FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Evaluating the Unikernel Concept for the Deployment of Software on IoT Devices

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Jan Amort, BSc.
Matrikelnummer 0425530

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Univ.Prof. Dr. Schahram Dustdar

Wien, 1. Dezember 2017

Jan Amort	Schahram Dustdar

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Evaluating the Unikernel Concept for the Deployment of Software on IoT Devices

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Jan Amort, BSc.
Registration Number 0425530

to the Faculty of Informatics
at the TU Wien
Advisor: Univ.Prof. Dr. Schahram Dustdar

Vienna, 1st December, 2017

Jan Amort

Schahram Dustdar

Erklärung zur Verfassung der Arbeit

Jan Amort, BSc.
Lienfeldergasse 3/8, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Dezember 2017

Jan Amort

Acknowledgements

I would like to thank Prof. Schahram Dustar for agreeing to supervise this diploma thesis, for his advice and valuable feedback.

I thank Martina for encouraging me to finish my study and her support throughout this journey. Particularly for her support, patience and understanding when I had to work late during the diploma thesis.

To my parents I want to say thank you for allowing me to study, their patience and support and allowing me to pursuit my own ways.

When it comes to writing a thesis it can be a lonesome endeavor. I want to thank Alex for her input, feedback, encouragements and being a good companion during writing of this thesis.

Kurzfassung

IoT Geräte sind nicht mehr wegzudenken aus dem modernen Alltagsleben. Als integraler Bestandteil des täglichen Lebens haben Sicherheitsprobleme dieser Geräte eine große Auswirkung auf deren User.

Die relativ neue Verfügbarkeit von Hypervisoren für die vergleichsweise leistungsschwachen CPUs von IoT Geräten bringt Vorteile bezüglich Sicherheit und der Bereitstellung von Software auf den Geräten. Andererseits beeinträchtigt der Einsatz eines vollumfänglichen Betriebssystems auf einem Hypervisor die Leistungsfähigkeit des IoT Geräts.

In den letzten Jahren wurde das Unikernel Konzept neu entdeckt. Unikernels kombinieren eine Applikation zusammen mit einem kleinen Umfang an Betriebssystemfunktionen zu einer untrennbaren, leistungsstarken Einheit. Dies macht Unikernels zu einer interessanten Option für die Bereitstellung von Software auf IoT Geräten.

Diese Arbeit untersucht ob Unikernels dafür gut geeignet sind, Software auf IoT Geräten bereitzustellen und dazu beitragen können die Sicherheit des Gerätes zu erhöhen indem Updates zeitnah ausgeliefert werden können.

Im ersten Teil der Arbeit werden relevante Performancecharakteristiken für IoT Geräte definiert und umfangreiche Tests durchgeführt. Die Ergebnisse der Unikernels werden mit den Ergebnissen von Virtuellen Maschinen verglichen und stellen die Grundlage für die weiteren Erwägungen dar.

Die sicherheitsrelevanten Eigenschaften von Unikernels werden in einem weiteren Hauptteil untersucht und dargestellt, welche Auswirkungen sich daraus für die Sicherheit von IoT Geräten ergeben.

Da die Sicherheit des System entscheidend davon abhängt ob Aktualisierungen der Software zeitnah für eine große Anzahl von Geräten bereitgestellt werden können, wird in einem weiteren Kapitel untersucht wie ein Prozess von der Entwicklung bis zur Installation von Software unter Einbeziehung von Unikernels gestaltet werden könnte.

Abstract

The ever increasing number of IoT devices makes it clear that a lack of security in these systems has a big impact on the lives of people. The availability of hypervisors for low-power CPUs used on IoT devices brings advantages regarding security and deployment strategies but bear the question how the use of fully fledged operations systems would impact the performance of these low-power CPUs.
In recent years the concept of a Unikernel has reemerged, challenging the idea of a general purpose operating system. A Unikernel combines an application with a minimal set of operating system functionalities needed to run the application on a hypervisor. This makes Unikernels an interesting option for deploying applications on a hypervisor-based IoT device.
This work evaluates whether Unikernels are a good fit for deployments of software on IoT devices and can contribute to the security of IoT devices by facilitating the timely rollout of new versions of an application.
In the first part this work conducts a series of comprehensive performance tests for a range of characteristics relevant in the field of IoT devices and compares the results of Unikernels with virtual machines. Next this work evaluates the problems by IoT devices regarding security, takes a look at current incidents involving IoT devices and the claims of Unikernels with respect to security. The chapter takes the inherent properties of Unikernels in consideration and evaluates whether these properties can contribute to the security of the overall system.
The security of the system depends upon the ability of rolling out new versions of an application in a timely manner whenever bugs are discovered. Therefore the third part lays out how Unikernels could be integrated into a modern software development process based on continuous integration and -delivery principles to enable the frictionless deployment, testing and distribution of new versions of an application.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The term Internet of Things (IoT) refers to a range of heterogeneous devices that have in common, that they are all connected to some type of Internet service and have relatively low system resources. IoT devices are deployed in smart homes as part of the climate system for the heating control system, operating security cameras or as part of the entertainment system. The term IoT applies to industrial control systems (Industry 4.0) as well as for interconnected autonomous driving systems.
A Gartner study estimates the number of IoT devices deployed in 2017 to be 8,4 billion with a forecast of 20 billion in the year 2020 (41). With the widespread presence of IoT devices we also have to face security problems, that for a long time have been haunting desktop and server environments, entering our homes and critical infrastructure. IoT devices operate on general-purpose hardware and run applications on top of an operating system like any other IT system and like any type of software it contains bugs that pose potential security risks.

In the recent months there have been a number of high profile attacks on IT systems of companies and home users through ransomware trojans. These campaigns use zero-day exploits for bugs that have not yet been fixed. The cycles between the publication of a bug in a system and the development and use of a corresponding exploit became shorter through the professionalisation of this industry. Criminal groups make large amounts of money with large-scale attacks on systems and thereby raise the market prices for bugs and exploits. It's an ongoing armsrace between criminals trying to exploit systems and the software companies fixing bugs in their software and rolling it out to their users.
IoT devices are specially valuable targets as they are part of the inner network of a company and, therefore, can be the launchingpads for further attacks on the network. They are a vital part of the infrastructure and due to their heterogeneity and lack of accessible user interfaces are harder to keep updated.

This recent development shows that timely updates and automatic rollouts are an important part of the security strategy and reduce the likelihood to become victim of a ransomware campaign.

Since IoT devices are part of the IT infrastructure of a company and, therefore, are connected on various points to other IT systems, they become interesting targets for attacks and therefore call for a security strategy with a focus on hardening the system and timely rollouts of new software versions as soon as bugfixes are available.

1.2 Problem Statement

The availability of virtualization for low-power CPUs used on IoT devices brings advantages like improved security and easier software deployments known from the world of cloud computing to IoT devices. The use of fully-fledged virtual machines on these devices raises questions regarding performance and feasibility of deployments over the network. The goal of this thesis is to evaluate whether the concept of Unikernels is able to address the problems arising from the use of traditional virtual machines on IoT devices and inquiry if Unikernels can bring advantages regarding security, performance and deployment strategies to x86 based IoT devices.

Since the currently most advanced Unikernel implementations OSv and Rumprun do not support ARM architectures yet (see 2.4.4), the focus will be on Unikernels for the x86 processor architecture.

The use of virtualization technologies on IoT devices brings performance penalties through the additional layer between the application and the underlying hardware. The question arises how the different Unikernels perform compared to a virtual machine running a standard Linux system. There have been performance measurements and comparisons before in the literature but they either focused on one particular Unikernel implementation (79) (56), compare Unikernels with other systems like Docker or focus on a small selection of performance characteristics (16) (82). The question remains how the performance characteristics of Unikernels compare to a standard Linux system in a range of categories particularly relevant for IoT devices like the handling of various network protocols, boot time and image size. To this end this thesis will present an evaluation framework for performance characteristics of IoT devices and conduct comprehensive series of tests in the categories defined in the evaluation framework in order to draw conclusions on the ability of Unikernels in the handling of IoT workloads.

For an accurate performance comparison between Unikernels and a Linux VM the same application has to be deployed to all tested systems. For this the Unikernels need to be POSIX compatible. Therefore, this work focuses on the two POSIX compatible Unikernel projects OSv and Rumprun during the performance evaluation. Other evaluated aspects like the security properties apply to other Unikernels as well.

The properties of Unikernels differentiating them from modern operating systems have security implications, positive and negative. This work will evaluate the impact of these

properties on the security and assess whether they can contribute to the overall security of the system and which type of attacks could be mitigated by deploying software in Unikernels.

The security of an IoT system relies heavily on the timely rollout of updates whenever bugs are found. This work will inquiry how a software development and deployment process based on Unikernels could look like, where sourcecode would be compiled into Unikernels and what benefits a Unikernel based process could bring for software deployments on IoT devices.

1.3 Methodology

In a first step, a literature research is conducted in order to get a comprehensive overview of the problems and inefficiencies of IoT devices regarding security and deployment management. Research on virtualization support for low power CPUs will be included to define appropriate type 1 hypervisors (XEN, KVM) for the use on a IoT field device.

As a second step, commonalities and differences of different implementations (MirageOS, Rumprun, OSv) of the Unikernel concept are being compared and evaluated for their applicability for the problems of IoT security and deployment management. As some implementations are designed for a specific purpose or programming language, not all of them are equally applicable for the deployment of software on IoT devices.

An evaluation framework will be defined allowing the comparison of different deployment strategies (virtual machines running full operating systems, Unikernel) regarding performance and defines relevant metrics in the IoT world regarding execution speed, network handling (protocols relevant in the sphere of IoT devices) and updatability.

Based on the evaluation framework, comprehensive tests of the performance characteristics of the different Unikernels will be conducted. A range of tests for different protocol implementations used in IoT devices will lead to a detailed view on the networking capabilities of the Unikernels and allow for a comparison of the results with a reference Linux system.

Next, a real-world IoT application will be developed simulating a scenario of an IoT device collecting temperature sensory data. The application will be developed in two different languages and frameworks that are supported by both Unikernel projects and the Linux system. Experiments will be conducted deploying these applications on a virtual machine running Linux and as a Unikernel. These tests will allow to draw further conclusions about the performance of different aspects of the Unikernels since the applications go beyond the simple handling of protocol requests. The evaluation framework will be used to assess the performance of the different deployment strategies in the dimensions relevant for IoT devices.

In a next step, a prototype for deploying software on IoT devices will be built. This will show the applicability of the Unikernel concept for the deployment management of IoT

devices with virtualized x86 hardware. The deployment system manages the sourcecode for the different applications running on a IoT device in software repositories. Whenever a new version, tagged as a release, is pushed, the deployment system checks out the code, compiles it into a Unikernel and presents it in a web interface as a new update available for the IoT devices. Since we assume that the IoT devices are not directly connected to the Internet and therefore cannot be reached by a server by directly addressing the device, an update service running on the IoT devices need to query the central deployment system regularly to check for new available updates. As soon as the IoT field device realizes that there is a new update available, it downloads the updated image, verifies the image by calculating a hash and starts the new image on the hypervisor. This will lead to a handover from the old Unikernel instance to the new and the old instance can be shut down. In order to assess the deployment prototype, a IoT application will be implemented exposing a webservice and acts as a proof-of-concept for the deployment system.

CHAPTER 2

Fundamentals

2.1 Evolution of Operating Systems

In the early days of computing the computers were big mainframes focused on completing a single job at a time. A user was attributed a certain time slot and could load the computer with the program (punched paper cards) and data. The computer would calculate a result and print it out. Since computers were big and expensive only a small number of organizations could afford it and a large number of users needed to share the same computer. Timesharing was an idea by Bob Bemer in 1957 allowing multiple users to share the same computer and interact with the mainframe through a terminal. The amount of real memory in a system was limiting the timesharing capabilities of the systems. The publication of Arden et al. (7) in 1966 describing virtual memory and address translation for the first time was the starting point facilitating timesharing in mainframes (77).

IBM designed their IBM System/360 Model 67 for the needs of time sharing and implemented virtual memory for the first time. The operating system "Time Sharing System Monitor" regulated the access to the hardware and supported a multiprocessor system. The virtual memory system implemented segment- and page tables and the operating system translated the virtual addresses into real addresses with the "Dynamic Adress Translation Box" (DAT box). With its 32bit virtual address space it could address 4GB of memory. The first operating system supporting operating system virtualization was the CP-40/CMS that was deployed for the System/360. It allowed the execution of multiple operating systems in separate virtual machines. The virtual address space and other features allowed the system to virtualize IO and interrupt handling (42).

The era of the home and personal computing began with the availability of cheaper commodity hardware using common interconnection. Early operating systems like the CP/M-80 for the 8080 processor family and later Microsofts MSDOS designed for the

IBM PC provided a standard operating system working for a variety of hardware. These systems were single user systems and the user interacted with the operating system via a commandline. MSDOS did not provide memory protection or a scheduler. There was only one process running at a given time and it did so until the process has finished. There was no scheduler that could preempt the process and share the CPU with other running processes (77).

In the 1980s the graphical user interface was introduced to the personal computer by multiple vendors. This led to the use of the personal computer by multiple users and the operating systems started to introduce additional protection mechanisms. Windows 3.1 was the first system from Microsoft introducing a non-preemptive scheduler that would allow to run several applications in parallel. Operating system manufacturers began to implement protection mechanisms limiting the access of processes to the memory. With the introduction of the 80286 processor (member of the x86 architecture) two important security features were introduced: Rings and the CPU mode "protected mode". The concept of protection rings allowed to attribute privilege levels (rings) to code blocks (see 2.2.2 for detail). The protected mode regulated access to memory segments with the help of a newly introduced memory management unit (MMU). With this new mode the operating system could for the first time protect privileged operating system processes from unprivileged code. The next generation 80386 then introduced new paging mechanisms that made virtual memory easier and faster on x86 based systems (77).

On the server side commodity servers based on x86 system architectures began entering the market. They were used to serve personal homepages and small commercial websites and provided webspace to multiple users on the same operating system. Access to the resources like harddisk and memory was regulated by operating system capabilities attributing quotas to the users and separating the users from each other with filesystem permissions. Users only got access to certain folders on the system via ftp to store the content of the website. The computing services provided to the customer were limited to isolated services like webhosting and e-mail services.

While virtualization was available on mainframes since the 70s, on the x86 processor architecture used for personal computers and servers the concept of virtualization became popular only in the late 1990s with hypervisors by VMware. The availability of hypervisors for x86 based server systems made it possible to run multiple virtual machines on a hypervisor on cheap commodity hardware. The system provider could sell complete virtual machines with varying resource attributions to the customer. The customer got complete root access to the operating system and could buy additional computing power or memory when needed (77).

An application running on an operating system requires a number of libraries to execute. When several applications run in parallel requiring different versions of the same library, this could lead to conflict and result in harder to maintain system setups. This and the fact that different setups on developer and production systems could bring undetected conflicts on production machines lead to the widespread adoption of container systems for software deployments. These container systems like Docker are often deployed on

top of virtual machines. The abstraction provided by the container is implemented on the operating system level. Containers run as normal processes on an underlying Linux system, using the kernel of the host (see 3.2.2).

This development over time shows that a system running Docker on a virtual machine has a number of redundant protection mechanisms collected throughout the history of computing systems. The Hardware is encapsulated by a hypervisor running multiple guest systems in parallel and separates itself from the guests through privileged CPU modes. The hypervisor separates the memory pages of the guests from each other and from the memory pages used by the hypervisor. The guest system then runs a full operating system in the VM using CPU privilege levels to separate itself from the running processes in the VM, and the processes from each other. One of these processes is a Docker container running an operating system environment inside the container using the kernel of the host (that is a guest of the hypervisor) to separate the multiple containers from each other and simulating hardware resources.

Unikernels in contrast to this example only rely on the hypervisor for resource separation and protection. All operating system level mechanisms like privilege transitions between operating system and application, separated virtual memory and user management is abandoned for the gain of speed, reduced attack surface and smaller system images. The goal is simply to run a single application on top of a hypervisor.

2.2 Operating System Concepts

Unikernels see operating system functionalities as a set of libraries used to aid the execution of the one application running in the Unikernel. They break with a number of operating system concepts and protection mechanisms in order to gain advantages regarding execution speed and optimize the Unikernel for running on a hypervisor. This chapter takes a look at traditional operating system concepts that later in chapter 2.4 get reevaluated and challenged by the Unikernel projects.

2.2.1 Kernels

The kernel is the central piece of an operating system interacting with the hardware on behalf of the other software components. It controls peripherals like keyboard, network cards, graphic cards and manages the memory for the programs. System calls are used by programs to interact with the kernel and request an operation. There have been several design principles in the past. Most modern kernels fall into one of the following two design principles: monolithic- or micro kernel.

2.2.1.1 Monolithic Kernel

Monolithic kernels are kernels that include a wide range of functionalities directly into the kernel itself and thereby letting them run in kernel space with the corresponding privileges. Until recently most operating systems were monolithic kernels. Typically, a

monolithic kernel is implemented as a single process, with all elements sharing the same address space. Examples of implementations of this design principle are Linux, Windows 9x and MS DOS (95).

Linux has addressed the problems and shortcomings of the monolithic model by developing loadable kernel modules that can be loaded into the kernel only when needed. This reduces the size of the codebase running in the monolithic kernel simultaneously. An advantage of the monolithic kernel architecture is the fact that all the code is running in kernel mode and thus the costly privilege transitions into and out of ring0 (see 2.2.2) can be omitted. Another advantage is the easier communication between parts of the operating system. A disadvantage of this architecture model occurs when parts of the kernel get changed or need to be patched. This requires a recompilation of the whole kernel instead of just patching a module (64).

2.2.1.2 Microkernel

A microkernel externalizes as much functionality as possible into the user space and provides a minimal coordinating set of functions in kernel space. The microkernel provides functionalities for memory management, multi-tasking and inter process communication in the kernel and externalizes functions like networking into so called "servers" in user space. These user space servers are treated like any other user space application by the microkernel. An advantage of the microkernel architecture is the smaller trusted computing base. This is the amount of code the kernel needs to trust in order to function. The microkernel design simplifies the development by separating the kernel development from the server development (95).

A disadvantage of this concept is the reduced execution speed. The functionalities of the kernel are separated into several userland processes and therefore result in a larger number of context switches (see 2.2.5) and inter-process communication. Subsequently the system is slower than a comparable monolithic kernel.

Implementations of the microkernel principle are systems like QNX, L4 and the Mach kernel.

2.2.2 CPU-Modes and Rings

CPU modes (also called CPU privilege levels) are operation modes implemented in the CPU regulating which operations can be executed in a particular mode and which parts of the virtual address space can be accessed. This allows an operating system to run with more privileges than a normal application and at the same time to control the access of unprivileged processes to hardware resources. Each process gets assigned a privilege level which it is running in and stores this information in the segment descriptor (64).

The IA-32 architecture uses a system of four privilege levels that can be visualized as rings. These rings are numbered from 0 to 3, where ring0 is the most and ring3 the least privileged. This mechanism is in place to protect hardware resources from user processes

and prohibit unprivileged processes from accessing virtual address space pages used by the kernel. Through this separation the operating system can ensure that only processes running on ring0 are allowed to access the hardware directly and thereby enforce security policies. Most modern operating systems like Windows and Linux use only two of the four available rings: ring0 for privileged access and ring3 for user mode. The resources protected by the ring model are memory, I/O ports, some registers and certain machine instructions. The ring3 process thus cannot execute instructions that would result in more rights for the process like changing the privilege level, changing pagetables or registering interrupt handlers. When programs try to run these protected instructions outside of ring0 this leads to a general-protection exception, like when programs use invalid memory addresses (53) (64).

When an application, running in the unprivileged ring3, needs to access the hardware (eg. write data to the disk) it issues a so called system call. This results in executing a function in the kernel running with the heightened privilege level ring0 and therefore has access to all resources without restrictions. When the kernel function is done executing, the execution in the user space process is continued and the privilege level drops to ring3 again (64).

2.2.3 Privilege Transitions through Syscalls

Modern operating systems encapsulate the parts of the software talking directly to the hardware (drivers) in the kernel. As described in 2.2.2 these instructions run in a higher privilege level than ordinary userspace programs and the CPU supports this separation through the concept of rings.

When a user process has to access hardware or requires services provided by the kernel, it does so by calling a system call (or syscall for short). A syscall is a function running in kernel space and therefore has access to parts of the memory belonging to the kernel and can execute any instruction since it runs in ring0. The user process initiates a syscall by setting the kernel function and the parameters in the corresponding registers and executing a software interrupt. This triggers the interrupt handler of the kernel that then resolves the function and executes the corresponding code in kernel space. When the code is finished running, it gives back control to the user process (95).

The way a syscall was implemented in Linux in previous times (and still is for older CPUs that do not support the corresponding instructions) was to load the system call number into the register EAX and execute the instruction “int 0x80”. This generates the interrupt 0x80 and an interrupt service routine in the kernel is called. This then saves the current state of the process and calls the system call handler for the system call defined in the register EAX. This method is now deprecated for modern CPUs and replaced by a faster approach with less overhead supported by new hardware instructions (40).

Modern CPUs by Intel and AMD provide a new set of instructions for faster syscalls. Current Linux systems use the hardware instructions SYSENTER/SYSEXIT for Intel CPUs and SYSCALL/SYSRET for AMD CPUs to transition between user- and kernel

space. The intention for the new instructions was to speed up system calls by eliminating unneeded checks and loading predetermined values into the CS and SS segment registers. As a result, SYSCALL and SYSRET can take fewer than one-fourth of the number of internal clock cycles to complete compared to the legacy method of CALL and RET (6).

Linux setups the entry/exit points for the syscalls by attaching a single memory page called VDSO to the address space of all user space processes. This page contains the actual implementation of the syscalls enter/exit mechanism. Through this mechanism the system can avoid context switches when calling a syscall and thereby improve the performance (40).

2.2.4 Memory Management

Most modern computing architectures use the concept of virtual memory for handling its memory requirements. This concept distinguishes between the addresses of the physical memory blocks built into the system and the (virtual) addresses used by the processes to access the memory. The process running on the CPU sees memory as a single contiguous address space with contiguous memory segments. The important advantage the system gets by using virtual memory, is that the amount of memory accessible to a process does not need to match the amount af memory physically present in the system. By splitting memory into blocks and writing unused blocks to the disk, a process can access more pages than would fit into the physical memory (also called “real memory”). The translation of virtual addresses into physical addresses is done via a hardware part called memory management unit (MMU) (95).

