Unikernel Firewall Performance Evaluation: IncludeOS vs Linux

Tobias Tambs

Thesis submitted for the degree of
Master in Network and System Administration
30 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2018

Unikernel Firewall Performance Evaluation: IncludeOS vs Linux

Tobias Tambs

Unikernel Firewall Performance Evaluation: IncludeOS vs Linux

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

In today's digital world cloud computing is key and it is growing. At the same time we are trying to maximize resource utilization. Running big, old general-purpose operating systems on virtual machines in the cloud is not a good way to do this. Using for instance Linux and iptables for firewalling can very much limit throughput and latency on a network. Using unikernels instead can revolutionize cloud computing, saving a huge amount of resources while providing better performance and security.

In this thesis, we look at unikernels for enhancing network performance in router- and firewall-VMs, while greatly minimizing resource usage compared to Linux' and iptables, ipset, and the newer nftables. Using a server running VMware's ESXi hypervisor, we set up a network of VMs consisting of a client and a target running Ubuntu and firewalls running Ubuntu Server, IncludeOS and Alpine Linux. Iperf, Netperf and hping3 was used to measure network performance.

Using only a fraction of the resources of the Linux VMs, the IncludeOS unikernel showed that it can manage large traffic volumes while blocking thousands of ports or IPs without negatively affecting throughput. In fact, our IncludeOS image of just over 3 MB in size managed 15 times the throughput of Ubuntu Server (850 MB image size) when running an iptables firewall with 50 000 blocked IP addresses. Nftables and ipset were quite closely matched, but they are still slower than IncludeOS. Iptables severely limits throughput when handling large sets of filter rules.

Using unikernels like the tiny but powerful IncludeOS can very much help cut costs in data centers running thousands or more single-purpose VMs like firewalls by providing better network performance and security while imposing almost no overhead.

Contents

List of Figures

Listings

Preface

This master thesis is written as part of the master's programme *Network and System Administration* in the spring semester of 2018. The master's programme is a collaboration between the Faculty of Mathematics and Natural Sciences at UiO and the Faculty of Technology, Art and Design at Oslo Metropolitan University.

Acknowledgements

I would like to thank my supervisors Hårek Haugerud and Anis Yazidi for their support and interesting and fun conversations throughout the semester. Thank you for introducing me to the great team behind the IncludeOS unikernel. Thanks Hårek for providing me with the physical hardware I needed to conduct my experiments. Thanks to IncludeOS and CEO Per Buer for helping me realize this project by providing software, training, support and insights.

Many thanks to Amir Maqbool Ahmed, department engineer at Oslo Metropolitan University for being my friend and teacher, for motivating me and for helping me with the thesis.

Thanks to my fellow students, who have become my good friends over the course of these past two years. We've had a lot of good times together, both in and out of the classroom!

To my girlfriend and all of my friends that have helped and motivated me throughout this master's programme – thank you for making this an exciting and memorable time!

Thanks to my family for supporting me and for always being there for me. Thank you dad for proofreading my thesis.

To my dear uncle Kristian who passed away last summer. You have always been an inspiration to me. Thank you for all your kindness, knowledge and advice, and for pushing me to continue my studies.

Chapter 1

Introduction

Since the first computers went on sale, the price of computer hardware has been decreasing rapidly, while the availability has been shooting towards the skies. In the early days of computers people had to share large, expensive servers and mainframes, but now most of us have our own smartphones, tablets and laptops.

The majority of these devices are designed and used to perform many different tasks for multiple users, like making phone calls, sending messages, navigating and running a host of other applications and programs. Thus the operating systems of such devices must be designed to serve a wide set of users, programs and hardware, providing many different underlying services. Take Linux as an example. This is an operating system used on everything from low-power mobile devices to personal computers to powerful servers running in huge data centers [1]. We call these types of OSs for general-purpose operating systems or GPOS for short.

These general-purpose operating systems are flexible, supporting many different standards, technologies, hardware, protocols, services etc., but they also have some serious drawbacks. Having to support such a wide array of platforms and uses, the operating systems tend to become big, bloated and heavy to run. Windows 10, for instance, requires at least 2 GB of RAM and 20 GB of available hard drive space for a base installation of the 64-bit version [2]. Providing tons of different services also means that general-purpose OSs have large attack surfaces – making it relatively easy for an attacker to find vulnerabilities that can be exploited [3]. A number of these exploits can quite easily be acquired on the dark web these days, allowing even intermediately skilled hackers to perform advanced attacks on existing platforms [4].

Many businesses rely on their applications to take care of time-sensitive tasks. These tasks may include networking, firewalling, running databases, web servers, trading platforms and different military applications. One thing all of these tasks have in common is that you want them to run

as quickly and securely as possible, which is not exactly GPOS' strong suits. The same is true for lots of different sensor and information systems. Even simple information screens and ticket machines used in public transportation relies on general-purpose operating systems, making them open to a number of different types of attacks. This was the case when public transport information screens fell victim to the big 'WannaCry' attack in 2017.[5]

As more of our computing is shifting towards mobile devices, battery life has also become a critical factor for many. With the rise of the Internet of Things (IoT) we see a proliferation of typically smaller, battery powered devices which could greatly benefit from running a lightweight OS which is not using power on a host of unnecessary features.

The same is true for virtual machines, which in most cases don't really need to run these big, old GPOSs, since most VMs now are single-purpose, meaning they only have a very specific task, like running a web server or a database.

Utilizing up and coming unikernel technologies, many of these problems probably can be remedied. A unikernel is designed specifically for running one application, without loads of interfering processes and unnecessary drivers, services and resulting context switches. The unikernel allows applications to be deployed directly on a hypervisor or even run on bare metal (hardware).

One area that may see big performance improvements with the use of unikernels is networking. Using unikernels for handling web and DNS servers, load balancing, firewalls etc. can potentially bring huge advantages when it comes to latency, throughput and security. In this thesis, firewall performance on IncludeOS' unikernel will be tested and compared with the widely used Linux firewall based on netfilter.

1.1 Problem definition

Since the introduction and commercialization of virtualization on a large scale, the way we do computing has changed drastically. Where people before had to buy their own infrastructure, including servers, network equipment, etc. and operate and maintain this hardware, they can now choose to rent only the resources they need from cloud providers such as Google, Amazon and Microsoft – greatly improving cost efficiency. We can now easily deploy, manage and migrate virtual machines based on current needs, like latency, traffic, temperature and power management.

While this is all well and good, there are aspects of cloud computing that can be streamlined and improved upon. One big issue with today's virtualization is that we are virtualizing software that was never meant to operate in a virtual environment, whereof the operating systems

themselves are probably one of the biggest drawbacks. "Old" general-purpose operating systems were developed to accommodate multiple users and support a wide range of hardware and services – which of course was very important when companies had to buy millions of dollars worth of equipment and try to utilize that for everything it was worth.

Today, a big portion of VMs are meant to be single-purpose appliances, providing just one service; like a web server, load balancer, DNS server, database, etc. Many of these services benefit from being highly flexible, relying for instance on instant spawning/booting of VMs as traffic load to a web service increases. Here also lies the problem: Spawning and booting VMs that contain large operating systems with tons of unnecessary features greatly reduce the benefits we should see from virtualizing. What we should have is specialized images, designed to do only the job they are meant to do, and nothing else.

According to several recent studies, articles, tech blogs and -websites, the introduction of the relatively new unikernel technology has the possibility to drastically improve performance in single-purpose appliances while using only a fraction of hardware resources like CPU, memory and disk space. Unikernels ditches all unnecessary software components and integrates with an application to form a bootable image containing only the bare minimum of code it takes to run the application.

The main goal of this thesis is to evaluate the network and firewall performance of the IncludeOS unikernel and to see how it compares to the traditional and widely used Linux iptables. In addition to this, tests will be performed on the newer nftables, and also on iptables with ipset.

Chapter 2

Background

2.1 Technologies

2.1.1 General-purpose operating systems

An operating system is what lies between the applications/users and the lower software and hardware layers. An OS incorporates drivers that communicate with the underlying hardware, making it easier for developers to write applications. It also usually provides a user interface and some means for users to interact with the OS. For regular users, OSs include graphical user interfaces (GUI) that make interaction easier than with the more traditional command line.

A general-purpose operating system (GPOS) is an operating system designed to be able to support multiple users, applications and services. To do this, a GPOS needs to be able to support a wide range of applications, services and hardware – often including thousands of different device drivers for things like monitors, keyboards, mice, microphones, NICs, webcams and other peripheral devices and integrated hardware.

The most common GPOSs for personal computers are Microsoft Windows, macOS (previously OS X) and various Linux distributions, like Ubuntu, Fedora, Red Hat, Debian etc. For mobile phones, Android and iOS are by far the most common.

A GPOS is typically split in two: An operating system kernel that runs in *privileged* mode and a user space that runs in *unprivileged* mode. The main reason for building an OS this way is for security reasons, restricting user processes from accessing the kernel, and therefore the hardware. The drawback of this scheme is the speed/latency penalties it introduces. The process of transferring data between the kernel and the user space is expensive, wasting resources on context switches etc. A context switch happens when the CPU switches from working on one process or thread to another. This process involves saving the state of the previous process so the CPU can continue working on that process from the same point later

on, effectively "pausing" the process. Context switches can be invoked in different ways, for instance when the CPU receives an interrupt or when switching from user mode to kernel mode.

Most computing done today also does not require a multi-user environment. Usually new applications are not added to existing servers or VMs, but rather deployed to new servers or new VMs, effectively making them "single purpose".

2.1.2 Linux

Linux was initially just an operating system *kernel* but over time has evolved to become a whole family of distributions (or distros) built around the Linux kernel. The first Linux kernel was released by Linus Torvalds on September 17, 1991. As opposed to Windows and MacOS, Linux is essentially free and open-source, which has led to numerous distributions from many different developers. Because of its open-source and highly flexible nature, Linux has become extremely popular in a wide range of devices and systems, and as of today it is ported to more platforms than any other operating system – from TOP500 supercomuters to desktop PCs, cellphones and embedded devices. Much because of the Android OS, Linux is also the operating system installed on most devices worldwide.

2.1.3 Netfilter

Netfilter is a framework implemented in the Linux kernel that provides several different network components, including connection tracking, packet filtering, network address translation, port translation etc. Because of netfilter being a part of the Linux kernel, it is widely used in networking applications. All the components needed for building your own router or firewall is included with netfilter and for many or most uses, there is no need to download additional packages.

Netfilter's user utilities consists of iptables, ip6tables, ebtables, arptables, ipset and nftables.

2.1.4 Containers

A container is an executable package of software which includes all components necessary to run it, including code, runtime, system tools, system libraries and settings. Software inside a container will always run the same way, no matter what platform the container runs on. The container provide isolation between its running software and the software layers below.

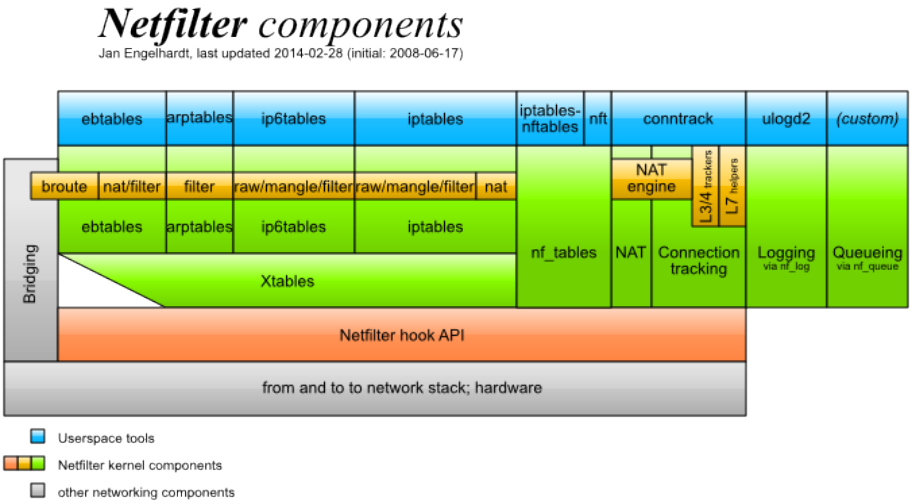

Figure 2.1: Netfilter components [6]

Where virtual machines abstract physical hardware, containers provide abstraction at the application layer, packaging code and dependencies together. Multiple containers can run on the same host while for instance running different Linux distros inside the containers, sharing a Linux kernel but running as individual processes in user space. VMs on the other hand include the whole software stack, from a guest OS to the top-level applications.

2.1.5 Unikernels

A unikernel is a kind of a merge between an application and an operating system into one image. This essentially makes the application bootable, since it includes (only!) the necessary drivers and libraries for the underlying hardware or hypervisor.

Unikernels have some native advantages when it comes to security. These advantages come from the fact that a unikernel typically contains much less code than a general-purpose operating system, including only the code and libraries required to run the app it is built to run. This means that there are no unnecessary drivers for Bluetooth, floppy, CD-ROMs, NICs or the thousands of other hardware devices out there. There is also no file sharing, shells or other protocols that could provide additional attack vectors. In other words: The attack surface is much smaller in a unikernel than in a traditional OS running the same application, though this of course does not mitigate potential threats against the hypervisor it could be running on.

Unikernels are also immutable, which means that the running code cannot be altered. Updating a unikernel involves making your changes to the code, building it into a new image, downloading the new image to the

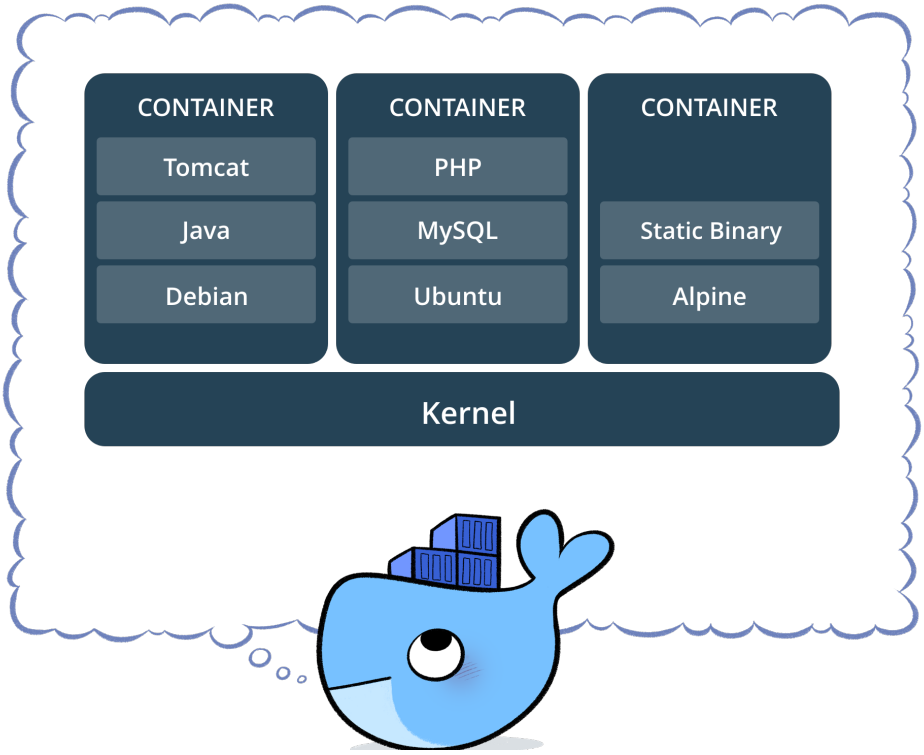

Figure 2.2: Containers [7]

VM and replacing the old image with the new one. In IncludeOS with LiveUpdate, this is done by pushing the new image to the IncludeOS instance, where it is saved in memory. The state of the running application is then stored in memory as well, including open sockets, file descriptors etc. After that, the new image is booted, the state is restored and the application resumes execution. Replacing an image happens in a few hundreds of a second, and can maintain active TCP connections in a router or firewall.

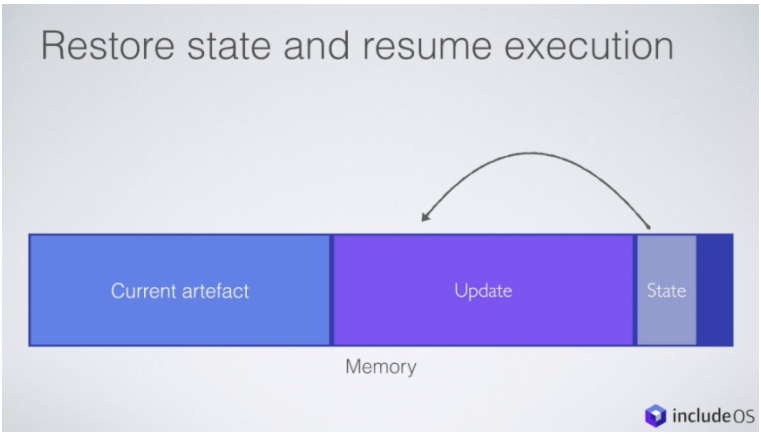

Figure 2.3: IncludeOS LiveUpdate [8]

Because there are no system calls or context switches in unikernels – they instead just switch to another part of the program – there is no need to flush caches and page tables etc., giving them a performance edge over GPOSs [9].

Unikernels/libOS also allow applications to access hardware resources "directly", without having to make privilege transitions for moving data between a user and a kernel space, making them a potential great choice for applications where performance and predictable performance is essential.[1]

Since unikernels do not need to initialize and run lots of unnecessary services, they can achieve incredibly fast boot times. Thanks to IBM Research, IncludeOS is now able to run the Solo5 unikernel interface with the ukvm back end, cutting the already low boot times of 300 ms down to just 11 ms [10]. This in turn allows IncludeOS to be used in an even wider set of applications an creates new possibilities in for instance handling web traffic. With boot times that fast, a web server could be booted as late as when a DNS request for a web site comes in.

