the network for all domains using this device, including the host domain. In `XenBPF` we therefore detect the used network interfaces and apply limitations to BPF programs on which devices can be used and allow retransmission only in outgoing direction. Each BPF program loaded by a guest domain is passed through a custom verifier. When the usage of packet retransmission helpers is detected, we rewrite the BPF program to use a fixed interface in outgoing direction only. Even if the program tries to cause a disconnect by passing different arguments to these helper functions, it will be executed with the fixed values[1]. The implementation is further explained in Section 5.5.

Both memory and CPU usage of `bpfd` should not have a big impact on the host system. We especially do not want to allocate an unlimited amount of memory in `bpfd` and therefore apply hard limits on buffer sizes internally. BPF programs have a hard limit of 4096 instructions, enforced by the kernel. Longer programs will be rejected by the in-kernel verifier. Some network functions might require more instructions to fulfill their task. One possible way to circumvent this restriction is to split the program into multiple BPF programs and use tail-calling to jump into dedicated functionality. As described in Section 2.2.1, tail-calling is achieved by storing the file descriptor of loaded BPF programs into a special BPF map. The actual tail-call requires a BPF helper function. The kernel looks up the BPF program in the program map, translates the value into a pointer to the program and uses that to jump into its execution. As all BPF programs are loaded by the single `bpfd` daemon, they all are in the same namespace. In theory every loaded BPF program in an application namespace can be the target of a tail-call. User-space applications as well as BPF programs themselves can insert file descriptors into BPF program maps. User-space applications have to use the API provided by `XenBPF`, where it can check the arguments before actually executing them. Inserting file descriptors not owned by the requesting application can be prohibited. However, this only covers one side of the API. An user-supplied BPF program could insert data just as

[1] At the time of writing our rewritten BPF programs hit a bug in the kernel verifier and therefore our own verifier is disabled for tests.

easily and our application does not have a way to stop this. The simplest mitigation method to prevent tail-calling into other programs is thus to reject programs that do any tail-calls. This method can be implemented purely in the custom verifier in `bpfd`. In order to eventually allow tail-calls even from BPF programs initiated by guest domains, kernel changes would be required. A new helper function has to be added that can check the validity and access rights of BPF programs before doing the tail-call. `bpfd` already has information on which BPF program belongs to which domain and can pass the required information down to the kernel. Our user-space verifier can replace regular tail-calls with the new modified helper call and no changes from the guest would be necessary.

4.4 libxenbpf: User-space Library and API

In order to communicate with the privileged domain, user-space applications in guest domains need to use the established communication channel to send requests. We provide our own library, `libxenbpf`, abstracting this communication and providing a simple to use API. User-space applications that want to offload network functions into the host load this library. On initialization the library will use the `XenStore` to look up the settings for the communication channel and then connects to it. Once the communication is established, the user application can invoke functions to send command requests. The library encodes these requests and passes them to the host machine, waiting for a response. The user-facing API functions of this library are as close as possible to the kernel API. This allows a replacement of BPF functionality in existing programs with only minimal changes to the source code.

For regular BPF syscalls no special synchronization needs to be done, as the switch between user-space and kernel-space and back can be considered atomic. A simultaneous BPF syscall from another thread will not corrupt any communication state, even though data races for map contents can happen. However, for our host communication a single communication channel is used with an asynchronous protocol send over shared memory buffers. If two or more threads try to use this channel, the serialized protocol will be corrupted. Therefore, we need to ensure mutual exclusion around every call to the host domain. If used in a threaded environment, this can lead to contention when trying to load or store data into BPF maps. In general this is a minor issue, because the user-space side of a BPF program is not directly involved in processing network packets and thus not in the performance-critical path.

4.5 Summary

We implemented XenBPF, a framework on top of Xen, that allows guest domains to send BPF programs to the host domain, where they are attached as network filters on the host's NIC. The BPF programs should implement a specific task of a network function. State can be shared with the offloading guest domain through BPF maps. Through careful separation of tasks and using available information from the network packets, we ensure the separation of network traffic of different guest domains. By reusing the available BPF execution environment in Linux and the traffic control

subsystem, combined with additional checks in our software, we provide a safe, sandboxed environment for executing unknown code as network functions on incoming network traffic. We identified potential security problems with this approach and showed how these problems can be mitigated.

The following Chapter 5 describes parts of the implementation of `XenBPF` in more detail. Chapter 6 then evaluates the performance of our approach.

5

Implementation

In the previous chapter we presented the general design of `XenBPF` and discussed some security aspects. This chapter will describe some parts of our implementation in more detail.

5.1 Overview

`XenBPF` consists of the `bpfd` daemon, running in the host domain, and `libxenbpf`, a library for integration into user-space application in guest domains. Both parts are implemented in C. This eases integration with the available libraries for Xen. The daemon will be started at boot of the host and waits for input from guest domains. Inter-domain communication over shared memory is then used to exchange data between `bpfd` and applications in guest domains. The high-level design of both parts is explained in Chapter 4. The following sections provide further details on the communication channel implementation, CPU accounting and additional checks on BPF programs as well as the user library.

5.2 Communication Channel

The main task of `bpfd` is to provide the inter-domain communication channel to guest domains and to load and attach BPF programs on their behalf. The channel for communication between domains relies on functionality provided by Xen. Memory pages can be shared between domains. These memory pages are then used for a ring buffer into which the communicating domains can write and read from. The same mechanism is also used by the network driver of Xen to move network packets into a guest domain. We use `libxenvchan` to handle inter-domain communication. This library relies on other abstractions. The library `libxenevtchn` provides access to the event channel mechanism in Xen. Configuration and state of event channels is

stored in `XenStore` and available to both the host and the guest domain. Actual data is passed through the ring buffer located in shared memory.

With this functionality in place, our own `libxenbpf` can establish the communication channel and pass its protocol between domains. The communication channel is always started from the host domain's side on boot of the guest domain.

The communication channel is completely transparent to the end user and could easily be replaced with another communication mechanism on other VMMs. This allows an alternative implementation for other hypervisors.

5.3 Communication Protocol

Communication between guest domains and the host domain requires a structured protocol to encode data. In order to call functions over the inter-domain communication channel, user applications send a request and receive a response from `bpfd`. Requests and responses get serialized into a simple protocol that can encode different data types. All buffers have explicit bound checks and the protocol uses a length-prefixed encoding for the contained data. In addition, hard limits on buffer sizes allocated from requests are enforced. The main communication should only require a limited amount of memory. The only size we cannot control is the size of passed object files containing the compiled BPF code. As we know the upper limit of instructions allowed in a single BPF program, the only additional overhead in an object file should be metadata about BPF maps. Even this metadata should only amount for a small part of the size of the file. We can therefore limit the overall size to a little more than what is required for the maximum of 4096 BPF instructions and reject everything bigger than that. Applications have to restrict their BPF programs to only include the necessary data. Size restrictions are applied on both sides to avoid misuse of the library and misuse when the communication channel is used without assistance from `libxenbpf`.

Individual commands are identified by a simple numeric identifier. Each command explicitly defines the number of arguments it expects as well as the type of every argument. If the number of received arguments is wrong or any of the arguments are of the wrong type, the command is immediately rejected. A command can either return an error response with an error code or return an individual success response with additional data attached. Again, the number of arguments and their types in the response are fixed and checked on the user-space side to avoid misuse.

5.4 CPU Overhead Accounting

Executing BPF programs in the host on behalf of guest domains raises the used CPU time, which is not accounted for the guest domain by Xen. `XenBPF` can measure this CPU overhead and report it back. In March 2017 the BPF syscall was extended with a simple testing framework [34]. Using a new subcommand of the `bpf` syscall, a loaded BPF program is executed a number of times with a constructed network packet. The time to execute is measured and returned back to the caller. Using this

functionality, `bpfd` measures the time it takes to execute a BPF program. To get a rough approximation of the runtime, we use a BPF program that rewrites an ICMP `echo-request` packet into an ICMP `echo-reply` packet and retransmits it. While this exact use case does not represent every BPF program a guest domain would supply, it exercises some of the general used code paths. To identify the packet it first has to read the packet header and parse out contained data. It then needs to rewrite parts of the packet and force a retransmission of the packet. Other network functions will do similar operations on network packets.

The program is run for 1000 iterations to measure the time to execute. On the test machine it takes between 20 and 24 ns to execute a single run of the BPF program. The dispatcher program we insert into the network packet processing chain records the number of processed packet per MAC address. With the sampled processing time for a single packet and the total number of packets processed for any given MAC address, and therefore guest domain, we can account for the CPU time the host domain uses on behalf of a guest domain. This information could be used to adjust CPU time assignment, but is not automatically passed to the Xen scheduler at the moment.

5.5 User-space Verifier

Even though the BPF kernel environment already provides a limited execution engine and ensures the program contains no loops, unreachable states or invalid memory accesses by running a verifier when loading programs, additional checks in `bpfd` on BPF programs are necessary to prevent malicious behavior in guest-supplied programs. We add our own user-space verifier to filter potentially harmful code. As BPF bytecode instructions are always of a fixed size, we can easily iterate over the given bytecode and extract single instructions. The instructions can then be validated and if necessary be replaced, new instructions can be added, or the full program can be rejected.