The memory is divided into equally sized blocks of memory called pages. Operating system functionalities can extend the capabilities of the MMU to load and save memory pages to the harddrive and thereby dynamically exceed the physically present memory. The MMU uses pagetables to translate virtual addresses into physical (also called “real”) addresses. The pagetable contains for each page a flag indicating if the page is currently in real memory and therefore can be accessed, or is swapped out to the harddisk (page eviction). In this case the MMU raises a page fault and the paging component of the operating system loads the missing page into physical memory by swapping out a currently unused page. Pages can be flaged as being writable, only readable or executable if they contain program code. This is a protection mechanism making it harder for attackers to inject code into memory and redirect the control flow of the program to execute the injected code. All input read by a program is stored temporarily in a memory page, but if this page is not executable, the attacker has not gained any advantage by being able to load his code into a non executable memory page. A virtual memory address consists of a virtual page frame number and an offset into the page. The MMU translates the virtual page frame number into a physical page via a lookup in the page table by using the virtual page frame number as an index into the page table. Modern processors support a physical address mode and a virtual address mode. If the CPU runs in physical address mode it does not attempt to translate addresses into physical addresses via the MMU. The Linux kernel uses this mode to load itself for performance reasons (95).

In Linux each process has its own virtual address space. The virtual address spaces are completely separated from each other and so a process cannot interfere with the address space of another process (91).

Processors have a number of caches for fast access to small amounts of data. One of these caches is the translation lookaside buffer (TLB) caching recently accessed entries in the page table from one or more processes. When a reference to a virtual address is made, the processor first tries to find a matching entry in the TLB. If this succeeds (called a "TLB hit"), the address can directly be translated into a physical address. If it however fails (called a "TLB miss"), a time consuming process called "page walk" starts. A signal is issued to the operating system that a TLB miss has occurred. The OS then has to lookup the physical page via the page table and calculate the real address from the offset. This address is then stored in the TLB and further can be directly translated. When a memory page is altered by the process while in real memory, it is called "dirty" and flagged as such. When a context switch happens these dirty pages need to be saved to secondary memory in order to be later restored. The TLB entries implicitly refer to the current address space. If now a context switch happens the TLB is flushed and has to be rebuilt for the new process. This is the reason why context switches are computationally expensive (91).

When a binary is executed on a system, first the image of the executable has to be loaded into memory. This process is called memory mapping. This does not mean that the image is completely loaded into physical memory, instead the image is linked into the virtual address space of the process and the corresponding pages of the image are loaded as they are accessed (95).

2.2.5 Context Switch

A context switch occurs when multiple threads share the same processor. When the scheduler decides to pause the currently running thread and continue with another thread (rescheduling), the current state of the running thread or process needs to be saved in order to later be able to continue where the thread or process was paused. A context switch is the act of saving the processors state of the thread and loading the saved state of another thread. The data saved for a thread includes the program counter (pointer to the next instruction that should be executed), stack pointer and all registers. When a thread gets rescheduled this data gets saved to the thread control block (TCB) containing the state of the thread. If the new thread is associated with a different virtual address space the context switch also includes the switching of the address translation maps used by the processor. In Linux this happens when the thread belongs to a different user process.

A process switch is more expensive than a thread switch because more data needs to be saved and the virtual memory for the new process needs to be loaded. Two threads belonging to the same process use the same virtual address space, therefore the caches do not need to be cleared. If the new thread belongs to another process the state of the old

process is saved to the process control block (PCB), the dirty memory pages are written to secondary memory, the TLB entries for the old thread are invalidated and therefore the TLB gets flushed, the pagetable for the new process gets loaded into the MMU and the data (stackpointer, program counter, registers) from the new process gets loaded from the processes PCB.
The switch from one virtual address space to another including the flushing of the caches is the reason for the higher costs for process switches than for thread switches (28).

2.2.6 POSIX Interface

The Portable Operating System Interface (POSIX (47)) is a standard to maintain compatibility between operating systems. It specifies application programming interfaces (API), utility interfaces and command line interpreter (shell). A wide range of operating systems are POSIX compatible like Linux, OpenBSD, AIX or MacOS. Other systems like the Microsoft Windows NT kernel are only compliant with a compatibility feature.

The standard defines operating system properties like filename conventions, character sets, regular expressions, path structures and environment variables. It specifies APIs called by userspace programs for handling threads, requesting or freeing memory, synchronizing between threads and accessing IO devices.
An operating system is said to be POSIX compatible if it provides all expected APIs to a userspace program and thereby allowing the POSIX compliant code to be compiled on the system without any modifications (64).

2.2.7 Spinlocks and Synchronization

When code blocks need exclusive access to a resource they need some type of mutual exclusion mechanism. Unlike sempaphores spinlocks may be used in code that cannot sleep such as interrupt handlers. Spinlocks offer higher performance then semaphores.

Spinlocks are mutual exclusion devices that can have only two states: locked and unlocked. It is usually implemented as a single bit in an integer. When a thread tries to access a resource protected by a spinlock, it tests for the spinlock and when available, locks the spinlock and continues in the critical section therefore only one thread can hold the spinlock at any given time. When on the other hand the spinlock is locked, it repeatedly tries for the lock (“spins”) until the lock gets available. In order to avoid deadlocks the “test and set” operation has to be atomic, meaning that the two operations have to be executed without interruption. In a scenario where a process acquires the spinlock and then goes to sleep or being preempted by the kernel for calling a long-running I/O operation, another process waiting for the lock could be waiting forever when the first process dies. This would result in a deadlock situation and this is the reason why code holding the spinlock has to be atomic and cannot go to sleep. A disadvantage of spinlocks is that the thread waiting to gain the lock is continuously testing for the lock and is therefore in a busy-waiting mode. The information in this chapter is taken from (60) and (64).

2.3 Virtualization

Virtualization is a concept where operating systems are executed on top of virtualized hardware instead of running on the hardware directly. Through virtualization several fully fledged operating systems can run in parallel on the same hardware. A hypervisor or virtual machine monitor (vmm) is a software layer running on top of the hardware to manage the interaction between the guest systems and the underlying hardware. The hypervisor attributes resources to the guest systems and queries hardware on behalf of the guest system. A range of CPUs by Intel, AMD and ARM provide features to support hardware virtualization on host systems. The CPUs provide hardware virtualization capabilities through dedicated instruction sets for memory management, page table virtualization and execution modes facilitating the interaction between guest and host and allows a hypervisor to efficiently manage and schedule concurrently running virtual machines.

Intel introduced with their VT-X CPU extension new instructions for the handling of virtual machines. One important new concept are two additional privilege levels “VMX Root Operation” and “VMX non-Root Operation” additionally to the Ring levels. The hypervisor runs in the VMX Root Operation mode and all virtual machines run in the VMX non-Root Operation mode. Within those modes the usual Rings 0-3 are available for the protection of processes. The big difference is that instructions executed in ring0 in the VMX non-Root mode can be trapped by the hypervisor running in VMX Root mode. This is an implementation of the trap-and-emulate procedure and solves the depreviligation problem (27).

AMD has with AMD-V a similar hardware extension implementing the trap-and-emulate principle. The “hypervisor mode” is used to distinguish between guest privileges and the hypervisor running with full access to the hardware (6).

Small devices that should fulfill a small set of tasks require suitable CPUs with lower power requirements than regular desktop- or server CPUs. In the world of mobile- and IoT devices the CPUs by the company ARM are widely used. The more recent CPU series v7 and v8 include features supporting hardware virtualization. The “ARM Virtualization Extension” introduces a new operation mode (hypervisor mode) for the CPU running with higher privileges than the previous modes and allows the hypervisor to trap and emulate instructions. Additionally the extension brings new facilities for memory management and address translation (67).

For the MIPS32 and MIPS64 processor architecture there exists an extension called “VZ” bringing hardware virtualization to MIPS processors. These processors have an additional operation mode called “quest mode” additionally to the “root mode” allowing the execution of guest systems and trapping of guest-instructions through the hypervisor running in root-mode (48).

2.3.1 Full-, Para- and Hardware Assisted Virtualization

In full virtualization the guest system is not aware that it is running in a hypervisor and thinks it is running on real hardware. The hardware is completely abstracted away by the hypervisor. The hypervisor runs in ring0 (see 2.2.2) and provides the needed virtualization infrastructure. The guest operating system resides in ring1 and the applications running inside are executed in ring3. Since the guest is not aware of the virtualization and thinks it is running in ring0 and therefore has access to the hardware, the hypervisor needs to translate the instructions on the fly (called binary-translation). The non virtualizable instructions get translated into a sequence of instructions that have the same effect (21).

Paravirtualization or partial virtualization is a concept where the guest is aware of running on a hypervisor layer. The guest system gets modified in a way that replaces non-virtualizable instructions and lets it directly interact with the hypervisor through hypercalls. Hypercalls are similar to system calls and let the guest directly talk to the hypervisor. This technique improves the performance of the system and eliminates additional overheads (21).

All of the big CPU producing companies like Intel, AMD and ARM have implemented special instructions into their CPU instruction set supporting and facilitating the execution of virtualized guest systems (called “hardware assisted virtualization”). The instructions include operations for page table management of the guest systems, storing and managing the state of the virtual machines and additional execution modes allowing a hypervisor to trap access to hardware without binary translation. This leads to performance improvements and makes hardware assisted virtualization the fastest of the three options (21).

2.3.2 Hypervisors

A hypervisor is an abstraction layer between the guest systems and the hardware. To the guest system it provides a uniform interface, emulates hardware parts and manages the access to the hardware resources of the host.

Hypervisors can be distinguished into two categories: type1 and type2. A type1 hypervisor runs directly on the hardware and is therefore also called bare-metal hypervisor. Examples for this type are XEN, Oracle VM Server and Microsoft Hyper-V. Type2 hypervisors run on top of an operating system as a normal process on the host next to other applications. Examples include VM Ware Workstation, VirtualBox and Qemu. For KVM (details see below) and FreeBSDs bhyve the case is not so clear. They use an existing operating system indicating a type 2 hypervisor but they get loaded as a kernel module and thereby turn the operating system into a type1 hypervisor (105).

Two of the most prominent open source hypervisors are XEN an KVM. They both are supported by most Unikernel projects and, therefore, are examined in more detail in the following.

XEN is a hypervisor developed as an open source project. It provides full virtualization if the required hardware (CPU extensions) is available and paravirtualisation if the hardware is not available. XEN differentiates between paravirtualized guests (so called PV guests) and hardware virtual machines (HVM). In a paravirtualized environment the guest operating system is aware of the virtualization and the hypervisor and needs to be ported to the specific hypervisor in order to work. Instead of issuing syscalls, the guest OS sends so called hypercalls to the hypervisor in order to interact with the hardware (104).

The kernel based virtual machine (KVM) is a virtualization technology based on the Linux kernel. It uses the hardware virtualization extensions Intel VT and AMD-V to run virtual machines. After loading the KVM kernel module the Linux kernel plays the role of a hypervisor for the guest systems and thereby it can be argued that KVM is a type1 hypervisor. The started guest systems appear on the host system as normal processes(55).

While the forementioned hypervisors mainly target the x86 processor architecture, there exists with the prpl hypervisor a hypervisor specifically targeting low power MIPS CPUs (87). Since MIPS processors are widely used in embedded devices like IoT devices, this hypervisor could become an interesting target for Unikernels once they support the MIPS architecture.

2.3.3 Virtio

Virtio (1) is a set of efficient drivers for Linux intended to be used by a variety of hypervisors. Before Virtio each hypervisor would implement its own block, network, console and other drivers.

Full virtualization where the hypervisor simulates a certain hardware for the guest is not efficient and therefore slow. To optimize running a guest on a hypervisor Virtio is implemented as a set of paravirtualized drivers where the guest knows that it is running on a hypervisor and both sides (guest and hypervisor) use a Virtio driver to interact with one another. This leads to a significant performance improvement compared to the full virtualization environment. The interaction between guest and host uses buffers and virtual queues to send commands to the host and receive data from the host. A device driver can use several virtual queues for the interaction. The network device driver for example uses one queue for receiving packets and one for sending network packets via the hypervisor to the network (92).

2.4 Unikernel Concept

“Unikernels are single-purpose appliances that are compile-time specialised into standalone kernels, and sealed against modification when deployed to a cloud platform. In return they offer significant reduction in image sizes, improved efficiency and security, and should reduce operational costs.” (63).

The term Unikernel refers to a design principle for a system combining all aspects of an operating system together with an application into one inseparable execution unit. An alternative name for the same concept often found in the literature is the term "Library Operating System" (84) (62). There are several projects implementing this principle and each has a different focus (see chapter 2.4.1). The projects can be separated into two categories: Unikernels aiming to be POSIX (see 2.2.6) compatible and thereby support the execution of unchanged applications, meaning an application can be ported from running on a bare metal Linux to running in a Unikernel without changing the code. The OSv Unikernel and Rumprun are members of this group and are therefore able to run unmodified applications that were not specifically written for a Unikernel. The second category are Unikernels where the application is specifically written to be run on that Unikernel implementation by using specific libraries developed for this Unikernel. Most of them require the application to be written in the same programming language as the Unikernel itself. MirageOS for example is written in the functional language OCaml and requires its applications to be written in OCaml as well (63).
Unikernels directly target hypervisors as platforms for the executions, thereby Unikernels avoid the hardware compatibility problems which older approaches to the library OS like Nexus and Nemesis had. Traditional operating systems have multiple layers of protection and thereby allow to run untrusted applications side by side. Unikernels execute only one application and therefore reduce the protection boundaries between kernel- and user space for the benefit of improved execution speed. Unikernels running on a hypervisor rely on the hypervisor to introduce the needed protection boundaries between mutually untrusting systems (62).
The chapter "Unikernel Implementations" 2.4.1 gives an overview over the available Unikernel projects, their aims and project status and goes into detail on the three most advanced projects OSv, Rumprun and MirageOS. Based on these projects chapter 2.4.2 extracts common properties and design principles found in all Unikernel implementations and defines the characteristics and aims uniting all projects.

2.4.1 Unikernel Implementations

The Unikernel design is a concept implemented by a range of projects with different focuses. This chapter will give an overview on the currently available implementations and explore some of the implementations in depth highlighting the interesting features distinguishing them from general purpose operating systems.

To this date there are the following projects implementing the Unikernel concept:

Project	Version	Description	State
MirageOS (69)	3.0	A Unikernel developed in the functional language OCaml designed to run in the cloud. see 2.4.1.2	active

OSv (78)	0.24	OSv is one of the more advanced Unikernel projects. I supports a wide range of programming languages and execution environments (including Java) by being POSIX compatible. see 2.4.1.1	active
Rumprun (89)	08/2017	A project based on the NetBSD Rump kernel. Like OSv, this Unikernel aims to be POSIX compatible. see 2.4.1.3	active
HalVM (39)	2.4.0	The Haswell Lightweight Virtual Machine (HalVM) is an Unikernel project designed to run Haskell code on top of a Xen hypervisor.	active
ClickOS (23)	2.0.1	A modular Unikernel for network function virtualization (NFV).	inactive - merged into MiniOS
Drawbridge (66)	-	A research prototype by Microsoft combining a picoprocess with the concept of a library OS based on Windows. Code is not available.	inactive
IncludeOS (49)	0.10.0	A project that allows to compile c++ programs into Unikernels by simply including the library in the sourcecode. It is designed to run on top of a hypervisor in the cloud.	active
LING (35)	0	LING allows to run an erlang application on top of a XEN hypervisor.	active
runtime.js (90)	0.2.14	A library OS running a JavaScript application based on the V8 JavaScript engine running on a KVM hypervisior.	active
Arrakis (11)	-	A research prototype developed to investigate if the influence of the kernel in the execution of an apllication can be reduced and thus speedup the application.	inactive - merged with Barrelfish OS
Clive (24)	-	A research prototype written in go. It aims to compile applications written in go into a Unikernel and run it on a hypervisor or baremetal.	active

2.4.1.1 OSv

The OSv project (78) describes itself as an operating system for the cloud. It is designed to be run on a hypervisor and does not include drivers needed to run on bare metal. It runs the kernel and multiple threads all in the same address space and thereby makes system calls as efficient as function calls. OSv aims to be able to run unchanged POSIX compliant Linux ELF binaries and therefore provides most of the standard POSIX

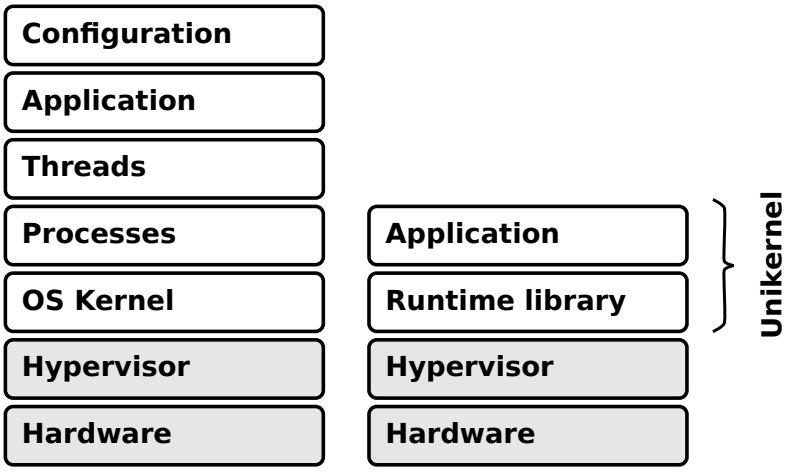

Figure 2.1: Structure of traditional OS and a Unikernel (from (16))

interface (see 2.2.6) to the application. One crucial difference to a standard POSIX compliant system is that OSv is designed to be a single process system and does not support multiple processes running in parallel, therefore the POSIX APIs initiating new processes as fork() and exec() are not supported. The same applies for signals that are partially supported in order to support the execution of a Java Virtual Machine (JVM) on the OSv system since one major goal of the project is to support JVM applications (86).

Traditional general purpose operating systems take a lot of effort to make sure that processes running in parallel are separated from one another and one cannot interfere with the memory of the other. The same applies to the separation of user-land processes and kernel-land functions. When a process in a traditional operating system like Linux tries to interact with the hardware, it issues a system call to the kernel. Since the kernel runs on a different ring (ring 0) (see 2.2.2) a privilege transition has to occur where the control is passed to the kernel function, the command on the hardware is executed and the result is passed back to the process in user-mode (in Linux running on ring 3). Since OSv Unikernel runs a single application in a single process, there is no need to separate processes from one another or the user-mode processes from kernel-mode functions. The complete Unikernel is seen as one single application either working as expected or failing. If the application does not function properly there is no need for the underlying operating system kernel to continue to function, therefore there is no need to protect the kernel from user-process behavior. OSv has implemented a range of performance improvements targeted at services running in the cloud.

System calls in traditional operating systems produce a significant overhead for the operating system through context switches and parameter copying (see chapter 2.2.3). In OSv all the expected syscall APIs are available to the application but are implemented as normal function calls. This improves the performance of the OSv Unikernel by reducing

the overhead of a system call to the costs of a normal function call without address page swapping (56).

The OSv project rewrote the network stack with a network channels based design. A channel is a single producer, single consumer queue forwarding the packets to the application. The network channels approach leads to a reduction of required locks on the resources: Socket receive buffer lock and -send buffer lock are now populated by the same thread either running the syscall recv() or send(). The interleave prevention lock has been replaced by a waiting queue using the socket lock for synchronization. The TCP layer lock has been merged with the socket lock since TCP processing now always happens within a socket call (56).

OSv has implemented a custom thread scheduler that is described as lock-free, preemptive, tick-less, fair, scalable and efficient. Where traditional scheduling mechanisms use spinlocks for synchronizing threads (see 2.2.7), OSv avoids them by introducing runnable queues for each CPU. While the use of spinlocks is unproblematic when the code runs on a physical hardware, it gets problematic when the CPUs are virtual and the code runs on top of a hypervisor. The problem arises when the vCPU is preempted in the guest system while holding the lock. The other vCPUs of the same guest system have to wait until the CPU holding the lock is executed again and is freeing the lock. OSv addresses the spinlock problem through runnable queues and by implementing a lock-free algorithm for resource access (56). In order to ensure fairness for threads in the queues a load-balancer thread runs 10 times in a second to reassign threads to runnable queues if one queue has more waiting threads than another. All threads can be preempted, there is no difference between an application and kernel thread. A thread can temporarily avoid preemption by increasing the per-thread preempt-disable counter. Classic operating systems use periodic timer interrupts (ticks) to cause a reschedule. The scheduler accounts the amount of time for each thread and uses these counts to determine which thread to schedule next. Especially on hypervisors these frequent timer interrupts waste CPU time since interrupts lead to hypervisor exits and costs valuable CPU time. OSv uses a tick-less design by using a high-resolution clock to account the actual time each thread used, instead of approximating it by ticks. Fairness in scheduling is achieved by calculating a moving average over the recent history of the runtime of each thread. The thread with the lowest average is scheduled next (56).

OSv uses a single virtual address space for all threads, kernel- or userspace threads. As described in chapter 2.2.4 and 2.2.5 this reduces the costs of context switches for threads since the operating system does not need to load page tables and flush the TLB (Translation Lookaside Buffer) and needs to store less state since all code runs in the same process.

OSv describes itself as the operating system for the cloud and therefore seeks to support a number of popular programming languages and frameworks. One specific goal is to be optimized for the execution of Java applications on a JVM in the OSv Unikernel by providing optimized APIs for memory usage and garbage collection. OSv supports

memory mapping via the mmap API to specifically support JVM based applications, like Apache Cassandra, managing their own memory over JNI.

While the project supports a large set of POSIX APIs (see chapter 2.2.6) and thereby aims at being able to run unchanged POSIX compatible applications, it introduced new APIs for memory management and networking deriving advantages from the reduced complexity in the kernel functionalities by not needing to support multi-process and multi-user functionalities. The socket API on a Linux system has significant overhead in the handling of packets resulting from not being able to share a buffer between kernel- and userspace. When data is read from or written to a socket, the buffer has to be copied between kernel- and userspace. The single-addressspace design of OSv allows to develop zero-copy APIs and share a common buffer without copying. This even allows the OSv kernel to expose the Virtio rings (see chapter 2.3.3) to the application and thereby further improve the performance of the network stack. It was demonstrated in (56) that this non-POSIX API can lead to a 4-fold increase in packet throughput for a Memcached implementation.