2.1.6 IncludeOS

Built on C++, IncludeOS is a "zero overhead", single address space library operating system created by Alfred Bratterud et al. [11]. Already in use with Basefarm for firewall and load-balancing, IncludeOS shows great promise in improving latency characteristics while requiring a fraction of the resources used by other systems/OSs [12]. Networking is a key aspect for the IncludeOS team, and at the time of writing, IncludeOS is probably the only unikernel available which can (easily) be used as a router/firewall/load-balancer.

Features like the in-house developed LiveUpdate feature for quick-and-easy IncludeOS updates and the NaCl configuration language should make IncludeOS a strong competitor in the quite new unikernel market.

A minimal version of IncludeOS including network stack can be as small as 700KB, with a memory footprint of no more than 16 MB for a simple web API.

2.1.7 Configuration as Code – VCL and NaCl

Varnish Software develops and delivers caching technology for some of the world's most visited websites. One of the reasons why Varnish is so fast is that their (domain specific) configuration language *Varnish Configuration Language (VCL)* is based on C, which is then compiled – making it "lightning fast"[13].

IncludeOS does the same ting with their unikernel configuration language NaCl. While originally creating a router, firewall and load-balancer

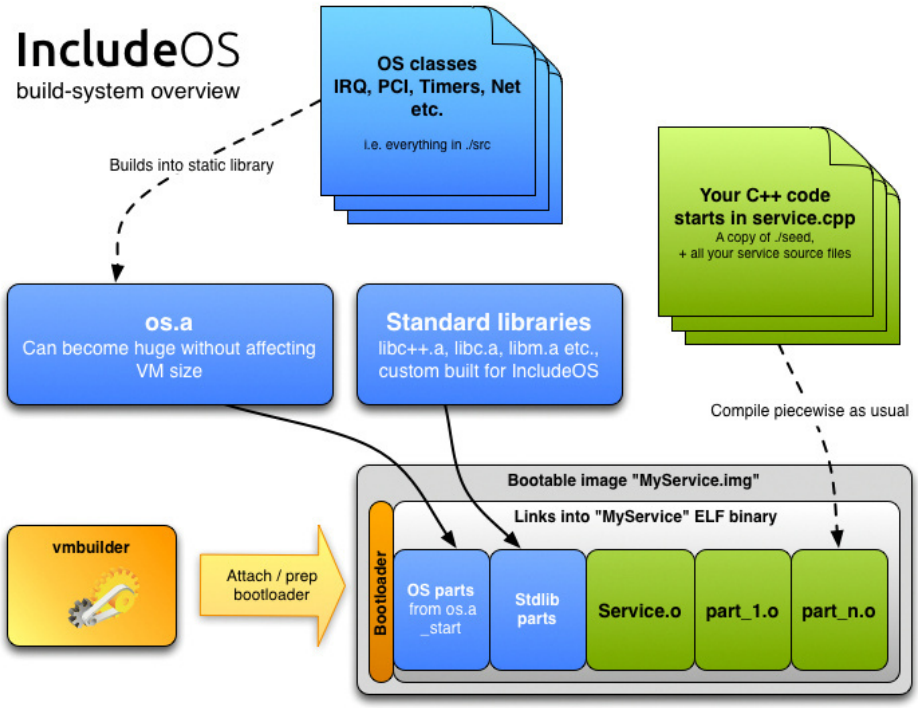

Figure 2.4: IncludeOS build-system overview

in C++, this was not deemed ideal for neither IncludeOS nor their customers.[14]

Creating a new language for configuration in a unikernel, some considerations were needed. One of the fundamental characteristics of a unikernel is that there is at runtime no distinction between code and configuration. Each time a unikernel needs to be updated, the whole image needs to be rebuilt and deployed – a process made easy in IncludeOS using their LiveUpdate feature. This should then also be a feature of the configuration language. Another motivation is that running the configuration through a compiler can optimize the code for that specific system, resulting in a performance advantage over traditional systems using configuration files.

The result was for IncldeOS to use the same principle as Varnish VCL: Let users write configuration in the (high-level) NaCl configuration language, transpile the config into C++, compile the resulting C++ and link it into the binary.[15]

2.1.8 Virtual Machines

Virtual Machines (VMs) allow for much better resource utilization than traditional servers provided strong algorithms for power management, migration of VMs across physical hardware, cooling schemes, etc. Instead

of companies running their own servers, wasting resources when those are not being used or are underutilized, these companies can instead rent hardware resources from cloud providers such as Amazon, Google and Microsoft, making sure they only pay for the resources they need.

Typically, VMs contain "large" guest OSs like Windows or Linux, even though a VM's purpose may only be to execute a simple task like hosting a web server or database (single-purpose), wasting a lot of resources on booting and running the OS itself. Additionally virtualizing introduces an extra software layer that typically sits between the guest OS and the hardware, taking up some resources in itself.

A key advantage of virtualizing is the possibility to spawn and tear down VMs on demand. A company providing a web service can easily build a load balancing system that automatically spawns new VMs as traffic to the service increases, and tear down VMs as traffic decreases. This helps free up resources and allows the company to efficiently run their service without having to pay for resources they do not use.

In the above example, it is probably vital for the company providing the service that VMs can be spawned quickly enough to manage the incoming traffic in real time. Spawning new VMs that themselves have to boot GPOSs like Linux and Windows with all the necessary components inflicts a huge time penalty, and may in some cases take too long for the service to seamlessly handle fluctuations in traffic. The result is often that the company would need to run more VMs than necessary to be able to cope with traffic spikes – causing a wasteful resource usage.

Using unikernels instead of traditional OSs in VMs may prove to be a huge advantage here, with boot times often in the order of milliseconds.

2.2 Securing virtual machines

Virtualization is an essential part of computing today but, as for physical machines, VMs also have to be properly secured. In addition to the OS and software running inside VMs, what lies beneath the VMs – the hypervisor – also needs to be secured.

As opposed to earlier times when virtualization was done in software, virtualization capabilities are now built into most CPUs. The hypervisor's primary job is to provide a management interface to the hardware primitives. The result of the isolation of CPU, memory and I/O being done at the hardware level is a considerably smaller attack surface than with previous techniques. Virtualization extensions in Intel (VT-x) and AMD (AMD-V) CPUs don't enable VMs to run at Ring-0. Only the Virtual Machine Monitor (VMM) runs at a hardware privilege level while guest OSs run at a virtualized privilege level. When a privileged instruction is executed in a guest OS, it is trapped and emulated by the VMM.[16]

In addition to securing a server with a "physical" firewall, traffic between VMs on the same server needs to be filtered, to prevent malware on one VM to spread *horizontally* to other VMs. Both antivirus software and firewalls can be implemented in software in the VMs, improving security inside the server [17]. Another approach to inter-VM security is to deploy dedicated firewall-VMs. Using unikernels for this purpose could provide ultra fast firewalls with minimal overhead.

Several virtualization and security vendors provide security solutions aimed towards hypervisors and virtual machines:

Check Point *CloudGuard* protects against "lateral spread of threats within virtualized environments and private cloud datacenters"[18].

Working with Amazon Web Services (AWS), Microsoft Azure and VMware Cloud on AWS, Trend Micro has developed and optimized *Deep Security*, that focuses specifically on datacenter and cloud security.[19]

McAfee, Citrix, VMware and others also have similar products securing cloud infrastructure.

ESXi security

In the whitepaper titeled *Security of the VMware vSphere Hypervisor*, VMware explains the security features of its vSphere software suite, including the ESXi hypervisor. In the executive summary of the whitepaper, they list the following features (citation)[16]:

- Secure isolation of virtual machines at the virtualization layer. This includes secure instruction isolation, memory isolation, device isolation, and managed resource usage and network isolation.

- Configurable secure management of the virtualized environment. This includes secure communication between virtualization components via SSL; host protection via lockdown mode; and least privilege by a fine-grained, role-based access-control mechanism.

- Secure deployment of the ESXi software on servers through use of various platform-integrity mechanisms such as digitally signed software packages and Intel Trusted Platform Module (TPM)–based trusted boot.

- Rigorous secure software development life cycle that enables developers to create software using secure design and coding principles such as minimum attack surface, least privilege, and defense in depth.

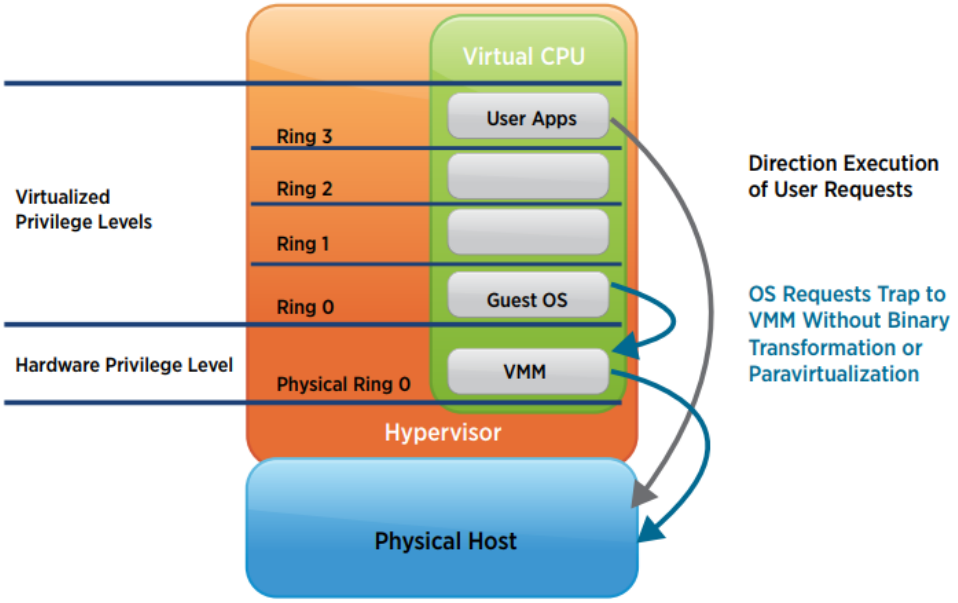

Figure 2.5: VMware instruction isolation [16]

2.2.1 Next-Generation Firewalls

A Next-Generation Firewall (NGFW) is a network security system that is able to enforce security policies on multiple levels of the OSI model, protecting networks and devices against modern, sophisticated attacks. Paloalto's NGFW for instance "classifies all traffic, including encrypted traffic, based on application, application function, user and content"[20]. That means Next-Generation firewalls must be able to inspect traffic all the way up to the application level.

In Check Point's NGFW, administrators have the ability to look at users, groups, applications, machines and connection types and assign permissions based on these[21], giving a high degree of granularity. NGFW typically includes both intrusion prevention and intrusion detection systems with deep packet inspection. Junpier's NGFW is available in both virtual and physical form factors.[22]

2.2.2 Virtual firewalls

Firewalls are typically sold as physical devices, but today, several virtual firewall products exist. Trend Micro has a system where a driver applies security updates to a host. The host then applies these updates directly to VMs without agents running on each VM, so automatic security updates are instantly available when new VMs come online. This is a huge advantage for virtualized systems, where VMs are frequently spawned and teared down.

VyOS

There are several approaches to virtual firewalling. One of them – VyOS – is a network operating system based on GNU/Linux that is built for securing networks. According to the creators, VyOS can be installed on physical hardware or it can be run in a VM, for instance in a cloud environment[23]

2.3 Related works

2.3.1 iptables, nftables and ipset

A comparison between iptables and nftables in Red Hat Linux was done by a developer in the Red Hat Developer Blog[24] to show the differences in performance in the two filtering tools under different circumstances. Several different tests were run through several different firewalls comprised of address and port filters. Scaling the number of rules from zero to 1500, throughput tests were run at 20 different points, ex. at 75 rules, 150 rules, 225 rules etc., and used as points in a graph, showing the performance as the rule count increased. The filters used were iptables, both with and without ipset, and nftables also both with and without nftset.

With simple rules, filtering on source IP addresses, iptables performed somewhat better than nftables, with iptables increasing the lead as the rule set grew larger.

In similar tests, filtering on destination ports instead of IP addresses and using multiport in iptables and set in nftables, iptables showed a linear decrease in performance as the number of rules increased, while nftables started out with a bit lower throughput than ipset, but maintained the same performance all the way through the test.

2.3.2 Performance Evaluation of Netfilter

In the paper *Performance Evaluation of Netfilter* by Raik Niemann, Udo Pfingst and Richard Göbel, the authors are evaluating the throughput performance of netfilter under different conditions and using UDP, TCP and SCTP. They built their own testing tool that resembles iPerf and netperf, but their testing tool also incorporates a gateway in addition to just a client and a server, to allow extended testing. The testing complies with the guidelines in RFC3511

Testing is performed with 1: the router performing only forwarding, 2: stateless iptables filter with source, destination and protocol specified, and 3: the same as 2 plus QoS marks.

The results in this paper show that SCTP was in average 32 percent slower than TCP, which itself was 8 percent slower than UDP across all experiments. IPv6 was on average 9 percent slower than IPv4. The overhead when running netfilter was minimal compared to when the router host was idling; but the router host in this case used network adapters with network processors that validate network data packets, verify network headers and calculate checksums – tasks that would otherwise be done by the operating system.

The main findings were that the throughput performance of netfilter was independent of the transport protocol, frame size and address family with simple netfilter rules, and that the decrease in throughput as the number of *simple* netfilter rules increase are roughly linear. The throughput loss per simple netfilter rule was 0.05 percent for IPv4 and 0.03 percent for IPv6.[25]

2.3.3 A Performance Evaluation of Unikernels

Ian Briggs, Matt Day, Yuankai Guo, Peter Marheine and Eric Eide, the authors of *A Performance Evaluation of Unikernels*, looked at network performance of two unikernels, namely MirageOS and OSv compared with Ubuntu Linux. The tools they used were iPerf for bandwidth measurements, queryperf for DNS response measurements and httperf for http/server performance testing.

In the conclusion, the authors points out the immaturity of the unikernels they tested, and that it is unreasonable to draw general conclusions about unikernels, but that they still show some promising results.

OSv significantly outperformed Linux in every test category, and it didn't require much work to port the test tools to this unikernel. Due to OSv being in alpha at the time of testing, the authors experienced some bugs, but they conclude that the unikernel will probably be very attractive for high-performance applications at a later stage [26].

2.3.4 Netfilter Performance Testing

Netfilter Performance Testing by József Kadlecsik and György Pásztor is a study of the performance of netfilter performance compared to several other "solutions", including nf-hipac, Compact Filter, iptables with classifiers and ipset. The tests looked at requests per second measured with httperf and were performed both with and without conntrack (connection tracking), NAT and firewall filters.

The tests in this paper show that on their test setup the maximal performance were halved when using conntrack, compared to just using plain routing. They also find little difference between tests where just conntrack is enabled and where conntrack is enabled in addition to

filtering, when not using "excessive filtering", meaning not too many filter rules. When running tests with NAT also enabled, in addition to conntrack and filter, the performance went down another 50 - 60 percent.

Looking at iptables scaling, the researchers show the "clearly non-scaling behavior of iptables" caused by the way iptables processes rules. Iptables processes rules linearly, meaning that increasing the number of rules decreases both req/s and throughput dramatically. Testing of ipset and nf-hipac shows virtually no decrease in performance even when the rule sets get very big (16k+ rules).[27]

2.3.5 Unikernels: Library operating system for the cloud

Building on previous work with library operating systems, Madhavapeddy et al. in *Unikernels: Library operating system for the cloud* created the Mirage unikernel to address the lack of specialization in current VMs that are meant to be single-purpose. Their main contributions are 1: the unikernel approach for providing sealed, single-purpose appliances suitable for cloud services, 2: evaluation of an implementation of the techniques using OCaml, showing the performance benefits, and 3: libraries and language extensions supporting systems programming in OCaml.

The authors unikernel architecture combines static type-safety with a single address-space layout that can also be made immutable. With Mirage's suite of type-safe protocol libraries, they demonstrate that running on a hypervisor, they can overcome hardware compatibility issues that made earlier library operating systems impractical to deploy.