We implemented two verifier passes. The first one checks for calls to the redirect helper functions in BPF, `clone_redirect` and `redirect`. When either one is called, we insert new instructions into the program right before the function call. In these instructions we override the register holding the interface index that should be used for the redirect, to contain a fixed interface index as configured on start of `bpfd`. By default this is the same interface traffic is received on. We also make sure that packets are retransmitted as outgoing packets. Passing them on as incoming packets should not need a retransmission in the BPF program. When inserting additional instructions, all jump operations in the bytecode need to be checked. Jumps in BPF are relative and the jump offset needs to be adjusted. Jumps before the inserted code that jump beyond the function call need to be adjusted accordingly. The same offset adjustments is applied to backwards jumps after the call. While our code for this verifier pass correctly identifies the helper function calls and can replace the register values, some resulting BPF programs are rejected on load by the kernel verifier due to a bug in the Linux kernel verifier. For now, this verifier pass can be disabled.

The second pass checks for arbitrary tail-calls, again identified by a helper function call. When the verifier detects such a call, the program is immediately rejected.

Tail-calling is not supported in `XenBPF` due to security concerns discussed in Section 4.3.4. In the future, the verifier could replace the tail-call with a custom helper call that can check access permissions for BPF programs at runtime.

5.6 libxenbpf: User-space Library and API

To ease integration into applications in guest domains, we provide a library to handle the inter-domain communication. Our user-space library `libxenbpf` is a near drop-in replacement for functionality normally provided by `libbpf`, a library for creating, loading and handling BPF programs and maps, that is available as part of the Linux kernel. `libxenbpf` is designed to replace the used functionality with little code modifications and the inclusion of a shared library. The only dependencies for `libxenbpf` are Xen libraries that are included in the official distribution.

The library offers convenient wrappers around the commands accepted by `bpfd`, as discussed in Section 4.3.1. It also provides functions to establish the communication channel to the host. Commands can only be executed with an established connection. The full list of available functions:

1. `xenbpf_connect`: Connects to the host domain and returns a handle to the opened communication channel.
2. `xenbpf_disconnect`: Closes the communication channel and forces unloading of all offloaded BPF programs.
3. `xenbpf_load_elf`: Sends the provided ELF file to load in the host domain. This also loads BPF maps as described in the ELF file. Returns the BPF program identifier and identifiers for loaded BPF maps.
4. `xenbpf_load_bytecode`: Sends the provided BPF bytecode to load in the host domain. Returns the BPF program identifier.
5. `xenbpf_lookup_elem`: Looks up an element in the referred map. Returns the value from the map or an error.
6. `xenbpf_update_elem`: Updates an element in the referred map with the provided value.
7. `xenbpf_delete_elem`: Deletes an element in the referred map.

The communication channel is again provided by `libxenvchan`. The command functions take the parameters and serialize them into the communication protocol, which is then written to the shared ring buffer. When data is available, it is read and decoded and then returned to the caller. Every command function is protected by a mutex lock to ensure exclusive access to the channel.

When integrating `libxenbpf` into programs that already use BPF programs attached to network interfaces simple code changes are necessary to use it. First, a connection to the communication channel needs to be established. This should be done early

in the program. Next, calls to load BPF programs into the kernel can be replaced with calls to the above load functions. Last, code to manually attach loaded BPF programs as network filters can be removed, as it is automatically handled by `bpfd`. Calls to map functions can be replaced with the appropriate functions from `libxenbpf`. The above mentioned restrictions regarding retransmissions and tail-calls need to be respected in BPF code.

Using `libxenbpf` provides a simple way to offload network functionality. It adds a single dependency to user-space applications, but does not restrict other parts of the application or tooling in the guest domain. In the next chapter we will evaluate the performance of `bpfd` and show how to integrate `libxenbpf` into existing applications.

6

Evaluation

We evaluated the performance of offloading network functions from guest domains to the host domain using `XenBPF` with multiple benchmarks. We measure the latency improvements when responding to ICMP ping requests and measure the throughput, packet loss and effect on neighboring domains when blocking traffic using a BPF-powered firewall. We conclude this chapter with two case studies to show the integration of `XenBPF` into a load balancer system and the key-value data store *Memcached*.

6.1 General Setup

All tests were run using a single machine as the Xen host. The exact hardware specifications of this machine are presented in Table 6.1. The host was running an unmodified Xen 4.7 as available in the Ubuntu repositories. All guest domains where created from a bare Ubuntu 16.10 image using the same Linux kernel v4.10 as the host system. The host domain was assigned all four available cores, whereas guest domains were only assigned two cores. The other configuration settings for domains can be found in Table 6.1 as well. The host machine's first NIC was configured with a static IPv4 address. A Linux software bridge enables network communication for guest domains. Each guest domain was instantiated with a hardcoded IPv4 address from the configured network. IPv6 was disabled for the tests.

Two additional machines are used to generate network traffic. Both are equipped with Gigabit network cards and are connected over a Gigabit switch directly to the Xen host machine and configured with IPs from the same subnet.

6.2 Reducing Ping Latency

Processing and answering network packets early in the processing pipeline should result in lower latency. To measure the reduced latency, a BPF program is used to

Configuration	Host machine	dom0	dom U
Processor type	Intel Core 2 Quad Q9400	-	-
Clock rate	2.66 GHz	-	-
Cores / vCPUs	4	4	2
Memory	8 GB	(auto)	1 GB
NIC	Intel 82572EI (Gigabit)	-	-
Operating system	Ubuntu 16.10	Ubuntu 16.10	Ubuntu 16.10
Linux Kernel	v4.10	v4.10	v4.10

Table 6.1 The hardware configuration of the host machine and domains running under Xen.

answer ICMP packets. An unmodified guest system without the BPF program is used to measure the baseline latency.

6.2.1 Benchmark Model

In order to measure the overhead of traversing the network stack, the bridge interface and the Xen network front/backend driver, we measure the latency of ICMP ping requests. One external machine sends a configurable amount of ICMP ping packets to a specified IP and measures the roundtrip time to receive an ICMP pong packet. In order to get exact timing, network packets are recorded by `tcpdump` with hardware timestamps. Those timestamps were extracted and used to calculate the ping latency. The receiver IP was either the IP of a guest domain or of the Xen host domain itself. The target machine and the traffic-generating machine are connected through a switch. No other devices are attached to the network and we ensure no other network traffic is sent between the machines.

6.2.2 Run Configurations

Five different scenarios where tested to show the different latencies. The baseline is measured in an unmodified Linux system in a guest domain without additional software running (domU Plain). This way an ICMP ping request was handled in the kernel of the guest domain and is first passed through the host's kernel. In the second run, the ICMP ping request is answered using a BPF program (domU BPF). The BPF program was inserted into the ingress chain of the guest domain's kernel. The third configuration used the same BPF program, but offloaded it to the host domain through `XenBPF`, where it was then inserted into the traffic control subsystem of the host's kernel by `bpfd` (XenBPF). For these three configurations the target IP was the one of the guest domain. The fourth and fifth configurations used the host's IP as the target. In the fourth run again no additional software was running. Guest domains were stopped (dom0 plain). The fifth test used the BPF program in the host to respond to ICMP packets early in the packet processing pipeline (dom0 BPF).

For each configuration run the external machine sent 10000 ICMP ping packets and recorded the timings. A warmup round with 100 ICMP ping packets was done before the measurement.

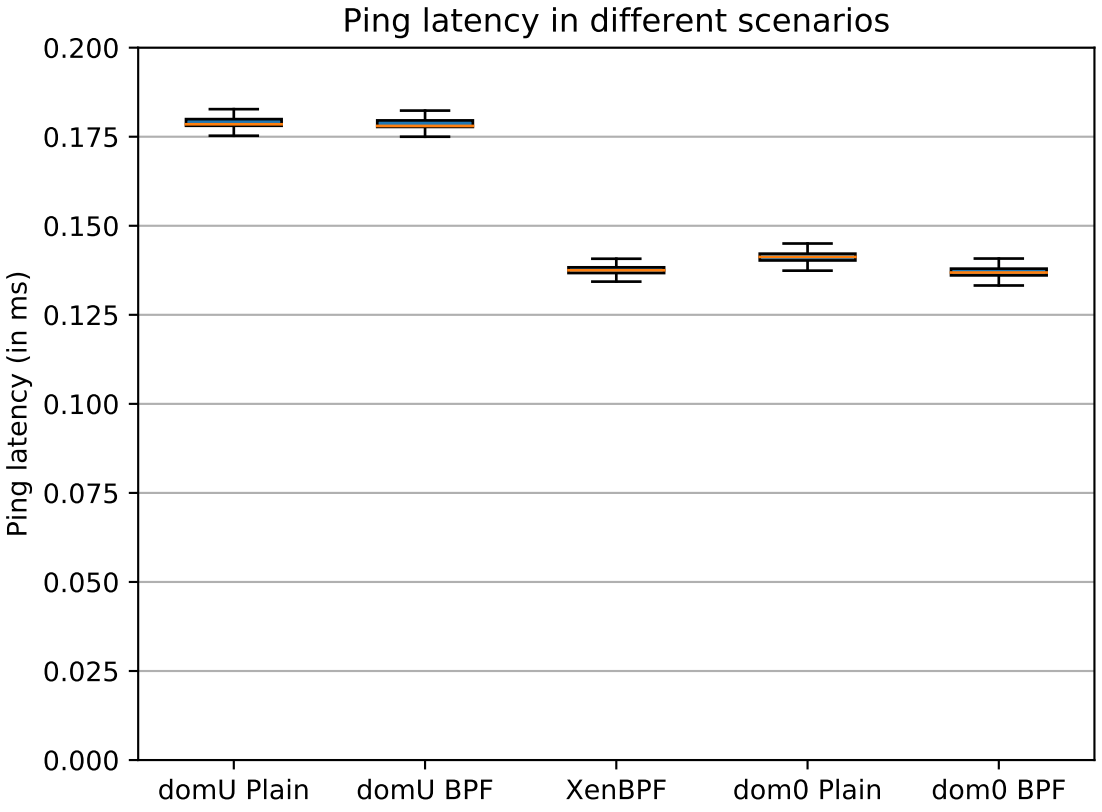


Figure 6.1 Ping latency for the different configurations. It shows the round trip time for ping packets in Milliseconds. Lower round trip times are better. Parsing incoming packets and recycling them for the response reduces the latency. This holds true for offloaded network functions as well.