2.4.1.2 MirageOS

MirageOS (69) was one of the earlier implementations of the Unikernel concept. It was developed by Madvahapeddy and is written in the high-level functional language OCaml.

The goal of MirageOS is to restructure the functionalities that virtual appliances provide into modular, flexible and secure components. To this end the MirageOS project takes the core functionalities of a standard kernel and surrounding drivers and implements them into libraries written in the type-safe high-level language OCaml (63). The approach of MirageOS for the implementation of larger applications with multiple services that traditionally would run in a single virtual machine is to split it up into multiple small Unikernels running on a hypervisor and interacting with each other over the network interface.

In contrast to other implementations of the Unikernel concept like OSv or Rumprun, MirageOS does not support porting existing applications directly into MirageOS since it does not provide the standard POSIX interface needed for Linux binaries to interact with the system. The project aims to optimize for execution speed and safety of applications executed on a hypervisor. To achieve this goal, the project has rewritten libraries for handling TCP/IP connections in the typesafe language Ocaml and could show that they were able to improve the network performance of MirageOS by that. A traditional operating system has protection mechanisms on several levels. It aims to protect kernel functionalities from rogue user processes by using different privilege levels and a clear separation between kernel-space and user-space. MirageOS does not differentiate between the both since the only purpose of running a Unikernel is to execute the one application. If the application is no longer functioning correctly through a misbehaving process, there is no purpose for the rest of the system to continue to work. MirageOS Unikernels therefore have only one single address space where all code is executed.

MirageOS provides a collection of libraries for the developer ranging from block device drivers and network drivers to more specific ones as drivers for key-value stores. Since the system cannot rely on additional system services a normal Linux system would provide through separate processes running in parallel, the libraries have to include implementations of popular protocols like TCP/IP, DNS and TLS for communication with other services over the network. One property contributing to the boottime advantage compared to traditional operating systems is the missing configuration phase during the boot process. MirageOS Unikernels do not read configuration files during bootup, instead the configuration is compiled into the Unikernel during the build phase (62).

When a developer writes an application for MirageOS, the code will be written in the OCaml language and declares the needed libraries for the compilation. When the developer starts the building process, the compilation target for the code is specified. The building system for MirageOS can produce Unikernels for XEN, UKVM or Virtio and includes only the necessary drivers for the chosen target environment. During this process the Unikernel is linked against a minimal runtime, providing garbage collection and boot support and the device drivers for the defined compilation target and thus a Unikernel specific for that target environment is compiled. The Unikernel does not include capabilities for dynamic linking at runtime since all configuration is resolved at compiletime and compiled into the resulting system image. The developer can choose the compile target “Unix” and the building system compiles into a POSIX compliant Linux binary that is linked against the Linux device drivers. This facilitates testing on the developer system and is often used during the development process (68).

MirageOS Unikernels are treated as immutable objects. Since all configuration takes place at compile time, a reconfiguration of an existing system image is not possible. In this case a new Unikernel with the new configuration is compiled and replaces the existing system image.

2.4.1.3 Rumprun

The Rumprun Unikernel (89) is an operating system framework built on top of NetBSD Rump kernel drivers to compile applications into a Unikernel running on bare metal or a hypervisor. It uses Rumps modular drivers to build an application together with additional platform specific operating system functionalities into a minimal system image (53).

The Rump kernel is a kernel that is designed as an “anykernel” through componentizing its functionalities, meaning it can compile the needed drivers into the monolithic kernel or run the drivers in userspace on top of a lightweight Rump kernel (and thereby function as a microkernel) (54). The term “anykernel” refers to a kernel type codebase from which drivers can be extracted and integrated into any operating system model (53).

The Rump kernel provides a way to run drivers outside the kernel in user space. The goal is to provide drivers as a set of libraries for an application. An important differentiation to a traditional operating system are the functionalities it does not provide. It does not

include virtual memory, a scheduler, thread handling, lock management and interrupt handling. The Rump kernel expects the platform it runs on to provide these functionalities and thereby the Rump kernel alone is not a complete kernel. The Rump kernel depends on a slim hypercall layer to provide these functionalities to the Rump kernel. Hypercalls are the interfaces the CPU specific code provides to interact with the Rump kernel and lets the kernel interact with the hardware. The platform could either be a regular operating system like Linux or NetBSD where the Rump kernel runs as a user space process or the platform could be a system on top of a XEN hypervisor providing the hypercall interface. A number of other microkernel projects like Genode use the drivers from the Rump project in their projects to interact with the hardware (26).

The Rump kernel and its modularized drivers are the basis for the Rumprun Unikernel. Since the Rump kernel alone is not a complete kernel through the lack of forementioned operating system functionalities, the Rumprun framework provides the missing environment and operating system functionalities the Rump kernel needs through a so called bare metal kernel (bmk). The project includes several platform specific bare metal kernels for platforms like XEN or KVM. Figure 2.2 shows the internal structure as layers of a Rumprun Unikernel and the abstraction it uses to provide a POSIX interface to the running application. It includes a unmodified libc library from NetBSD allowing the applications running in the Rumprun kernel to use the same POSIX syscalls as on other POSIX compatible systems like Linux or NetBSD. Where in other Unikernels the applications need to be specifically developed for this Unikernel (for example MirageOS 2.4.1.2), because the drivers are provided as libraries with newly designed APIs, an application in Rumprun does not need to be tailored for the system and can be compiled without changes in the sourcecode (53).

Rumprun Unikernels are always crosscompiled on a different system. The building system provides custom wrappers for the appropriate toolchain. The application is compiled for the target platform together with the Rump drivers and bare metal kernel into a bootable system image through a “bake” process (103).

2.4.2 Common Properties of Unikernels

The previous chapter described three of the most advanced Unikernel projects. Each of these projects has a different focus. OSv lays its focus on becoming the operating system for the cloud and therefore aims at supporting a wide range of frameworks and languages. MirageOS on the other hand explores the advantages of writing operating system functionalities in a typesafe language and requires the applications running in MirageOS to be written in this language. Rumprun builds upon an existing wide ranging repository of drivers which allows them to be able to execute on bare metal. All of these projects, as different as they may be, have as Unikernels common properties. This chapter explores these properties and lays out differences in the implementations where they arise.

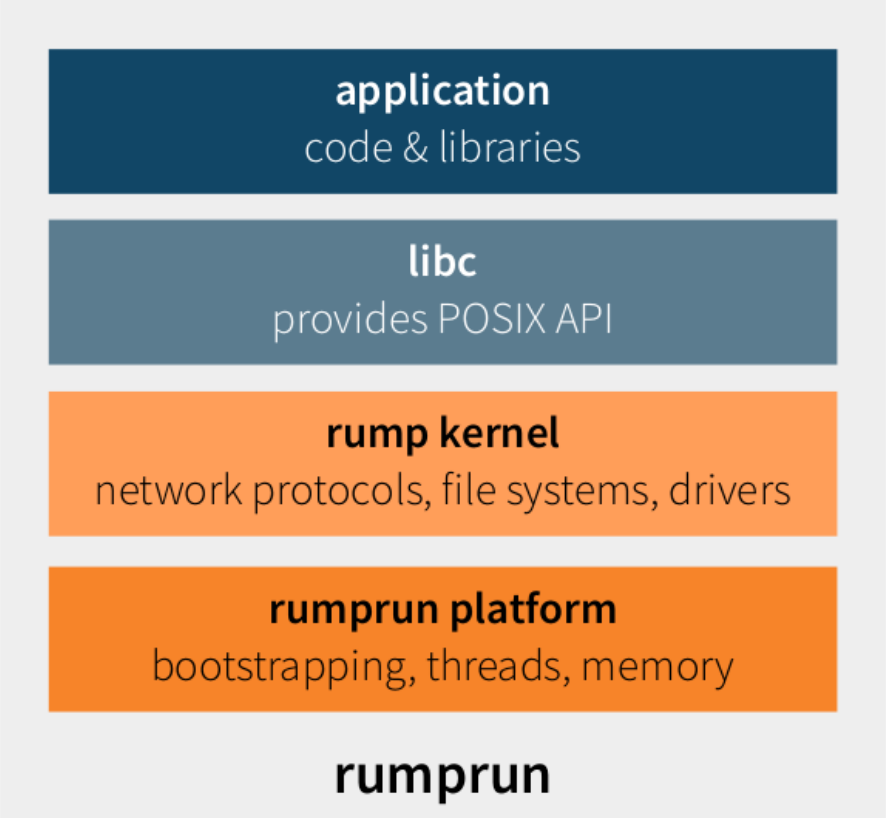

Figure 2.2: Layout of a Rumprun Unikernel (from (103))

2.4.2.1 Single virtual address space

One property all Unikernel implementations have in common is the lacking separation between kernel- and userspace memory. A traditional operating system will go over great lengths to protect the core system from the users and different processes from each other. It has clear protection boundaries and trust zones. If a program needs to access a hardware resource, it has to ask the kernel for the resource by the way of a syscall. In that way the kernel is the gatekeeper managing access to the hardware and grants or denies access. Through this strict separation the operating system makes sure that a rogue program cannot compromise other processes and bring the system to a hold. A Unikernel by contrast is seen as one single unit that either functions together or fails together. From the viewpoint of a Unikernel there is no need for the operating system to continue to function if the application fails since there is no separation between the kernel and the application and therefore if the application is terminated the system-kernel has no point of continuing the work since there are no other tasks for it to fulfill.

Unikernels use a single virtual address space for the execution of the application and all threads concerned with managing the system (63) (16). As described in chapter 2.2.4

during a context switch from one process to another, the fast cache for memory addresses TLB is flushed and needs to be rebuilt for the newly loaded process. These TLB misses are costly and reduce the executions speed of a process. In a Unikernel with a single virtual address space the TLB never needs to be flushed, which leads to the claims by several Unikernel projects that context switches in Unikernels are cheaper and therefore Unikernels have performance advantages over traditional systems.

2.4.2.2 No privilege separation

The threat model of traditional operating systems knows different untrusting parts inside an operating system that need to be protected from one another. Userspace processes running side by side cannot trust one another and therefore each need to be protected through mechanisms like virtual memory and privilege separation. On systems like Linux each process gets its own virtual memory (see 2.2.4). This prohibits a process from accessing and interfering with the memory of other processes. The separation through address translation and page tables provides clear boundaries between multiple concurrent processes running on the same CPU.

In the Unikernel world the system is seen as a inseperable unit that cannot be divided into separate layers of operating system and application. Unikernels run in one virtual address space 2.4.2.1 and all instructions run on ring 0 (62). This reduces the overhead of a syscall on a standard Linux system to a normal function call within the same address space.

2.4.2.3 Single process

Unikernels execute all running code in a single process (15). This improves the performance through the lack of costly context switches between processes (switching between threads is less costly since fewer data needs to be saved and restored (60)). The context switch between two concurrently running processes is the costly transition from the execution of one process to the execution of another by saving the state of the former process an load the previously saved state of the later. By avoiding context switches in Unikernels, the page table does not need to be switched and the TLB (Translation Lookaside Buffer) does not need to be flushed and rebuilt.

2.4.2.4 Optimized for Hypervisor

The Linux operating system aims at being able to run any software and to be able to run on any hardware. This is achieved through a large number of drivers and libraries provided and delivered with the system image. The large number of drivers increase the size of the system image and makes it harder to distribute a new system image over the network. Earlier approaches at implementing the library operating system concept as the projects Nexus and Nemesis struggled with keeping up with the demands for drivers for new hardware. Whenever new hardware was released, new drivers needed to be developed.

Since the dawn of virtualization on commodity hardware this reduced the requirement for new drivers for each individual operating system. The hypervisor presents to the guest system an abstracted view on the hardware and therefore the guest systems need to support only the drivers needed for the hypervisor. Unikernels for the most part are developed to be executed on top of a hypervisor. The hypervisor provides a standardized interface to the Unikernel such that the system image only needs to include a small number of drivers for the interaction with the hypervisor. The specific drivers are determined, configured and included at compiletime into the Unikernel.

2.4.3 Claims of Unikernels

The Unikernel community claims performance advantages over traditional operating systems through the reduction of protection mechanisms between processes, faster APIs and whole system optimizations and thereby reducing system startup time and resource usage.

The claims in detail:

Faster boot time: Unikernel projects aim at being able to boot quickly. This is achieved though a reduction of the amount of code running at boottime and configuration being done at compiletime. Claim for OSv in (56), for Rumprun in (53) and MirageOS in (62).

Smaller image size: The size of the system image is relevant when the image has to be transfered to the system over the network, the bigger the image the more load is produced. Claims of the Unikernel projects regarding smaller image sizes can be found in (62) (MirageOS), (56) (OSv) and (53) (Rumprun).

Increased performance: By getting rid of the protection boundaries between kernel- and userspace and using a single virtual address space the Unikernel projects claim to have improved the systems performance. OSv in (56), MirageOS in (62) (63).

Security: Security plays an important role in computing environments handling valuable data. Operating systems go over great lengths to address security issues. Unikernels address security primarily through a reduction of the amount of code present in a Unikernel. Security in OSv is addressed in (56), for MirageOS in (62).

These claims make the Unikernel principle a potential solution to the challenges of running virtual machines on a hypervisor on top of an IoT device.

2.4.4 Unikernels on ARM

IoT devices are often embedded systems with low system resources fulfilling a small set of specific tasks. In order to reduce the power consumption of the system the device manufacturers often use low power CPUs based on the ARM architecture for the IoT

devices. These CPUs operate on an instruction set different from the x86 architecture dominant in the desktop- and server world and therefore the software running on these systems needs to be adapted for the ARM architecture to be able to run it.
This work has chosen the two projects OSv and Rumprun as examples for the Unikernel principle because they are developed relatively far and are designed to be POSIX compatible. This means that a binary has the same interface to talk to the system as on a regular Linux. This allows for a comparison of performance metrics between Unikernels and a standard Linux system. Both Unikernel projects (89) describe themselves as being able to compile for several architectures including ARM (53). Therefore, the first idea was to use an ARM based device like the Raspberry Pi v3 as a model for an IoT device, run the tests on the device and demonstrate the deployment of software to this device through the deployment prototype described in 6. The Raspberry Pi v3 seemed to be a good fit for this purpose through the use of an ARMv8 CPU (9) supporting hardware virtualization with a dedicated instruction set for memory management and interrupt handling in virtual machines. During the compilation attempts it turned out that the the projects were not as advanced on the ARM architecture as they are for their x86 codebase.

While the Rumprun system itself could be crosscompiled for the ARM64 architecture, when it came to the compilation of an application into a Rumprun compatible binary, the compiler produced a binary that got rejected by the next step of the creation of a Rumprun image “rumprun-bake”.
Both projects state that the ARM port is an ongoing effort and while they were able to demonstrate the possibility for running the respective Unikernel on an ARM CPU for some chosen appliances, the codebase is not yet ready to compile a suitable Unikernel for this architecture.

2.5 Internet of Things

The idea behind the Internet of Things (IoT) is that physical objects of our daily lives get connected to Internet over a local network and thereby increase their utility to the user and become “smart”. An example for such devices is an integrated temperature control system in a house with a number of sensors delivering data to a control device regulating the heating and ventilation system according to a defined temperature goal.
An ever increasing number of devices in people’s homes get connected to the Internet for additional services. Smart refrigerators can transfer a live video feed to the user over the Internet to inform him about the products present at home. A smart coffee maker can receive commands to produce coffee from remote, the home security system transfers alerts and video feeds to the user by sending data from the home to a webservice. The TV receives movies over video streams directly from Youtube and smart speaker systems like Google Home or Amazons Echo have a bidirectional connection to the cloud. They are receive spoken commands from the user, decode them in the cloud and can stream music or receive information from the Internet. While these services so far have been voluntary choices by users, through the rollout of smart metering systems for the electricity grid

IoT devices become unavoidable (2).
IoT devices are not just present in homes but are omnipresent in the corporate world as well. Access systems process authentication information and unlock doors accordingly. In the manufacturing industry where the term is tightly connected to the buzzword Industry 4.0, IoT devices are used to collect measurement data and initiate actions as part of the manufacturing process or the surrounding environment. The idea here is to tightly interconnect the systems such that the overall processes can be optimized and shortages and overstockage can be avoided.
The integration of IoT devices in healthcare environments like hospitals makes clear that privacy and security play a crucial role in the acceptance of the added value the devices can provide.
In applications like smart agriculture and smart animal farming the goal is to supervise the whole production process, detect problems early on and automate as many tasks as possible. The large amount of data that can be collected through the widespread use of sensors allow deep insights into the production process and reveal optimization potential. This shows that IoT devices are already omnipresent in our daily lives and their number will dramatically increase over the near future as the devices get ever smaller and cheaper.

2.5.1 Hardware

There are several hardware platform available for the development of IoT devices. Several manufacturers provide boards with different computational power and power requirements like Arduino, UDOO, FriendlyARM, Intel Galileo, Raspberry PI, Gadgeteer, BeagleBone, Cubieboard, Z1, WiSense, Mulle, and T-Mote Sky (2).

IoT devices are designed to be small in size and consume less resources than a regular system. The core of each system is the CPU managing all surrounding devices. There are a range of CPUs with low power requirements on the market. One of the biggest player is the company ARM. Their ARM CPUs (10) are widespread in mobile devices like smartphones, tablets, NAS systems and IoT devices (70). The newer versions ARMv8 and ARMv9 have support for hardware virtualization build in through a set of instructions facilitating memory management and interrupt handling. The company MIPS Computer systems include in their processors support for virtualization in their newer products of the P- and I-series, M5100 and M5150 CPUs. Intel has some CPUs in their low power Atom series that support virtualization as well. Their system-on-a-chip “Quark” designed for low power consumption in the embedded world however does not support virtualization yet (51).

The Arduino project enters the market of IoT devices with their “Yùn” called project. It aims to provide a development platform for IoT devices and includes a resource efficient ATmega32u4 CPU at its core (8).

2.5.2 Operating Systems

Operating systems for IoT devices are tailored to the needs of low power devices. There are several realtime operating systems (RTOS) like Contiki in use on IoT devices. Operating systems like TinyOS, LightOS, Riot OS, Android (43) or some flavor of Linux are other candidates for the use on devices (2).

Several companies run projects aiming to define the future operating system for IoT devices. Google is developing with "Android Things "(codename Brillo) (43) a platform for embedded operating systems working with systems that have low system resources (32-64MB RAM) and supports wireless connections with Bluetooth Low Energy and Wifi. The company Canonical has a project by the name "Ubuntu Core for IoT" (18) aiming at providing an integrated platform for IoT devices. It uses Canonicals snap system to isolate applications from each other through the use of application containers and thereby tries to implement heightened security mechanisms protecting concurrently running applications from one another.

Microsoft enters the IoT market with its Windows 10 for IoT operating system (65) integrating the devices running the operating system with cloud services running in its Azure cloud. It aims to support devices running on x86 and ARM processors.

2.5.3 Network Connection and Communication Protocols

IoT devices use various technologies to connect to other devices or directly to the cloud. Apart from cable based ethernet connections, there are a range of wireless technologies available. On close proximity wireless technologies like RFID, UWB (Ultra Wide Bandwidth) or NFC are used where for larger distances systems like Bluetooth (including the near-range low power variant Bluetooth LE), Wi-Fi, Z-Wave or Zigbee suit the need (2).

There are several communication protocols in use for the communication among IoT devices and with the cloud. Depending upon the positioning of the device within the architecture of the network (sensor/actor, network edge, gateway, ...) and the processing capabilities of the devices different protocols are used. In the communication between the "Thing" and the gateway, lowlevel protocols like CAN-Bus or LonWorks are used. For the communication between the gateways and cloud services protocols like CoAP, MQTT or REST/HTTP based protocols are used (101).

2.5.3.1 CoAP Protocol

In the world of IoT devices the communication protocols present in the web-based Internet communication confront environments with constraints on the available computing power and network connectivity. Nodes often use 8-bit microcontrollers with low amounts of RAM and ROM and can have lossy network connections over low-power wireless personal area networks (6LowPANs). A high error rate in the transmission of packets and a low throughput lessens the reliability of the network connection.

To address these issues a new protocol called “Constrained Applications Protocol (CoAP)” was developed and specified in the RFC-Standard 7252 (see (94)). The protocol is designed for machine-to-machine communication in an IoT environment. The protocol adopted key concepts of the web like URI (“Unified Resourcee Identifiers”) and media types. It is based on a request/response model between application endpoints and supports built-in service discovery. The low overhead and good integration with the HTTP protocol makes the protocol a good fit for constrained environments like IoT devices. (Information in this chapter taken from (94)).

2.5.3.2 MQTT Protocol

The MQTT (Message Queue Telemetry Transport) protocol was developed by IBM and Arcom Control Systems originally for the supervision of a oil pipeline. In 2013 it was standardized at OASIS (12). In contrast to the HTTP protocol it uses a publish/subscribe model instead of a request/response model. It is event-driven and allows for messages to be pushed to the clients. A MQTT installation consists of three components: publisher, subscriber and broker. The broker sits in the middle of the architecture and passes messages between publisher and subscriber. Messages are associated with so called “topics”. Each subscriber subscribes to a certain topic with the broker and receives all messages with this topic from the broker whenever they arrive at the broker. Publishers send messages with a specific topic to the broker to be forwarded (13).

2.5.4 Update Mechanisms

IoT devices like any other computing device runs software that potentially contains bugs. When the manufacturer issues updates for the software the system needs a way to receive these updates and patching the system. IoT devices normally do not include user accessible userinterfaces that would allow a manual update through the user. When an update fails and renders the device disconnected from the network or unable to continue to execute the operating system, the IoT device needs to be accessed physically by attaching interaction devices like a screen and keyboard to try to recover the system from the fail state. This problematic recovery mechanisms makes an update on a IoT device risky.

2.5.5 Security

Due to the integration of some IoT devices with the private lifes of the users or internal business processes through collecting and processing data, IoT devices may arguably become a major target for attackers (71).

Home automation systems and network connected entertainment systems are part of the homes of many users and therefore touch the private lifes of the users. All devices that are interacting of that area therefore bring concerns regarding privacy for the users and security of the devices. When IoT devices are deployed as part of the infrastructure of a company, either as part of the manufacturing process of the building management system

(heating/cooling, light management, door access systems), the IoT systems could be a target for attackers as a stepping stone for further attacks into the companies network.

Compromised devices can pose risks to the confidentiality/privacy of the collected and transmitted data, modify traversing data (authenticity) and causing malfunction on the devices or one of the connected devices (71).