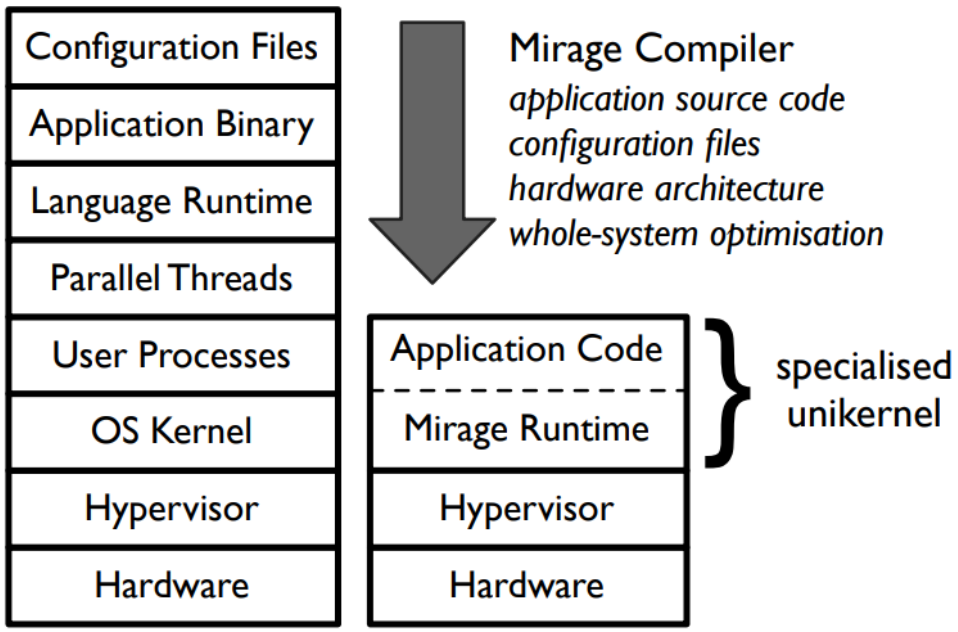

Figure 2.6: Traditional VM vs unikernel approach [28]

Evaluating their prototype unikernel with micro-benchmarks and applications that test DNS, OpenFlow and HTTP performance, the authors find that sacrificing backward compatibility allow them to increase performance while also improving security of cloud services facing towards the Internet. This is done using standard network protocols such as TCP/IP, and the authors are able outperform standard Linux network tools while using much smaller VM images, both in regards to boot times, DNS performance, httperf session create rate and throughput (connections per second). [28]

2.3.6 Cloud Cyber Security: Finding an Effective Approach with Unikernels

The goal of the study *Cloud Cyber Security: Finding an Effective Approach with Unikernels* by Bob Duncan, Andreas Happe and Alfred Bratterud is to identify and tackle some of the privacy and security issues related to cloud computing and the Internet of Things.

While security in IT is always improving, many IT users are not well informed about information security, and people themselves are often the biggest security vulnerability for companies. Not aware that they sit behind a company firewall, employees of the company may well export data to clouds, where they may not be aware of security implications like not being in control of who runs and has access to the cloud software and underlying hardware. Complex security systems, documentation and regulations may also well be a overwhelming for users not educated in IT, making it easy to lose oversight.

With the rapid expansion of Internet of Things (IoT)-devices, security is often overlooked in favor of pushing devices out as fast as possible. Take for instance web cameras, where search engines exists for finding and accessing devices using that either uses default user names and passwords or are completely open [29]. There have also been examples lately of IoT devices being used in massive botnets, launching huge DDOS attacks on Internet infrastructure [30] [31].

The researchers suggests an approach to addressing these problems by using unikernel-based systems, reducing complexity, attack surface and resource usage compared to traditional systems. The following definition of a unikernel is used in the study:

- a minimal execution environment for a service
- providing resource isolation between those services
- offering no data manipulation on persistent state within the unikernel, i.e. the unikernel image is immutable
- being the synthesis of an operating system and the user application

- only offering a single execution flow within the unikernels, i.e. no multitasking is performed

With these advantages in security, including isolation, immutability etc. and also the performance gains and energy efficiency compared to traditional virtualization appliances, the team concludes with unikernels being a smart approach to solving many of the current issues in cloud computing.[32]

Chapter 3

Approach

3.1 Hardware and network setup

3.1.1 Server and hypervisor

Considering available time and equipment, it was decided to set up a virtual network on one of OsloMet's Intel servers. The IncludeOS team has shown good results with the VMware ESXi hypervisor, and they recommended it for the setup. VMware is also one of the hypervisors currently supported by the IncludeOS unikernel. Having access to the server, the latest version of ESXi was downloaded and installed on the server. The server specifications are as follows:

- Dell PowerEdge R630
- 40 CPUs x Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz
- 128 GB DDR4
- ESXi-6.5.0-20170104001-standard (VMware, Inc.)

Managing the server can be done either via an ESXi shell which provides a browser GUI or a SSH shell. The ESXi shell is pretty straight-forward to use, and didn't take a long time to understand, so that is what was used for management, setting up the network and VMs, throughout the project.

3.1.2 Network and VMs

The test network topology outline was proposed by Per Buer at IncludeOS, and complies with the dual-homed setup in RFC 3511.

Networks can be set up in ESXi by creating virtual switches and port groups, connecting VMs to port groups and port groups to switches.

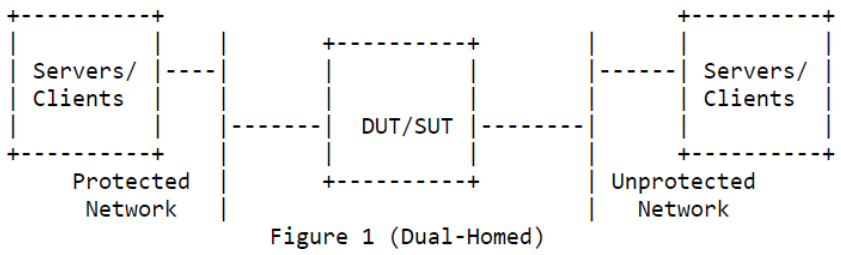

Figure 3.1: RFC 3511 Dual-Homed test setup [33]

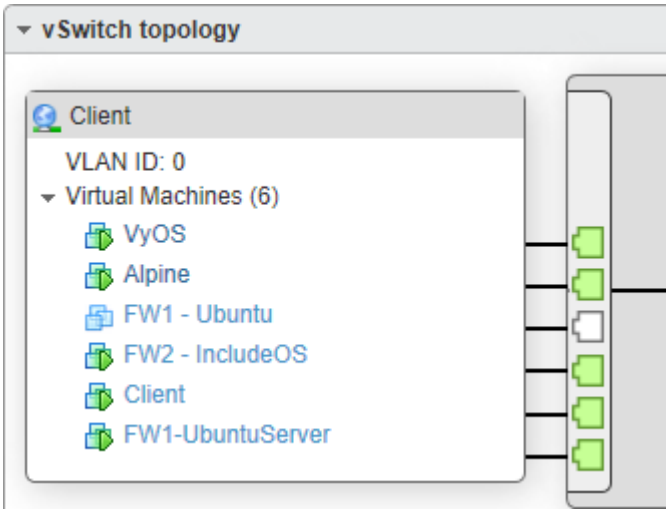

Figure 3.2: Port group "Client" connected to vSw1

For the initial setup, five VMs were created, all using vmxnet3 virtual NICs:

- One VM called **Client**

  Running Ubuntu Desktop 16.04

  Traffic generator

- One VM called **Target**

  Running Ubuntu Desktop 16.04

  Server-side, receiving traffic

- One VM called **Fw1**

  Running Ubuntu Server 16.04

  Acts as Firewall 1

- One VM called **Fw2**

  Running IncludeOS v0.12.0-rc.2

Table 3.1: Firewall-net

Host 1	Interface	IP address	Host 2	Interface	IP address	vSw
client	ens160	10.0.0.2 ->	FW1	ens192	10.0.0.1	1
FW1	ens256	10.0.1.1 ->	Target	ens160	10.0.1.2	2
client	ens160:1	10.3.0.2 ->	FW2	client	10.3.0.1	1
FW2	target	10.3.1.1 ->	Target	ens160:1	10.3.1.2	2
client	ens160:2	10.4.0.2 ->	FW3	eth1	10.4.1.1	1
Fw3	eth2	10.4.0.1 ->	Target	ens160:2	10.4.1.2	2

Includes network stack

Acts as Firewall 2

- One VM called **Mothership**

  Running Ubuntu Desktop 16.04

  Manages IncludeOS unikernel on Fw2

One additional VM was later created, called **Fw3**. This will serve as a way to test other OSes, initially Alpine Linux. Alpine Linux is "a security-oriented, lightweight Linux distribution based on musl libc and busybox"[34]. The idea was to test if this could run a router/firewall with better performance than Ubuntu Server, which by default includes a lot more functionality.

The client and the target both have one "physical" interface ens160 with two subinterfaces; ens160:1 and ens160:2. The interfaces can be set up with different IP addresses, providing an easy way to specify which router/firewall to send traffic through. The ens160-interface should go towards FW1-Ubuntu, subinterface ens160:1 towards Fw2-IncludeOS and ens160:2 towards Fw3-Alpine.

To prevent unnecessary overhead, Fw1 Ubuntu Server was cleaned of services with the following script:

```
1 #! /bin/bash
2
3 services="cron atd rsyslog acpid libvirt-bin libvirt-guests
4 apparmor ebtables friendly-recovery resolvconf ntp atop
5 pmlogger pmie pmcd pmproxy open-iscsi openipmi lxcfs bind9
6 accounts-daemon apache2 metricbeat redis-server collectl"
7
8 for service in $services; do
9     service $service stop
10 done
11
12  apt purge snapd ubuntu-core-launcher squashfs-tools
13  systemctl mask accounts-daemon.service
14  apt remove policykit-1 -y --purge
```

Listing 3.1: Clean-script

3.1.3 VM management

For easy management of the VMs, SSH access was set up and the VMs themselves reachable via three public IP addresses. The network interfaces on the Linux VMs was configured using the /etc/network/interfaces config file on each VM.

IncludeOS configuration can be done either with a GUI accessible via a browser or via command line on the Mothership. Port forwarding to the Mothership-VM with *ssh -L 8080:localhost:8080 128.39.120.12* was used for remote management with the browser GUI. From there, multiple IncludeOS instances can be configured and updated. The IncludeOS has also built an easy-to-use editor for their NaCl (network) configuration.

Updating the IncludeOS instance in the Fw2 VM with new network parameters involves writing or changing the configuration in the NaCl tab of the GUI, building a new image including the NaCl file and deploying the new image to the running VM. The update process, which includes IncludeOS' LiveUpdate feature, is described in section 2.1.5. The actual update of the VM, replacing an old image with a new one, takes only a fraction of a second. This feature is extremely useful – the alternative would be to edit the config, build an image, download the image locally, upload the image to the server and boot the image in ESXi – a process which would have taken many times as long to perform.

3.2 Tools

3.2.1 iPerf, TCP and UDP

iPerf is a network performance tool available for multiple operating systems and platforms, including Windows, MacOS, Android, iOS, and several Linux distros. Through the command line, iPerf can be run with various parameters to test and log many different aspects of a network connection, like timings, buffers and bandwidth, and it supports both TCP, UDP, SCTP with IPv4 and IPv6.

iPerf's TCP test mode will be used to measure maximum throughput of the different firewall setups. As most of the traffic on the Internet use the TCP transport protocol, these tests will be the most important for most people. UDP has traditionally been used for applications that require very little overhead and where some packet loss is accepted, like in audio and video streaming, but TCP has lately been the primary protocol for some of these tasks as well, including for YouTube. The iPerf UDP mode creates a constant bitrate UDP stream similar to voice communication, and measures packet loss and jitter in addition to throughput.[35]

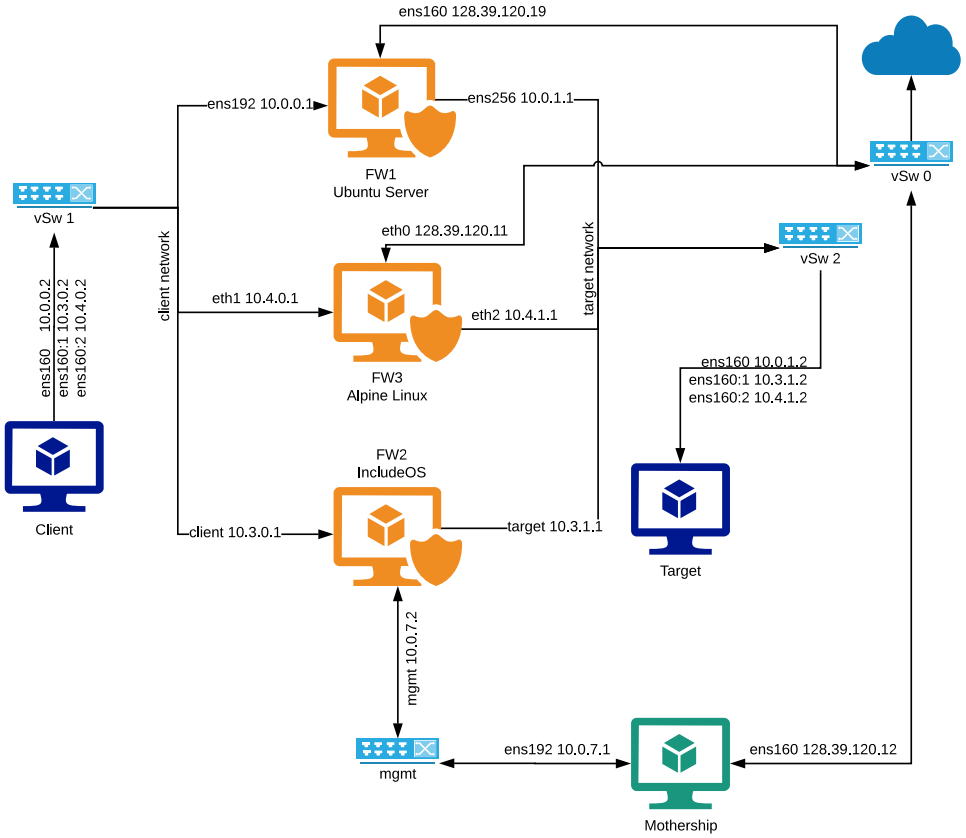

Figure 3.3: Network setup model

3.2.2 hping3

Hping3 is used to check that the firewall actually drops packets that match the blocked ports and IP addresses. We also use hping3 to measure latency in the TCP latency tests.

3.2.3 Netfilter

Netfilter is the main networking framework in Linux and performs all network related tasks in the Linux routers/firewalls. Userspace programs like iptables and nftables will be used to create firewall filters that block certain types of network traffic between the client and target VMs.

3.2.4 ipset

ipset is a netfilter module that allows for quick and efficient processing of large rule sets. Ipset may store IP addresses, networks, TCP and UDP port numbers, MAC addresses, interface names or combinations of these. Ipset uses either hashing or bitmaps to store and quickly look up information.

Bitmaps were used for both ports (bitmap:port) and IPs (bitmap:ip) in this project.

A bitmap set uses a memory range where each bit represents one IP, MAC or port, depending on the chosen set type. A hash set uses hashes to store IP addresses, CIDR netblocks, port numbers, interfaces or a combination of these.

3.2.5 IncludeOS Starbase image

Starbase is a minimal IncludeOS image with network stack and drivers for the VMXNET Generation 3 (vmxnet3) virtual network device from VMware. This image is going to be configured with NaCl to assign interfaces, network configuration including routing and firewall filters. The image size was only about 3 megabytes when compiled with interfaces and routing config.

3.3 Testing methodology

3.3.1 RFC 3511

RFC 3511 "Benchmarking Methodology for Firewall Performance" from 2003 defines tests for firewall performance testing. Important parts for this thesis are:

- Dual-homed vs tri-homed
- NAT vs no NAT
- Testing SHOULD be performed using different size rule sets to determine its impact on the performance of the DUT/SUT
- Rule sets MUST be configured in a manner which enables rules associated with actual test traffic to be configured at the end of the rule set and not at the beginning.
- The same TCP parameters MUST be used on all firewalls
- The duration of the test portion of each trial MUST be at least 30 seconds.

This will be a dual-homed test setup with NAT disabled. Different size rule sets with different types of rules will be compared to observe both how well the different technologies scale and to find out if some rule types are heavier to process than others, ex. TCP destination port vs. source IP address filtering. UDP and TCP performance will be tested.

3.3.2 Firewall verification

Hping3 is used to verify that the firewalls blocked the intended traffic. In addition, ipset, nftables and iptables were never used simultaneously, except for the one rule in iptables that refers to the ipset set, when running the ipset tests. Configuration from one filter were removed before configuring and testing a different filter.

3.3.3 iptables setup

The firewall filters will be applied to the FORWARD chain, since traffic passes through the firewall VM from one interface through another. The policy will be to accept packets that do not match any rule in the chain. Scripts are used for creating large rule sets quickly.

TCP destination port rules:

```
1 for z in {3000..3999}
2 do
3         iptables -A FORWARD -i ens192 -o ens256 -p tcp --dport $z
      -j DROP
4 done
```

Listing 3.2: iptables TCP dport script

Source address rules:

```
1 for x in {0..99}
2 do
3         for y in {1..10}; do
4                 iptables -A FORWARD -i ens192 -o ens256 -s 10.0.$x
      .$y -j DROP
5         done
6 done
```

Listing 3.3: iptables sAddr script

Iptables will have to try to match each incoming packet against every rule. Larger rule sets should therefore lead to more overhead and probably lower throughput.

3.3.4 ipset setup

The set types bitmap:port will be used for destination port matching and bitmap:ip for IP address matching.

Create set

```
1 ipset create ports bitmap:port range TCP:3000-7999
```

Listing 3.4: ipset: create set

Add ports

```
1 ipset add ports 3000-7999
```

Listing 3.5: ipset: add ports

After one or more sets are created in ipset, iptables is used to point to that rule set. Example:

Add iptables rule that points to the set

```
1 iptables -A FORWARD -i ens192 -o ens256 -m set --match-set ports
      dst -j DROP
```

Listing 3.6: iptables -> ipset

Structure

```
1 Name: ports
2 Type: bitmap:port
3 Revision: 3
4 Header: range 3000-7999
5 Size in memory: 732
6 References: 0
7 Members:
8 3000
9 3001
10 3002
11 ...
```

3.3.5 nftables setup

Nftables are the newest of the three Linux firewall types used in this thesis. As such, it took a little longer to find the best way of implementing nftables rulesets. The following commands creates sets that is compiled into bytecode by the nft command line tool. There are also other ways of implementing the same rules in nftables, but using this setup should provide fast lookup.