6.2.3 Latency Results

Figure 6.1 shows the results of every configuration run of the ping latency benchmark in a boxplot. Lower ping latency is better. When the ping packets are answered by the kernel in the guest domain (domU Plain), it takes close to 0.18 ms to receive the response. Using a BPF program attached to the interface inside the guest domain does not improve the situation significantly (domU BPF). If ping packets to the host domain are answered inside the host kernel, the latency is reduced (dom0 Plain). It takes slightly more than 0.14 ms for a pong packet to be received at the sender. When ping packets are answered using a BPF program in the host domain, the latency is a bit lower (dom0 BPF). Running guest domain offloaded BPF programs to answer ping packets achieves about the same latency, as it effectively runs at the same stage (XenBPF). The small difference is explained by the additional lookups the dispatch program of `XenBPF` has to perform.

Offloading BPF programs into the host domain using `XenBPF` can reduce the latency for handling network packets by handling packets early, without the need to copy data to the guest machine's memory. The overhead of the dispatcher program is minimal. This is in line with our previously stated goals.

6.3 Firewall & Packet Dropping

As described in Chapter 3, network operators have to deal with an increasing number of Distributed Denial of Service (DDoS) attacks against their networks. To maintain availability of services in the network and to not overload single systems, the excess traffic has to be filtered and blocked. Firewalls in front of services are used to filter incoming traffic, block unwanted traffic and let wanted traffic pass through to an user-space application.

Now the task of a firewall should be handled by a network function implemented in software and running in a virtualized environment, such as the one provided by Xen. With the following benchmark we test the efficiency of a simple firewall implemented as a BPF program and offloaded into the host using `XenBPF`, when a guest domain is hit with large amounts of attack traffic. We measure the effect of dropping packets as early as possible on other network traffic hitting the same guest domain as well as the effect on CPU-bound applications running in neighboring guest domains.

6.3.1 Benchmark Model

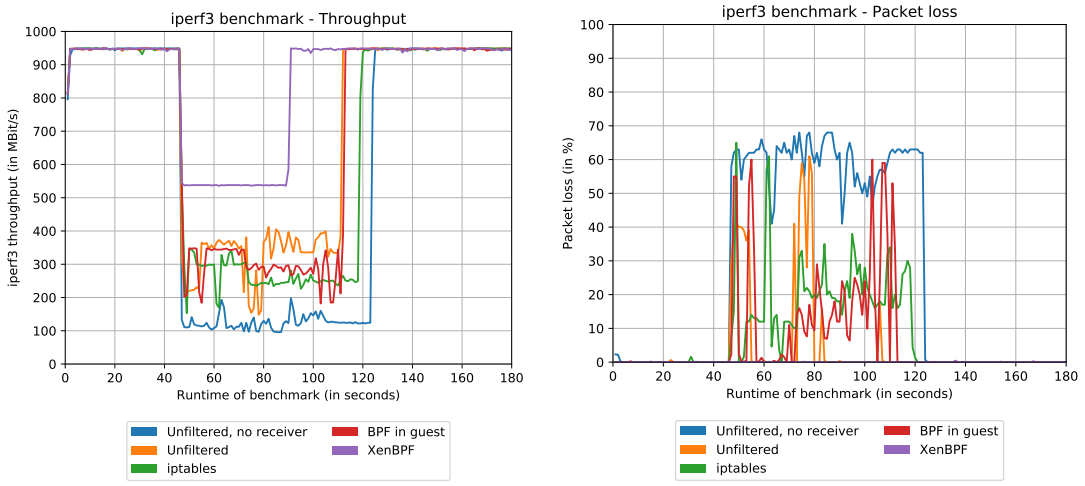
Incoming traffic should pass through to an application listening for the traffic. Without a bottleneck in the network, all sent traffic should be received and processed. However, when under attack, the number of malicious network packets sent to the attack target is increased. The attack target has to cope with this additional traffic and drop it if possible. If buffers run full or the bandwidth of the network link is exhausted, packets are dropped. In the worst case the malicious traffic will fill up all buffers and bandwidth and valid traffic is dropped before reaching its destination. This leads to a denial of service, as no valid traffic can be processed.

To prevent processing unwanted traffic and to drop unwanted network packets, we use a software firewall to drop packets as early as possible. For the following benchmark test, we measure the achieved bandwidth for valid traffic sent to the guest domain, while at the same time sending large amounts of attack traffic.

Packets can be dropped using different mechanism. If UDP traffic is sent to a specific port, but no application is listening on that port, the kernel will reject those packets and respond with a *Destination unreachable* packet. This is the case if arbitrary traffic is sent to a server without any application targeted and an attacker tries to overload the server's bandwidth. If an application is listening, these packets are passed through to user-space. This is the case when an attacker tries to overload a single application. To drop unwanted traffic on a certain port, `iptables` is used. In another configuration, a custom BPF program parses network packets and drops the attack traffic.

6.3.2 Run configurations

For this test we run five configurations with different ways to drop incoming traffic. To measure throughput and packet loss for valid traffic we use `iperf3`. The `iperf3` server is launched inside a guest domain and a client connects from an external



(a) iperf3 Throughput

(b) iperf3 Packet loss

Figure 6.2 Dropping UDP attack traffic using a firewall and impact on concurrent running bandwidth benchmark. Figure (a) shows the achieved bandwidth, Figure (b) the packet loss for the same benchmark run. Higher Throughput is better, the packet loss should be kept minimal. When dropping traffic early in the host domain using an offloaded BPF program, more valid traffic can pass to the guest domain. Nearly no packet loss for the valid flow is measured.

machine, connected to the Xen machine through a network switch. A second external machine is used to generate the attack traffic targeted to the guest domain's IP and a fixed port.

The `iperf3` benchmark is run for a total of 180 seconds. The bandwidth is set to 1000 Mbit/s. We first verify that the maximum bandwidth is reached constantly without packet loss. The packet size is set to 1448 byte.

Attack traffic is generated using the Linux kernel's `pktgen` module. It bypasses the usual network stack and generates network packets directly in kernel and writes them out to the used NIC. Firewalling is less bound by the size of individual packets, but more by the total amount of packets it has to handle. In order to maximize the number of packets sent out, the packet size is set to only 72 bytes. We want to stress the attacked host with a maximum number of packets per second instead of only using up available bandwidth. Using `pktgen` a total of 25M packets are sent out. The `pktgen` traffic is started 45 seconds into the `iperf3` benchmark. That is enough warmup time for a stable flow of valid traffic.

6.3.3 Results

Figure 6.2 shows the throughput and perceived packet loss as reported by the `iperf3` bandwidth benchmark over the time of the benchmark. Higher throughput is better, with a maximum close to 1000 MBit/s. Packet loss should be minimized.

For the first two runs, incoming attack traffic is not dropped. When no application in the guest domain is set up to receive traffic on the attacked UDP port, bandwidth drops as low as 100 Mbit/s, with packet loss reaching as high as 70% (Unfiltered, no receiver). The kernel is busy responding to packets and is not fast enough

to keep up with the incoming rate of packets. With an application receiving all incoming UDP packets, throughput is higher, but fluctuates more between 150 and 400 Mbit/s (Unfiltered). Packet loss is unsteady, ranging from near no packet loss up to 60% in short bursts. A software firewall like `iptables` cannot increase the achieved bandwidth, it reaches a throughput of 270 Mbit/s on average, with peaks up to 410 Mbit/s (iptables). Packet loss again shows peaks at 65% once and stays below 40% for most of the time. Dropping the unwanted traffic using a BPF program inside the guest shows only a small improvement compared to the `iptables` approach (BPF in guest). Once we offload firewalling to the host domain, throughput increases significantly (XenBPF). Throughput is constant at 550 Mbit/s and packet loss dropped below 1%. From the graphs we can also see that the time frame for attack traffic is different between the runs. For the baseline run the attack traffic is applied for about 80 seconds, whereas for the run with `XenBPF` it finishes after 45 seconds. This is explained by the flow control of Ethernet. When buffers fill up on the receiving side, it sends back a pause frame, asking the sender to reduce the send rate. By default, NICs adhere to these frames. With the higher drop rate on the receiver side, buffers do not fill up and no pause frame is sent back. The attack traffic rate is not reduced.