(93)

There have been a number of attacks on IoT devices mainly focused on the use of these devices as part of a botnet to attack IT systems with a DDOS attack. Details of these incidents are described in 5.1.

CHAPTER 3

State of the Art and Related Work

3.1 State of the Art

There are two prominent projects concerning the management and orchestration of IoT devices and their connection to the cloud: IBM Bluemix and Amazons AWS IoT cloud. Both provide software development kits (SDK) for the application development on the devices and focus on the communication of the devices with the cloud services and vice versa. The Unik project aims at unifying the creation and management of Unikernels regardless of the Unikernel implementation in use. It allows the developer to be agnostic of the underlying Unikernel and manages the compiled Unikernels in a Docker like repository system from where the images are distributed to the devices.

3.1.1 Unik

The project Unik (34) aims at implementing a Docker like system for creating, distributing and starting of new Unikernels in the same way as containers are managed.
Where Docker uses a Dockerfile to specify the layout of the system, Unik has a file called manifest.yaml as part of the application sourcecode where the basesystem of the future Unikernel is specifies, how the sources are compiled and which compiled binaries should be included. The basesystem is a template for the Unikernel and depends upon the language environment the application needs. There are templates for languages like Java including the needed JVM and its dependencies. With the command “unik build” the yaml file is read, the basesystem gets downloaded, the sourcecode compiled according to the manifest file and the resulting binaries get combined with the basesystem into a Unikernel. Unik supports at this point the compilation of applications into the Unikernels OSv, Rumprun, MirageOS and IncludeOS and can compile them for a number of platforms including XEN,

KVM, Amazon Webservices, Openstack, Virtualbox, vSphere and Photon Controller. It presents a frontend for the creation of Unikernels and delegates the creation to the corresponding toolkits (44).
Like the Docker engine Unik runs as a daemon on the host system in the background and is controlled by a client command line program. It manages the virtual machine images, start and stop the Unikernels and provides facilities for monitoring and logging.

3.1.2 Amazon AWS IoT

Amazon has developed an IoT platform as part of their AWS infrastructure going by the name "Amazon AWS IoT". It connects IoT devices to the cloud and allows a bidirectional connection sending collected data from the devices to the cloud and allowing the user to send instructions from the cloud to the connected actors. The infrastructure consists of device gateways controlling a number of devices and providing a secure connection to the cloud, a message broker, rule engine, identity services, a registry for the things and a shadowing service storing the last state of each device in case of their outage. The communication is secured through a x509 certificate that is managed by the registry that is part of the IoT cloud services (5).
Amazon provides a SDK (Software Development Kit) for a range of programming languages and platforms including Arduino Yún, Android, Java and iOS. The APIs allow the programmer to interact with the Amazon hosted webservices and messagebrokers to interact with a cloud based application (61). The IoT platform by Amazon does not include specifics for the device itself like an operating system beyond the SDK provided through the project. The deployment of the compiled application to the device is not part of the infrastructure.

3.2 Related Work

This chapter gives an overview over technologies and platforms used for deployments of software on IoT devices. The prerequisite for the deployment of Unikernels on hypervisors on IoT devices is the availability of hypervisors for the low-power CPUs used for the devices. Docker is the most prominent system for deployments of applications in virtualized environments via a centralized repository and is similar in the handling to the Unikernel based architecture proposed in 6. Resin.io is a project demonstrating the process of software deployments to IoT devices using the Docker system for the distribution of images and thereby demonstrates the process an application traverses from the developers machine to the repository on to the devices.

3.2.1 Virtualization on IoT devices

In their 2016 paper (72) Moratelli et al describe the hypervisor PRPL (87) for MIPS processors and discuss the benefits of using a a hypervisor on IoT devices. They argue for security improvements for the applications on IoT devices through security by separation through the use of hypervisors.

The deployment of software on IoT devices through container technologies is discussed by Celesti et al. in 2016 (20). They demonstrate the use of Docker on an IoT device as part of a cloud IoT infrastructure and use a Raspberry Pi as a model for an IoT device. The conducted experiments highlight the overhead introduced by the use of container based virtualization.

3.2.2 Docker

Docker (29) is a system for operating system level virtualization with containers. Contrary to virtual machines it does not run on top of a hypervisor, but instead uses an underlying Linux system and the features of its kernel to run a container as a separated process on the host machine (30).
Docker containers are derived from a base image providing a standard environment for the application. These base images include all features an application expects from its environment and provides the needed frameworks and libraries to the application. Similar to virtual machines running on hypervisors this clearly separates the guest environment from the host environment and allows for running different versions of services and libraries in a contained environment.
From a developers perspective this is an interesting feature for deploying software. In the past software was compiled on a developer machine having a certain version of the dependencies, surrounding services and frameworks installed. This binary then got deployed to a production machine and being executed. A common problem was that only at this point did the operations team realize that the version of the dependent libraries and services on the production machine the binary expects did not fit with the binaries expectations. Between two version of a library the API might have changed or the interaction with the library got redesigned completely. This lead to conflicts between developer- and operations teams (30).
The other problem with library version was that a production system did run more than one service at a time. With different software requiring different versions of the same library installed on the system this could lead to a conflict on the system and some trickery was required to deploy both versions side by side. Docker addresses these conflicts by specifying the required dependencies at creation of a container. The developer writes the software, specifies the layout and dependencies of the container and assembles a local container on his developer machine. Upon starting the container on the developers machine potential conflicts resulting from library version mismatches arise at that point and the developer can address them. The sourcecode and the container specification gets deployed to a build system where the exact same container gets build again, tested and deployed to the production environment. Thereby no discrepancy between library versions on the developer side and production side can arise. Since these containers normally only contain one application and its dependencies, there can not arise a conflict between two different library versions being required by two pieces of software on the same machine (73).
The crucial difference to virtual machines is the kernel. Where virtual machines include a complete operating system inside the vm including a separate kernel, the Docker system

uses the kernel of the host system for the interaction with hardware. Since the kernel of the host system is already initialized when a Docker container is started, the guest system does not need to initialize the hardware and thereby is very fast at booting up the application running in the container. Docker uses two features from the Linux kernel to separate the process running the container from the other operating system processes: cgroups and namespaces. Cgroups are a feature implemented in the kernel of the host system allowing to specify the amount of system resources a process can access.

In 2017 a paper was published comparing the performance of Docker containers against a bare metal deployment on SBCs (Single board computers) that could be the base of an IoT edge device. The paper compared five devices including two generations of the popular Raspberry Pi and three Odroid devices. All devices use a ARM v7 or V8 CPU at their core. The study found that the use of container technologies on the devices bring only a negligible performance penalty when compared to the bare metal performance and the container activation time is relatively small even when the SBC is overloaded (70).

3.2.3 Resin.io

Resin.io (88) is a platform for software deployments on IoT devices. It compiles the application into containers and relies on Docker 3.2.2 for this task. The platform consists of three components: client, server and device. The developer specifies through a Dockerfile the layout and build instructions for the system and after checking in the code the build system creates the container accordingly.

The process from development to application deployment begins at the developer machine. The developer writes the application code and pushes it into a git repository. The resin.io builder system receives the code from the repository and builds it according to the instructions for a specified platform. The builder are capable to build for several IoT specific platforms. The result of the build process is a Docker image. An agent running on the IoT device gets aware of the newly created image and downloads it to the device.

3.2.4 Amazon Greengrass

With the product "Greengrass"(4) Amazon aims at integrating IoT devices with their AWS cloud infrastructure but taking into account that the devices need to continue to function without an Internet connection. It is an extension to their existing product AWS IoT (see 3.1.2 for details) and extends the functionalities through an edge gateway connected to the IoT devices. Greengrass allows the devices to execute AWS Lambda functions using the same programming model on the device like in the cloud even if the devices are not connected to the Internet (4). AWS Lambda is a product by Amazon that follows the serverless-computing paradigm. Code is stored on the server side and only executed when triggered by a request. Contrary to classic server applications Lambdas do not need a user owned server constantly running and waiting for incoming requests. The users of the service are billed by the amount of calls processed instead of per-VM basis. The system extends the AWS IoT SDK powered devices through local edge devices

called “Greengrass Cores” that play the role of a hub connecting all devices of a local network that are combined into so called “Greengrass groups”. Compared to the previous solutions based only on AWS IoT, Greengrass reduces the latency between devices and data processing layer by shifting functionalities from the cloud to the local Greengrass Cores. At the same time it reduces bandwidth costs by reducing the amount of data that needs to be transfered between the local network and the cloud service (74).

CHAPTER 4

Performance & Footprint - Evaluation of Unikernels

One of the advantages all Unikernel implementations claim to have, is improved performance compared to a standard Linux system (for the claims see 2.4.3). This chapter defines metrics relevant in the context of IoT devices based on the claims of Unikernels and evaluates the performance of Unikernels in seven relevant categories. Of all the Unikernel projects currently available only two (Rumprun 2.4.1.3 and OSv 2.4.1.1) are POSIX compatible 2.2.6 and therefore can run unchanged Linux applications. All other Unikernels require the application to be written specifically for this system and a specific programming language and, therefore, are not directly comparable with an application running on a Linux system.

4.1 Evaluation Framework

To compare Unikernel implementations with a standard Linux, one first needs to define a framework of relevant metrics for the context of IoT devices. IoT devices are by definition connected to a network and almost all interaction is done over some network protocol. Thereby the performance of the network stack is particular relevant in order to compare performance characteristics of Unikernels with a standard Linux system. The tests for the network stack include low level tests for the processing of TCP and UDP connections (4.1.2) and tests for higher level protocols like the HTTP-protocol (4.1.3), CoAP protocol (4.1.4) and the MQTT protocol (4.1.5).

The tests for the low level protocols described above take an isolated view on the protocol implementations within the Unikernels and therefore allow for making statements about the processing of requests by a specific part of the system. In a real world application the ability of a system to respond to an incoming request is the result of several components

of the system working together. The response by the protocol handler can only be sent after e.g. data is received from a the database, values calculated and transformed into the requested format. To test the capabilities of Unikernels in a real world scenario, an IoT-application simulating the collection and storage of sensor data was developed in two prominent languages (4.1.6).

One advantage of hypervisor based architectures is that a new instance of a system can be started when needed in order to address an increased number of incoming requests (called scaling out). To be able to react in a timely manner to the rising demand, the boot time of a system becomes relevant. Therefore, the evaluation framework includes the boot time as a metric for the performance of the Unikernels.

When a new version of a virtual machine should be executed on top of a hypervisor, the image file of the VM first needs to be downloaded to the device over the network connection. If several IoT devices download an image over the network at the same time, this produces a heavy load on the network and can influence the throughput of the data on the network. This makes the size (see 4.1.7) of the operating system image a relevant metric since smaller images produce less load on the network.

4.1.1 Boot Time

During the boot process the system queries the available hardware, loads the appropriate drivers and initializes the devices. A standard Linux system starts a number of background services for different maintenance tasks. This process takes time. In an environment with varying workloads a strategy to cope with a rising number of incoming requests could be to spin up new virtual machines on the fly and shut them down as soon as they are no longer needed. Fast boot times allow to spin up new instances on demand and therefore allows for a flexible handling of workload spikes.

The time measured as “boot time” in this test series is defined as the time difference between issuing the command to start the virtual machine or Unikernel and the time where the service running inside the VM is starting to respond to incoming requests. To this end a script was written taking time x, starting the system and immediately start querying for the service. As soon as an answer is received, the time y is taken again and the difference between the two values (y-x) is stored as boot time. The boot time thereby not only includes the time it takes a system to initialize the interaction with the hypervisor, but includes the startup process of the running application as well.

4.1.2 TCP/UDP Connection

Both, TCP and UDP, are the fundamental protocols of modern computer networks like the Internet. They both belong to the transport layer in the protocol stack. The Transmission Control Protocol (TCP) (50) is a connection oriented protocol that can detect packet loss and retransmits the lost packages if needed. An example for a higher level protocol relying on TCP for the transportation of data is the HTTP protocol.
The User Datagram Protocol (UDP) is a so called connectionless protocol that, contrary

to the TCP protocol, does not keep track of the sent packets and does not verify with the recipient if the packet was received. UDP sends the content in datagrams to the receiver (85). A number of higher level protocols use UDP for the transportation of their data: The DNS protocol used for name resolution in a network, the DHCP protocol for assigning dynamic ip addresses in a network and a number of routing protocols to negotiate the routes a packet can take through the network (46).

Since all application level protocols (HTTP, DHCP, DNS, CoAP, MQTT) rely on the two transport layer protocols for the data delivery, the performance of the network connection fundamentally relies on the implementation of the two low level network protocols. Therefore, the two layers of network protocols get tested separately to draw conclusions on the implementations of subcomponents of the network stack.

For IoT devices the network stack plays a central role, because all communications with the device happens over the network. Therefore, the performance of the networkstack and its subcomponents is an important criteria for evaluating a Unikernel implementation for the use on a IoT device.

For the testseries evaluating the transport layer protocols (TCP and UDP) the tool Netperf (76) is used. It consists of two executables (netserver and netperf) and is designed as a client-server model where the server (called netserver) is executed on the system under test and the client (netperf) is executed on a second system running the tests. Netperf allows to test different test profiles and thereby measuring certain characteristics of an TCP and UDP connection. The following description of the testprofiles used during the testseries is taken from (52).

TCP_RR The TCP request-response test executes one request at a time synchronously. The resulting measurement expresses the average number of completed transactions per second.

TCP_STREAM In this test some quantity of data is sent from the client to the server and measured on the server side. The initialization of the connection does not count to the measurement. The measuring unit of the result is megabit per second.

UDP_STREAM During the UDP_STREAM test the client sends data to the server. Since the UDP protocol has no end-to-end control flow the netperf tool has no way of knowing from the protocol if the data was received. To mitigate this netperf shows the sending and receiving throughput.

UDP_RR A UDP request-response test is similar to the TCP_RR test in the sense that data is sent in both directions. The difference is found in the characteristic of the UDP protocol that is a connectionless protocol meaning that it does not make sure that the data is actually received by the receiving system. If a UDP datagram is lost it is not retransmitted as with the TCP protocol.

TCP_MAERTS This testprofile is the reverse of the TCP_STREAM test. Instead of streaming from the client to the netserver application running in the system under test, it streams from the netserver to the client and thereby testing the capabilities of the system to send TCP packets. The measured result represents the average number of megabits successfully transfered during the test.

There are more testprofiles available in the netperf testsuit, but the above listing contains the most common and useful for the intended testscenario. Additional profiles cover other protocols or operating system specific functions.

4.1.3 HTTP Connection

The preferred way to interact with a data providing service these days is via a REST interface. Theses interfaces communicate over the HTTP protocol (36) and mainly deliver a JSON file as result.

In the context of an IoT device this could be a service aggregating data from different sensors and stores it in a local database. Other centralized services could then fetch the data from this service over the REST interface.
The performance metric for the HTTP connection is different from the tests for the TCP stack since the TCP tests only measure the performance of the networking components and the implementation of the low level protocols where the HTTP response times are the result of components involved in the handling of higher level protocols. Isolated tests for the handling of the HTTP protocol allows to derive additional conclusions on top of the results for the TCP tests. Where the results for the TCP connection mainly involves the handling of the network drivers implemented in the Unikernel projects, the tests for the HTTP protocol additionally include the communication between the networking component with the Java-VM and the handling of the HTTP-requests inside the VM.

The tool used to test the HTTP connection of Unikernels and the reference system is httperf (45). It collects a number of data points for a testrun and calculates average values for metrics as reply time. For this testseries two related metrics were chosen allowing to make a statement about the ability of the system to process incoming requests and define the basis for comparing the performance of the tested systems.

- Reply time: The reply time of a single request is the time it takes from sending a request to receiving an answer. The tool calculates the average value of all requests sent during a testrun and prints it after the test. The measurement unit for this metric is milliseconds (ms).
- Request rate: The value measured as request rate is the average number of requests the server can process in a second. This is highly correlated with the reply time in the sense that the faster a single request can be replied, the more requests can be processed in a second. The unit for this measurement is requests per second (req/s).

This testseries was conducted with different system configurations (see 4.1) to see if differences in the amount of RAM or number of vCPUs have an impact on the measured network performance. For each system configuration five tests were executed and the average of these values was calculated. The tool httperf allows to specify the number of connections that are established during a single testrun. For all tests with the tool httperf the fixed number of 3000 connections was chosen as it is a reasonably high number to get a good average value over all connections but is low enough for the slower systems to complete the test in time.

4.1.4 CoAP Protocol

The CoAP protocol is a machine-to-machine communication protocol designed for constrained environments (see 2.5.3.1 for details).

The client-server architecture allows to design a test for the performance of Unikernels in the handing of CoAP requests where a simple server is implemented through a library and compiled into a Unikernel. A client benchmark-application then queries the server repeatedly and measures the time it takes the server to complete the request. An important distinction from the MQTT protocol is the use of the underlying transport layer protocol. Here CoAP uses the connectionless UDP protocol instead of TCP. The project "Californium" (37) implements a library for the handling of CoAP requests. Next to the library itself they have implemented a range of tools to test the installation. One of the tools is a simple benchmark application (client and server) written in Java called "TcpThroughput" allowing to measure the throughput between client and server. During the tests the server application was packaged into a JAR archive and compiled into both Unikernels OSv and Rumprun.
The fact that CoAP is widely used in IoT projects and is a more lightweight protocol than HTTP makes it an interesting candidate for testing the network capabilities of Unikernels. Therefore, the CoAP tests were included in the evaluation framework to get a comprehensive view on the different aspects of the networkstack implemented in the Unikernel projects.

4.1.5 MQTT Broker

Like the CoAP protocol, the MQTT protocol is a protocol for constrained environments with low network reliability (see 2.5.3.2 for details). The architecture of a MQTT based system is different from CoAP in that it has a broker in the middle between the client nodes.

The MQTT protocol is widely used for the communication and integration of low-power devices into a larger network and therefore the performance of Unikernels in the processing of MQTT messages becomes relevant.

In an IoT environment a Unikernel could be a sensor or actor interacting with a MQTT broker or a Unikernel could be used to host the MQTT broker for other clients. In a

scenario where the Unikernel is the broker, the throughput capabilities of the Unikernel becomes relevant to the performance of the overall system.

One of the most prominent implementations of a MQTT broker is the Mosquitto project (33) by the Eclipse Foundation. Therefore, this broker was chosen for the throughput-capability tests.

4.1.6 Real World Application

The tests for the networking components specified before all try to take an isolated view on their component. The transport layer protocols (OSI layer 4) TCP and UDP lay the groundwork for the higher level protocols to work. So the results for the higher level protocols need to take into account the results for the lower level protocols since they build ontop of them. The tests for the higher level protocols HTTP, CoAP and MQTT all use lightweight implementations of these protocols to isolate the tests from interfering components as good as possible.

After the isolated tests for the network protocols the question remains how Unikernels would handle a real world application consisting of more than just a simple response server and is implemented on top of a more resource-demanding framework. To this end an application was developed in two languages with two different frameworks. The application simulates the collection of sensor data in regular intervals, saves the received data in an in-memory relational database and provides a REST interface over the HTTP protocol to any client querying for the data. The application was implemented on the one hand in the language Java and uses the framework “spring-boot”. This application is further called “IoT-App-Java”. To rule out language or framework specific effects, the same application was reimplemented in the language JavaScript with the use of the framework Node.js. This application is further called “IoT-App-Node.js”.

The comparison of the simple handling of HTTP requests through the tests specified in 4.1.3 with the real world applications and the comparison among both implementations of the application should allow for drawing conclusions about the performance of Unikernels in the handling of more complex applications.

4.1.7 Image Size

When a new virtual machine is deployed on a system, the host system first needs to download the VM image in order to execute it. With an IoT device possibly low on system resources and restricted on the available network bandwidth this could be a significant bottleneck for the deployment of new virtual machines on a hypervisor running on an IoT device.
The system images of Unikernels are much smaller through the radical reduction of drivers, tools and services needed for the task at hand. A comparison of image sizes should give a good indicator if the deployment of software through a Unikernel would be feasible even if the network connection does not allow for the download of large images.

4.2 Measurements and Comparison

For the assessment of the metrics two Unikernel implementations (OSv 2.4.1.1 and Rumprun 2.4.1.3) were selected for their ability to execute applications without making changes to the sourcecode and thereby allow for a direct comparison of the performance of the same software on different systems. The OSv Unikernels were built with the current version 0.24 of the project. For Rumprun the current code version from August 2017 was used. As a reference system for the comparison a standard Ubuntu Linux was chosen in the server version 16.04.

4.2.1 Test Setup

For an accurate comparison of Unikernels against a standard Linux system, one has to ensure that each system gets the same amount of resources allocated. The virtual machines and Unikernels were started on a system with an Intel i7 5820k CPU and 32GB RAM. KVM was chosen as the hypervisor on which the systems were run. The tests were executed for different VM configurations (assignments of virtual CPUs and amount of RAM) in order to compare the behavior of the tested systems under different resource attributions (see table 4.1 for all configurations). Each test was repeated five times to get an accurate average value. The machine running the VMs and Unikernels was connected via an ethernet cable with a laptop (Intel i7 3517U) that ran the tests. On the hypervisor the VMs and Unikernels were using a bridge interface to interact with the physical ethernet device.

For the HTTP tests and boot time tests a small sample IoT application further called "IoT-App" was developed in two different languages and frameworks. One application is written in the language Java and based on the Spring-Boot framework and therefore the system running the application ("IoT-App-Java") has to include a JVM for the execution of the program. The other application ("IoT-App-Nodejs") is written in JavaScript with the currently popular event based server-side framework Node.js. Both applications fulfill the same requirements. The applications simulate the collection of temperature data that are stored in an in-memory database. It offers a REST interface to query the stored data from the database. Whenever an HTTP-GET request reaches the interface the average of all collected temperature data is calculated and delivered to the client in the form of a JSON file. This produces a load on the system so that a realistic scenario with changing data can be tested and the efficiency of the complete system is measured instead of evaluating only the network stack or the delivery of static content. Both Unikernel projects OSv and Rumprun currently only support a smaller number of languages and frameworks and the chosen ones (Java and Node.js) were some of the few that are supported by both.
It would have been interesting to see the performance of an application running directly on the CPU without a runtime environment between. Unfortunately it was not possible to compile the same application written in the language Go into a runnable Unikernel.