Create table:

```
1 nft add table ip filter
```

Listing 3.7: nftables: Create table

Create chain:

```
1 nft add chain ip filter forward { type filter hook forward
      priority 0 \; policy drop\; }
```

Listing 3.8: nftables: Create chain

Add rules:

```
1 sudo nft add rule ip filter forward ip protocol tcp tcp dport {
      3000-7999 } counter drop
```

Listing 3.9: nftables: Add multiport

Structure:

```
1 table ip filter {
2         chain forward {
3                 type filter hook forward priority 0; policy accept
      ;
4                 tcp dport { 3000-7999} counter packets 0 bytes 0
      drop # handle 4
5         }
6 }
```

Listing 3.10: nftables: Structure

NaCl setup

In IncludeOS' Mothership – which manages the IncludeOS instance – one has the opportunity to use the command line to configure, build and upload images. This would in theory make it easy to do a scripted approach along the lines of what was done in the Linux VMs. Since Mothership in this case resides in a Docker container, it required more work to automate the "add 100 rules, run 30 tests times 180 seconds". The GUI was therefore used to configure and update IncludeOS with new rulesets. Both the CLI and the GUI approach makes use of IncludeOS' LiveUpdate feature, which stores the running state in the VMs RAM, uploads the new image, switches over to the new image and restores the state. NaCl config examples are found in the appendix.

3.3.6 Sample size, testing length, scaling

To get a good sense of the throughput pattern of Fw1 Ubuntu Server with iptables and to see if there was any clear drops in throughput as the number of rules increased over a certain limit, a scaling test was scripted that runs a throughput test for every new iptables rule created.

```
1 testNo=1
2
3 for z in {1..100}; do
4  for x in {1..10}; do
5   echo "Test $testNo" >> $log
6   iptables -A FORWARD -i ens192 -o ens256 -p tcp -s 9.4.$z.$x -j
      DROP
7   ssh -i .ssh/master.key tobias@10.0.0.2 'iperf -c 10.0.1.2 -t 30'
       >> $log
8   echo -e
     '-------------------------------------------------------------\
     n' >> $log
9   (( testNo++ ))
10  sleep 2
11 done
12 done
```

Listing 3.11: iptables: Test script: One test per rule

To provide a more thorough comparison of Fw1 Ubuntu Server and Fw2 IncludeOS and provide cleaner graphs, we also need a test scheme that provides solid source data without taking too long to run. For the Central Limit Theorem to apply, we should have sample sizes of at least 30 for the distribution of the sample means to be fairly normally distributed. Because of time constraints, we used the minimum of 30 tests for each run, and each throughput test is run for 30 seconds, which is the minimum testing time specified in RF 3511.

The script below will run an iPerf test for every 100th iptables rule applied.

```
1 testNo=1
2 dport=3000
3
4 for x in {3000..3999..100}; do
5  for z in {1..100}; do
6   iptables -A FORWARD -i ens192 -o ens256 -p udp --dport $x -j
      DROP
7  done
8
9  for y in {1..30}; do
10  echo "Test $testNo" >> $log
11  ssh -i .ssh/master.key 10.0.0.2 "iperf -c -u -m 5000 10.3.1.2 -t
      30" >> $log
12  echo -e
      '------------------------------------------------------------\
      n' >> $log
13  (( testNo++ ))
14  sleep 5
15 done
16 done
```

Listing 3.12: iptables: Test script: Test every 100th rule

3.3.7 Misc network settings

Maximum number of simulatious connections in conntrack:

```
1 tobias@Fw1-Userver:~$ cat /proc/sys/net/nf_conntrack_max
2 262144
```

Listing 3.13: Conntrack: Maximum connections

UDP connection timeout:

```
1 tobias@Fw1-Userver:~$ cat /proc/sys/net/netfilter/
    nf_conntrack_udp_timeout
2 30
```

Listing 3.14: UDP connection timeout

Chapter 4

Results

All test results are presented and briefly discussed in this chapter. The tests consists of bulk throughput, a transactions-per-second test and latency tests. The throughput tests use iperf on both client and target and measure both TCP and UDP performance. The transactions-per-second tests are run with netperf and netserver and measures per-second request and reply performance. Latency tests are performed with the ping and hping3 utilities.

- 4.1: Early tests with iptables only, using the scale-and-test script presented in 3.3.3. The script runs an iPerf TCP throughput test for each new iptables rule.
- 4.2: Baseline tests. Pure throughput, no filter. iPerf TCP throughput.
- 4.3: IP address rules. Firewall rules that matches packets against source addresses/IPs. iPerf TCP throughput.
- 4.4: Port rules. Firewall rules that matches packets against TCP destination ports. iPerf TCP throughput.
- 4.5: Large rulesets. 10 000 and 50 000 blocked IPs. Source IP filters, iPerf TCP throughput.
- 4.6: Requests/transactions per second. Netperf used to measure requests and replies per second.
- 4.7: Latency tests. Ping and hping3 used to test ICMP and TCP latency.
- 4.8: UDP throughput. iPerf UDP throughput and CPU usage. 1 vs 4 vCPU on Fw1 Ubuntu Server.

The "Second run" sections contain tests that were done with an increased test time of 180 seconds per test compared to the previous 30 seconds per test. These tests were done to minimize variances and verify previous findings, and are discussed in more detail later.

4.1 Early iptables testing

These initial tests (figure 4.1 and 4.2) were run by a script that created iptables rules one by one and ran an iPerf test for each new rule for a total of 1000 tests. The resulting graphs were not as we expected as we were hoping for a much "cleaner" graph. Still we can see a clear trend in that the throughput decreases almost linearly with the number of rules, and we can also see that TCP dport filtering restricts throughput more than running IP address filtering only.

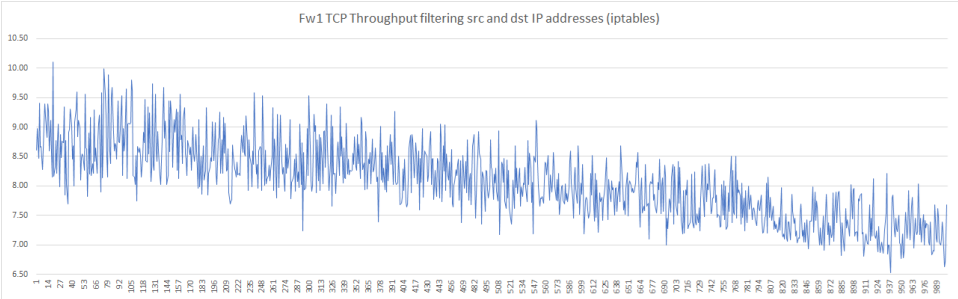

Figure 4.1: Fw1 iptables IP address filtering

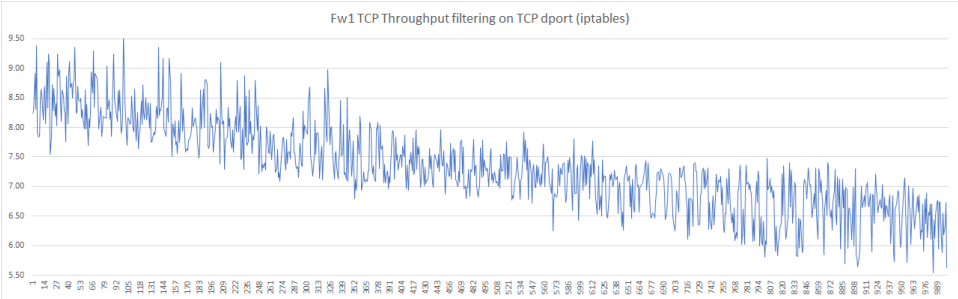

Figure 4.2: Fw1 iptables TCP destination port filtering

4.2 Baseline tests

The pre-planned testing schemes were used for all testing to ensure that they were performed in the exact same way and under the same conditions. That means using a scripted approach which performs 30 tests that each runs for 30 seconds. Some results are verified by running 180 seconds (under the "Second run") sections. Iperf was run with default settings, meaning a TCP window size of 85 kB at the client and 85.3 kB at the target.

4.2.1 TCP throughput

First run

Looking at the data from the baseline tests (figure 4.3), we see that Fw2 IncludeOS outperforms Fw1 Ubuntu Server and Fw3 Alpine Linux when not running any firewall filter. Fw2 IncludeOS manages 9.29 Gbps, Fw1 Ubuntu Server manages 8.41 Gbps and Fw3 Alpine Linux manages 8.52 Gbps. In these tests all the VMs are acting as nothing more than forwarding routers, with no filter applied.

Standard deviations are 0.388 for Fw1, 0.301 for Fw2 and 0.292 for Fw3.

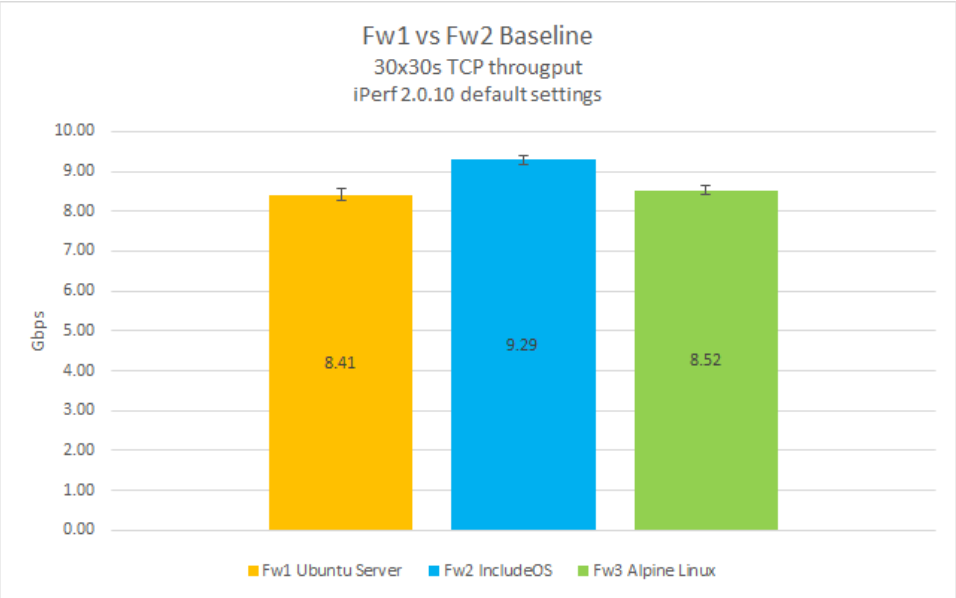

Figure 4.3: Baseline test results 1st run with 95 percent CI error bars

Second run

To verify previous findings, the previous "test groups" of (30 tests times 30 seconds) were run 10 times for a total of 10 groups times (30 tests times 30 seconds). E.g. Test group 1: (30 tests x 30s), test group 2: (30 tests x 30s), ..., test group 10: (30 tests x 30s).

For the second round of tests (figure 4.4), we observe that the IncludeOS VM still outperforms the Linux VM – though with a bit lower margin, managing 9.67 Gbps versus Ubuntu's 9.12 Gbps. It was not found exactly why the results were higher in the second run, though possible reasons are discussed later.

Standard deviations are 0.532 for Fw1 and 0.658 for Fw2, which means that both throughput results are inside each others standard deviations.

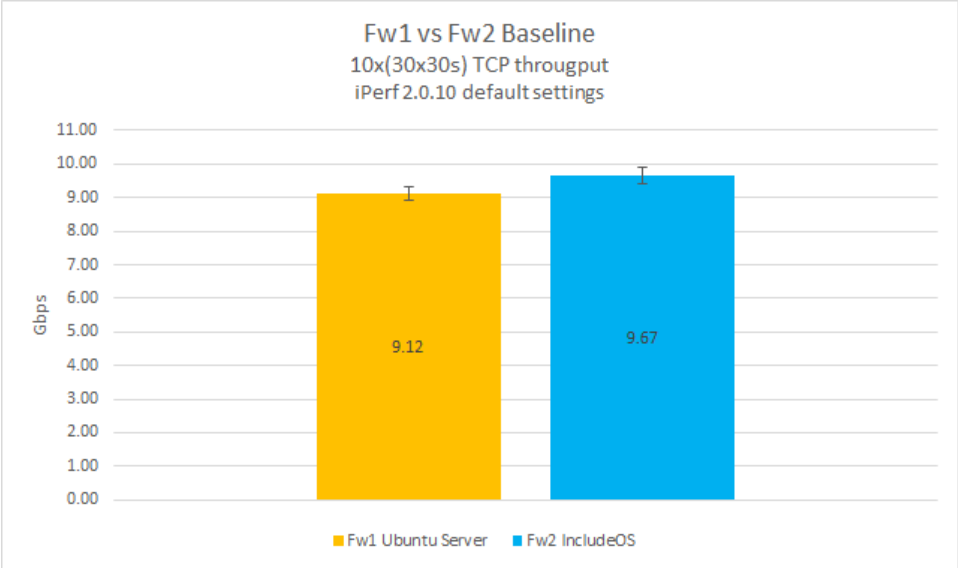

Figure 4.4: Baseline test results 2nd run with 95 percent CI error bars

4.3 Address rules

A common firewall scenario is to have a blacklist of IP-addresses and block traffic from those addresses. OsloMet for instance has such a blacklist of IP-addresses that is built from various traffic patterns and known threats. Ranges of source IPs are therefore blocked in these tests.

4.3.1 First run

Figure 4.5 shows the results from doing source IP address filtering exclusively. Looking at the line that represents Ubuntu Server, we can see a close to linear decrease in throughput as the number of rules increases. With 5000 source address rules, throughput is down almost 50 percent, from 8.41 Gbps with no rules to 4.24 Gbps.

IncludeOS on the other hand shows practically no decrease in throughput, pushing the same 9+ Gbps all the way to 5000 rules.

4.3.2 Second run

The second batch of tests (figure 4.6) resembles the first one, with Fw2 IncludeOS managing well over 9 Gbps throughout the test, while Fw1 Ubuntu Server with iptables see a close to linear drop in performance as the number of rules increase. The difference in this run is that Fw1 Ubuntu Server's throughput does not drop as much after 3000 rules, and ends on around 5 Gbps, while in the previous run, it ended at 4.3 Gbps with 5000 iptables rules in place.

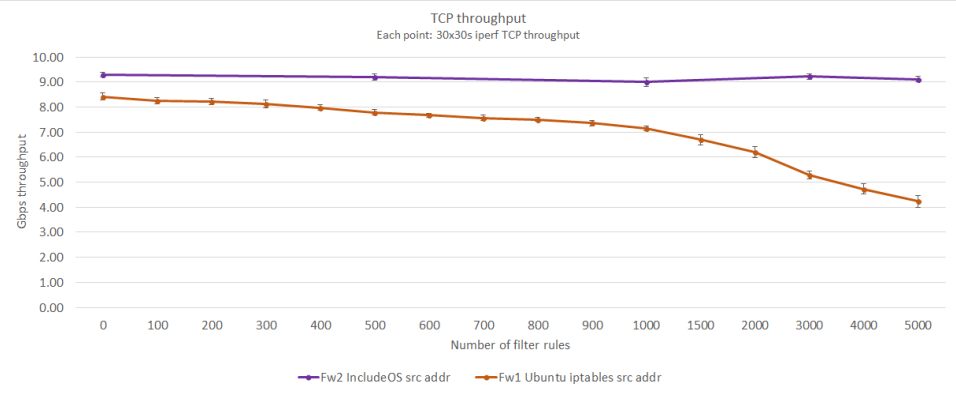

Figure 4.5: Source address filtering

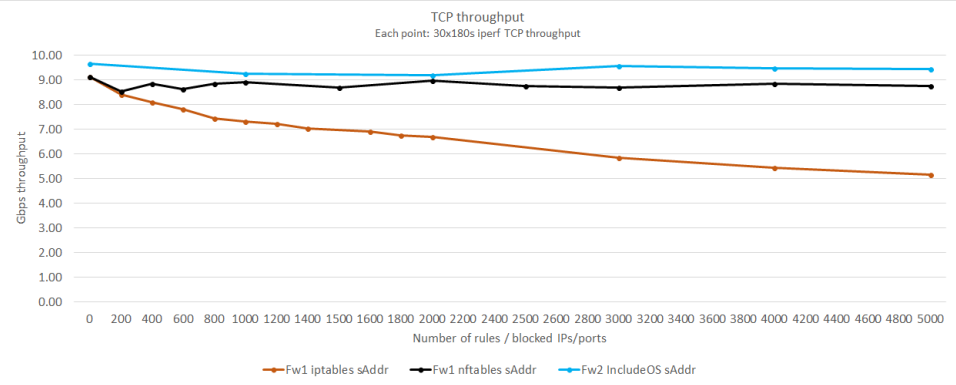

Figure 4.6: Source address filtering - run 2

This time, nftables was also tested with up to 5000 blocked source IP addresses. Fw1 shows no decrease in throughput as the number of blocked IPs increases, and manages almost 9 Gbps at 5000 blocked IPs with nftables.

4.4 Port rules

Going one layer further up the network stack and creating rules that check for TCP destination port matches may impose a bigger overhead than rules that only look at IP-addresses.

4.4.1 TCP throughput w/ destination port only rules

First run

Looking at the graph in figure 4.7, we can see that the assumption that processing layer 4 rules takes a bigger toll than processing layer 3 rules is indeed true – at least for the Ubuntu VMs running iptables. Where Ubuntu

Server managed a throughput of 4.24 Gbps with 5000 IP source address rules, it only manages a throughput of 2.78 Gbps when switching out -s <IP> with -p tcp –dport <port>.