Discarding packets before they are transmitted to the guest domain shows a large improvement in this test. More consistent and higher throughput and less packet loss show the advantages of our approach.

6.3.4 Effect on neighboring domains

Measuring the pure network throughput on an otherwise idle machine with only a single guest domain running gives us a good first overview of the impact of offloading work to the host domain. Up to now we only ran a single guest domain on Xen. However, in more real world scenarios the available resources of a machine are shared between multiple running guest domains. With multiple guest domains running, scheduling gets more complex. The different workloads of different guest domains can have a large impact on scheduling decisions.

In order to tests this in combination with `XenBPF`, we now run two domains on the Xen host. The first guest domain, Domain 1, is hit by a constant stream of network traffic. The second guest domain, Domain 2, runs a CPU-bound benchmark. To measure the impact of `XenBPF`, we run the different configurations of the previous firewall test. The second domain is booted on the same host, running the same bare-bones Ubuntu 16.10 image, and will run the classic `UNIXbench` benchmark tool with a reduced set of tests. Only CPU-intensive tests are enabled and any graphic or disk tests are disabled.

The external hosts sends constant traffic targeting the first guest domain. `UNIXbench` is started in the second guest domain and the index score of the benchmark is recorded. A baseline score is recorded without any network traffic targeting a guest domain (No Traffic). We reuse the different configuration setups from the previous test. First, no packet filtering is enabled and no receiving application started (Traffic). Second, a user-space application receives data on the attacked machine (Traffic recv). Third, `iptables` is configured to drop the unwanted traffic in the

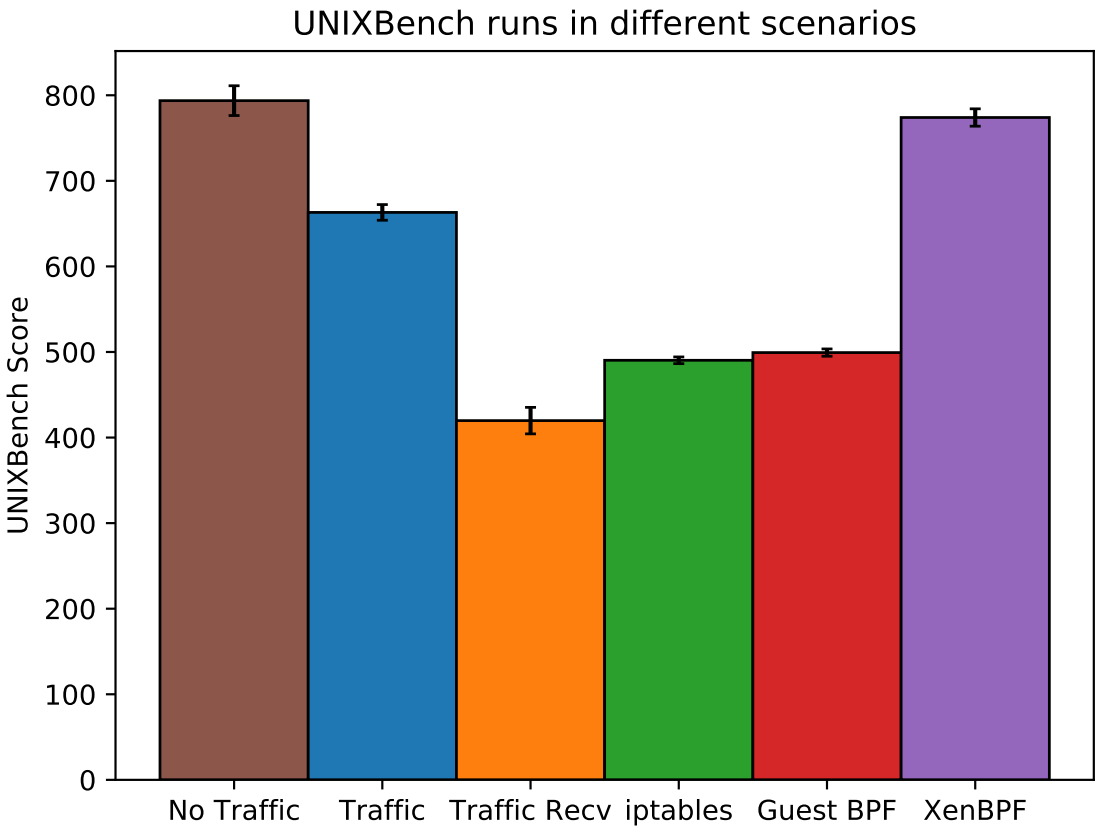


Figure 6.3 UNIXbench score when running on a guest domain co-located to a domain with constant network traffic, that gets firewalled. A higher score is better. Blocking packets in the host domain saves CPU time, that can be allocated to more CPU-intensive domains.

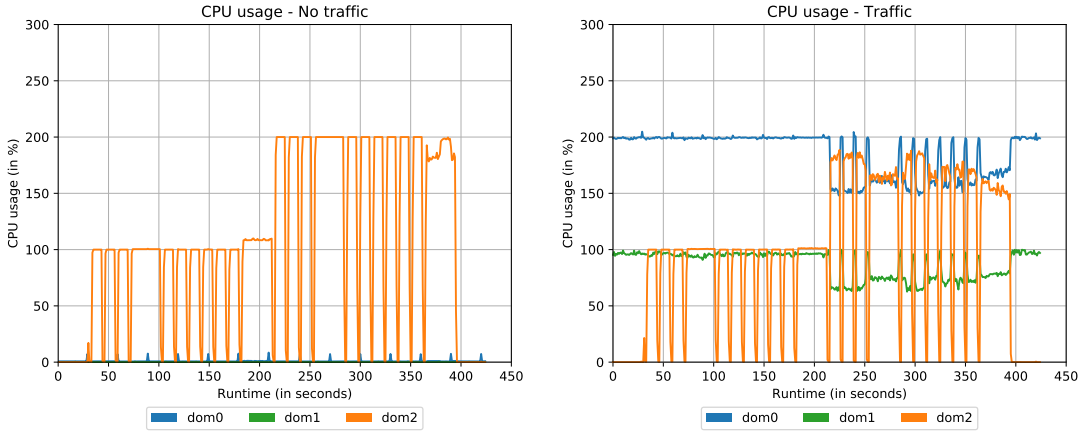
guest (iptables). Next, the BPF firewall program is launched in the guest domain (Guest BPF). Finally, the same BPF firewall program is offloaded through XenBPF (XenBPF). All test were run five times and the average index score was used.

In addition to the benchmark score, we also monitor CPU usage of all running domains using `xentop`. It reports different real-time statistics about running domains, which includes the total time a domain was running on a CPU and the percentage of CPU time used by any domain, including the host domain. Data is collected every second.

6.3.5 Results for CPU Performance Benchmark

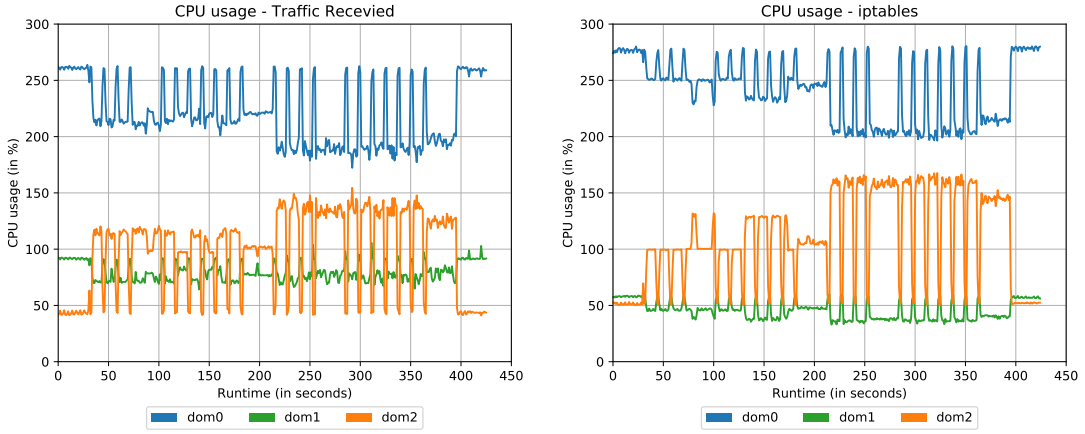
The reported benchmark score is an indicator for the performance of the processor. A higher benchmark score is better. As we use only a reduced set of the available benchmark tests, the score be compared to publicly available scores of other systems. However, the score can be compared against results of the same benchmark when running under different scenarios.