4.2.1.1 Resource Attributions

A hypervisor allows to attribute certain amounts of resources (number of virtual CPUs and amount of RAM) to a virtual machine. Each individual test was repeated under different resource attributions to the VMs in order to measure performance differences under certain conditions. The following configurations were used for the tests:

Setup nr.	Number of vCPUs	Amount of RAM (MB)
1.	1	512
2.	1	1024
3.	1	2048
4.	2	2048

Table 4.1: Hardware configurations used in the tests

4.3 Measurement: Boot Time

The time measured as "boot time" here is the time it takes from issuing the start command for the VM to the hypervisor until the service running in the VM (either the netserver service or the REST interface of the IoT-App) is responding to incoming requests. Three testseries were conducted for boot time measurements. One series with the netserver binary compiled into the Unikernels, a second testseries with the Java IoT-App as reference point and a third with the Node.js based application. The idea is to compare a fast booting service like the netserver application with more heavyweight applications needing to initialize more resources like the in-memory database during the boot-process and thereby differentiate between the provision time of the system on the hypervisor and the initialization of an application within the Unikernel.

4.3.1 Boot Time Netserver

The variation between different configurations of any system was very low such that the differentiation by resources can be neglected for this test. The OSv Unikernel was the fastest to boot with an average time of 2,01 seconds from initiating the start of the Unikernel on the hypervisor until the netserver application responds to queries. The Rumprun system was only slightly slower with an average boot time of 2,55 seconds. Compared to both Unikernels the standard Ubuntu Linux was over 15 times slower than the OSv Unikernel (12 times for Rumprun) with an average boot time of 32,06 seconds.

System	Average Boot time (sec)	in percent of ref. system
OSv	2,01	6,28%
Rumprun	2,55	7,97%
Ubuntu	32,06	-

Table 4.2: Boot time Measurements Netserver-series (1CPU 1GB RAM)

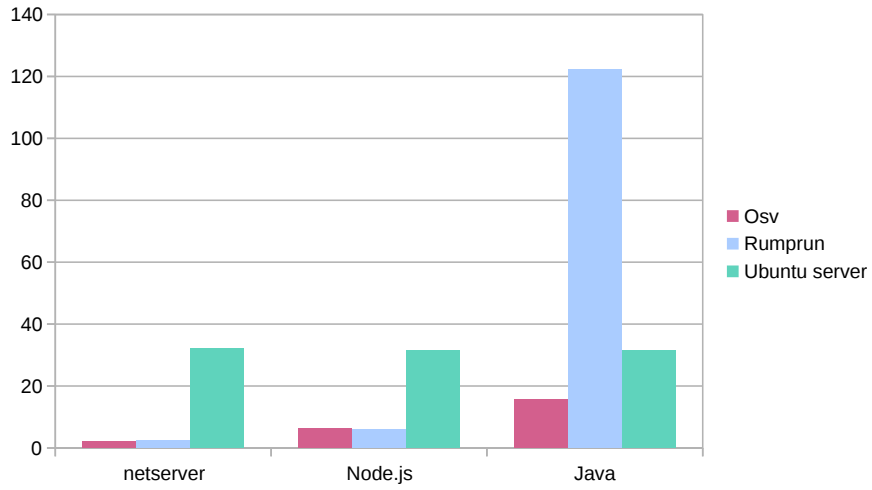

Figure 4.1: Boot time measurements (1 vCPU 1024MB RAM)

4.3.2 Boottime IoT-App-Java

The second testseries for the systems boot time lead to dramatically different results (see table 4.3). While the Ubuntu system was consistent with the boot time in the netserver test with an average boot time of 31 seconds, the results for the Unikernels are much slower and the results show differences in the configurations as well.
The OSv system reached a 15 second boot time for the lower system resources (configurations 2 and 3 in table 4.1) that dropped to 10,77 seconds for the higher configuration (configuration 4 4.1). Thereby OSv needs only 50% (35% respectively for the higher configurations) of the boot time of the Linux system. Compared to the Netserver results it shows that while the Linux system is consistent in the boot time, the Unikernel takes much more time to initialize the IoT application compared to the netserver application. The reduction of boot time for the higher configuration indicates that OSv is able to effectively use the second processor to its advantage and thereby speedup the initialization of the system.

The most eye catching result is the performance drop for the Rumprun Unikernel. It took the Unikernel on average 122 seconds to respond to the queries on the REST interface. This is four times the amount of time needed by the Ubuntu server. Contrary to the OSv measurements the attribution of more resources did not lead to a change in the measurements for Rumprun. The comparison of the netserver result for Rumprun and the IoT-App result makes clear that the Rumprun system is able to boot fast on a hypervisor but takes significantly more time initializing a heavy-weight Java application.

The explanation of the differences here between OSv and Rumprun might be the optimizations specifically for the JVM implemented in OSv (see 2.4.1.1).

CPU	RAM (MB)	System	Average Boottime (sec)	in percent of ref. system
1	1024	OSv	15,53	49,51%
1	1024	Rumprun	122,13	389,40%
1	1024	Ubuntu	31,36	-
2	2048	OSv	10,91	34,88%
2	2048	Rumprun	126,47	404,51%
2	2048	Ubuntu	31,26	-

Table 4.3: Boottime Measurements IoT-App-Java series

4.3.3 Boottime IoT-App-Node.js

The results for the Ubuntu reference system in this test series is consistent with the results in the Java-series, meaning the type of the application has no influence on the boot time on a Ubuntu system.

System	Average Boottime (sec)	in percent of ref. system
OSv	6,23	19,85%
Rumprun	5,86	18,67%
Ubuntu	31,56	-

Table 4.4: Boottime Measurements IoT-App-Node.js series (1vCPU 1GB RAM)

The boot time of the OSv Unikernel improved by 9 seconds compared to the Java-testseries resulting in only needing twenty percent of the boot time of the reference system. The improvement of the Rumprun Unikernel is even more remarkable as it only takes a 20th of the time (5,86 seconds compared to 122.13 seconds) it took to run the Java application in a Rumprun system on the same configuration (compare table 4.4). This indicates that the significantly slower boot time for the Java application is not a result of a slow initialization process with the hypervisor but has more to do with the initialization of the more heavy weight and memory intensive Java application.

Compared to the boot time of the netserver application 4.3.1 this shows an overhead of starting the Node.js environment on a OSv Unikernel of an additional 4,13 seconds adding to the boot time.

4.4 Measurement: Network Protocols

The defining characteristic of IoT devices is their connection to a network. For the communication between a IoT device and a gateway, server or other IoT device a number of protocols can be used. Current webapplications often use a REST interface over the HTTP protocol to exchange data with a service. In the resource constrained environment of IoT devices two alternative protocols have emerged that specifically address the needs of devices with few resources and varying availability on the network: MQTT and CoAP.

This chapter presents measurements of the performance of OSv and Rumprun Unikernels in the handling of low-level TCP and UDP connections, MQTT connections, the CoAP protocol as well as the HTTP protocol.

4.4.1 Performance of TCP and UDP Connections

When data is sent over a network different protocols on different layers of the network stack are involved. The two most used protocols on the transport-layer of the OSI-model are TCP and UDP. To evaluate the performance of the network connection of Unikernels this chapter takes an isolated view on the handling of those two protocols and measures and compares their performance.

4.4.1.1 Testsetup

For the measurement and evaluation of the network stack of a system, the company Hewlett Packard developed a tool called netperf (76). It consists of two components: a server replying to incoming TCP or UDP connection requests (called netserver) and a client allowing the user to execute test profiles for a certain period of time.

The two Unikernels OSv and Rumprun were compiled with the netserver binary and on the reference Linux system the binary gets started as daemon on startup.

4.4.1.2 Results

For the reference Linux system the performance under different configurations was relative consistent, meaning that an increase in the amount of RAM or number of virtual CPUs did not lead to a significant increase in throughput. The same can be said about the OSv Unikernel with the exception of a 18 percent improvement for the TCP_RR profile and a 32 percent improvement of the UDP_STREAM profile when comparing the second configuration (1GB RAM and 1 vCPU) with the first configuration (see table 4.1 for all tested configurations).

The OSv Unikernel outperformed the Linux system in four of the five categories. With a throughput of 138 percent compared to the reference system in the TCP_MAERTS (138,65) and TCP_RR(138,01) tests, a 133,88 percent performance in the UDP_RR test the Unikernel performed significantly better than the Linux system. In the UDP_STREAM test the Unikernel could only reach a slight improvement of 102,25 percent and in the TCP_STREAM category it reached 67,51 percent of the performance of the Linux system. These findings are consistent with the tests conducted by the OSv project themselves (see chapter 4.7).

The Rumprun Unikernel was not able to outperform the Linux system in any of the tested categories (see figure 4.2). While it reached in the UDP_RR test only 89 percent of the transfer rate of the reference system when started with the second configuration (1 vCPU 1GB RAM), it could at least keep up with the result of the Ubuntu system when a higher configuration (1 vCPU 2GB RAM) was used. For the TCP equivalent

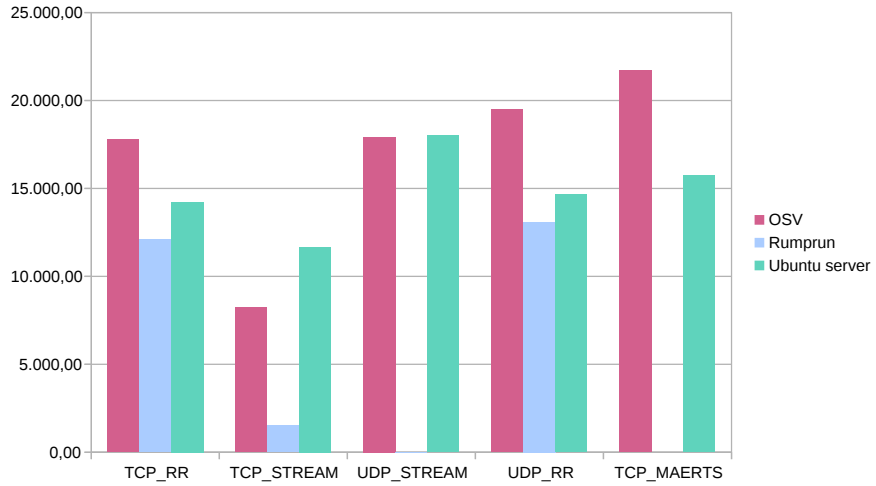

Figure 4.2: Results of the netperf testseries (req/s) (1 vCPU 1024MB RAM)

the results show a similar behavior: 85% of the transfer rate of the reference system for the lower configurations (configuration 2 in table 4.1) the results increased to 93% with more attributed RAM (configuration 3). Both streaming tests resulted in much worse performance. With an average result of 1.532 Mbit per second in the TCP_STREAM test the Unikernel only got 13 percent compared to the Ubuntu system. Even lesser was the result for the UDP_STREAM test resulting in 0,01 percent (1,82 Mbit) of the throughput the reference system reached for the same configuration (18.013 Mbit). The Unikernel failed the TCP_MAERTS test completely by not producing any results.

4.4.2 Performance of MQTT Connections

The MQTT protocol is a protocol specifically developed with IoT devices in mind. It aims at facilitating the interaction of a device with a broker distributing the data. On the transport layer of the OSI (see (46)) model it uses the TCP protocol to transport the packets on the network. An IoT device collecting data from sensors could be tasked to run a MQTT broker as an intermediary between the sensors and a cloud service.

4.4.2.1 Testsetup

For this testseries the prominent MQTT broker Mosquitto (33) was used. The broker was compiled for the OSv Unikernel with slight modifications to the build process since the binary used in the OSv image needs to be position independent (modifications taken from (80)). For the comparison of the results the Mosquitto broker was installed on a Ubuntu server virtual machine. The Rumprun build system uses a custom compiler that was not able to build the Mosquitto broker from source. Therefore, the measurement results for Rumprun are missing. To test the Mosquitto broker with requests the tool mqtt-bench (96) was used. It sends 1000 messages with 10 concurrently connected clients

and measures the time the messages need to traverse the broker. From the resulting data the throughput in requests per second are measured.

4.4.2.2 Results

The measurement results for the MQTT broker 4.5 show that the throughput is unaffected by the amount of resources attributed to the virtual machines. The throughput is consistent all throughout the measurement series for the systems OSv and Ubuntu.

CPU	RAM (MB)	System	Throughput (req/s)
1	512	OSv	25.994
1	1024	OSv	25.211
1	2048	OSv	26.800
2	2048	OSv	25.176
1	1024	Rumprun	-
1	512	Ubuntu	27.511
1	1024	Ubuntu	27.133
1	2048	Ubuntu	28.768
2	2048	Ubuntu	25.464

Table 4.5: Results of the MQTT tests with the Moquitto broker

When the results of the OSv Unikernel are compared to the counterparts from the Ubuntu system, it becomes clear that the throughput of the OSv system is only slightly less than that of the Ubuntu system. The throughput (reqests per second) stays within a range of 92-98 percent compared to the reference system.

4.4.3 Performance of CoAP Connections

The CoAP protocol is a service level protocol that is intended to be used to connect IoT devices. An interesting property differentiating CoAP from MQTT and HTTP is the underlying transport layer protocol it uses. Instead of using TCP the CoAP protocol uses the connectionless UDP protocol.

4.4.3.1 Testsetup

The project “Californium” (37) by the Eclipse Foundation has implemented the CoAP protocol as a Java library. As part of the project, benchmarking tools were developed allowing for a measurement of the throughput of a server. For the tests on the Unikernels the benchmarking tool “TcpThroughputServer” by the Californium project was compiled into a executable JAR-file together with the Californium libraries and deployed on a OSv and Rumprun Unikernel. For the performance comparison the same JAR was deployed to a Ubuntu server as a reference machine for the test. The project includes next to the server a benchmarking client “TcpThroughputClient” as counterpart for the test.

During a test the client sends 2000 requests to the server and measures the time it takes to complete. From this data the throughput is calculated. For each configuration the tests were repeated five times. The results shown in table 4.6 show the average of the five results.

4.4.3.2 Results

The results for the CoAP tests show a similar behaviour to the results from the MQTT test 4.4.2.2 in that they are not responsive to changes in the resource attributions. Neither an increase in the amount of RAM attributed to a virtual machine, nor an additional virtual CPU (vCPU) for the vm has any impact on the amount of requests the Unikernels or the Ubuntu system are able to process.

CPU	RAM (MB)	System	avg Throughput (req/s)
1	512	OSv	327,60
1	1024	OSv	332,80
1	2048	OSv	336,60
2	2048	OSv	322,80
1	512	Rumprun	164,40
1	1024	Rumprun	159,60
1	2048	Rumprun	162,80
2	2048	Rumprun	162,40
1	512	Ubuntu	337,60
1	1024	Ubuntu	339,00
1	2048	Ubuntu	346,80
2	2048	Ubuntu	332,80

Table 4.6: Results of the CoAP tests with the TcpThroughputServer

The measurements results presented in 4.6 show that the results for the OSv Unikernel is within a slim margin of the results of the reference system. The Rumprun system scored significantly worse than the reference- and OSv system (see figure 4.3). With an average throughput of around 160 requests per second it reaches only half or the throughput of both the reference system and the OSv Unikernel.

4.4.4 Performance of HTTP Connections

The HTTP protocol is the backbone of todays Internet. Every website is transported to the browser over the HTTP protocol and most webservices exchanging JSON structured data do this using the HTTP protocol. A pattern that has emerged in recent years is the use of REST-style interfaces on data providing webservices making use of the HTTP protocol to retrieve and store data and query data in a predefined structure. The clients query data by directly using the HTTP protocol instead of merely using it as a means of transportation.

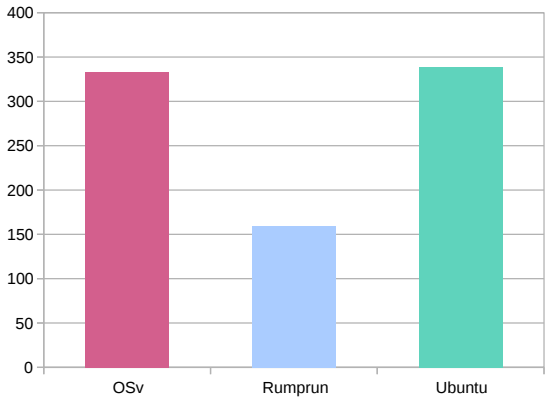

Figure 4.3: Throughput CoAP server in req/s (1 vCPU 1024MB RAM)

4.4.4.1 Testsetup

To test the performance of Unikernels in the processing of HTTP requests a simple application was written in the language Java and deployed to both Unikernels and the reference system. Java was used to ensure that the same application could be deployed to all three platforms without any changes to the code. The test application "iot-spark-static" exhibits a simple REST interface providing a static value in the JSON format. This allows for an isolated view on the processing of HTTP requests and can factor out other influences (eg. database queries) that could play a role in the performance. The framework "Spark" (97) (not Apache Spark) was used for the creation of the REST interface.
To test the performance of the REST interface the tool httperf was used executing 1000 requests and measuring the time needed for completion. The tool then calculates the request rate from the collected data.

4.4.4.2 Results

As before in the test of the CoAP protocol 4.4.3.2 and the MQTT protocol 4.4.2.2 the amount of resources attributed to a system has next to no influence on the performance measurements of the processing of the HTTP protocol. The testseries were conducted with all configurations defined in 4.2.1.1 leading to a result where changes in the configurations did not significantly impact the performance.

The results shown in figure 4.4 show no significant performance differences between the OSv Unikernel and the Ubuntu reference system. While Ubuntu slightly outperformed OSv by a slim margin for the configuration with 1 virtual CPU and 512MB RAM and OSv outperformed Ubuntu in the configuration with double the amount of RAM, both results are within a slim margin so that none of the two system can be called outperforming the other. The Rumprun results on the other hand show a completely different picture. While the results themselves show unresponsive to changes in the configuration the performance

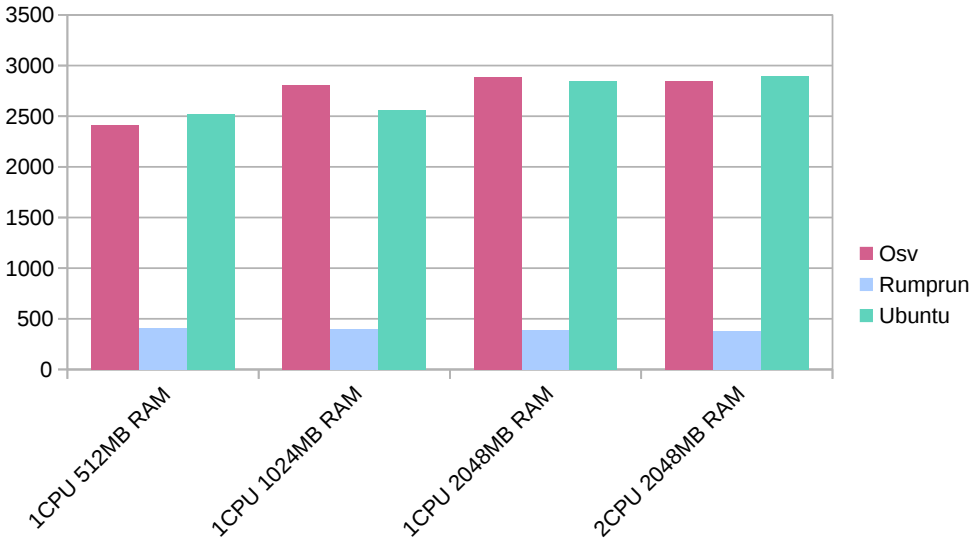

Figure 4.4: Request rate (req/s) for the HTTP protocol

measurement for the REST interface show significantly lower request rates than for OSv or Ubuntu. On average Rumprun could only score around 15 percent of the result of the reference (Ubuntu) system.

4.5 Measurement: Image Size

When a new virtual machine or Unikernel gets deployed onto a device, the machine image first has to be transfered to the device. This often happens over a network where the device downloads the system image from a server. In a scenario where the virtual machine is treated as immutable infrastructure, and therefore only the system as a whole can be updated, the size of the delivered system image gets relevant. In this scenario when software updates are necessary the existing image is not altered but instead gets thrown away and is replaced by a new image including the new software version.

System	Application	Image size in MB
Ubuntu server 16.04	Netserver	3.776
Ubuntu server 16.04	IoT-App Java	3.776
Ubuntu server 16.04	IoT-App Node.js	3.754
OSv	Netserver	29
OSv	IoT-App Java	123
OSv	IoT-App Node.js	99,1
Rumprun	Netserver	19
Rumprun	IoT-App Java	189
Rumprun	IoT-App Node.js	58

Table 4.7: Image size

4.6 Measurement: A Real-World Application

The TCP and UDP tests conducted in chapter 4.4.1 and the HTTP tests in 4.4.4 did focus on the protocol implementations of the systems and try to take an isolated view on the network stack of the system. However the overall performance of a system is influenced by more than just the network drivers and protocol handlers. To get a more holistic view on the handling of real life scenarios by the tested systems a more complex application was developed. These tests are designed to not only evaluate the performance of the network stack of the Unikernel but to take the complete system in account and thereby evaluate how the Unikernel performs in a realistic scenario.

To test the capabilities of Unikernels in processing HTTP connections the same near real-world application was developed in Java and Node.js. Both use rather heavy-weight frameworks and aim to simulate an application that could be run on a IoT device aggregating temperature data with the use of an in-memory database.

4.6.1 Java IoT-App

Java was chosen as an example because it is a common language in software development, has frameworks for a wide range of tasks and is portable to different systems. Both Unikernels OSv and Rumprun have build systems allowing the creation of Unikernels including the Java runtime environment and can run arbitrary JAR-archives.

4.6.1.1 Test setup

To test Unikernels with a real world application an application was developed in the language Java. The applications builds upon the widely used but rather heavy-weight framework “sping-boot” (81). The application simulates the collection of temperature data from sensors by creating random temperature data in regular intervals and saving the data in an in-memory database. The application provides a REST interface over the HTTP protocol to query for the saved data. When a HTTP-GET request reaches the application, the average of all stored values is calculated and sent back as a response in the form of a JSON file. A query to the service thereby leads to a number of internal processes contributing to the query-time.

4.6.1.2 Results

The test results for the application written in Java shows a clear performance advantage of the standard Linux system over both Unikernel implementations in all of the collected metrics.