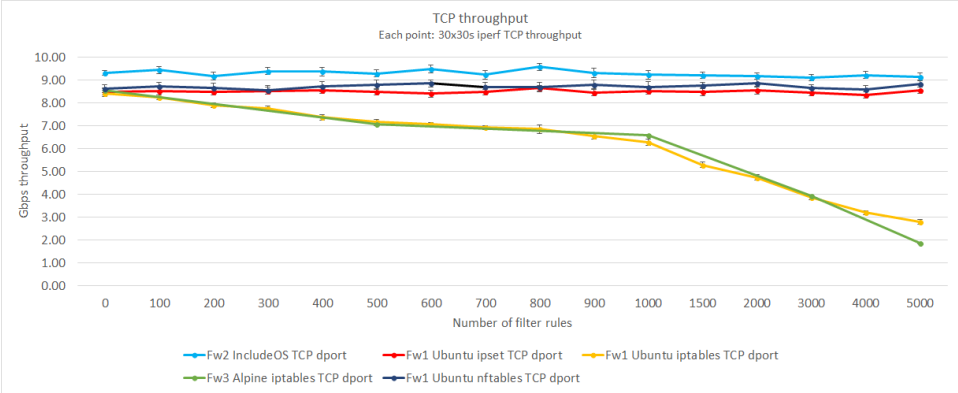

Figure 4.7: TCP dport filtering

While iptables on the Linux VMs takes a big hit when the rule sets starts to grow, IncludeOS and ipset shows absolutely no drop in throughput – even when running 5000 rules. Ipset is configured with the bitmap:port set type for the TCP dport tests.

Looking at Fw1 Ubuntu Server only (figure 4.8), we can clearly see the difference between the iptables, ipset and nftables, though ipset and nftables are quite closely matched.

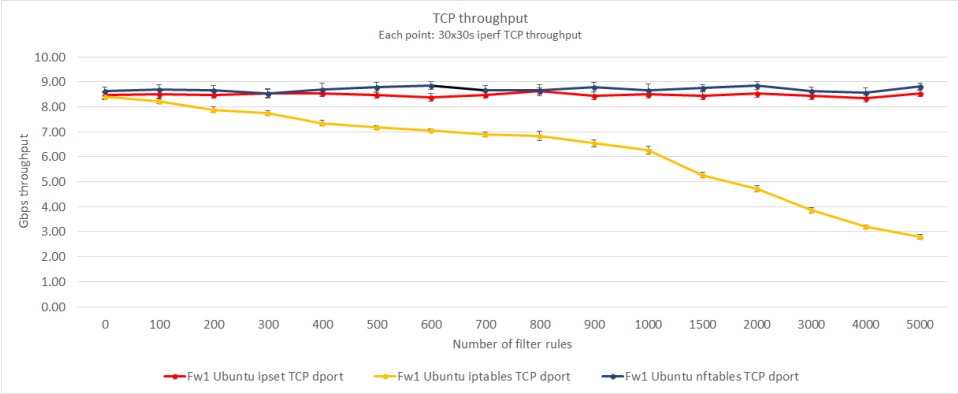

Figure 4.8: Fw1 TCP dport filtering

Second run

The second batch of tests (figure 4.9) confirm the findings in the first run. Except from some variations up to 1400 rules, we can see a quite linear decrease in the throughput as the number of iptables rules increase. At 5000 rules, the throughput is measured at approximately 3 Gbps.

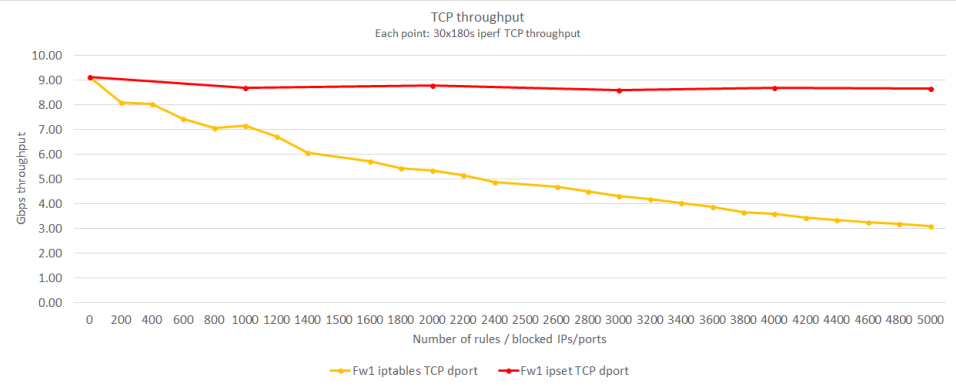

Figure 4.9: Fw1 TCP dport filtering - run 2

4.5 Large rulesets

Because of all the malicious traffic coming from and targeting specific IP addresses, there is often a need for firewalls that block thousands of different IPs. These tests show how the different firewalls perform with 10 000 and 50 000 rules respectively.

4.5.1 10k blocked IPs

Running filters that block 10 000 source IP addresses, we observe (figure 4.10) that Fw2 still outperforms the other firewall setups at 9 Gbps throughput. Nftables and ipset follows with 8.7 and 8.5 Gbps respectively. Iptables is the slowest, with 3.5 Gbps throughput at 10k rules.

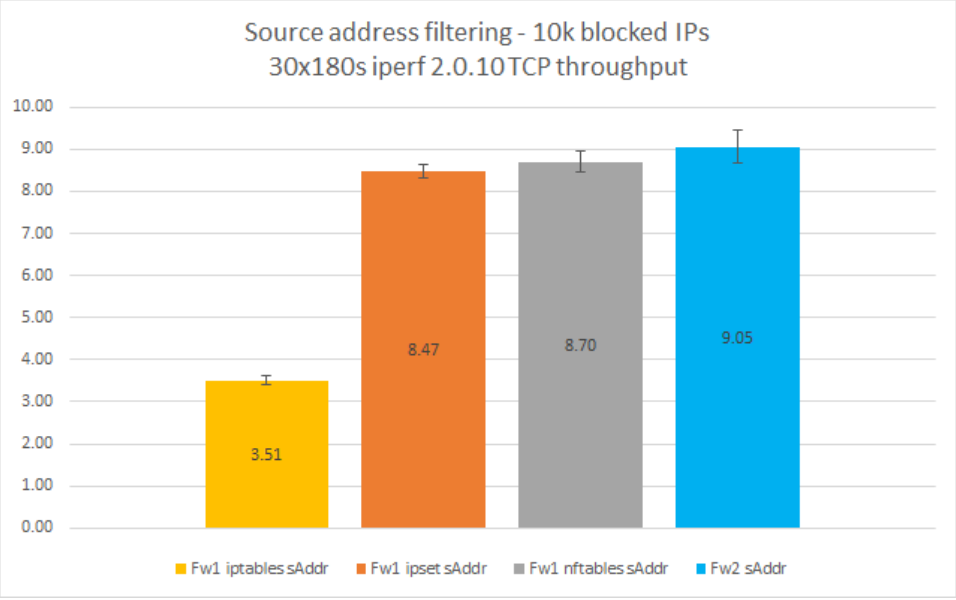

Figure 4.10: 10k blocked IPs comparison

The graph includes error bars that represent the 95 percent confidence intervals. The CI is quite a lot bigger for Fw2 IncludeOS than the others, which is probably caused by the network throughput in ESXi being maxed out – introducing larger variations/noise in throughput than for the other firewalls.

4.5.2 50k blocked IPs

Figure 4.11 shows throughput when running filters that block 50 000 IP addresses. Fw2 IncludeOS manages 9.25 Gbps and outperforms Fw1 with nftables at 8.40 Gbps.

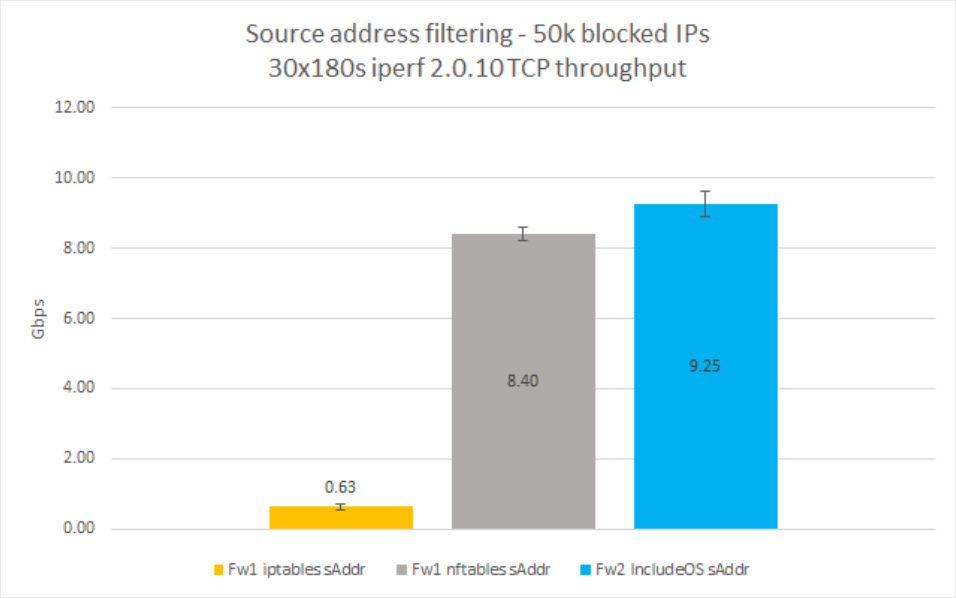

Figure 4.11: 50k blocked IPs comparison

4.6 Requests per second

Netperf was used to test the number of TCP transactions per second each of the firewalls were able to handle. A transaction is defined as a single request and a single reply.

Running Netperf -H <server IP> -t TCP_RR gave the results seen in figure 4.12. The results are averages of 30 tests.

From the chart, we see that when not running any filters both Fw1 and Fw2 manage about the same transaction rate – 15400 and 15470 transactions/s respectively.

Blocking 5000 source IP addresses Fw2 IncludeOS does not seem to restrict performance, averaging 15750 transactions/s. Ipset and nftables are quite

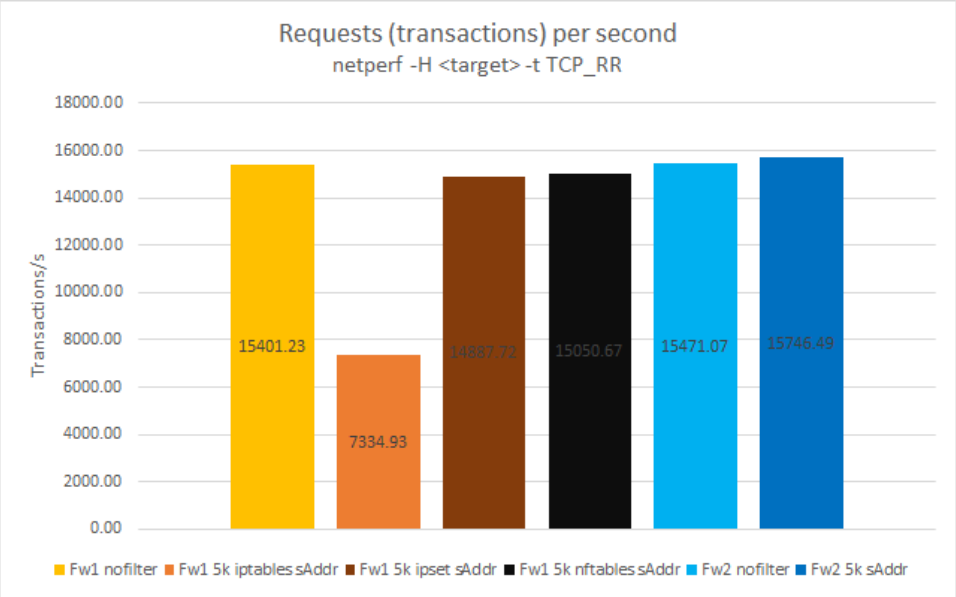

Figure 4.12: Transactions/s

closely matched, managing 14890 and 15050 transactions/s. Fw1 Ubuntu Server with 5000 iptables rules sees the transaction rate more than halved (7335 transactions/s).

4.7 Latency tests

Simple latency tests run with Linux' built-in ping tool show that iptables not only restricts throughput when large rule sets are applied, but also latency increases. In these tests, ping was run with the following command:

```
1 sudo ping -i 0.1 -c 1000 <IP address>
```

Listing 4.1: latency: Ping command

which sends 1000 pings with 0.1s intervals. The average of the 1000 pings is then calculated, and is what you see in the graph (figure 4.13).

In this test, Fw2 IncludeOS starts out with a tiny bit higher latency than Fw1 Ubuntu Server, which may just be due to internal noise and variations. When applying 1000 and 5000 rules respectively, it becomes obvious that iptables is (by far) the biggest loser, while the others are quite evenly matched.

Marked on each bar is the standard deviation, calculated automatically by the ping tool.

Not sure if the TCP port sets created in ipset and nftables actually comes into play when sending ICMP traffic to the firewall, we decided to run the tests again with hping3, sending TCP traffic instead.

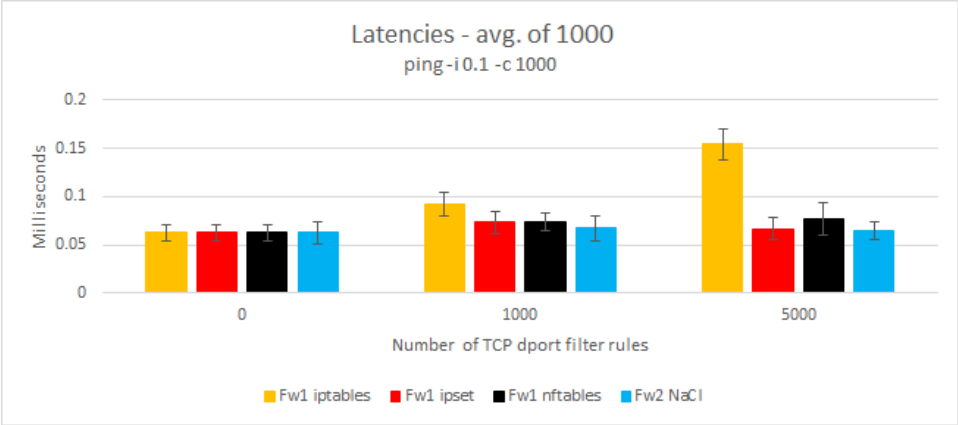

Figure 4.13: Latencies

Running the second batch of test – with TCP traffic – on Fw2 IncludeOS did cause some headaches at first. When running

```
1 sudo hping3 -c 10000 -i u10000 10.3.1.2 -p 8000
```

Listing 4.2: latency: hping3 TCP

the client VM reported exactly 50 percent packet loss every time we tried to send 10k packets. After some discussion with the IncludeOS team, trying to figure out the cause of the packet loss (even blaming conntrack for a while), it became apparent that the IncludeOS firewall actually blocked packets in **both** directions. Hping3 sends all the packets to the specified port, which in this case is TCP 8000, but the target host uses different ports for each reply (TCP RST). What actually happened in this case was that the return traffic from the target host was blocked when it entered the specified port range (set to drop), which was of course meant to only block traffic in one direction, namely from the "client" interface towards the "target" interface.

In short, the filter blocked the specified port range in both directions on Fw2, while on Fw1 Ubuntu Server, the iptables rules specify an input and an output interface, i.e. the traffic direction.

To solve this, the filter was applied to the prerouting chain on the incoming interface instead of the "gateway" block in the NaCl code (ref. appendix).

The exact same problem was encountered when testing with nftables. Setting up a port range in the forward chain, traffic in both directions was blocked. This is what the setup looked like:

```
1 table ip filter {
2     chain forward {
3         type filter hook forward priority 0; policy accept;
4         tcp dport { 3000-7999} drop
5     }
6 }
```

Listing 4.3: nftables: Forward filter

After some research, the following solution was found:

```
1 nft add rule ip filter forward oif ens192 ip protocol tcp tcp
    dport { 3000-7999 } drop
```

Listing 4.4: nftables: If port

specifying which interface the filter applies to.

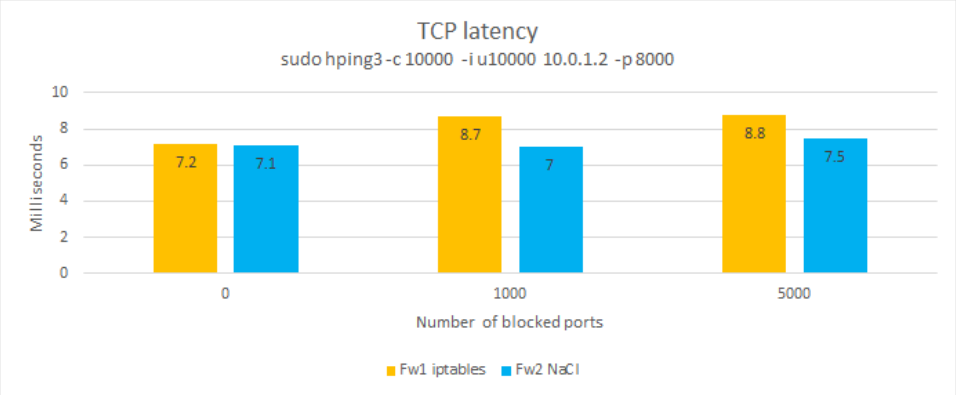

Figure 4.14: TCP Latencies

The results from sending 10 000 TCP pings with hping3 can be seen in figure 4.14. It certainly looks like iptables introduces a latency penalty, but the ping results varies quite a lot on the ESXi network, and they probably should not be trusted too much in this case.

4.8 UDP throughput

Even though TCP is the most used transport protocol, UDP is still very much in use. These tests should show how the different firewalls handles large volumes of UDP traffic, and what happens when reducing the number of vCPUs on Fw1 Ubuntu Server from four to one.