Figure 6.3 shows the mean benchmark score for each test run as well as 95% confidence intervals. The baseline with no traffic or firewalls running reports a score of 800 (No Traffic). When the colocated domain is hit with constant traffic, CPU performance in the second domain suffers (Traffic). The score drops below 700. If an



(a) No traffic (b) With unfiltered traffic

Figure 6.4 CPU usage of the host domain and two guest domains, when Domain 2 is running a CPU-intensive benchmark. Graph (a) shows the baseline measurement without traffic hitting the machine. Graph (b) shows the CPU usage when traffic is hitting Domain 1, while Domain 2 runs the CPU benchmark.

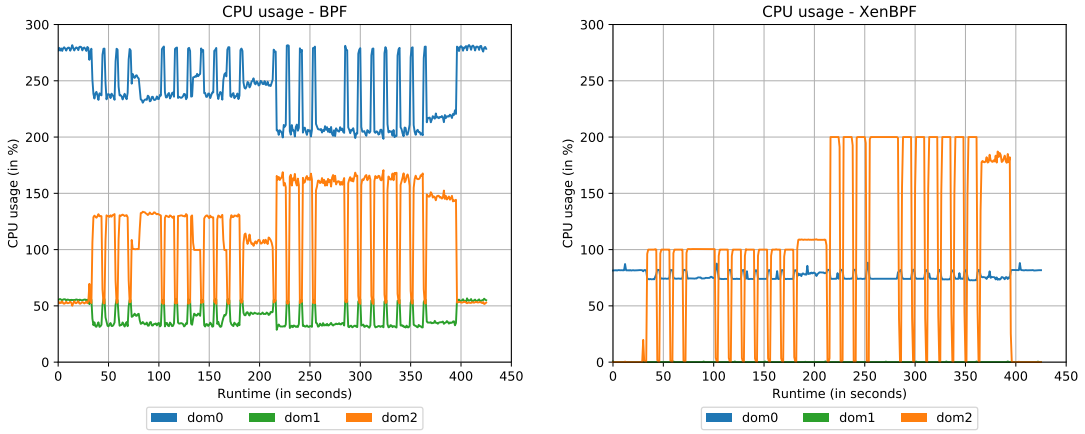


(a) Unfiltered traffic, received by application (b) Traffic filtered using iptables

Figure 6.5 CPU usage of the host domain and two guest domains, when Domain 2 is running a CPU-intensive benchmark and Domain 1 is the target of high attack traffic. Graph (a) shows CPU usage when the traffic to Domain 1 is received by a user-space application. Graph (b) shows the CPU usage when traffic is filtered using iptables in the guest domain.

application in the network-intensive domain accepts the traffic, CPU performance in the second domain suffers even more and the score drops to 400 (Traffic Recv). Now all running domains need their fair share of CPU time to get work done. Filtering the traffic in Domain 1 using either `iptables` or the BPF program (Guest BPF) has a positive effect on the more CPU-intensive domain. The benchmark reports a score of 500. Once we move packet handling out of the guest domain and into the host domain, the CPU benchmark is nearly back to the baseline results (XenBPF).

In addition to the simple benchmark score we also measured the CPU usage of domains. With four cores in the host machine, the sum of CPU usage across all domains cannot be larger than 400%. Graphs of the different runs always show the CPU share of each of the three domains.



(a) Traffic filtered using BPF in the guest

(b) Traffic filtered using XenBPF

Figure 6.6 CPU usage of the host domain and two guest domains, when Domain 2 is running a CPU-intensive benchmark and Domain 1 is the target of high attack traffic. In the run of Graph (a), incoming traffic is filtered by a BPF program in the guest domain. In the run of Graph (b) the BPF filter program is offloaded into the host domain using `XenBPF`. The latter one does not require any CPU for Domain 1.

The graph in Figure 6.4(a) shows the baseline measurement. No traffic is sent in this run and the CPU benchmark is the only running application in the guest domain. The benchmark runs in waves, launching test after test. For the first few rounds it uses a single vCPU completely, later tests use both vCPUs of the domain, resulting in 200% of all available CPU resources. The host domain is nearly idle, with only minimal CPU activity due to the measurement in this domain.

Figure 6.4(b) shows the test with ongoing traffic to Domain 1 and the CPU benchmark on Domain 2. No packet filtering is applied. All three domains consume some amount of CPU time. The CPU-benchmarked domain again shows the same waves of CPU usage, however the second half of the benchmark is unable to use both assigned vCPUs completely. For the first half, `UNIXBench` only occupies one of the four cores of the host and thus three cores can be used for other work. The host domain takes two cores completely for processing the incoming traffic and forwarding it to the first guest domain, where it is then handled and answered from within a kernel. Domain 1 uses up the remaining core to do the processing. Once the CPU benchmark starts using multiple threads, the CPU time has to be split between the domains. All three domains get their fair share for short times of work, but performance in all three domains suffers.

Figure 6.5(a) shows the test with ongoing traffic, that is now received by an application running inside Domain 1. The CPU benchmark runs in Domain 2. Receiving the traffic in Domain 1 requires CPU time, but it is less than when the kernel would handle it. In case of a receiving application, no *Destination unreachable* packet is sent, saving processor time. With less work to handle outgoing traffic in the first guest domain, the host domain now uses even more CPU time to handle the incoming traffic. `UNIXBench` uses a maximum of 150% of CPU, as the rest is already in use. This results in a far lower benchmark score, as reported before.

The fourth graph, Figure 6.5(b), shows a run when traffic to Domain 1 is dropped using `iptables`. CPU usage of Domain 1 drops to 50%. The host domain now peaks at 270% usage of all CPU time. The faster drop rate in the guest allows the host to transmit more packages into the guest domain, but requires more CPU. The CPU-benchmarked Domain 2 now shows small spikes at 120% of CPU usage for the first half. For the second half, Domain 2 is assigned up to 160% CPU time, as the other portion of CPU time is used up by the host machine again.

Dropping incoming traffic in the guest domain using a BPF program shows similar results as shown in Figure 6.6(a). CPU usage in Domain 1 is slightly lower than in the `iptables` run and Domain 2 has slightly higher peaks for the first half of the CPU benchmark. Both `iptables` and the BPF program do their work early in the network stack of the kernel in a pretty efficient way.

For the last test, results shown in Figure 6.6(b), the BPF firewall program was offloaded to the host using `XenBPF`. With traffic being handled in the host domain, there is no need to wake up Domain 1 and it gets no CPU time assigned. Dropping the traffic using BPF requires 80-90% of CPU usage in the host domain, but it can drop traffic at a much higher rate than previously achieved. As `UNIXBench` at most uses two cores, that leaves one complete core of the host idle.

Handling packets in the host domain and never invoking the guest domain not only has a positive effect on latency and throughput. It also frees up CPU time, that can be used by other more CPU-intensive workloads in other domains. Previously, handling network traffic required CPU time to be assigned to two domains, the host domain and the guest domain receiving the traffic in the end. With offloading BPF programs to the host, CPU time is only required in the host domain.

6.4 Case Study: Load Balancer

Now that we have shown the effectiveness of handling packet processing as early as possible, which can reduce latency and packet loss and increase throughput, we now go on to show how `XenBPF` can be integrated into existing applications.

Another use case for network functions are load balancers. Load balancer frontends are used to balance incoming traffic between multiple backend hosts. Traffic should be equally distributed across available backends to not overload a single system. Additionally, load balancers need to react to changing conditions of the backends. Backends can become unavailable and traffic needs to be sent to the next available backend. Dynamic configuration can extend or reduce the number of available backends. If backends can communicate back to the load balancer, they can signal when they become overloaded and traffic should be redirect to other backends. The load balancer has to reconfigure after these notifications.

The Cilium Project [35] uses BPF for network policing between application containers. It comes with a load balancer implementation in BPF, which can be dynamically configured through BPF maps. The load balancer is not tied to the Cilium project and can be used stand-alone.

We deploy the load balancer implementation into our guest domain. Using `XenBPF` we offload the BPF program into the host domain. Using an additional tool, `cilium-lb-cli`, we configure the load balancer's frontend and backend mappings.

When traffic hits the load balancer it looks up if any service is registered for the targeted IP and port. If the service is registered, it chooses one of the available backends to use. We modified the Cilium load balancer to pick backends in a round-robin fashion in order to work similar to the user-space load balancer we tested against. By default the Cilium load balancer implementation would use a flow hash to choose the backend. With little changes to the implementation it could take other external factors into account. The incoming network packet is modified to be addressed to the chosen backend IP and port and retransmitted in the direction of the backend. At no point the packet is passed to user-space of the guest domain running the load balancer.

For full production usage a reverse translation would need to be applied to response packets from the backends to get back to the client. The necessary data for this reverse translation is stored in BPF maps. Currently, we do not deploy reverse resolution of the modified packages. This means that while incoming traffic can be load balanced to backend servers, responses from these servers will not get back to the requesting client. The Cilium Project provides additional BPF programs to apply the reverse resolution. In our test scenario the reverse translation would need to be applied in the backend guest domains. This would require additional communication between the load balancer domain and backend domains, as sharing of BPF maps between domains is currently not supported.

Nonetheless, we show that it is possible to adopt our approach to existing applications with minimal effort. The following benchmark tests the performance of the offloaded load balancer.