As outlined in chapter 4.1.1, changing hardware configurations had not much impact on the measured data of the Unikernels. Only for the Ubuntu system an increase in the

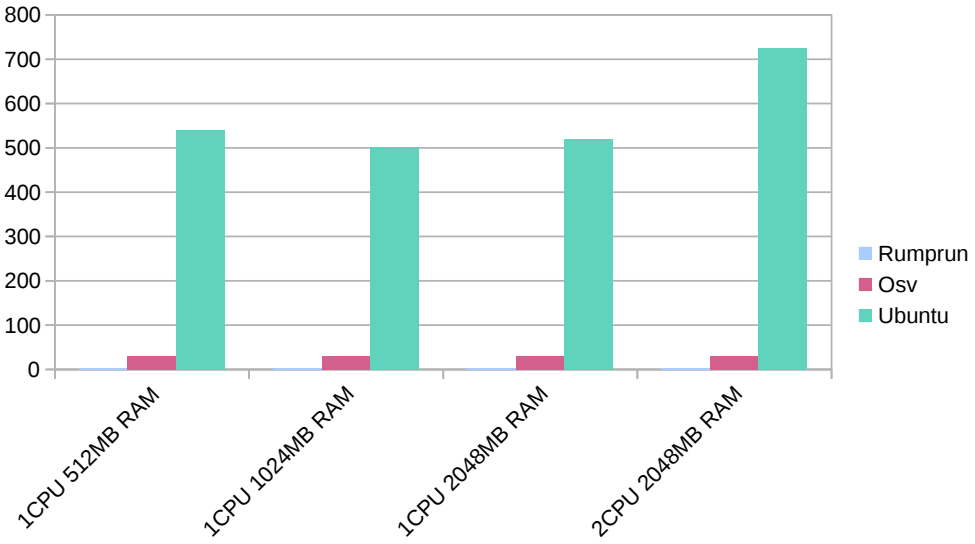

Figure 4.5: Request rate (req/s) of the Java-IoT application for all configurations

throughput could be measured. The switch from one virtual CPU to two virtual CPUs lead to a request rate at 140 percent of the 1 CPU throughput. Since the values did not vary much within the Unikernels and the lower configurations of reference system, the configuration 2 (see table 4.1) was chosen as comparison in table 4.8.

CPU	RAM (MB)	System	Reply time(ms)	Request rate (req/s)
1	1024	OSv	15,98	41,1
1	1024	Rumprun	219,54	2,82
1	1024	Ubuntu	1,44	506,96

Table 4.8: Httperf results for Java IoT-app: Reply time

Both measured values are highly correlated. The lower the reply time the higher the request rate and vice versa. With a reply time of 15,98 seconds the OSv Unikernel was the better of the two Unikernels but eleven times slower than the reference system. Rumprun did not perform well in this testseries. With a reply time of 219,54 it needed 152 times the time Ubuntu needed for a request on average. This is consistent with the boot time measurements results for Rumprun where the Java application did much worse than the other options.

4.6.2 Node.js IoT-App

Node.js is an application framework based on the ECMA Script (JavaScript) language and allows the creation of serverside applications. It became popular in the recent years in web applications for its reactive programming style.

4.6.2.1 Test setup

The logic implemented in the "IoT-Node.js" application is the same as in the Java equivalent 4.6.1. The application collects temperature data from simulated sensors into an in-memory database and provides a REST interface to query the data. Whenever a request reaches the REST interface the database calculates the average value of all collected temperature data and responds with a JSON representation of this data. Like in the Java application a request to this service involves a number of different parts of the system including the in-memory database and therefore a measurement of the performance characteristics of this application allows to draw further conclusions about the performance of the overall system.

4.6.2.2 Results

The results of the Node.js-based IoT application show that the amount of resources attributed has next to no influence on the measurement results of the Unikernels. The average reply time for a request and the request rate is relatively stable for increasing system resources, indicating that resource attributions are not the bottleneck for the performance. Therefore, the following data is selected from the results for the second configuration in table 4.1. The only significant differences between configurations is the same behavior perceived in the Java results 4.6.1.2 for the Ubuntu system. In the Node.js results the jump from one virtual CPU to two virtual CPUs has lead to a throughput of 140 percent compared to the result of one CPU.
The average reply time (see table 4.9) for a request to the REST-interface of a OSv Unikernel is more than three and a half times the time it takes a request to the reference system. This gets toped by the average reply time of the Rumprun system that results with 5,92ms in a 17 times higher reply time than the Ubuntu system.

Compared to the respective results of the Java application the results show an improvement in the reply time for all systems.

CPU	RAM (MB)	System	Reply time(ms)	in percent of ref. system
1	1024	OSv	1,2	352,94%
1	1024	Rumprun	5,92	1741,17%
1	1024	Ubuntu	0,34	-

Table 4.9: Httperf results for Node.js IoT-app: Reply time

The request rate is the number of requests that can be executed within a second and gives an indication on the amount of traffic a system is able to process. The results (table 4.10) show that OSv is able to process a third of the request the Ubuntu system can process in the same time. Rumprun is with 147,60 request per second much slower than the reference system and reaches only 12,80 percent of the throughput of the Ubuntu Linux.

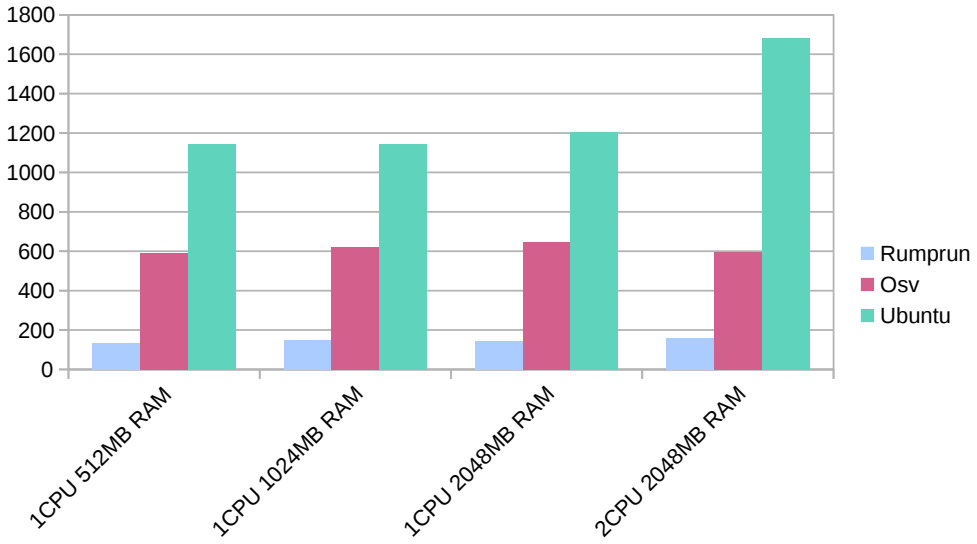

Figure 4.6: Request rate (req/s) of the Node.js-IoT application for all configurations

CPU	RAM (MB)	System	Request rate	in percent of ref. system
1	1024	OSv	622,68	54,41%
1	1024	Rumprun	147,60	12,89%
1	1024	Ubuntu	1144,24	-

Table 4.10: Httperf results for Node.js IoT-app: Request rate

Figure 4.6 shows the request rate of the Node.js IoT application on both Unikernels and the Ubuntu server reference system with all tested configurations.

4.7 Performance Measurements in other Works

The performance of Unikernels was measured before in different works and led to varying results.

In their 2015 paper "A performance evaluation of Unikernels" (16) Briggs et al. compared the two Unikernels MirageOS and OSv with a standard Ubuntu Linux 14.04. The conducted tests include tests for the TCP/UDP stack with the bandwidth benchmark tool iperf, test for a DNS server with the toolsuit queryperf and test for a HTTP-server delivering static html pages with the tool httperf. The paper found that OSv exeeds the performance of the Linux system in all categories but they had some problems porting some applications to OSv through the early version they were using. For MirageOS the paper concludes that the response rate for the DNS test was much higher than for the other systems but they discovered severe bugs prohibiting the comparison of the HTTP server.

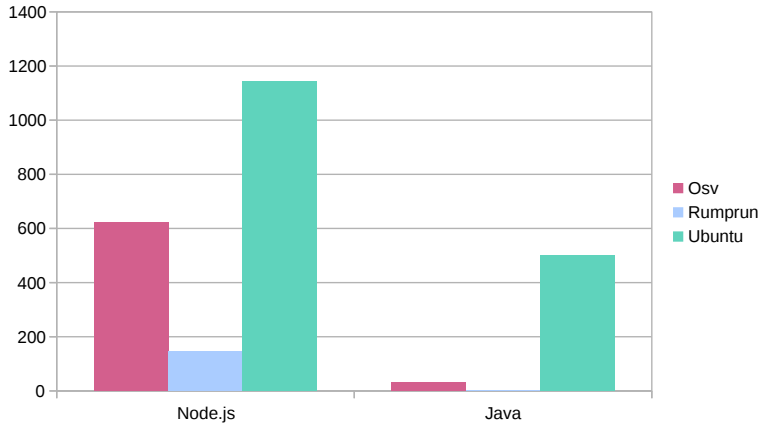

Figure 4.7: Comparison: Node.js and Java-IoT app. Request rate (req/s) for configuration 1CPU 1024MB RAM

The OSv project conducted a series of tests (79) in 2014 with the netperf testsuit comparing the performance of the network stack of an OSv Unikernel with a Fedora Linux version 20. They conducted two testseries, one with one virtual CPU and a second with four virtual CPUs both on a KVM hypervisor. The results show a relative advantage for OSv of 110 percent for the MAERTS test profile with one CPU (103 percent for 4 CPUs), a 163 percent relative performance for the TCP_RR profile (151 percent for 4 CPUs) and 177 percent advantage for the UDP_RR profile (162 percent advantage for OSv with 4 CPUs). These results are consistent with the findings in this work as described in chapter 4.4.1.

In 2014 Kivity et al. published a paper (56) benchmarking some performance characteristics of the OSv Unikernel. The tests included benchmarks on the TCP/UDP networkstack with the tool netperf. For the TCP_STREAM profile they concluded that the throughput of OSv is 24-25 percent higher than the Linux system. For the request-response test profile they found a 37-47 percent reduction in latency for both TCP and UDP.

Plauth et al. have conducted a study (82) in 2017 where they compared containers, virtual machines and the two Unikernels OSv and Rumprun. One of the testseries tested the performance of the HTTP server nginx serving static content on the systems. The results show for Rumprun that contrary to their expectations it could not outperform containers and both, containers and Unikernels, were outperformed by a Ubuntu Linux running on KVM. They suspected that this could be due to the highly optimized network stack found on a Linux system in comparison to the NetBSD network drivers used by Rumprun. In a followup test they compared the network performance of the Rumprun system with the performance of a full NetBSD system and found a significant performance advantage for the Rumprun Unikernel. They concluded that with further improvements on the network stack Rumprun could keep up or outperform a Linux virtual machine in

the future.

4.8 Evaluation of the Results

The tests conducted in this chapter fall into four categories: Tests for network protocol handlers, measurements of the boot time with different running applications, measurements of the image sizes with changing frameworks and applications and tests with real world applications to get a more comprehensive overview. In the following chapters the results are discussed in relation to each other and conclusions about the performance of Unikernels are drawn. Table 4.11 gives an overview over all tests that were conducted and presents the results of the two Unikernels in percent of the reference system (Ubuntu).

Test	Unit	OSv	Rumprun
TCP_RR	throughput	138,01%	106%
TCP_STREAM	throughput	67,51%	25,71%
TCP_MAERTS	throughput	138,65%	nA
UDP_RR	throughput	133,88%	99,44%
UDP_STREAM	throughput	102,25%	23,29%
HTTP	throughput	109,80%	15,56%
CoAP	throughput	98,17%	47,07%
MQTT	throughput	92,91%	nA
IoT-Java	throughput	6,10%	0,57%
IoT-Node.js	throughput	54,42%	12,90%
Boot time: netserver	time	6,28%	7,97%
Boot time: IoT-Java	time	49,51%	389,40%
Boot time: IoT-Node.js	time	19,85%	18,67%
Image size: netserver	size	0,76%	0,50%
Image size: IoT-Java	size	3,25%	5%
Image size: IoT-Node.js	size	2,60%	1,50%

Table 4.11: Overview over all testresults for configuration 2. The data is presented in percent of the reference system (Ubuntu).

The choice for languages and frameworks were limited by the support of the Unikernels in building images with these frameworks. There could have been other interesting programming languages to be used in the tests of the real-world applications like python or go but had to be ruled out for a lack of support through the projects. At this point the Unikernel projects OSv and Rumprun do not fully support the language Go that could potentially improve the boot time and performance in the HTTP-testseries as it is compiled into a binary running directly on the CPU instead of running on a heavyweight virtual machine (Java) or being interpreted on the fly (Node.js).

4.8.1 Network Protocols

The defining characteristic for all IoT devices is their network connection. The tests for the handling of network connections through Unikernels therefore play an important role in evaluating whether Unikernels are a good fit for IoT devices. During the tests five protocols from different OSI-layers were tested and evaluated for their performance.

On the transport layer the two most prominent protocols TCP and UDP were evaluated. For the OSv Unikernel the test results show that for the network tests (chapter 4.4.1) the Unikernel has clear performance advantage over the reference system in most (but not all) of the tests. This shows that the zero-copy API for socket connections implemented in OSv (described in chapter 2.4.1.1), enabled by the lacking separation between kernel- and userspace in Unikernels, results in clear performance advantages over a comparable Linux system with stricter separation and privilege transitions. With a throughput of 138% compared to the reference system in the TCP_RR profiles OSv clearly outperformed the Ubuntu system where Rumprun could with 106% perform only slightly better.

The MQTT broker used during the tests 4.4.2.2 is called Mosquitto and is an application written in the programming language C and therefore is compiled into a binary instead of bytecode as with Java. The results show that OSv can with 92,91% almost keep up with the performance of the reference system. Unfortunately the broker could not be compiled for Rumprun since the NetBSD based Rumprun compiler was not able to work with the build configuration of the Mosquitto project. Therefore, no Rumprun results could be gathered.

The CoAP server compiled into the Unikernels and Ubuntu system shows different testresults for OSv and Rumprun. Where OSv could score a draw with the Ubuntu system by reaching a result of 98,17%, the Rumprun system could barely reach half of the throughput. Since CoAP is based on the UDP protocol one might be able to see similar behavior as in the UDP tests. Where the results for OSv show a similar relation between TCP_RR and MQTT (TCP based) as in the relation between UDP_RR and CoAP, the results for Rumprun clearly break such a correlation.

The ability of the OSv Unikernel to outperform the reference system during the HTTP tests (4.4.4.2 and score a draw in the CoAP category (a Java application as well) shows that OSv can handle Java applications and the interaction between the network drivers and the JVM (Java virtual machine) does not lead to performance penalties. The Rumprun Unikernel on the other hand seems to have difficulties in the handling of the JVM. The results for the simple HTTP application written in Java show a throughput of only 15 percent compared to the reference system. Contrasted with the good results for the TCP_RR tests and the results for the CoAP tests this indicates that Rumprun is not as well equipped as OSv in handling Java based systems. Contrary to the OSv project, the Rumprun project has not focused on optimizations for the JVM.

4.8.2 Boot Time

When the Node.js application was used for the boot time test, Rumprun was able to beat both other systems with a time of 5,86 seconds (18,67% of the time Ubuntu required for the bootprocess). The good results could not be repeated when the Java application was used in the tests. With an average boot time of 122,13 seconds it needed 3,9 fold the time it took the Ubuntu system to boot. This is an additional datapoint for the above conclusion that Rumprun has difficulties with the handling of Java applications.

The boot time measurements for Rumprun show a good result for the native application (netserver) with an average boot time of 2,55 seconds (compared to a slightly faster OSv with 2,01 sec and a slow 32,06 seconds for the Ubuntu system).

4.8.3 Image Size

In the test category image size Rumprun has shown that it needs only 65% of the image size OSv needs for the native application (netserver) and 58% of the storage in the case of the Node.js application.

With an image size (chapter 4.5)) of only 100MB the OSv system shows a significant advantage over the reference system by using only 2,6 percent of the image size the Ubuntu system needs.

4.8.4 Real World Applications

When it comes to the more holistic tests for the REST-interface over the HTTP protocol taking other system components into account, the OSv Unikernel cannot keep up with the reference system. With needing 3,5 times the average reply time of the reference system (11 times for the Java application) and only a third of the request rate (8% for the Java application) the OSv Unikernel cannot compete with a Ubuntu system in this category. When it comes to boot- and initialization time OSv clearly outperforms the Ubuntu system and takes only half of the time to boot, initialize the application and respond to requests. This time goes down to 35% for higher resource attributions to the Java application and to 19% for the Node.js implementation.

The Rumprun Unikernel could nearly reach the performance of the reference system in two of the five network testseries (TCP and UDP request-response tests) as described in 4.4.1. In two other tests the system did manage to complete the tests but showed a much worse performance than the other two systems (TCP and UDP stream tests). During the HTTP testseries the average reply time of a request and the request rate was measured for both applications Java and Node.js. Compared to the Ubuntu system and OSv kernel the Rumprun Unikernel lost in all categories by a big margin. It took over 17 times the reply time the Ubuntu system needed for the Node.js application and got only an average of 2,82 requests per second completed for the Java application (Ubuntu got 506,96 req/s). A notable improvement can be seen when the results for the two applications are compared

against each other. The Ubuntu system could improve its request rate from 506,96 req/s for the Java application to 2314,98 for the Node.js application leading to a 4,5 times improvement. When the Rumprun Unikernel was started with the Node.js application it reached an average request rate of 143,04. Compared to the Rumprun result for the Java application (2,82 req/s) this is a 50 times improvement. The same comparison for the OSv Unikernel results in a 18 times better result for the Node.js application. The devastating performance of the Rumprun system in the Java test and the 50 times improvement for the Node.js test indicates that Rumprun has a particular problem with the Java system. While the OSv project explicitly supports Java applications and describes optimizations it has implemented for this environment (56), the Rumprun project has not mentioned any performance optimizations for Java specifically. This could give an indication for the reason of the much poorer performance the Java application has shown on the Rumprun Unikernel.

The differences between the result for the HTTP tests (see 4.11) and the IoT-Java results show for both Unikernels the difficulties they have in handling a more heavy-weight framework. For the HTTP tests the lightweight framework “Spark” was used focusing on facilitating REST requests but does not do much beyond. IoT-Java on the other hand was developed in the Spring-Boot framework providing a wide range of functionalities used during software development. Besides providing ways to easily develop REST interfaces it includes a comprehensive dependency injection framework and ways to abstract a database connection. The use of this framework requires a longer booting and initialization phase for the application since it has to configure and initialize the application by evaluating the runtime configuration.

CHAPTER 5

Security of Unikernels on IoT Devices

IoT devices are a part of the everyday lifes of people through the central integration of IoT devices into home automation, smart cars, smart meters, smart cities and part of the manufacturing process in factories. This is leading to concerns regarding privacy for the users and security and accountability for the devices.

The future homes of people increasingly will be smart homes. These smart homes will have a multitude of local sensors and actors connected to a local smart home hub. The hub can open doors, interact with the integrated kitchen system and use the collected data from the sensors to operate feedback loops connecting temperature sensors with the heating system. The smart hub is connected to a cloud uplink and thereby allows the user to remote control the system and read data from the devices. This makes clear that whoever has control over the system, has access to the most private surroundings of a user. This leads to smart homes being a prime target for an attacker as it allows access to private life of a target person (32).

On a larger scale IoT devices are, as part of the industry 4.0 movement, an integral part of the manufacturing facilities integrating the production systems with the IT landscape of companies. The IoT driven manufacturing facilities are not isolated from the outer world, but are connected over the Internet to other IT systems within or between companies to deliver realtime data for production planning. Data is constantly uploaded to the cloud, in order to organize the flow of data and manufacturing parts for improved efficiency of the manufacturing process. Traditional IT defense systems struggle to keep up with the range of newly connected devices. The industrial control systems contain vulnerabilities like the traditional IT systems assume the local network to be protected (32).

The industrialization of the process of finding and exploiting bugs in a system shortened the timespan between finding a bug and using the exploit for criminal gains and thereby

created a profitable market for bugs and corresponding exploits. Recent widespread attacks like the ransomware campaign going by the name "WannaCry" have shown a professionalization of attacks on systems. Security relevant bugs in software get traded for high prices over trading platforms leading to increased efforts to find bugs and selling them on market places. These bugs then get bought by criminals using them for campaigns infecting a vast number of systems. Any system connected to a network needs to address the possibility of an attack and how to recover from being compromised.

This chapter discusses security incidents involving IoT devices, the aims of the attackers and highlights implications of compromised devices in a corporate IT infrastructure. Since Unikernel projects claim advantages regarding security over conventional operating systems, this chapter evaluates the properties and claims of Unikernels for software deployments on hypervisor based systems. While the properties of Unikernels aim at hardening the system against an attack, they cannot prevent the existence of softwarebugs in the application or the Unikernel code. The ability to rollout new versions of an application in time is paramount to the security of the system. Chapter 5.5 discusses the contributions of Unikernels in this field.

5.1 Security Incidents involving IoT devices

In 2016 a large-scale DDOS (distributed denial of service) attack took place involving thousands of IoT devices sending requests to websites and subsequently leading to their unavailability (100). Internet connected DVR and web-enabled cameras by the Chinese manufacturer XiongMai Technologies got hijacked by a group of attackers and connected to a botnet. The botnet software Mirai was used to connect the IoT devices and launch coordinated attacks on its targets through the use of the network connection of thousands of captured IoT devices. The attack on the DNS service provider "Dyn" lead to availability problems for big companies like Netflix, Twitter and Github (57).
Researchers from the security company Flashpoint have found over 515.000 vulnerable devices by this manufacturer accessible over the Internet, that could potentially be attacked and integrated into the botnet (58). Since the device manufacturer has no means in place to update the flawed software on the devices, they had to recall a large number of devices in the US (19).

Verizon describes in their 2017 cybercrime report an incident involving IoT devices taking place in the network of an unnamed university (102). The devices affected by the attack ranged from network enabled light bulbs to vending machines for soda. These IoT devices were all connected to a subsegment of the network and the administration staff began to notice an unusual amount of traffic to the DNS server coming from the IoT devices. The devices were controlled by a botnet launching DDOS attacks on certain websites. The botnet spread from device to device by brute-forcing default and weak passwords. After the malware gained full control over the device, it connected to the central server for updates and instructions.