4.8.1 UDP throughput tests (iPerf 2.0.5)

UDP throughput tests with iPerf 2.0.5 show that we for Fw1 Ubuntu server can achieve a throughput of around 730 Mbps without packet loss when there are no iptables rules set. Running the exact same tests after creating 1000 UDP rules with iptables in Fw1, we observe a packet loss of between 9 and 10 percent.

4.8.2 UDP throughput tests (iPerf 2.0.10)

After researching a bit, trying to find out why the UDP throughput was so much lower than the TCP throughput, we found on source stating

that the UDP performance in iPerf 2.0.5 is low "due to mutex contention between the client thread and the reporter thread. The shared memory between these two threads was increased to address the issue [in version 2.0.10]."

After upgrading to 2.0.10, these were the results on Fw1, when trying to push 5 Gbps (figure 4.15):

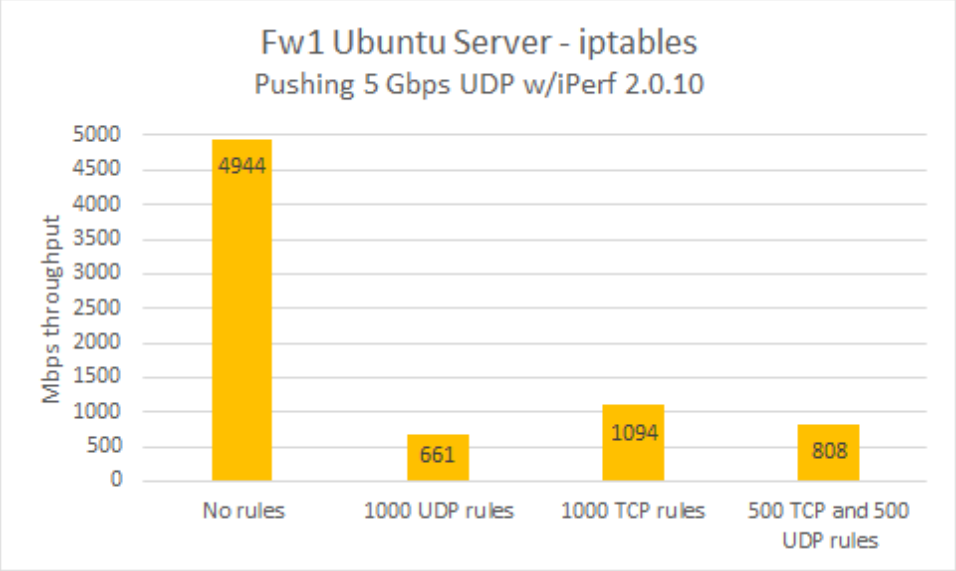

Figure 4.15: Fw1 UDP throughput tests

The first block of tests is run through Fw1 without any iptables filter. We can see that the throughput is close to 5 Gbps with little packet loss.

The second block of tests is run with 1000 UDP port rules on Fw1. The throughput in these tests are reduced to around 660 Mbps.

The third block of tests is run with 1000 TCP port rules Fw1. Here we can see that the throughput is a little bit higher than with the UDP rules, averaging close to 1100 Mbps.

The fourth block of tests is run with 1000 rules, where the first 500 are TCP rules and the second 500 are UDP rules. In regards to speed and packet loss, these average lies between the pure UDP and TCP filter tests, with a throughput of about 800 Mbps.

Looking at the packet loss chart (figure 4.16), we see a clear relationship between throughput and packet loss. Fw1 with iptables struggles to cope with the amount of UDP traffic, and the result is that most of the packets are dropped whith only 1000 iptables rules. While dropping around 86 percent of the packets with only UDP dport rules, the packet loss decreases to around 78 percent when running TCP dport rules instead.

ESXi reported a CPU usage on Fw1 of about 30 percent for each of the tests, regardless of rules being applied or not and also what kind of rules were

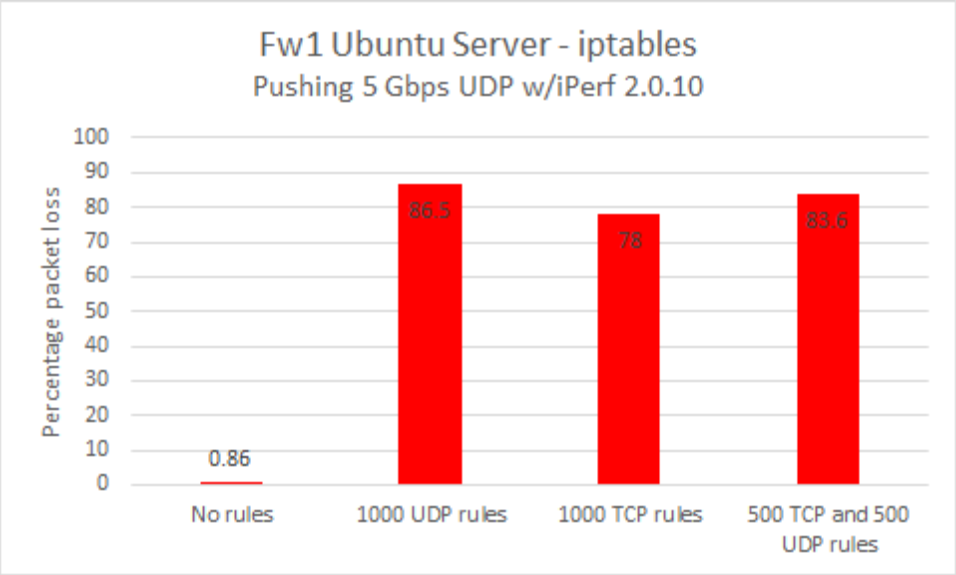

Figure 4.16: Fw1 UDP throughput tests - packet loss

applied (figure 4.17).

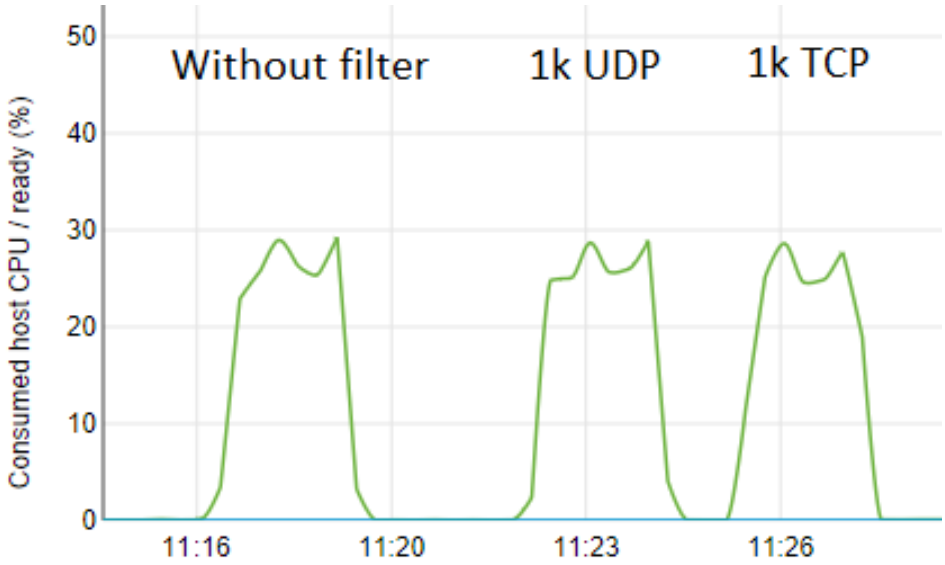

Figure 4.17: Fw1 UDP throughput tests CPU comparison 1

Also interesting is to see the difference in reported CPU usage when cutting the number of vCPUs from 4 to 1. Looking at the CPU chart (figure 4.18), it is very clear that Ubuntu Server is very CPU-bound when it comes to handling network traffic – at least UDP traffic – and can utilize at least four CPU cores.

Observing that the CPU hits 100 percent for the UDP 5 Gbps test when limiting Ubuntu Server to one vCPU, another round of tests was run to see if this had any impact on the throughput.

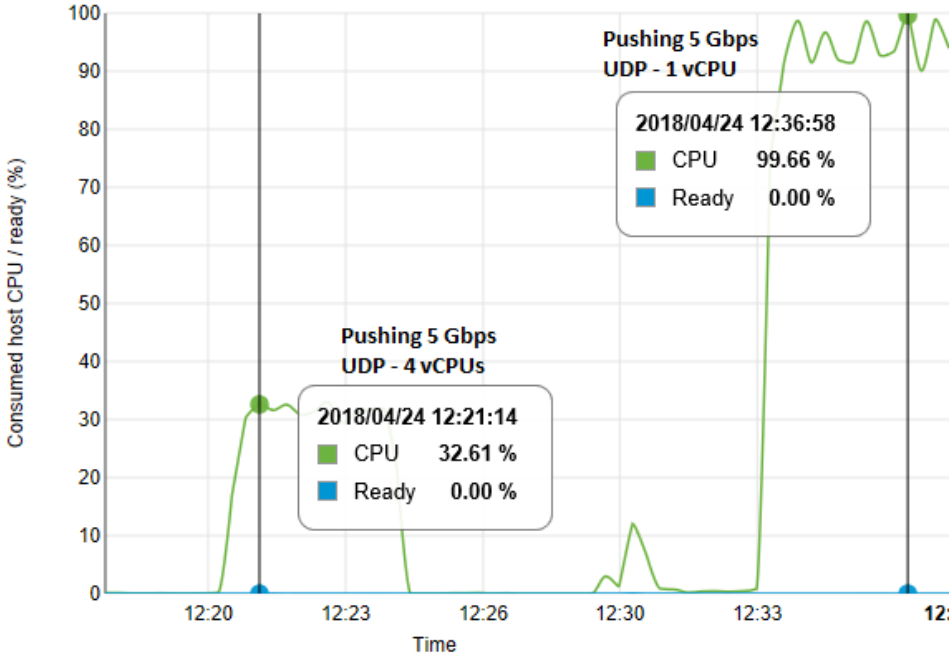

Figure 4.18: Fw1 UDP CPU usage 4 vs 1 vCPU

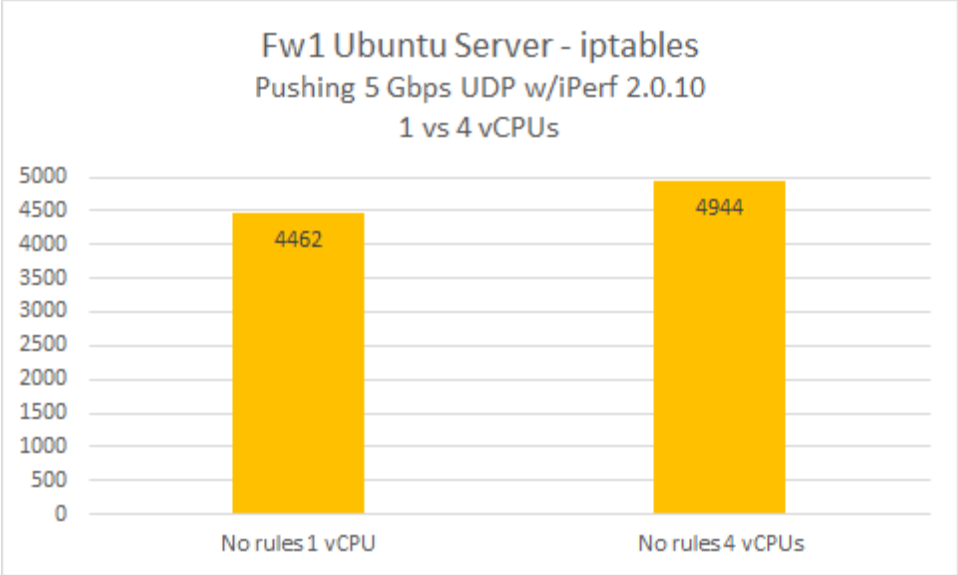

Figure 4.19: Fw1 UDP throughput test 4 vs 1 vCPU

As we can see from figure 4.19, the throughput decreased with about 10 percent to ~4.4 Gbps (from ~4.9 Gbps) when limiting the number of vCPUs to just one. The reason for this is the packet loss that occurred when Ubuntu ran out of CPU resources. The packet loss amounted to about 10 percent.

Based on these results, our theory is that the packet loss would increase

when pushing even more data through Fw1. With 5 Gbps and 4 vCPUs, the reported total CPU usage was about 33 percent. Limiting the vCPUs to just one, we saw an increase in CPU usage of around four times. We can then assume that Fw1 Ubuntu Server (if not running out of RAM) could handle quite a lot more throughput with the original four vCPUs without experiencing heavy packet loss. Limited to one vCPU, max UDP throughput with minimal packet loss lies somewhere under 5 Gbps. Pushing 5 Gbps UDP traffic through Fw2 gave some interesting results.

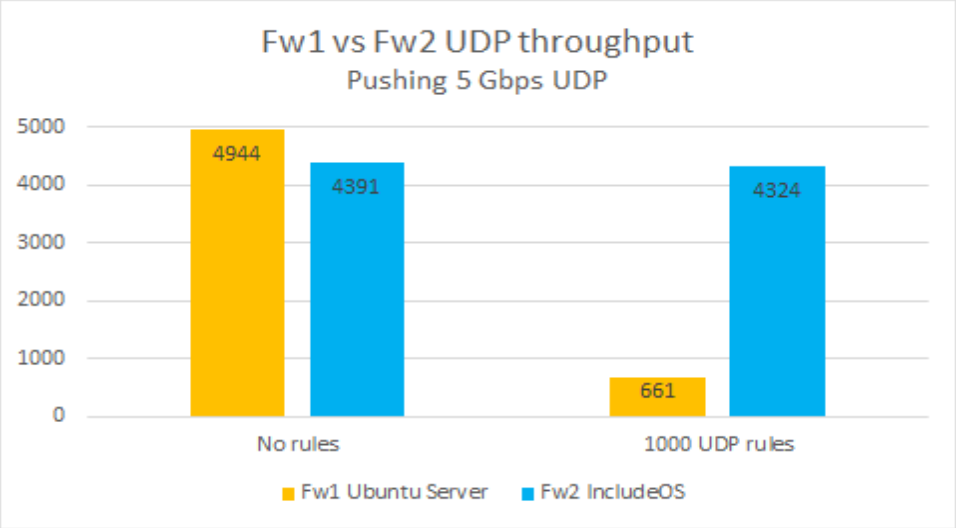

Figure 4.20: Fw1 vs Fw2 UDP throughput

Comparing Fw1 and Fw2 (figure 4.20), we observe that Fw1 Ubuntu Server actually experienced less packet loss than Fw2 IncludeOS in the first set of tests – that is without any filter applied. As soon as we start adding iptables rules to Fw1 and blocking the same UDP port range in NaCl, the results show quite the opposite; Fw2 manages about the same throughput while Fw1's throughput decreases to just 13 percent of what it managed without a filter.

We can also see a clear difference in Fw2 IncludeOS' CPU usage when pushing different amounts of UDP traffic. Figure 4.21 shows the ESXi's reported CPU usage (40-55 percent) when testing 1000+ TCP dport rules on Fw1 with iPerf TCP throughput.

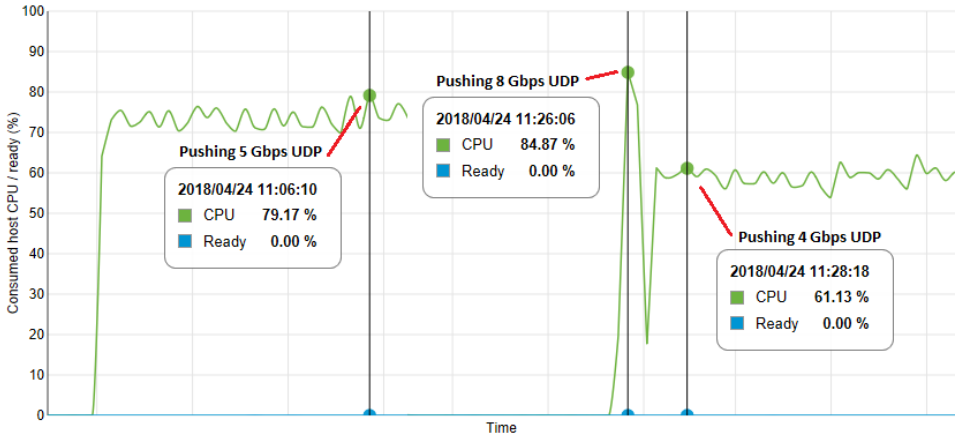

Figure 4.21: Fw2 UDP throughput tests CPU comparison 2

Chapter 5

Discussion

5.1 Testing methodology

There were more considerations to take into account than expected when starting the project. Varying results in throughput and latency resulted the decision of redoing some of the tests with increased run time (180s) for each test (still 30 tests per data point) to provide better source data. A number of factors come into play when performing these kinds of tests, even when performing the tests inside a closed setup like the one used in this project. The exact cause(s) of the inconsistencies were not pinpointed since the large number of components makes it hard to troubleshoot. Causes for the inconsistencies may include hardware, hypervisor (ESXi), virtual NICs, resource distribution, virtual switches, protocols, TCP congestion control, etc. Traffic shaping is disabled in both the vSwitches and the port groups/links, so that should not limit the throughput.

5.1.1 ESXi network planning

VMware's ESXi hypervisor is optimized for high-speed and cost-effective networking between VMs[36] and should thus provide a solid platform for the kind of testing done in this project.