6.4.1 Benchmarking a Load Balancer

To show how effective the BPF implementation of a load balancer is, we compare the Cilium load balancer, offloaded to the host domain, to a user-space load balancer. As a user-space load balancer we use *Nginx* [28], a web server and reverse proxy, which gained UDP support in v1.10. We run Nginx v1.10.1 in one guest domain and run two simple UDP receivers in another guest domain on two different ports. Nginx is configured to load balance the incoming traffic between both backends in a round-robin fashion. It does health checks by waiting for a response from the backend servers. To satisfy these health checks, the UDP receivers respond with a static message. We drop the outgoing response on the guest domain, so no traffic is sent back to clients to keep it similar to the BPF approach. The Cilium load balancer is offloaded to the host domain through `XenBPF`.

For each run, the external server generates ten million UDP packets directed to the guest domain running the load balancer. We measure the number of received packets on the listening ports on all domains using `tcpdump`. In addition to that the UDP receivers measure the number of packets they processed. Results show the mean values as well as 95% confidence intervals for the measurement.

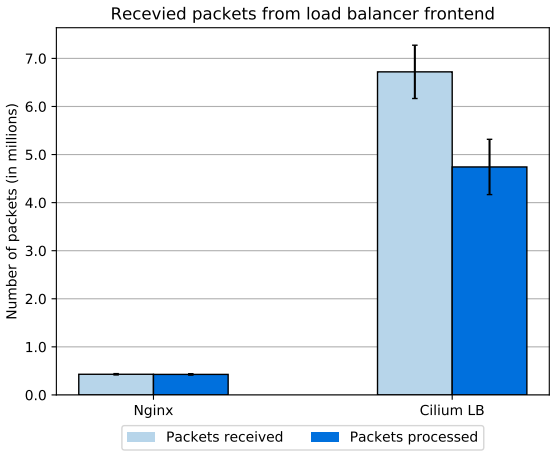


Figure 6.7 Number of packets received and processed by the two backends of the load balancer. The Cilium load balancer is able to sent out ten times as many packets as nginx, but the backend servers also drop a higher percentage of packets, because the user-space application cannot keep up with processing the traffic.

6.4.2 Results

Figure 6.7 shows the number of packets received on the system as reported by `tcpdump` and how many packets the backends where able to process. With ten million UDP packets sent out by the external server, the best result would be if all packets reach one of the receiver backends and get processed.

When Nginx does the load balancing, only 450000 packets are received on the backend machine. A small fraction of these packets, less than 1%, are dropped in the guest machine's kernel, the rest gets processed by the backend applications. Nginx does not rewrite any incoming packets, but instead sends the packet to the backend machine, waits for the response and then sends the response back to the client. The Cilium load balancer is much more effective in redirecting packets to backends. Between 60% and 70% of packets are received on the backend machine. However, the backend machine is not able to sustain the high rate of incoming packets. A large part of packets are dropped before the application is able to handle them. Between 40% and 50% of the original packets are handled in the receiving application.

This benchmark shows the effectiveness of handling packet processing and retransmission in the host, even though we run into bottlenecks in the user-space applications later on. Tuning the user-space applications for this high amount of traffic is required. The current BPF load balancer is configured for UDP only. With some additional setup and reverse translation it can be extended to work for TCP as well. Nginx can already handle both UDP and TCP connections.

6.5 Case Study: Key-Value Store Offloading

To test `XenBPF` with another existing application, we integrated offloading support into *Memcached*, a simple in-memory key-value store. Memcached is used by big companies such as Facebook to cache data and quickly retrieve it in applications [25].

The server listens on both TCP and UDP interfaces and provides a simple API to set keys to a value and retrieve the value again. The fork we base our work on already integrates a kernel caching layer into the code base. Inserted key-value pairs are cached in a BPF map. A BPF program responds to incoming UDP packets. If a packet contains a *GET* request wrapped in the Memcached binary protocol, the requested key is looked up in the BPF map. If a value is found, the buffer space of the incoming UDP packet is reused to generate a response packet. This packet is then sent out again and never gets to the user-space application. If at any stage parsing fails, no cached value is found, or the response would be too big, the packet is passed on unmodified and will be handled by Memcached in user-space. Due to the limited resources available to BPF programs, the size of keys and values in the BPF map as well as the number of pairs is limited. Memcached in the guest domain will periodically check the validity of cached key-value pairs and tries to keep most-frequently requested pairs cached there. This should speed up retrieval of the most frequently used values without the need to pass the network stack of the kernel. A similar approach was taken by Xu et al. [39], where a Linux kernel module responds to Memcached requests from within the kernel.

We replaced the BPF handling code in the modified Memcached with `libxenbpf`. The implemented kernel caching layer in Memcached runs multi-threaded. We therefore need the explicit locking around the communication channel with the host domain to avoid corruption of data in the shared memory. With these changes applied the extended Memcached was able to respond to requests with cached values from within the offloaded BPF program.

6.5.1 Benchmarking Memcached with XenBPF

For this case study, we test how many `GET` requests Memcached can process per second. We compare the unmodified Memcached running in a guest domain to Memcached with kernel offloading, once through a BPF program in the guest domain and once through `XenBPF`. Everything is based on Memcached v1.4.25. For the benchmark we use `memaslap`, a load testing and benchmark tool for Memcached. For the test `memaslap` sends only `GET` requests. The size of keys is limited to 22 bytes and the size of values to 30 bytes in order to fit into the BPF-powered in-kernel cache. Data is sent over UDP and uses the binary protocol for serialization. At beginning of the test a warmup round fills the cache with key/value pairs.

Each benchmark runs for 60 seconds and is repeated 10 times. Memcached is restarted between runs. We record the number of operations per second.

6.5.2 Results

Figure 6.8 shows the average operations per second and the 95% confidence intervals each test run achieved. On the test setup, plain Memcached running in a guest domain reaches a rate of about 74000 operations per second. The kernel cache using BPF, when running in the guest domain, is slightly lower at 72000 operations per second. When offloading the BPF program to answer Memcached requests to

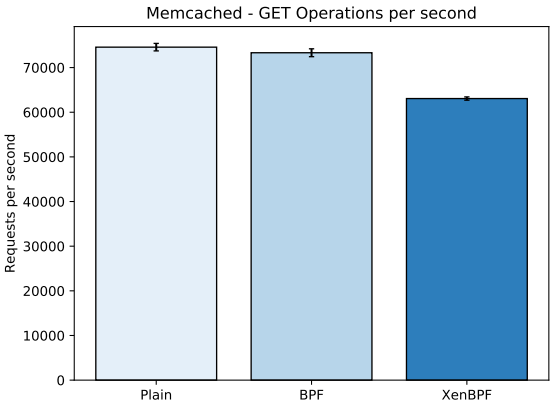


Figure 6.8 Number of `GET` requests per second Memcached can answer. Using the kernel cache results in less performance. Offloading into the host takes a huge performance hit.

the host domain, performance drops. Only 63000 operations per second can be performed.

In this test offloading does not improve the performance of the application. It even reduces performance significantly when offloaded into the host domain. The achieved performance is heavily dependent on the distribution of operations. The small size of the in-kernel cache means that not every request can be answered from within the BPF program. However, the BPF program has to process each incoming packet before it knows that it cannot answer it and has to pass it on to Memcached in the guest domain. This additional processing for cache misses is costly.

This benchmark shows that offloading work into the kernel of the host domain does not always result in a clear performance win. It depends on the intended use case and the work the BPF program does.

6.6 Summary

The results from the performed benchmark tests are promising. Offloading BPF programs from guest domains into the host's kernel reduces latency and CPU usage. Firewalling network packets as early as possible has a clear benefit for overall throughput and packet loss for the whole system and frees up CPU time, which can be assigned to other domains. The two case studies show that we can integrate our approach into existing software with minimal changes. Offloading network functions in a virtualized environment into the kernel can speed up network applications, but needs to be done with the use case in mind.

7

Related Work

In this chapter we present research related to offloading network functionality in virtualized environments. First, we discuss general research and application of NFV. Second, we look into research investigating optimizations for networking in virtualized systems. Finally, we show some of the work regarding recent developments regarding BPF.

7.1 Network Function Virtualization

Since the initial whitepaper describing NFV [10], some work has been done to investigate different approaches for implementation and deployment of network functions. One promising approach is ClickOS [21]. Martin et al. built a software middlebox platform on top of MiniOS, a unikernel bundled with Xen. Their minimal system provides the necessary functionality for network applications based on Click modules. In their evaluation they show that services built on ClickOS can achieve near line-rate performance. One huge advantage of the minimal unikernel approach is the reduced time to boot up new instances. This allows for rapid creation of instances based on demand. Contrary to our approach, the unikernel approach does not allow for easy reuse of existing applications.

Siracusano et al. follow a similar approach with *Miniproxy* [31]. They built a TCP proxy to accelerate TCP connections between endpoints. *Miniproxy* is built on top of the same MiniOS unikernel as ClickOS, that runs on top of Xen. It starts in tens of milliseconds and can run with just 6 MB of memory. These characteristics allow it to be deployed on demand in the cloud when necessary. The small resource requirements reduce the costs of operating it on cloud instances, where usage is billed by required memory, CPU and running time. Again, this work shows one implementation of a specific network function by reusing an existing unikernel operating system. It cannot interoperate with existing software developed for general purpose operating systems and cannot easily be reused for different tasks.