In October of 2017 reports of a new botnet by the name "IoT reaper" or "IoTroop"

emerged that is based on the Mirai botnet sourcecode attacking a range of IoT devices by numerous manufacturers. At this point the botnet has grown to a size of 2 million infected devices. The botnet attacks known weaknesses in the devices where patches already exist. It uses an arsenal of nine publicly known exploits to attack devices. This shows that the problem is not the availability of patches that would prevent the attack, but the lack of awareness by the users and lack of a method to roll out updates in masses to the devices (22) (75) (59).

These incidents show that with the large-scale rollout of IoT devices to consumers and in the industry, the threat for attacks on the infrastructure of the Internet rises. While the capturing of a single consumer IoT device by an attacker could be just an inconvenience for the user without further threat for this user, the device as part of a larger botnet can pose a serious risk to companies, government agencies and the infrastructure of the Internet. This makes clear that every device connected to the Internet entails the responsibility of making sure that the device is up to date and can be recovered upon a successful attack.

5.2 Implications of compromised IoT devices

IoT devices can be the target of attacks for two reasons: On the one hand for gaining access to - or manipulating data traversing the device, on the other hand to use the captured device as a launchpad for further attacks on other systems.

If the device is processing data interesting to an attacker, the motive for an attack could simply be to gain access to the traversing data and exfiltrate them to destinations outside the security perimeter. If the IoT device is an actor controlling a physical object directly, the goal of an attacker could be to manipulate the device itself in order to trigger a physical object like a door lock that is the target of an attacker. The aim of the attack in these scenarios is the device itself.

As part of industrial control systems IoT devices are integrated into the inner network of a company, making them an ideal launchpad for attacks on more valuable systems in the network. An IoT device that just collects data could be seen by the administrators to have a low impact on the security because the task handled by the device is not security critical. This could lead to neglecting the otherwise strict update policies. By compromising the device an attacker has an entrypoint to the wider network and a launchpad for attacks on bigger targets. Another use of a compromised IoT device is the collection of a large number of similar devices into a botnet in order to use the large number of network connections to overload a target system. Distributed denial of service attacks (DDOS) connect a large number of compromised systems called “bots” to a central control unit. When the attack gets launched, all the bots start a large number of requests to the system under attack (a website for example). This load is more than the system can handle and the system fails to respond to legitimate requests. IoT devices are an ideal target for being made into bots since they are by definition connected to the network and always available. They often form large networks of homogeneous devices

and if an attacker has a bug compromising one system the attack can be scaled up to take over all of the similar devices. The incidents described in 5.1 show that this scenario is realistic and already being exploited by criminal groups.

5.3 The OWASP Top 10

Attacks on IT systems can occur on many different levels. On a low level errors in the handling of memory can be exploited in binaries and on a higher level the protocol handling and configuration errors can lead to compromised systems.

The OWASP Foundation periodically publishes a list of the ten most common security vulnerability categories in webapplications ordered by their abundance called "OWASP Top 10". Since IoT devices often present content for users through a webapplication or provide a web based management interface, the OWASP Top 10 is also relevant for IoT devices.

Code	Threat
A1	Injection
A2	Cross Site Scripting (XSS)
A3	Broken Authentication and Session Management
A4	Insecure direct object reference
A5	Cross Site Request Forgery (CSRF)
A6	Security Misconfiguration
A7	Insecure Cryptographic Storage
A8	Failure to Restrict URL Access
A9	Insufficient Transportlayer protection
A10	Unvalidated Redirects and Forwards

Table 5.1: The OWASP Top 10 - Vulnerability List 2010 (99)

In their 2017 paper (44) Happe, Duncan and Batterud try to group the ten OWASP vulnerabilities into three categories and discuss the contribution of Unikernels to the security of IT systems. The first category "low-level vulnerabilities" deals with vulnerabilities that can be mitigated by local defense measurements like input validation - and output sanitization libraries. The vulnerabilities A1, A2 and A8 are part of this category. The second category "high-level vulnerabilities" includes vulnerabilities that must be dealt with on an architectural level. The OWASP A2, A5 and A7 are part of this category. The third and final category of "application specific vulnerabilities" directly depends upon the application specific workflow and can only be dealt on an application specific basis. The vulnerabilities A4, A6, A10 belong to this category.

With the widespread rollout of IoT devices in industry, homes, cars and cities the OWASP Foundation started a new project specifically focusing on the vulnerabilities and security

concerns in the area of IoT devices. Table 5.2 shows the most recent Top 10 list of vulnerabilities compiled by the project in 2014.

Code	Threat
I1	Insecure Web Interface
I2	Insufficient Authentication/Authorization
I3	Insecure Network Services
I4	Lack of Transport Encryption
I5	Privacy Concerns
I6	Insecure Cloud Interface
I7	Insecure Mobile Interface
I8	Insufficient Security Configurability
I9	Insecure Software/Firmware
I10	Poor Physical Security

Table 5.2: The OWASP IoT Top 10 - Vulnerability List (38)

5.4 Security Properties of Unikernels

Bratterud et al. have identified six security properties that Unikernels exhibit (15). These properties constitute the security features claimed by the Unikernel projects for their products. The following security observations were identified:

1. Choice of service isolation mechanism
2. The concept of reduced software attack surface
3. The use of a single address space, shared between service and kernel
4. No shell by default, and the impact on debugging and forensics
5. Micro services architecture and immutable infrastructure
6. Single thread by default

An additional property relevant to the security of the system is the choice of programming language used for the development of an application on a Unikernel. Therefore, the implications of the use of certain languages will be discussed in an additional chapter below 5.4.6. The property of lacking privilege separation in Unikernels can potentially have a negative impact on the security since all code is running in ring0. Chapter 5.4.7 discusses the implications thereof.

5.4.1 Isolation

Modern operating systems have multiple layers of protection and isolation implemented in the kernel. They protect processes running in kernels space from rogue processes running in user space and protect user space processes from one another. This isolation is achieved by virtual memory where every process gets its own virtual memory space. New hardware instructions and support were introduced to facilitate this mechanisms in the CPU. The problem with process level isolation is that the processes still share the same kernel. If the execution stack gets compromised and one process is able to gain root privileges, all other processes are compromised as well.
In 1974 Popek and Goldberg (83) introduced a formal model describing complete instruction level isolation i.e. hardware virtualization (31). While hardware virtualization was present in mainframes for a long time, it was introduced by chipset manufacturers for desktop environments as late as 2005. In a hardware virtualized environment a hypervisor running on the hardware presents a uniform interface to the virtual machines. The dedicated instruction set for managing virtual machines allows the hypervisor to securely and efficiently separate the memory pages of the machines and manages interrupts for all running VMs.
Another isolation technology that became popular in the recent years are application containers. The most prominent implementation of this technology is the Docker system (see 3.2.2). The isolation relies on two technologies by the underlying Linux kernel: cgroups and namespaces. These kernel features allow to specify the amount of resources each virtual machine can access and isolate the running container from other containers and the underlying system through dedicated namespaces preventing the application from accessing resources not belonging to this container.

5.4.2 Reduced Attack Surface

One of the major arguments of the different Unikernel projects with respect to security, is the reduced number of services and processes running on the system and thereby reduced attack surface (63) (31). Bugs can occur in any of the components of a system and the effect is not limited to this component. The argument for the reduced attack surface states that if a system has less code either running or available in the system image to run, this reduces the amount of potentially runnable code and the points where bugs could be present.

Bratterud et al. describe in (15) a more formal approach to the considerations of attack surface on Unikernels. They define the attack surface of a system as the number of bytes physically available for reading, writing and executing on a given hardware architecture. One of the smallest Linux distributions “TinyOS” with an assumed size of 24MB executing a 1MB executable would thereby lead to an attack surface of 25MB. Assuming that there exists a Unikernel providing the same application with a size of 2MB, this would lead to a reduction of the attack surface of 92%.

5.4.3 No Shell in Unikernels

Most POSIX 2.2.6 operating systems include a command line interpreter (shell) that can start new processes. It is widely used to configure, update, maintain and debug in server environments as it provides fine grained control over the running system. A standard Linux system running on a server includes a variety of services and tools facilitating the maintenance of the system. One of the most common is the ability to connect to the system from remote via a secure shell connection (ssh). The client connects to the server with a cryptographically protected connection and executes a shell on the server. Anybody with access to a shell on a system can execute arbitrary commands on the system. This makes a shell an interesting goal for an attacker and many exploits aim to get a shell for the attacker. Contrary to most operating systems, a Unikernel does not provide this type of access and normally would not include a shell. A Unikernel is treated as immutable infrastructure and therefore there is no need for a shell to assist in any maintenance tasks. The other problem with shells is that they start new processes which is contradicting the Unikernel principle of a single process running in a single address space. This Feature of Unikernels reduces the options an attacker has on the system. Even if a bug is found on the system and the attacker can control the controlflow of a program, the lack of a shell greatly reduces the options for further exploiting the system.

5.4.4 Mutability and Microservices

Unikernels are treated as immutable infrastructure meaning that a once compiled Unikernel cannot be altered after the fact. Where a traditional operating system is configured after the installation and provisioning on a hypervisor or bare metal, the Unikernel is finally configured at compiletime and does not support altering the configuration once the Unikernel is deployed. This different approach is represented in the toolstack that is available to a user on a traditional operating system like a Linux system where one normally finds services for remote logins, tools to modify files and a shell to execute arbitrary commands. On a Unikernel none of these tools is provided by default. If it is specifically needed for a particular task, services like a ssh server can be compiled into a Unikernel, but by default no such services are available leading to the advantages described in 5.4.2.

This immutable infrastructure approach has consequences for the system security as well. If a bugreport is released by a manufacturer and a securityfix is rolled out to the clients, in a traditional operating system the engineer would connect to the system and install the patch on the system. The problem with this approach is that the bug could have been exploited before the patch was installed and there is no easy way to guarantee that the system has not been compromised before. With a Unikernel infrastructure on the other hand the security patch would result in a recompilation of a new Unikernel including the patched application and the flawed system running on a hypervisor would be shut down and replaced by the new system.

5.4.5 Single Thread by Default

This property applies to some of the Unikernel projects but not all. IncludeOS and MirageOS by default execute all running code in a single thread (15). Other Unikernels like OSv or Rumprun have their running code all in one process but several threads that get preempted and scheduled by a custom scheduler. The argument for a single thread regarding security layed out by (15) is similar to the argument of reduced attack surface. Through the use of a single thread, the complexity of the running system gets reduced as the error prone handling of multiple threads and their coordination does not need to be addressed.

5.4.6 Programming Language in Unikernels

The programming language used for the application in a Unikernel has a big impact on the security of the system. Depending on the type of Unikernel, the programmer can or can not choose the language for the application. Unikernels like MirageOS or IncludeOS require the application to be written in the same language as the Unikernel itself. For MirageOS 2.4.1.2 it is the type safe functional language OCaml and for IncludeOS it is the language C. Other Unikernels like OSv 2.4.1.1 or Rumprun 2.4.1.3 support a range of frameworks and languages by providing a POSIX interface to the applications and therefore allow to run POSIX compatible programming frameworks. Different languages have different properties regarding security. Especially error prone is the memory safety and memory management of languages. Flexible languages like C or C++ allowing the direct manipulation of memory cells, pointer arithmetic and manual memory allocation. This leads to higher performance in the execution but introduces a range of potential security problems like buffer overflows, use-after-free and dangling pointers (44).
Many highlevel languages disallow pointer arithmetic and manage memory access through the execution environment for the user. Garbage collectors free the memory after being used whereby the programmer does not need to free the memory manually after the use.(44)

5.4.7 No Privilege Separation - A problem?

In the world of desktop- and server-computing one of the most important security related lowlevel tasks of an operating system is the separation of processes running in the unprivileged user space from processes running in kernel space (explained in detail in 2.2.2) that need to be protected from rogue user space processes. Unikernels neglect the separation for the benefit of increased execution speed. Where in a traditional OS the architecture clearly separates between user- and kernel space, a Unikernel is seen as one single unit that either works as expected or fails. When a process in a traditional OS through a bug starts to consume all memory resources or tries to write in memory areas reserved for the kernel, this process gets killed when the process exceeds memory boundaries or other quotas. The Unikernel does not have this type of control over the running processes.

As described in chapter 2.4.2.2 Unikernels cannot make use of the hardware protection mechanisms through protection rings (see 2.2.2) because they do not separate between kernel and application like most modern operating systems do. All code in a Unikernel (the application as well as the drivers) run on ring0 with the highest privileges. This means that the protection boundaries described in 2.2.2 do not apply to Unikernels. User space processes can therefore access all memory pages of the Unikernel and can execute any otherwise restricted instructions. In a normal Linux system an attack would happen in two stages, first the attacker would try to get access to the system by exploiting a vulnerability this could result in a shell being opened for the attacker. If the process through which the attacker got access is unprivileged (meaning the user under which the process runs has no root privileges), the second stage would be to try to elevate the privileges to be able to execute commands with root privileges and therefore having access to the complete system.
Unikernels are no multiuser systems and thus have no users or file permissions. When the attacker is able to inject his code into an executable memory page and redirect the control flow of the application to jump to this code, the injected code runs with the highest privileges as well and therefore has access to all instructions by the CPU and all memory pages of the virtual single address space. The difference to the Linux example above is the shell. Unikernels per default do not include shells and therefore the attacker would need to inject the code of the shell command that should be executed as well. The second stage of the above attack is needless since the Unikernel does not have users or associated file permissions that would prevent access to certain resources.

5.5 Atomic Updates

IoT devices seldom provide a physical interface like a monitor and keyboard. If an update process fails to update the system and produces an error, this could lead to a device disconnected from the network and thereby needs to be physically accessed or replaced completely. The idea of atomic updates is that the system has a fallback if the update fails. The system initiates the update process, if it succeeds then it continues to run on the new version of the software. If on the other hand it fails to update the system properly, the update process gets canceled and since the old version of the software is still present, the device continues to run on the old version.
Google's Chromebook laptop with the ChromeOS follows this approach by having two partitions on the harddisk, both containing the operating system. In case of an update, the new version is installed to the currently unused partition. Upon reboot the partition with the new version is selected but if the boot fails, the old system is pre-selected as boot-option so the system can recover to the functioning version of the operating system (98). The Android project has adopted this update process for the release of Android N as well (3).
Unikernels running on top of a hypervisor on a IoT device have two layers that need updates: the Unikernel and the hypervisor. When a bug is reported in the application running in a Unikernel or a new feature is implemented and should be rolled out to

the device, the sourcecode gets changed on the developers machine and compiled into a Unikernel by the build system. This newly created Unikernel then gets downloaded to the IoT device and started on the hypervisor. Thereby the new system starts replying to incoming requests and takes over from the old Unikernel that can be shut down. The second layer needing updates is the hypervisor. Through the minimal size and hardening of the system this should be a more rare event. For this type of update the dual-boot model of ChromeOS could be employed (32).

5.6 Summary

All software contains bugs no matter how hard the developers try to prevent them. Therefore, a way to handle the problem is needed. Unikernels can contribute to the solution in two ways: on the one hand their system images contain only the software components needed for the task at hand and thereby reducing the attack surface to the absolute minimum. On the other hand are Unikernels designed as immutable infrastructures meaning that after the compilation of the application into a Unikernel, the system image is not intended to be altered. When a problem with a system is detected, an engineer does not connect to the system and tries to repair the problem like it might be done on a virtual machine, but instead fixes the problem in the application, compiles a new Unikernel with the changed software and replaces the running faulty Unikernel by the newly compiled.

CHAPTER 6

Deployment Prototype - Deploying Software to IoT Devices

To answer the question whether Unikernels are well suited for the use on IoT devices and can contribute to the deployment process going beyond what is currently possible through the use of classic virtual machines, the first two main chapters examined properties and features of the Unikernel implementations and compared different Unikernel projects against each other regarding performance and security. What is so far missing from a complete picture of Unikernels as a deployment method, is the question on which point in the development- and deployment process Unikernels are added and how a deployment process based on Unikernels could look like.
This chapter describes a prototype that was developed to demonstrates a deployment process involving Unikernels. A product feature developed on a developers machine gets sent through a code repository to a continuous integration system, where the application is packaged into a Unikernel and distributed through a central hub to a number of connected clients. The Unikernel chosen for this prototype allows the developer to be agnostic about the Unikernel during the development process, while on the other hand allows to compile and test a near production system on the developer machine and the CI system.

6.1 Devops & Unikernels in the Development process

A recent trend in software development is “devops” a term combining development and operations. Instead of each team working isolated in silos and having few conversation happening between each other, in teams following the devops principle developers and operations people work side by side. If the software a developer wrote fails in the

production environment this problem is not just a problem for the operations but gets back to the developer as well. This produces a feedback loop where developers and operations people are incentivised to work together and make sure that the software performs well on a production system. This process is supported by a range of tools and frameworks as continuous integration (CI) - and continuous delivery (CD) systems where all code that gets pushed to the code repository gets supervised by the CI system. The CI system gets triggered by a new commit to the central repository, receives the code from the repository, builds it and runs a range of unit- and integration tests on the complete codebase. If the tests fail, the developer responsible for the last commit gets notified automatically and has to fix the problem.
A prerequisite for this type of process is a uniform execution environment. If the versions of libraries and services (for example locally installed database systems) differentiates between the developer system, CI system and production system, compatibility problems that are hard to debug are likely to arise. This leads to a situations where a bug gets reported to the developer who cannot recreate the problem on the developers machine since the problem only arises from the interaction with certain versions of a library or service. To address this kind of problem containers like the Docker 3.2.2 system gets employed. The developer can specify the requirements for a service in a Dockerfile and explicitly request a certain version of a library. If the compiled Docker container passes all tests on the developer machine, it will do so on the CI and production system as well.

Unikernels run on virtualized environments with a uniform virtualization layer presented to the Unikernel by the hypervisor. This standardization of the interaction between operating system and hardware allows the developer of the application to run and test the system in an environment similar to the production system during the development process and thereby reduce the friction between the developer system and production environment.
The development process with the use of Unikernels is similar to the use of the Docker system depending on the build system. The OSv system uses with Capstan and Capstanfiles (see 6.5) a similar approach like Docker by declaring a base image, dependencies and build instructions in a central file and compiling all necessary parts into a runnable unit. On a higher level the project Unik 3.1.1 facilitates this process by providing a unified frontend for the compilation of a number of Unikernels. It creates a central repository for Unikernel images and can derive running Unikernel instances from them. It provides monitoring and logging capabilities to supervise the running Unikernels and help in the debugging of errors.

6.2 System Overview

The architecture of the prototype for deployments of applications to IoT devices through the use of Unikernels consists of five components that are described in the following chapters in more detail.

Figure 6.1 gives an overview over the components of the process distributing changes

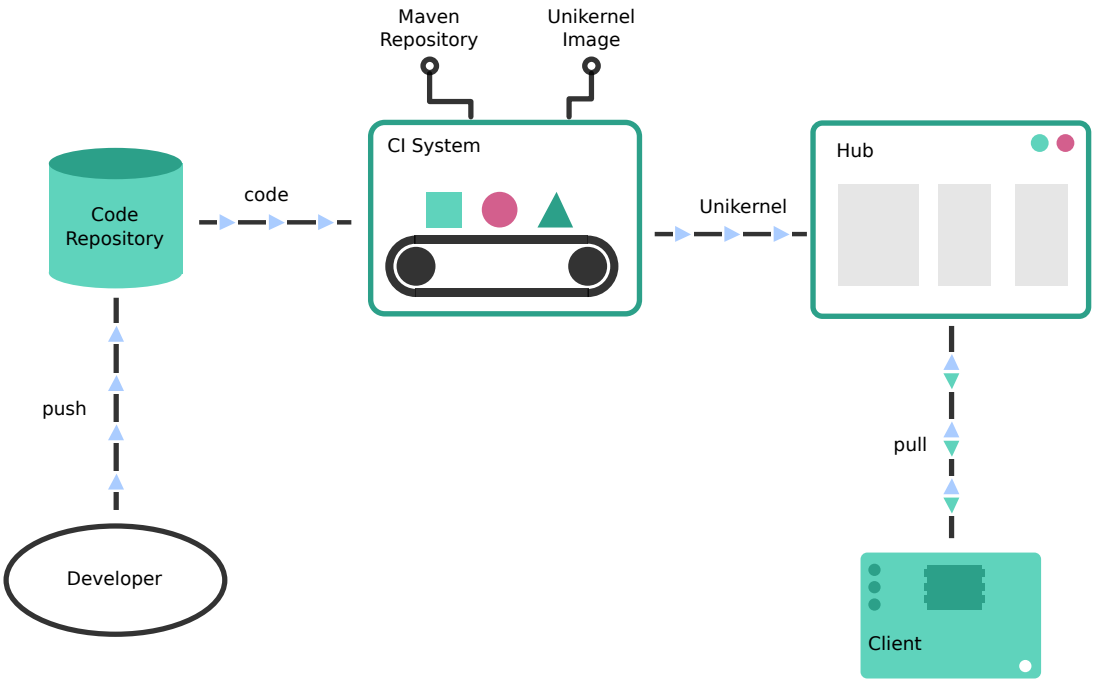

Figure 6.1: System architecture

made in the code to a number of clients. It consists of the following components:

1. Developer: All changes begin with a developer adding functionality to the application or fixing a bug.

2. Code repository: All sourcecode is managed by a centralized code repository containing and versioning changes by all involved developers.

3. CI System: A continuous integration system makes sure that the committed code is in a functioning state at any time and triggers the creation of a Unikernel after a successful build.

4. Hub: The Hub is the central connection point for all clients and responsible for distributing new versions of the Unikernel to the clients.

5. Client: The clients of the system are IoT devices running a hypervisor and receiving and executing Unikernels through an agent service.

6.3 Application Development

The process of deploying updates to clients depicted in figure 6.1 begins on the lower left corner with the developer. A developer fixes a bug in the code or develops a new feature on her local machine. In a next step new unit tests are added to the code covering the new features or reflect the changes made to existing code. When the build is successful on the local machine and all unit tests run without producing errors, the developer prepares a commit to the code repository and pushes the commit to the central version control system. For this deployment prototype the distributed version control system "git" was used where every developer has his own local repository where commits are added first and subsequently can be pushed to one or more central shared repositories. The next phase of the process, the CI system, is connected to this central repository and receives all code changes from it.

Depending on the Unikernel project used, the source code and therefore the developer can either be agnostic of the Unikernel technology, or has to develop her application specifically for the Unikernel in use (see 2.4.2 for details). The MirageOS Unikernel project (see 2.4.1.2) has rewritten all drivers from disk- to network drivers in the programming language OCaml and has deliberately broken the POSIX compatibility (see 2.2.6) of its API for the benefit of performance. The disadvantage from the viewpoint of a developer just wanting to deploy a feature to an IoT device, is that all sourcecode needs to be developed in the language OCaml and use the API only available on the MirageOS. The developer therefore binds himself to the use of a specific technology, making it harder to later migrate to a different Unikernel if needed.