5.1.2 Throughput

Testing primarily with iPerf should give a good idea of how the different firewalls perform under pressure, since the firewalls have to match every incoming packet with its specified rule set. VMware also use iPerf in an example for troubleshooting network latency in ESXi[37], which may imply that it is the right tool for the kind of testing done in this thesis. The VMware Network Throughput in a Virtual Infrastructure white paper states the importance of having sufficient CPU power for heavy network workloads. Maxing CPU usage may result in TCP treating the condition

as network congestion, applying it's flow/congestion control mechanisms to counter the issue. This should not be a problem here though, where the main goal is to show differences in throughput between the different firewalls and technologies – not to show exactly how much throughput can be achieved with this exact system. Anyway, ESXi reported well under 100 percent CPU usage for most tests except for one UDP test run on Ubuntu with one vCPU – which somewhat reduced the throughput, and for IncludeOS while blocking 20 000 IP addresses.

5.2 Linux firewalls – Current situation

A front-end to the kernel-level netfilter hooks, *iptables* has been widely adopted since its implementation in the 2.4.0 Linux kernel in 2001, and is now the standard firewall in many or most Linux distros.

Assumed by many to be the successor to iptables, *nftables* was implemented in the 3.13 kernel released early in 2014.

A third packet filter, *BPF*, was recently added by kernel developers, overlapping some of the functionality in nftables.[38]

5.2.1 Iptables, drawbacks

The most serious drawback of iptables is the performance hit that comes from its sequential processing of rules – easily seen from the graphs above. Every incoming and outgoing packet has to be matched against all iptables-rules, one by one, to determine what action to be taken on that specific packet. This may not be an issue for individuals building a small firewall at home, but for power users and businesses with higher security and performance requirements, iptables is certainly not the best solution.

In addition to the performance drawbacks, there are also security issues linked to the use of iptables.

5.2.2 Short term solution: ipset

Having the ability to create sets and store them in hash tables for fast lookup, ipset is golden for iptables users. Huge performance gains can result from cutting down on iptables rules and instead having just a few rules pointing to IP sets.

Since the ipset tool has been around for a while, it is also well documented, and it is not hard to find help in online guides and forums if needed.

5.2.3 nftables – Successor to iptables?

At the time of writing, the Netfilter project are working on getting nftables more widely adopted, trying to move people and companies over from iptables. A wiki page on Netfilter's nftables wiki contains a list over current adopters of nftables, but that list is currently quite short.[39]

Because of nftables not being that widely used, it is also much harder to find documentation and setup guides – in stark contrast to iptables, which is very well documented, widely used and which many sysadmins know well through many years.

Getting people to migrate from iptables to nftables are a priority for the Netfilter project, and while they are trying to make the change as easy as possible, many will probably still stick with iptables for years to come, at least when getting reasonable performance with iptables extensions like ipset.

Also with regards to security, nftables should in theory be a better choice than iptables. People are hard to change though, tending to stick with what works instead of pouring resources into something new and unfamiliar (if it ain't broke, don't fix it).

5.2.4 The rise of eBPF

Berkeley Packet Filter (BPF) started out as a way to optimize packet filtering by compiling expressions into optimal BPF bytecode, which is then executed in a sandboxed in-kernel VM [40]. Tcpdump is one tool that utilizes BPF, allowing it to for example return only packets specified by a filter, avoiding unnecessary copying of packets from the kernel to the process.

Several large service- and content providers are adopting (e)BPF [41] and XDP[42], among them is: Facebook (load balancing and DDoS mitigation)[43] and Cloudflare (DDoS mitigation)[44].

Advantages of eBPF:

- Just-in-time compilation
- Offloading of XDP-level programs to NIC, moving firewall processing off the CPU
- Writing firewall rules in C
- Subject to BPF verifier, providing an extra layer of security
- Existing iptables rules can be translated into BPF programs – moving entire configurations to BPF may be possible without sysadmins even knowing

Suggestions: Even though it is possible to implement the iptables API with bpfilter, some people in the community suggests that bpfilter should rather be implemented with the nftables API – encouraging and supporting the shift from the old iptables with its "design mistakes" over to the much newer nftables, which was built with iptables shortcomings in mind. [38]

Projects are underway for frameworks that works on top of bpfilter that translate nftables into BPF[45]

In a talk in FOSDEM 2018, the software engineer Quentin Monnet of Netronome talks about the advantages of eBPF and XDP [46]. The main idea is to have BPF programs talk with XDP in the kernel. XDP can intercept packets and make decisions before the packets reach the network stack, enabling faster packet processing for simple use cases. These use cases may include load balancing, DDoS protection/mitigation, firewalls, virtual switches, QoS and more.

An overview on XDP from 2016 by Alexei Starovoitov and Tom Herbert, both Facebook employees, describes XDP as "A programmable, high performance, specialized application, packet processor in the Linux networking data path" [47]. Like Monnet, they also list DOS mitigation and load balancing as use cases for the XDP packet processor, in addition to forwarding, flow sampling, monitoring and ULP processing. For packet drop, they list a target of 20 million packets per second per CPU as a performance goal, which will be important for DDoS mitigation.

DDoS

DDoS attacks have been more frequent and much bigger in "size" over the past few years than ever before. To be able to investigate and understand how these attacks work, the big service providers of the Internet has to be able to "absorb" the attacks, not simply "black hole" them, as Cloudflare writes in a blog post from 2016 [48]. In the first quarter of that year, Cloudflare saw an increase of 15x in individual DoS events (large attacks are registered as many separate events). Instead of using BGP blackholing – a technique that involves routing the traffic to an IP address which has no host attached to it – the scale and efficiency of Cloudflare's network allow them to monitor and analyze the traffic. Hitting peaks of 180 million packets per second, Cloudflare still managed to sustain and log the attack. With XDP and eBPF, (more) providers should get an even better chance to mitigate these kinds of attacks. Unikernels may also be able to provide low cost DDoS protection on the same level as more expensive systems, at least when handling large traffic volumes as good as IncludeOS did in this project.

One reason for the increase in DDoS frequency and size is the enormous increase in (badly secured) IoT devices. Improving security in IoT devices is therefore a hot topic these days, which Google's new IoT

OS[49] is a testimony of. Having limited system resources, IoT devices often can't run traditional operating systems, instead often relying on proprietary firmware/software. Unikernels should have an advantage in this regard, as some of their main traits are their minimal size, good resource utilization, optimized/compiled code and immutability.

It is apparent that XDP is getting a lot of attention these days. A list of XDP work and presentation can be found at https://www.iovisor.org/technology/xdp.

5.3 Networking in IncludeOS

5.3.1 TCP/IP stack

Contrary to Linux, which has one network stack shared by all interfaces, IncludeOS has one network stack for each interface, allowing tighter control and enhancing security.

5.3.2 In development

IncludeOS is still a new product, and new versions are being released quite quickly. The performance tested and shown in this thesis should give a good impression of the performance characteristics of the unikernel, but the results may vary in new versions.

5.4 Interrupts and throughput variations

After some experimenting, tweaking affinity and watching interrupts, we found that:
1. The NIC has four interrupt queues
2. Interrupts when testing throughput with iPerf seems to end up in two random interrupt queues. When the interrupts end up in two *different* queues on the client/sender the throughput (when measuring on a second to second basis) jumps up and down quite a lot:

```
1 $ taskset -c 3 iperf -c 10.0.1.2 -t 180 -i 1
2 ------------------------------------------------------------
3 Client connecting to 10.0.1.2, TCP port 5001
4 TCP window size: 85.0 KByte (default)
5 ------------------------------------------------------------
6 [  3] local 10.0.0.2 port 47042 connected with 10.0.1.2 port 5001
7 [ ID] Interval       Transfer     Bandwidth
8 [  3]  0.0- 1.0 sec  1.29 GBytes  11.0 Gbits/sec
9 [  3]  1.0- 2.0 sec   877 MBytes  7.36 Gbits/sec
10 [  3]  2.0- 3.0 sec  1.22 GBytes  10.5 Gbits/sec
11 [  3]  3.0- 4.0 sec   834 MBytes  7.00 Gbits/sec
12 [  3]  4.0- 5.0 sec  1.19 GBytes  10.2 Gbits/sec
```

```
13 [  3]  5.0- 6.0 sec   857 MBytes  7.19 Gbits/sec
14 [  3]  6.0- 7.0 sec  1.33 GBytes  11.4 Gbits/sec
15 [  3]  7.0- 8.0 sec   850 MBytes  7.13 Gbits/sec
```

Listing 5.1: Different interrupt queues

3. When the interrupts end up in the same queue, the traffic pattern is much more stable:

```
1 $ taskset -c 3 iperf -c 10.0.1.2 -t 180 -i 1
2 ------------------------------------------------------------
3 Client connecting to 10.0.1.2, TCP port 5001
4 TCP window size: 85.0 KByte (default)
5 ------------------------------------------------------------
6 [  3] local 10.0.0.2 port 47040 connected with 10.0.1.2 port 5001
7 [ ID] Interval       Transfer     Bandwidth
8 [  3]  0.0- 1.0 sec  1.18 GBytes  10.1 Gbits/sec
9 [  3]  1.0- 2.0 sec  1003 MBytes  8.41 Gbits/sec
10 [  3]  2.0- 3.0 sec  1.03 GBytes  8.82 Gbits/sec
11 [  3]  3.0- 4.0 sec  1020 MBytes  8.56 Gbits/sec
12 [  3]  4.0- 5.0 sec  1.02 GBytes  8.75 Gbits/sec
13 [  3]  5.0- 6.0 sec  1.00 GBytes  8.60 Gbits/sec
14 [  3]  6.0- 7.0 sec  1.04 GBytes  8.91 Gbits/sec
15 [  3]  7.0- 8.0 sec  1022 MBytes  8.57 Gbits/sec
```

Listing 5.2: One interrupt queue

This is what the interrupt queues look like for ens160 on the Client VM:

```
1 grep ens160 /proc/interrupts
2  56: 24  349993252  541823462  296032049  PCI-MSI 1572864-edge
       ens160-rxtx-0
3  57: 12  129686885 1026923095  36998141  PCI-MSI 1572865-edge
       ens160-rxtx-1
4  58: 6  67101899 1083994287  3642212  PCI-MSI 1572866-edge
       ens160-rxtx-2
5  59: 1  89156482  253644912  870415523  PCI-MSI 1572867-edge
       ens160-rxtx-3
```

Listing 5.3: Interrupt queues for NIC ens160

It should be possible to force the interrupts onto one queue with ethtool, but this may be NIC or driver dependant, and was seemingly not possible in this instance. We did try to set the irq affinities of the four interrupt queues to one CPU and pass traffic with another CPU too look if that did anything to the traffic pattern That did not help though, since the interrupts could still ended up in different queues on the same CPU.

Setting interrupt CPU affinity for the ens160 NIC queues:

```
1 echo 2 > /proc/irq/56/smp_affinity
2 echo 2 > /proc/irq/57/smp_affinity
3 echo 2 > /proc/irq/58/smp_affinity
4 echo 2 > /proc/irq/59/smp_affinity
```

Listing 5.4: Set irq affinity

Since we use standard vmxnet3 NICs, the same issue will probably apply for others running their network on VMware ESXi. The following error message was presented when trying to use flow steering to pin flows to one queue:

```
1 $ sudo ethtool -N ens160 flow-type tcp4 dst-ip 10.0.1.2 dst-port
      2999 action 1
2 rxclass: Cannot get RX class rule count: Operation not supported
3 Cannot insert classification rule
```

Listing 5.5: Flow steering error

5.5 Future work

Figuring out how to pin traffic flows to one NIC queue and thereby stabilizing traffic performance should be tested and verified. The NIC types would probably have to be changed in this instance.

Knowing that IncludeOS can handle large amounts of traffic even when compiled with large firewall filters, it should be possible to use the unikernel for DDoS protection/mitigation as well. We propose a study where IncludeOS is tested in a DDoS simulation with a whitelist of allowed IP addresses while blocking all other traffic.

XDP and eBPF is definitely worth watching in the coming years, as it may bring some serious competition to existing Linux-based firewalls and network equipment, and maybe also to unikernels. As XDP/eBPF is in its early stages, further research and development is needed.

Continued network performance testing on unikernels would also be valuable. A study looking at HTTP, DNS and load balancing performance could be an important contribution to the field. If more unikernels aimed at networking tasks are developed, a performance comparison between the unikernels would be interesting.

Next-generation firewalls with multi layer and deep packet inspection that includes intrusion prevention and detection mechanisms could be suited for unikernels. It would also be interesting to see if unikernels could handle other security related tasks like running antivirus and anti malware software.

Chapter 6

Conclusion

The tests conducted in this thesis has shown that the IncludeOS unikernel is more than capable of competing with existing firewall solutions available for Linux at this time. Using a fraction of the resources of a standard Ubuntu Server implementation, IncludeOS did not only match Ubuntu Server in these tests, but exceeded it in every test, though often not by much when compared to ipset and nftables.

The performance advantages observed with nftables and IncludeOS comes from the configurations being compiled before it is processed. Having configuration files that have to be checked while a program is running is inefficient. IncludeOS' NaCl configuration language is transpiled into C++ and compiled for a high degree of optimization and efficiency. Nftables is similar in that it uses the nft command line tool to compile a ruleset into bytecode and pushes that into the kernel, explaining its good performance.

Being immutable, unikernels have a security advantage over general-purpose operating systems like Linux. After a unikernel image is built (including all configuration) the image cannot be altered, but rather has to be replaced completely. Another key advantage of unikernels over traditional operating systems is their small size. Not having to include all sorts of drivers and unnecessary code, unikernels can be as little as a few kilobytes – compared to gigabytes for GPOSs.

For large cloud providers serving tens or hundreds of thousands of customers with even more virtual machines, switching to a unikernel like the one tested here could help save enormous amounts of resources and consequently cost. We gave the IncludeOS VM 128 MB RAM, but did not see it using more than a few megabytes, compared to Ubuntu's usage of over 1,3 gigabytes as reported by ESXi.

Building unikernels that incorporate other security related functions could also be beneficial for cloud providers. These unikernels could possibly address tasks like antivirus and anti malware, network offloading and load balancing inside a cloud. Being quick to build and update, these could be

deployed when needed both in front of LAN and WAN interfaces of cloud VMs.

Based on the results, we can safely say that iptables, which was introduced all the way back in 2001, is ready for retirement when it comes to enterprise-grade firewalls. Looking through filter rules one by one the way iptables does is slow and does not scale: Throughput on our Ubuntu Server firewall (Fw1) with iptables were down to 50% compared to having no rules applied when running a set of 3000 firewall rules blocking different TCP destination ports. At 10 000 rules/blocked IPs Fw1 Ubuntu Server managed only about a third of the original throughput, and when running a filter blocking 50 000 IP addresses, throughput went down to around 0.6 Gbps, while IncludeOS pushed 9.25 Gbps - 15 times more.

Iptables' bad performance when dealing with large rulesets is nothing new, and solutions that use sets have been available for a while. Ipset showed much better results than iptables from just a couple of hundred rules/blocked IPs/ports. Using the bitmap:port and ip:port set types, ipset performance was not degraded even when implementing thousands of ports or IPs.

The newer nftables, thought by some to replace iptables as the standard Linux firewall, achieved the best throughput of the Linux firewalls. Not as well documented as iptables and ipset, it can take a bit of Googling to set up a firewall from scratch using nftables, but the performance may be worth it; our nftables firewall was not at all affected by implementing up to 10 000 IPs matching incoming packets, but it was slower than IncludeOS at 50 000 blocked IP addresses.

The cloud and IaaS provider Basefarm has already been testing IncludeOS in a production environment for over a year, and holds the unikernel in high regard [50].

Bibliography

[1] Anil Madhavapeddy and David J. Scott. "Unikernels: Rise of the Virtual Library Operating System." In: *Queue* 11.11 (Dec. 2013), 30:30–30:44. ISSN: 1542-7730. DOI: 10.1145/2557963.2566628. URL: http://doi.acm.org/10.1145/2557963.2566628.

[2] *Windows 10 requirements*. URL: https://www.microsoft.com/en-us/windows/windows-10-specifications.

[3] *Attack Surface Analysis Cheat Sheet*. URL: https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet.

[4] Andy Greenberg. *New Dark-Web Market is Selling Zero-Day Exploits to Hackers*. URL: https://www.wired.com/2015/04/therealdeal-zero-day-exploits/.

[5] *Rail operators among those falling victim to world's biggest ever cyber-attack*. URL: http://www.transportsecurityworld.com/rail-operators-among-those-falling-victim-to-worlds-biggest-ever-cyber-attack-1.

[6] Jan Engelhardt. *Netfilter components*. 2014. URL: https://commons.wikimedia.org/wiki/File:Netfilter-components.svg.

[7] *What is a Container*. URL: https://www.docker.com/what-container.

[8] Per Buer. *Introducing Liveupdate*. 2017. URL: http://www.includeos.org/blog/2017/liveupdate.html.

[9] Per Buer IncludeOS. *What is a Unikernel?* 2017. URL: https://www.youtube.com/watch?v=TNjPkOvHO2c.

[10] Alfred Bratterud. *IncludeOS on Solo5 and ukvm*. 2017. URL: http://www.includeos.org/blog/2017/includeos-on-solo5-and-ukvm.html.

[11] A. Bratterud et al. "IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services." In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. Nov. 2015, pp. 250–257. DOI: 10.1109/CloudCom.2015.89.

[12] *IncludeOS.com*. URL: http://www.includeos.com/.

[13] *Varnish Software*. URL: https://www.varnish-software.com/.

[14] Per Buer. *NaCl - A unikernel configuration language*. 2017. URL: https://devel.unikernel.org/t/nacl-a-unikernel-configuration-language/285.