Cerrato et al. implemented a platform to run a large number of small network functions based on Intel's DPDK framework [9]. The DPDK framework provides software implementations for fast packet processing [17]. The platform built by Cerrato et al. consists of a virtual switch and network functions implemented as regular UNIX processes. They do not make use of a virtualization platform because of the overhead of regular VMs. Their idea is to have hundreds or thousands of small network functions in a single server, through which the virtual switch routes the incoming traffic. Using DPDK's network functionality they are able to reach a large throughput even with a large number of network functions on the same machine, but latency increases with the number of running network functions, making it less useful for production deployments. With our approach using Xen it is less likely to deploy hundreds or even thousands of offloaded network functions on a single machine.

7.2 Network Optimizations for Virtual Machines

In this thesis the networking for guest domains was identified as one of the major bottlenecks for efficient network functions in Xen. `XenBPF` can improve network processing by offloading it into the host. Other research has been done to improve network efficiency directly for both regular usage and usage in Xen.

The current implementation of networking under Xen is the result of work by Menon et al. [23]. They optimized the I/O channel between the driver domain and the guest domain by avoiding constant memory remappings and instead copy the data into shared memory pages. They also added functionality to use the natively available TCP checksum offloading and TCP segmentation offload from guest domains if available. Huge performance improvements are achieved even without the offload mechanisms. In this thesis, we identified the data copy in the virtual network device driver as one of the bottlenecks of guest networking in Xen and bypass it with offloaded programs.

While not directly related to improve networking for virtual machines, different solutions exist to improve performance of network functionality in general. We already mentioned DPDK, the *Data Plane Development Kit* by Intel [17]. The DPDK offers a framework for fast packet processing, where individual applications run to completion to process packets. Devices are accessed via polling. This eliminates the performance overhead of interrupt-based device control. It uses user-space I/O to have direct access to the network devices without additional overhead from switch to kernel-space. Netmap is a similar framework, providing fast network I/O exchange to user-space applications [29]. Netmap is implemented as a kernel module and eliminates bottlenecks of regular network processing in Linux and BSD systems. It provides preallocated resources to avoid costly allocation when handling network packets and eliminates memory copies by using shared buffers between the user-space application and the kernel. `XenBPF` does not directly optimize networking in Xen, but bypasses the bottleneck of guest domain networking by offloading functionality into the host. Improvements to the networking of Linux itself would immediately apply to `XenBPF` as well, whereas frameworks can provide the basis for other network function platforms.

Hwang et al. built *NetVM*, a platform for network functions based on the KVM hypervisor and the above mentioned DPDK [16]. An application, running in the hypervisor user-space, is responsible for polling the network device using DPDK functionality. Received network packets are read into a shared page area. The same application extracts basic information from the received packet to decide to which guest VM the packet should be sent. The targeted guest VM is then scheduled and the user-space application in that VM can process the packet further. When it finishes, it can ask the host to forward the packet to the next network function or transmit it over the network. NetVM is similar to `XenBPF` and allows network functions to run inside Virtual Machine (VM)s on top of a hypervisor. Network I/O is mainly handled in the hypervisor's user-space, similar to the host domain in Xen, and available to VMs through shared memory. In contrast to `XenBPF`, all guest VMs contain an application performing the task of a network function, whereas in `XenBPF` regular guests can run side-by-side to guest with offloaded functionality.

When implementing ClickOS, Martin et al. identified and eliminated additional bottlenecks in the Xen networking layer [21]. They minimized the number of hypercalls needed for transferring I/O data from the host to the guest by reusing already mapped memory pages. Additionally, they replaced the used software switch with their own implementation and removed other parts of the networking stack, such as the netback driver. Packet buffers are directly mapped from the switch implementation into the guest domain. The ClickOS switch is based on VALE, another software switch implementation, but is modified to the specific needs of ClickOS. To keep compatibility with Linux, they also provide a reimplementation of the frontend driver, that can interoperate with the new ClickOS switch. Replacing parts of the network stack of Xen with more efficient implementations will benefit `XenBPF` as well, as it does not eliminate all traffic to guest domains.

Ongaro et al. explore the relationship between efficient and fair scheduling in Xen and I/O performance for guest domains [26]. The default scheduler in Xen is able to fairly share processor resources between domains. However, with high network traffic the scheduler shows some shortcomings. Otherwise idle domains are immediately prioritized when they receive network packets. While the Xen scheduler achieves fairness for compute-intensive workloads, it cannot achieve the same fairness for I/O-intensive workloads. With their changes to the Xen scheduler I/O performance was improved. For `XenBPF` the scheduler changes are less relevant, as our offloading approach should avoid waking up the guest domains and instead do processing in the host machine. They did not address the problem of accounting time spent in `dom0` processing network packets on behalf of a guest domain, whereas `XenBPF` collects the necessary metrics and could potentially inform the scheduler.

Eiraku et al. proposed a method to improve networking for virtual machines running on hosted VMMs [13]. Their idea is to outsource the socket layer of a guest operating system and connect it directly to the socket layer of the host system. The socket outsourcing requires change to the guest operating system as well as the VMM. The low-level kernel primitives providing socket functionality are replaced with specialized functions, that directly call into the host system, where those calls are mapped to the host system's socket layer. By using shared memory additional copy operations can be avoided. Data written from the socket layer in the host is immediately available in the guest system as well. They briefly discuss a potential security issue. By combining the socket layer of the host and guest systems

they effectively share the same network configuration and therefore IPs. Dynamic rewrites of socket function calls restrict the usage to the guest machine's configured IP addresses. Other security issues are not discussed. Using shared memory for the packet buffers is similar to the shared memory used in the paravirtualized network driver in Xen. In `XenBPF` we avoided coupling guest operating systems directly and instead use offloading of programs to process packets.

7.3 BPF

The BPF environment is under heavy development and gains new features with every release of Linux. It is now used by multiple companies to speed up network processing. Bertin describes Cloudflare's use of cBPF for large-scale DDoS mitigation and presents a solution to use BPF and XDP to improve performance [4]. Cloudflare handles a huge amount of traffic to their content delivery network. Their reverse proxies handle traffic and forward it to their customers, first filtering malicious looking traffic. Edge servers send traffic to a central location for further analysis. The central server inspects the sampled traffics and generates rules for `iptables` with some parts compiled to BPF bytecode. These rules are pushed back to the edge servers, where they are applied to block traffic. They plan to switch to BPF in the future and use BPF maps to collect additional metrics on filtered traffic. With XDP, traffic could be filtered directly in the device driver, reducing CPU usage even further. The combination of user-space programs for metric collection and BPF programs for low-level packet filtering is enabled by `XenBPF` in virtualized environments. Further improvements to the BPF environment can be adopted to `XenBPF` as well.

Kicinski and Viljoen present a way of offloading BPF programs directly into hardware [19]. There exist NICs with integrated fully programmable processors (NPU). With hardware offloading, the kernel would be responsible to compile the BPF bytecode to the NIC's hardware after the verifier step. This compiled code is then transferred to the NIC directly, where it is executed by the NPU for every incoming network packet. If packets are dropped, they never leave the NIC and no further processing in the driver or kernel is necessary. One problem is the use of BPF maps, as access can happen form the BPF program on the NIC and the host system. The kernel would need to provide hooks for the general read and write operations to reflect changes to both sides. Because not every NIC will have BPF offloading capabilities, a fallback to in-kernel execution is provided. If all of this is provided transparently by the kernel, integration into `XenBPF` is straight forward. If a programmable NIC is detected, `XenBPF` would offload guest-supplied BPF programs to the device and otherwise would use XDP in the host or the current traffic control subsystem.

Ahmed et al. use BPF to dynamically build large virtualized networks [1]. Their network virtualization platform *InKeV* uses BPF implementations of network functions and dynamically inserts them into the kernel to build a larger virtualized network. A network graph can be stitched together using individual network functions. This way multiple virtual networks can be used over the same physical network. User-space components are used to provide configuration through BPF maps to the running BPF programs and collect statistics. Using BPF as their network functions

shows clear benefits over solutions that require kernel-/user-space context switching to process network packets. Latency in processing is reduced and the throughput increased. The reasons that BPF was used for InKeV are similar to the reasons it was chosen for `XenBPF`. BPF does not require changes to the networking stack of the operating system, while still providing better performance for packet processing. Additionally, existing user-space tools can be reused, as the higher layers of the system remain unchanged.

8

Conclusion

In this chapter we give a brief outlook on possible future work regarding the implemented framework to offload network functions into the host system in a virtualized environment. Finally, we summarize the results of this work.

8.1 Future Work

For future work, there are still interesting research topics left open, that could improve the approach. First off, `XenBPF` is built on top of Xen. It could similarly be implemented on top of other hypervisors, which face similar problems for their network traffic handling. Thus this approach could be easily adapted to VMs running on other platforms.

The additional security issues we mentioned in Chapter 4 regarding low-level handling of more sensitive BPF instructions should be implemented to harden the framework against misuse.

As every invocation on behalf of a guest domain is still accounted as CPU time of the host domain running `bpfd`, guest domains can now easily use more CPU time than what they are actually allowed to use. `XenBPF` already keeps track of the amount of packets processed per domain. This information could be fed into the Xen CPU scheduler. At the moment, Xen does not have an interface or mechanism to track this additional CPU time. Changes to the scheduler are necessary to allow this.

Lastly, upcoming changes in the toolchain should be addressed, such as using XDP for offloading directly into the driver of NICs or even offloading onto supported network cards.

8.2 Conclusion

In this thesis we presented the idea of deploying software implementations of network functions to Xen, a virtual machine monitor, able to host multiple guest systems on a single machine. We showed the shortcomings of the current networking stack and focused on improving performance of network packet processing by bypassing the existing network stack. We built `XenBPF`, a framework that allows guest machines running on Xen to offload implementations of network functions into the host machine. It uses an inter-domain communication channel to pass BPF programs from guest machines to the host domain, where they are attached to the NIC as network functions.

With our design of the framework we tackled all previously stated goals. Our evaluation demonstrated that `XenBPF` can reduce the latency when responding to network packets using offloaded BPF programs. Using it to drop incoming attack traffic has a positive effect on other valid traffic. Throughput was increased and packet loss was reduced significantly. By moving work into the host domain, less CPU time has to be assigned to a guest domain to which the traffic is targeted, freeing up resources that can be used by other colocated domains. We identified several potential security issues with our approach, showed ways to mitigate all of them and incorporated these mitigations into our design. By minimizing overall CPU usage for network processing and by measuring the CPU overhead of the host domain on behalf of guest domains, we can ensure fairness of packet handling and resource assignment. Additionally, we presented the user-space library `libxenbpf` and showed how it can be used to integrate `XenBPF` into existing applications.

With `XenBPF` we show that Xen is a suitable host to deploy software implementations of network functions and achieve competitive performance. However, offloading does not always have a positive effect and must be individually tuned to the intended use case. Further research and development in that area is necessary to bring it to a production-ready state and make it usable on general cloud networks.

Bibliography

[1] Ahmed, Z., Alizai, M. H., and Syed, A. A. InKeV: In-Kernel Distributed Network Virtualization for DCN. *ACM SIGCOMM Computer Communication Review 46*, 3 (2016).

[2] Amazon. Amazon Web Services: Overview of Security Processes, May 2017. `https://d0.awsstatic.com/whitepapers/Security/AWS_Security_Whitepaper.pdf`.

[3] Barham, P., et al. Xen and the Art of Virtualization. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 164–177.

[4] Bertin, G. XDP in Practice: Integrating XDP into our DDoS Mitigation Pipeline. In *Proceedings of netdev 2.1* (2017).

[5] Borkmann, D. act_bpf: add initial eBPF support for actions, Mar. 2015. Linux kernel, commit a8cb5f556b56.

[6] Borkmann, D. cls_bpf: add initial eBPF support for programmable classifiers, Mar. 2015. Linux kernel, commit e2e9b6541dd4.

[7] Borkmann, D. Advanced programmability and recent updates with tc's cls_bpf. In *Proceedings of netdev 1.2* (2016).

[8] Borkmann, D. On getting tc classifier fully programmable with cls_bpf. In *Proceedings of netdev 1.1* (2016).

[9] Cerrato, I., Annarumma, M., and Risso, F. Supporting fine-grained network functions through Intel DPDK. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on* (2014), IEEE, pp. 1–6.

[10] Chiosi, M., et al. Network Functions Virtualisation. `http://portal.etsi.org/NFV/NFV_White_Paper.pdf`.

[11] Chisnall, D. *The Definitive Guide to the Xen Hypervisor*. Pearson Education, 2008.

[12] Cisco Public. Cisco Global Cloud Index: Forecast and Methodology, 2015-2020. *White Paper* (2015).

[13] Eiraku, H., Shinjo, Y., Pu, C., Koh, Y., and Kato, K. Fast Networking with Socket-Outsourcing in Hosted Virtual Machine Environments. In *Proceedings of the 2009 ACM symposium on Applied Computing* (2009), ACM, pp. 310–317.

[14] Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A. Enforcing Performance Isolation across Virtual Machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware* (2006), Springer-Verlag New York, Inc., pp. 342–362.

[15] Han, B., Gopalakrishnan, V., and Ji, L. Network Functions Virtualisation (NFV); Terminlogy for Main Concepts in NFV. *ETSI GS NFV* (2013).

[16] Hwang, J., Ramakrishnan, K. K., and Wood, T. NetVM: High Performance and Flexible Networking using Virtualization on Commodity Platforms. *IEEE Transactions on Network and Service Management 12*, 1 (2015), 34–47.

[17] Intel. DPDK - Programmer's Guide, May 2017. Accessed: 2017-06-29. `http://fast.dpdk.org/doc/pdf-guides/prog_guide-17.05.pdf`.

[18] IO Visor Project. BPF Features by Linux Kernel Version, June 2017. Accessed: 2017-06-07. `https://github.com/iovisor/bcc/blob/e09564df601d6225b1c367d8faf45d6dcf9656f8/docs/kernel-versions.md`.

[19] Kicinski, J., and Viljoen, N. eBPF Hardware Offload to SmartNICs: cls_bpf and XDP. In *Proceedings of netdev 1.2* (2016).

[20] Lattner, C. *LLVM: An Infrastructure for Multi-Stage Optimization*. PhD thesis, Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, 2002.

[21] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), USENIX Association, pp. 459–473.

[22] McCanne, S., and Jacobson, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter* (1993), vol. 93.

[23] Menon, A., Cox, A. L., and Zwaenepoel, W. Optimizing Network Virtualization in Xen. In *USENIX Annual Technical Conference* (2006), no. LABOS-CONF-2006-003.

[24] Mijumbi, R., Serrat, J., Gorricho, J.-L., Bouten, N., De Turck, F., and Boutaba, R. Network Function Virtualization: State-of-the-art and Research Challenges. *IEEE Communications Surveys & Tutorials 18*, 1 (2016), 236–262.

[25] Nishtala, R., et al. Scaling Memcache at Facebook. In *nsdi* (2013), vol. 13, pp. 385–398.

[26] Ongaro, D., Cox, A. L., and Rixner, S. Scheduling I/O in Virtual Machine Monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (2008), ACM, pp. 1–10.

[27] Osanaiye, O., Choo, K.-K. R., and Dlodlo, M. Distributed Denial of Service (DDoS) Resilience in Cloud: Review and Conceptual Cloud DDoS Mitigation Framework. *Journal of Network and Computer Applications 67* (2016), 147–165.

[28] Reese, W. Nginx: the High-Performance Web Server and Reverse Proxy. *Linux Journal 2008*, 173 (2008), 2.

[29] Rizzo, L. Netmap: A Novel Framework for Fast Packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)* (2012), pp. 101–112.

[30] Salim, J. H. Linux Traffic Control Classifier-Action Subsystem Architecture. In *Proceedings of netdev 0.1* (2015).

[31] Siracusano, G., Bifulco, R., Kuenzer, S., Salsano, S., Melazzi, N. B., and Huici, F. On the Fly TCP Acceleration with Miniproxy. In *Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization* (2016), ACM, pp. 44–49.

[32] Starovoitov, A. eBPF syscall, verifier, testsuite, Sept. 2014. Linux kernel, commit b4fc1a460f.

[33] Starovoitov, A. net: filter: rework/optimize internal BPF interpreter's instruction set, Mar. 2014. Linux kernel, commit bd4cf0ed331a2.

[34] Starovoitov, A. bpf: introduce BPF_PROG_TEST_RUN command, Mar. 2017. Linux kernel, commit 1cf1cae963.

[35] The Cilium Project. Cilium, June 2017. Accessed: 2016-05-30. `https://github.com/cilium/cilium`.

[36] Whitaker, A., Shaw, M., Gribble, S. D., et al. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Tech. rep., Technical Report 02-02-01, University of Washington, 2002.

[37] Xen Wiki. Network Throughput and Performance Guide, June 2017. Accessed: 2016-05-31. `https://wiki.xen.org/wiki/Network_Throughput_and_Performance_Guide`.

[38] Xen Wiki. XenStore, June 2017. Accessed: 2016-06-29. `https://wiki.xen.org/wiki/XenStore`.

[39] Xu, Y., Frachtenberg, E., and Jiang, S. Building a high-performance key–value cache as an energy-efficient appliance. *Performance Evaluation 79* (2014), 24–37.

A

Appendix

A.1 List of Abbreviations

API Application Programming Interface
BPF Berkeley Packet Filter
DDoS Distributed Denial of Service
ETSI European Telecommunications Standards Institute
I/O Input / Output
JIT Just-in-Time
MAC Media Access Control
NFV Network Functions Virtualization
NF Network Function
NIC Network Interface Card
NPU Network Processor Unit
TSP Telecommunications Service Providers
VMM Virtual Machine Monitor
VM Virtual Machine
XDP Express Data Path
cBPF classic Berkeley Packet Filter
eBPF extended Berkeley Packet Filter
vCPU Virtual CPU