[15] Annika Hammervoll. *Introducing NaCl*. 2017. URL: http://www.includeos.org/blog/2017/introducing-nacl.html.

[16] VMware. *Security of the VMware vSphere Hypervisor*. 2014. URL: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/whitepaper/techpaper/vmw-white-paper-secrty-vsphr-hyprvsr-uslet-101.pdf.

[17] L. Garber. "The Challenges of Securing the Virtualized Environment." In: *Computer* 45.1 (Jan. 2012), pp. 17–20. ISSN: 0018-9162. DOI: 10.1109/MC.2012.27.

[18] *Private IaaS Security*. URL: https://www.checkpoint.com/products/iaas-private-cloud-security/.

[19] *Deep Security for the Data Center*. URL: https://www.trendmicro.com/en_no/business/products/hybrid-cloud/deep-security-data-center.html.

[20] *Next-Generation Firewall - Paloalto*. URL: https://www.paloaltonetworks.com/products/secure-the-network/next-generation-firewall.

[21] *Next-Generation Firewall - Check Point*. URL: https://www.checkpoint.com/products/next-generation-firewall/.

[22] *Next-Generation Firewall - Juniper*. URL: https://www.juniper.net/us/en/solutions/software-defined-secure-networks/next-gen-firewall/.

[23] *VyOS Home page*. URL: https://vyos.io/.

[24] Phil Sutter. *Benchmarking nftables*. 2017. URL: https://developers.redhat.com/blog/2017/04/11/benchmarking-nftables/.

[25] Raik Niemann, Udo Pfingst, and Richard Göbel. "Performance Evaluation of netfilter: A Study on the Performance Loss When Using netfilter as a Firewall." In: *CoRR* abs/1502.05487 (2015). arXiv: 1502.05487. URL: http://arxiv.org/abs/1502.05487.

[26] Ian Briggs et al. "A Performance Evaluation of Unikernels." In: *School of Computing, University of Utah* (2014). URL: http://media.taricorp.net/performance-evaluation-unikernels.pdf.

[27] József Kadlecsik and György Pásztor. *Netfilter Performance Testing*. 2004. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.109.6813.

[28] Anil Madhavapeddy et al. "Unikernels: Library Operating Systems for the Cloud." In: *SIGPLAN Not.* 48.4 (Mar. 2013), pp. 461–472. ISSN: 0362-1340. DOI: 10.1145/2499368.2451167. URL: http://doi.acm.org/10.1145/2499368.2451167.

[29] *Shodan - The Search Engine for Inter-connected Devices*. URL: https://www.shodan.io/.

[30] Alison DeNisco Rayome. *DDoS attacks increased 91% in 2017 thanks to IoT*. 2017. URL: https://www.techrepublic.com/article/ddos-attacks-increased-91-in-2017-thanks-to-iot/.

[31] Lily Hay Newman. *What we know about Friday's massive East Coast Internet outage*. 2016. URL: https://www.wired.com/2016/10/internet-outage-ddos-dns-dyn/.

[32] Bob Duncan, Andreas Happe, and Alfred Bratterud. "Cloud Cyber Security: Finding an Effective Approach with Unikernels." In: *Advances in Security in Computing and Communications*. Ed. by Jaydip Sen. Rijeka: InTech, 2017. Chap. 02. DOI: 10.5772/67801. URL: http://dx.doi.org/10.5772/67801.

[33] *RFC 3511*. URL: https://tools.ietf.org/html/rfc3511.

[34] *Alpine Linux*. URL: https://alpinelinux.org/.

[35] *Iperf*. URL: https://iperf.fr/.

[36] VMware. *Network Throughput in a Virtual Infrastructure*. URL: https://www.vmware.com/pdf/esx_network_planning.pdf.

[37] *Troubleshooting ESX/ESXi virtual machine performance issues (2001003)*. URL: https://kb.vmware.com/s/article/2001003#Network.

[38] Jonathan Corbet. *BPF Comes to Firewalls*. 2018. URL: https://lwn.net/Articles/747551/.

[39] Netfilter. *nftables adoption*. URL: https://wiki.nftables.org/wiki-nftables/index.php/Adoption.

[40] Brendan Gregg. *Linux Enhanced BPF (eBPF) Tracing Tools*. URL: http://www.brendangregg.com/ebpf.html.

[41] Thomas Graf. *Why is the kernel community replacing iptables with BPF?* 2018. URL: https://cilium.io/blog/2018/04/17/why-is-the-kernel-community-replacing-iptables/.

[42] *XDP - eXpress Data Path*. URL: https://www.iovisor.org/technology/xdp.

[43] Martin Lau Huapeng Zhou Nikita. *XDP Production Usage: DDoS Protection and L4LB*. 2017. URL: https://www.netdevconf.org/2.1/slides/apr6/zhou-netdev-xdp-2017.pdf.

[44] Gilberto Bertin. *XDP in Practice - Integrating XDP into our DDoS mitigation pipeline*. 2017. URL: https://www.netdevconf.org/2.1/slides/apr6/bertin_Netdev-XDP.pdf.

[45] Florian Westphal. *[RFC,POC 1/3] bpfilter: add experimental IMR bpf translator*. 2018. URL: https://www.spinics.net/lists/netdev/msg486873.html.

[46] Quentin Monnet. *The challenges of XDP hardware offload*. 2018. URL: https://fosdem.org/2018/schedule/event/xdp/.

[47] Alexei Starovoitov and Tom Herbert. *eXpress Data Path (XDP) - Programmable and high performance networking data path*. 2016. URL: https://github.com/iovisor/bpf-docs/blob/master/Express_Data_Path.pdf.

[48] Marek Majkowski. *400Gbps: Winter of Whopping Weekend DDoS Attacks*. 2016. URL: https://blog.cloudflare.com/a-winter-of-400gbps-weekend-ddos-attacks/.

[49] Ron Amadeo. *Android Things 1.0 launches, Google promises 3 years of updates for every device*. 2018. URL: https://arstechnica.com/gadgets/2018/05/android-things-hits-version-1-0-with-centralized-google-update-system/.

[50] *Har testet den norske teknologien hardt i over ett år. Mener resultatene er unike: – Dette kan bli en "game changer"*. 2018. URL: https://www.digi.no/artikler/har-testet-den-norske-teknologien-hardt-i-over-ett-ar-mener-resultatene-er-unike-includeos-kan-bli-en-game-changer/434315?key=YaDmOoHu.

Appendix A

Network setup and config

A.1 Network overview

A.1.1 Final network setup

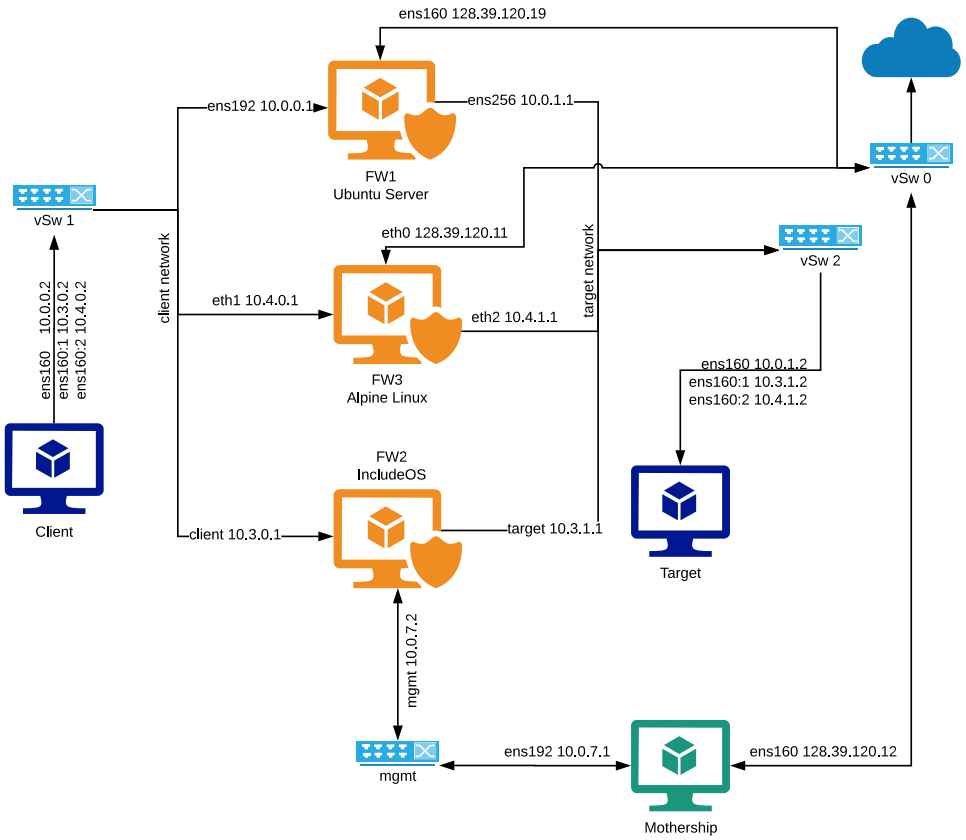

Figure A.1: Network setup model

A.2 Network config

Node network configuration (from /etc/network/interfaces)

A.2.1 Fw1 Ubuntu Server

```
1 # The loopback network interface
2 auto lo
3 iface lo inet loopback
4
5 # The primary network interface
6 auto ens160
7 iface ens160 inet static
8         address 128.39.120.19
9         netmask 255.255.255.0
10        network 128.39.120.0
11        broadcast 128.39.120.255
12        gateway 128.39.120.1
13        # dns-* options are implemented by the resolvconf package,
      if installed
14        dns-nameservers 8.8.8.8 8.8.4.4
15
16 #Towards Client
17 auto ens192
18 iface ens192 inet static
19        address 10.0.0.1
20        netmask 255.255.255.0
21        network 10.0.0.0
22        broadcast 10.0.0.255
23
24 #Towards Target
25 auto ens256
26 iface ens256 inet static
27        address 10.0.1.1
28        netmask 255.255.255.0
29        network 10.0.1.0
30        broadcast 10.0.1.255
```

Listing A.1: Fw1 Ubuntu Server nw config

A.2.2 Client

```
1 # interfaces(5) file used by ifup(8) and ifdown(8)
2 auto lo
3 iface lo inet loopback
4
5 #Towards Fw1-Ubuntu
6 auto ens160 ens160:1 ens160:2
7
8 iface ens160 inet static
9         address 10.0.0.2
10        netmask 255.255.255.0
11        network 10.0.0.0
12        broadcast 10.0.0.255
13
14        up ip route add 10.0.1.0/24 via 10.0.0.1
```

```
15
16 #Towards Fw2--IncludeOS
17 iface ens160:1 inet static
18         address 10.3.0.2
19         netmask 255.255.255.0
20         network 10.3.0.0
21         broadcast 10.3.0.255
22
23         up ip route add 10.3.1.0/24 via 10.3.0.1
24
25 #Towards Fw3--Alpine
26 iface ens160:2 inet static
27         address 10.4.0.2
28         netmask 255.255.255.0
29         network 10.4.0.0
30         broadcast 10.4.0.255
31
32         up ip route add 10.4.1.0/24 via 10.4.0.1
```

Listing A.2: Client nw config

A.2.3 Target

```
1 # interfaces(5) file used by ifup(8) and ifdown(8)
2 auto lo
3 iface lo inet loopback
4
5 #Towards FW1--Ubuntu
6 auto ens160 ens160:1 ens160:2
7
8 iface ens160 inet static
9         address 10.0.1.2
10        netmask 255.255.255.0
11        network 10.0.1.0
12        broadcast 10.0.1.255
13
14        up ip route add 10.0.0.0/24 via 10.0.1.1
15
16 #Towards FW2--IncludeOS
17 iface ens160:1 inet static
18         address 10.3.1.2
19         netmask 255.255.255.0
20         network 10.3.1.0
21         broadcast 10.3.1.255
22
23         up ip route add 10.3.0.0/24 via 10.3.1.1
24
25 #Towards FW2--Alpine
26 iface ens160:2 inet static
27         address 10.4.1.2
28         netmask 255.255.255.0
29         network 10.4.1.0
30         broadcast 10.4.1.255
31
32         up ip route add 10.4.0.0/24 via 10.4.1.1
```

Listing A.3: Target nw config

A.2.4 Fw2 IncludeOS

```
Iface mgmt {
    index:0,
    address:10.0.7.2,
    netmask:255.255.255.0,
    gateway:10.0.7.1
}

Iface client {
    index:1,
    address:10.3.0.1,
    netmask:255.255.255.0
}

Iface target {
    index:2,
    address:10.3.1.1,
    netmask:255.255.255.0
}

Gateway gw {
    r1: {
        net:10.3.0.0,
        netmask:255.255.255.0,
        iface:client
    },

    r2: {
        net:10.3.1.0,
        netmask:255.255.255.0,
        iface:target
    }
}
```

Listing A.4: Fw2 IncludeOS nw config

A.2.5 Fw2 IncludeOS w/prerouting filter

```
Iface mgmt {
    index:0,
    address:10.0.7.2,
    netmask:255.255.255.0,
    gateway:10.0.7.1
}

Iface client {
    index:1,
    address:10.3.0.1,
    netmask:255.255.255.0,
    prerouting:ipfilter
}

Iface target {
    index:2,
    address:10.3.1.1,
    netmask:255.255.255.0
}
```

```
20
21 Gateway gw {
22
23 #     forward:ipfilter,
24
25     r1: {
26         net:10.3.0.0,
27         netmask:255.255.255.0,
28         iface:client
29     },
30
31     r2: {
32         net:10.3.1.0,
33         netmask:255.255.255.0,
34         iface:target
35     }
36 }
37
38 my_ports: [
39     3000-7999
40 ]
41
42 Filter::TCP tcpfilter {
43     if (tcp.dport in my_ports) {
44         drop
45     }
46 }
47
48 filter::IP ipfilter {
49     tcpfilter()
50     accept
51 }
```

Listing A.5: Fw2 IncludeOS prerouting filter

A.2.6 Mothership

```
1 # interfaces(5) file used by ifup(8) and ifdown(8)
2 auto lo
3 iface lo inet loopback
4
5 #Towards Internet
6 auto ens160
7 iface ens160 inet static
8         address 128.39.120.12
9         netmask 255.255.255.0
10        network 128.39.120.0
11        broadcast 128.39.120.255
12        gateway 128.39.120.1
13
14 #Towards FW2
15 auto ens192
16 iface ens192 inet static
17        address 10.0.7.1
18        netmask 255.255.255.0
19        network 10.0.7.0
20        broadcast 10.0.7.255
```

Listing A.6: Mothership nw config

A.3 Testing scripts

A.3.1 ipset scale and iPerf tests

```
#!/bin/bash

iptables -F FORWARD      # Flush FORWARD table
ipset flush ports        # Flush ipset "ports"

sudo iptables -A FORWARD -i ens192 -o ens256 -m set --match-set
    ports dst -j DROP        # Match ipset

target='10.0.1.2'        # 10.0.1.2 for Fw1 || 10.3.1.2 for Fw2
port=8000                # iPerf target port
#bw='2500m'              # UDP bandwidth

today=`date +%Y-%m-%d.%H:%M:%S`                    # Can be used for
    logging
log="/home/tobias/log/Fw1.ipset.test.log"          # Log location
touch $log                                         # Create log file

testNo=1         # Appears in the log
dport=4000       # Start port

for x in {4000..4999..100}; do

        echo "Test number: $testNo"
        echo "dport value: $dport"

        (( toPort=dport+99 ))

        echo "toPort value: $toPort"

        ipset add ports TCP:$dport-$toPort
        (( dport=dport+100 ))

        for y in {1..30}; do
                echo "Test $testNo" >> $log
                ssh -i .ssh/master.key tobias@10.0.0.2 "iperf -c
    $target -p $port -t 30" >> $log
                echo -e
    '-------------------------------------------------------------\
    n' >> $log
                (( testNo++ ))
                (( y++ ))
                sleep 2
        done
done
```

Listing A.7: ipset scale and test script

A.3.2 iPerf TCP testing script

```
#!/bin/bash

target='10.0.1.2'
port=8000
```

```
5
6 today=`date +%Y-%m-%d_%H.%M.%S`
7 log="/home/tobias/log/Fw1.sAddr.10k.log"
8 touch $log
9
10 testNo=1
11
12 for y in {1..30}; do
13         echo "Test $testNo" >> $log
14         iperf -c $target -p $port -t 30 >> $log
15         echo -e
    '------------------------------------------------------------\
    n' >> $log
16         (( testNo++ ))
17         sleep 2
18 done
```

Listing A.8: iPerf TCP testing script

A.3.3 iPerf UDP testing script

```
1 #!/bin/bash
2
3 target="10.0.1.2"        #10.0.1.2 for fw1 (Ubuntu), 10.3.1.2 for
    fw2 (IncludeOS)
4 port=8000                #UDP port number (5001 is default)
5 noOfTests=30             #Number of tests
6 testNo=1                 #Test number (do not edit)
7 timePerTest=30           #Time per test in seconds
8 bandwidth="5000m"        #Mbps pushed by client
9
10 log="/home/tobias/log/Fw1.UDP.noRules.test.log"
11
12 while [ $noOfTests -gt 0 ]
13 do
14         #Run iperf throughput test
15         echo "Test $testNo $target">> $log
16         iperf -c $target -p $port -t $timePerTest -u -b $bandwidth
     >> $log
17         echo "
    ------------------------------------------------------------"
    >> $log
18         echo "" >> $log
19         sleep 2
20         (( noOfTests-- ))
21         (( testNo++ ))
22 done
```

Listing A.9: iPerf UDP testing script