Unikernel projects like OSv and Rumprun on the other hand put emphasis on their POSIX compatibility. This allows a developer to be agnostic of the underlying system and can deploy the same, unchanged application on a Linux based virtual machine as well as on the above mentioned Unikernels. Disadvantages could be fewer optimizations for compatibility reasons and missing out on MirageOS' advantages stemming from using one language all throughout the system (type safety).

For the deployment prototype the OSv Unikernel was selected since it has shown to be stable during the performance tests in chapter 4 and has the convenient property that the application does not need to be tailored to the Unikernel environment. For the demonstration the unchanged IoT-App, used during the performance tests, was used as a demonstration application that is built by the CI system and distributed to the client.

The only indication in the sourcecode for the use of a Unikernel is the Capstan file in the root directory. This is used by the build system to create an OSv Unikernel from the compiled application (see 6.5 for details on the Capstan file).

6.4 Continuous Integration

In former times the development of large software projects involving numerous developers meant that after the development of individual components was finished a critical

phase began where the components were integrated into one coherent application. The components were long in the making before an integration was tried for the first time. This meant that changes and assumptions in one component would break other components during the integration. The integration therefore was a much feared, complicated and long running process that could potentially jeopardize release dates.

A continuous integration system is a system used to make sure that the code in a repository at any given time can be built into a runnable application without error. To this end the CI system connects with the code repository of a project and receives the code upon being triggered. This can either happen periodically or through an event on the repository system. Each developer releases completed features to a centralized repository several times a day such that the current state of the repository represents a realistic picture of the currents state of the project.

The CI system builds the code from the central repository into a application by using the build system of the project (maven, gradle, ant or any other build system). When the build fails, the responsible developer gets notified and is tasked with fixing the code. A Dashboard visible to all project members is available to indicate the current health status of the project and the result of the last build.

For this deployment prototype the continuous integration system "Jenkins" was selected. It receives the code from the code repository hosted on GitHub whenever changes are pushed to the main branch. Jenkins then compiles the code with the maven dependency management and build system since the application is a Java application using the maven build- and dependency management system. During the build, the unit tests for the application are executed making sure that the changed code does not alter the expected behavior in other parts of the system. After a successful build, Jenkins triggers a script compiling the application into a Unikernel.

6.5 Creation of the Unikernel

The successful build of the code through the CI system triggers the creation of the Unikernel through a post-build script. The Unikernel selected for this deployment prototype is OSv and therefore the buildsystem Capstan is used to compile the sourcecode into the Unikernel, since Capstan was created for the OSv building process.

Capstan uses a file called "Capstanfile" (25) to specify how the code should be compiled into a Unikernel. The file structure is similar to the Dockerfile used by the application container system Docker (see 3.2.2) to specify how their containers should be built and what files and dependencies to include in the image. Both configuration files specify a base system that is used as a basis for compiling the new image. For the IoT-App the base system already includes an OpenJDK for executing JVM code used by the application. The Capstanfile specifies what build command should be executed to initiate the building process compiling the sourcecode into a runnable application (in the case

of the Java based IoT-App the goal is a JAR-file) and which of the resulting artifacts should be included in the Unikernel.

During the development phase the developer does not need to deal with the Unikernel since OSv provides standard APIs (see 6.3 for details on the development process). To test the creation of the Unikernel on the developer machine, the compilation of the compiled application into a Unikernel can be initiated through the use of the Capstan build system as well. For testing purposes a local installation of the hypervisors virtualbox or KVM could be used.

```
1 # Name of the base image. Capstan will download this automatically from
    Cloudius S3 repository.
2 base: cloudius/osv-openjdk
3 # The commandline passed to OSv to start up the application.
4 cmdline: /java.so -jar /iot-app.jar
5 # The command to use to build the application.
6 build: mvn package
7 # List of files that are included in the generated image.
8 files: /iot-app.jar: target/iot-app-0.0.1-SNAPSHOT.jar
```

Since the Capstan build system is only able to build an OSv Unikernel, the Unikernel compilation- and deployment platform Unik (see 3.1.1) could be used instead to build a wider variety of Unikernels like MirageOS, Rumprun or IncludeOS. The goal of this prototype was to demonstrate the use of Unikernels in the process and OSv was chosen as a demonstration object for reasons described in 6.3. Therefore, it was not necessary to support all kinds of Unikernels through the use of Unik and the Capstan system was chosen for the prototype for easier handling and integration in the custom distribution system (Hub). The Unik project takes control over the creation of Unikernels and the handling of images and instances in a way that was not fitting with the architecture in this prototype. Unik uses a central repository system for the distribution of Unikernels (similar to Dockerhub) that could be a replacement for the Hub, but has jet to be published.

6.6 Hub: Unikernel Distribution

The Hub is the central connection point for all connected clients. Whenever new versions of the system image are provided, the hub manages these versions and provides them to the clients.

In a real life scenario for IoT home automation devices this service would be provided to the customer as a service over the Internet. The devices would be connected to the Internet through a router in the home of the customer and would query the Hub for updates on a regular basis. In a setup where the devices are part of a larger IT

infrastructure in a company, the company would probably want to separate the devices from the rest of the network and prohibit direct connections to the Internet for security reasons. In this type of environment the Hub would be installed on a company internal system and cater updates to all company internal devices.

To make sure only allowed clients can retrieve the newly created Unikernel, the agent service running on the client has to authenticate itself against the hub.

The project resin.io (see 3.2.3) tackles the authentication of the provisioned clients by distributing provisioning keys within the system image. The device registers itself over the Internet using the provisioning keys. After the image is successfully installed and the device added to the VPN, the device receives unique API keys from the Hub.

6.7 The Agent

After the application changes were compiled into a Unikernel and uploaded to the distribution hub, the client needs to retrieve the image and replace the currently running system with the new one. To this end the client runs a software called “agent” that has the following tasks:

1. Periodically checking with the Hub for available updates
2. Downloading available system images
3. Verifying the image
4. Starting the new Unikernel
5. Shutting down the old system
6. Checking for available hypervisor updates

6.7.1 Authentication

A crucial aspect of the security infrastructure is the authentication of the clients against the server to make sure that the incoming requests are legitimate. If unauthenticated clients would be allowed to communicate with the server, the aggregated data on the server could contain misleading information about the state of the network.

There are several methods for authenticating clients to a server. One is to use client side x509 (14) certificates to provide identity information to the server. In this method the client trying to connect to a server supplies a certificate to the server in the process of establishing a connection in order to prove its identity to the server.

Client and server negotiate during the connection establishment the cipher and certificate authority they would use.

6.7.2 Verification

Verification of system images is an omnipresent problem in software distribution. Every available Linux distribution provided on a server also provides a way to verify if the image was altered during the download. A system image could either be damaged during the distribution over the network through a network problem, or being deliberately altered by an attacker on the server. The usual way to provide verification information to the receiver of an image is through a secure hash algorithm. A hash algorithm calculates a value (called "hash value") of a predetermined lengths from a given input of arbitrary length. A cryptographically secure hash algorithm calculates the hash value in a way that if a single bit of the input is altered, the hash value changes as well. This property is the basis for verifying that a distributed piece of data was not altered during the delivery. In order to use the hash value to verify a downloaded system image, one needs a way to securely distribute the hash value.

Over the years a number of cryptographically strong hash functions have been recommended for the use in cryptographic applications and later to be found to have weaknesses. The german federal agency "Federal Agency for Information Security" (BSI) publishes recommendations for the use of cryptographic algorithms (17). They currently recommend among others the SHA256 algorithm for the creation of cryptographically strong hashes. Therefore, the prototype uses a SHA256 on the Hub to create a hash value and on the client to verify the downloaded image.

6.7.3 Running the Unikernel

The hypervisor used for the prototype is KVM (see 2.3.2). In order to run a virtual machine it needs an emulator system providing drivers and the necessary environment. Here the widely known Qemu system will be used on the client. The agent service is responsible to instantiate the VM image into a running instance on the hypervisor. After downloading the image file and verifying its integrity the agent starts the Unikernel by invoking a startup script that uses the Qemu system and the KVM hypervisor.
For the network connection a bridge device is created on the client connecting the virtual devices to the physical network connection and all virtual machines receive an ip address from a centralized DHCP service.

```
1 qemu-system-x86_64 -m 1024 -smp 1 -cpu host,+x2apic -enable-kvm
  -nographic -device virtio-blk-pci,id=blk0,bootindex=0,drive=hd0 -drive
  file=/imgs/osv-iot2.qcow2,if=none,id=hd0,aio=native,cache=unsafe -chardev
  stdio,mux=on,id=stdio,signal=off -device isa-serial,chardev=stdio -netdev
  bridge,id=un0 -device virtio-net-pci,netdev=un0
```

6.7.4 Transportlayer Security

The network packets transmitted over the network can take unpredictable routes and traverse parts of the Internet that can not be trusted. To make sure that the data transmitted over the network is not altered, one approach is the verification of downloaded data presented in 6.7.2. While this approach is feasible for downloaded packets, it cannot guarantee the integrity of the data received from a call to for example a webservice API. The prototype uses TLS to encrypt the data transmitted between the hub and the agent service. In that way an attacker cannot easily intercept data or inject additional data into the packet stream.

6.8 Hypervisor Update

The Unikernel with the IoT service application is running on the IoT device on top of a hypervisor. While the process for fixing a bug in the code and distributing it to all clients has been demonstrated in the previous chapters (see 6.2), the question remains how to roll out updated on the layer beneath the application, namely the hypervisor. Hypervisors are themselves a type of operating system in that they interact with the hardware on behalf of higher software layers and manage, protect and attribute resources for the virtual machines.

A hypervisor is a complex system with a large codebase that needs to provide a vast number of hardware drivers to operate on diverse hardware. Although a systems administrator is well aware of the criticality of the security of the hypervisor and therefor is likely to take protective actions and heighten security, a hypervisor is like any other complex software and therefore bugs are inevitable. A system built upon a hypervisor infrastructure therefore needs a way to update the hypervisor as well.

The update process of an operating system on a IoT devices is more difficult than on other systems since they normally can only be reached over the network and do not provide any other means of interaction like a directly connected screen. This makes debugging and recovering from a failed system upgrade next to impossible without extracting the device from its environment and attaching it to a debug environment. To tackle this problem the ChromeOS project has divided the harddisk into two partitions. One partition contains the currently running system and the other is used for updates. When a new version gets released, the running system downloads the new system to the currently unused partition, unpacks it into a bootable system and adds this partition as the first boot option to the bootmanager. The old system then initiates a reboot. If all goes well, the new system gets booted from here on whenever the system starts up. When on the other hand the bootprocess fails, the old system is present as a backup and the default boot option gets reversed to the old system. This makes sure that there is a bootable system present at any time, even if the update process fails.

For hypervisor updates on IoT devices, a similar system could be used. The agent described in 6.7 could receive new system images of the hypervisor through the hub just

as it receives new version of a Unikernel. The client devices could use two partitions for the hypervisor updates and since the agent is running as a service on the operating system, it has (unlike the Unikernels running on the hypervisor) access to the harddisk.

From a security perspective a bug in the agent with access to the complete harddisk poses a higher risk the the security of the IoT device than a bug in the application or Unikernel since these are contained and separated through the hypervisor. A security breach in the application or Unikernel would affect the functioning and availability of the service provided by the application, but can be repaired through an update and new rollout.

6.9 Summary

The deployment prototype described in this chapter demonstrates how Unikernels can be used in a state of the art development environment to deploy an application to multiple clients.

Compared to the deployment on bare metal, the prototype has the advantage of separating the base system from the application through the hypervisor.

An application could be deployed as part of a classic virtual appliance using a complete operating system image. The comparison in 4.5 shows the advantages of a Unikernel compared to this deployment method. The system image of a standard Ubuntu server system is over 30 times larger than a Unikernel deploying the same application.

The process described in this chapter shows that the developer can be agnostic of the deployment method during the development process when the appropriate Unikernel is chosen. This is due to the selection of a Unikernel project that does not need the application to be tailored to the Unikernel. When other Unikernels like IncludeOS or MirageOS would be selected as a basis for the deployment, the layout and specific APIs of the Unikernels would have a big impact during the development and testing phase on the developers machine. The application would be developed with the specific system in mind.

The advantages regarding security and portability of a hypervisor do not come without costs. The additional layer comes with performance penalties where Unikernels can show their advantages over traditional virtual machines in some aspects (see 4.3 for details) and needs its own update process for rolling out updates. An update strategy for the application on top of a hypervisor without an update strategy for the hypervisor does not add additional security for the system. In 6.8 a strategy addressing these needs was described that would make sure that all layers of the system stay up to date.

CHAPTER 7

Future Work

When manufacturers build low power devices they often use an ARM-based processor for its low resource consumption and lower price. Both Unikernel projects (OSv and Rumprun) that were used in this work for the performance evaluation of Unikernels 4 are not supporting the ARM architecture at this point, but are working on supporting it in the future.
At the beginning of this work the intention was to take a Raspberry Pi v3 as a model for an IoT device and run the Unikernels on this device. It was possible to initialize a KVM hypervisor on top of a Linux system running on the device. The hypervisor was able to execute a Debian guest system and thereby demonstrated that virtualization on a Raspi is possible. Unfortunately the current state of the Unikernel projects do not allow the compilation of the Unikernels for the ARM platform and therefore the tests described in 4.2 could not be executed on the Raspberry Pi, but instead needed a 64bit x86 CPU.
It would be interesting to see the performance comparison conducted in chapter 4 repeated on an ARM based system and have a direct comparison of a Linux system running on a bare metal Raspberry pi v3 with its resource restrictions and Unikernels running on top of a hypervisor on this system.

The applications tested in 4.2 were both written in languages running in a heavyweight runtime environment. It would be interesting to see the performance of the same application written in a compiled language directly running on the CPU as for example Go. The application was already written to be used in the testseries but at the current moment the Unikernel projects OSv and Rumprun are not able to compile an application written in the language Go into a Unikernel in a stable manner.

CHAPTER 8

Conclusion

The idea of the library operating system has been known for a longer time and have been implemented in projects like Nemesis and Exokernel. In the current form implemented in Unikernel projects like OSv, Rumprun or MirageOS they draw from the benefits of long development of hypervisors through the support of virtualization in modern CPUs. This makes Unikernels an interesting solution to a range of problems in modern computing addressing concerns of security and the ever increasing layers of software on a modern operating system.

In this work the question was raised whether Unikernels could address the pressing problem of shipping new versions of an application to IoT devices in a timely manner and thereby improve the security of the systems. To this end Unikernels were evaluated in three dimensions relevant to the topic. The first main chapter 4 inquiried performance characteristics of Unikernels like boot time, ability to respond to network connections, handling of higher-level protocols and size of the system image and compared them with a standard Linux system in a virtual machine. The second main chapter 5 takes a look at incidents from the near past regarding IoT devices, evaluates inherent properties of Unikernels regarding security and lays out how the use of Unikernels in the deployment of software on IoT devices could contribute to the security of the system. In the third main chapter 6 the focus is on the process how software is deployed to the devices. The chapter shows a modern development process where the sourcecode written by a developer gets pushed to a repository, built by a continuous integration system, tested and deployed to a distribution hub from where the images get pulled by the devices. This process demonstrates where Unikernels could fit in the development process and facilitate the handling of application deployments.

One property of Unikernels that the Unikernel projects emphasize is the performance improvement in computation operations compared to regular operating systems. This claim was evaluated in the first main chapter 4 by conducting performance measures of real-world applications with different resource attributions. The claims could not

be verified in during the performance tests. While all Unikernel projects demonstrated performance improvements in some selected scenario (4.7), the selected POSIX-compatible Unikernel projects OSv and Rumprun were not able to outperform a classic Linux system running in a virtual machine in the tests conducted in this work. The aim of the tests was to conduct tests with near real-world applications using widely used programming languages and frameworks not optimized for the Unikernel environment and thereby get a realistic view on the deployment of real-world applications on IoT devices through Unikernels.

Chapter 5 examined the security risks and attack scenarios todays IoT devices face when deployed as part of the corporate IT infrastructure or running in the homes of users. The incidents described in 5.1 make clear that security of IoT devices already is a major problem not just for the owner of the devices, but for all other Internet connected potential targets of DDOS attacks as well. The projection of the number of deployed IoT devices in the near future (41) makes it clear that the need for security of the devices cannot be emphasized enough. Apart from the hardening of the systems themselves, the process of fixing bugs, developing patches and rolling out updates to all relevant devices in a timely manner is paramount to the security of the future Internet and all connected systems. Unikernels contribute in this field with advantageous security properties like reduced attack surface, isolation from the hardware through the use of a hypervisor and the lack of attacker-supporting management tools (like shells) on the systems. While it should be clear that the use of a new technology like Unikernels can never be a silver bullet to deal with all security problems, the small size of the Unikernel system images, abstracted environment for the developer and system hardening properties lets Unikernels contribute to the overall security of the system.

The third main chapter 6 demonstrated how Unikernels could be integrated into the development process. If the handling and building of the Unikernel would be to complicated during the development process, the benefits regarding performance and security would not justify the use of Unikernels in an IoT project. The chapter shows how Unikernels can be added to a modern development process through the use of a continuous integration system. The integration of Unikernels from a developers perspective is similar to the use of a Docker system in that it uses a configuration file for the creation of the OSv Unikernel.

Whether Unikernels are a good fit for software deployments on real world IoT devices will be dependent upon the availability of Unikernel implementations for the CPU architectures found in those devices 7. To this day the Unikernel projects support only x86 based architectures that are rather uncommon in the devices. Contrary to the claims by the Unikernel projects, the request-handling performance of a real-world application on a Unikernel was not significantly faster than the execution on a standard virtual machine with a full operating system. However the much smaller system image of the Unikernels (1/30 the size of the Ubuntu virtual machine image) has clear benefits for the large-scale distribution of new versions over the network to a large number of devices. The isolation between the hardware and application through a hypervisor has clear benefits for the

security of the system and allows the system to recover even after being compromised by shutting down the compromised virtual machine and start a new vm from the updated system image.

While the Unikernel concept might not be the silver bullet for software deployments on IoT devices, it has clear advantages over classical virtual machines with full operating systems in that it facilitates the rollout of new versions of the application and allows for the automation of large parts of the process. Thereby Unikernels can contribute to the security of IoT devices and with the availability of ARM-based versions and the continual improvement of the Unikernel projects they could be a good fit for the deployment of software to the upcoming network of 20 billion connected devices.

List of Figures

List of Tables

Bibliography

[1] Virtio project. URL https://www.linux-kvm.org/page/Virtio. Accessed: 2017-12-01.

[2] A. Al-Fuqaha, m. guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols and Applications. 17: Fourthquarter 2015, 2015.

[3] R. Amadeo. Android n borrows chrome os code for "seamless" update installation. *Ars Technica*, May 2016. URL https://arstechnica.com/gadgets/2016/05/android-n-borrows-chrome-os-code-for-seamless-update-installation/. Accessed: 2017-12-01.

[4] Amazon. Greengrass. URL https://aws.amazon.com/de/greengrass/. Accessed: 2017-12-01.

[5] Amazon Inc. Amazon AWS IoT. URL https://aws.amazon.com/de/iot-platform/how-it-works/. Accessed: 2017-12-01.

[6] AMD Staff. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Advanced Micro Devices, 2017. URL https://support.amd.com/TechDocs/24593.pdf.

[7] B. W. Arden, B. A. Galler, T. C. O'Brien, and F. H. Westervelt. Program and addressing structure in a time-sharing environment. *J. ACM*, 13(1):1–16, Jan. 1966. ISSN 0004-5411. doi: 10.1145/321312.321313. URL http://doi.acm.org/10.1145/321312.321313.

[8] Arduino. Arduino Yun. URL https://store.arduino.cc/arduino-yun. Accessed: 2017-12-01.

[9] ARM Inc. Arm architecture reference manual armv8, for armv8-a architecture profile. https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile. Accessed: 2017-12-01.

[10] ARM Inc. Arm architecture, 2017. URL https://developer.arm.com/products/architecture.

[11] Arrakis Project. Arrakis. URL http://arrakis.cs.washington.edu/. Accessed: 2017-12-01.

[12] A. Banks and R. Gupta. MQTT Version 3.1.1 Plus Errata 01. OASIS, December 2015. URL http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html. Accessed: 2017-12-01.

[13] J. Batalla, G. Mastorakis, C. Mavromoustakis, and E. Pallis. *Beyond the Internet of Things: Everything Interconnected*. Internet of Things. Springer International Publishing, 2016. ISBN 9783319507583.

[14] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008. URL https://rfc-editor.org/rfc/rfc5280.txt.

[15] A. Bratterud, A. Happe, R. Duncan, and A. Keith. Enhancing Cloud Security and Privacy: The Unikernel Solution. *Eighth International Conference on Cloud Computing, GRIDs, and Virtualization, 19 February 2017 - 23 February 2017, Athens, Greece*, 2017.

[16] I. Briggs, M. Day, Y. Guo, P. Marheine, and E. Eide. A Performance Evaluation of Unikernels. 2015.

[17] Bundesamt für Sicherheit in der Informationstechnik. Kryptographische verfahren: Empfehlungen und schlussellängen. January 2017. URL https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile. Accessed: 2017-12-01.

[18] Canonical Inc. Ubuntu for the Internet of Things, 2017. URL https://www.ubuntu.com/internet-of-things.

[19] A. Carman. Hacked webcams that helped shut down the internet last week are being recalled. *The Verge*, October 2016. URL https://www.theverge.com/2016/10/24/13383968/hangzhou-xiongmai-ddos-attack-iot-mirai. Accessed: 2017-12-01.

[20] A. Celesti, D. Mulfari, M. Fazio, M. Villari, and A. Puliafito. Exploring container virtualization in iot clouds. *Smart Computing (SMARTCOMP)*, 2016 IEEE International Conference on Smart Computing (SMARTCOMP):1–6, May 2016.

[21] K. Chandrasekaran. *Essentials of Cloud Computing*. CRC Press, 2014. ISBN 9781482205442.

[22] Check Point Research. A new iot botnet storm is coming, October 2017. URL https://research.checkpoint.com/new-iot-botnet-storm-coming/. Accessed: 2017-12-01.

[23] ClickOS Project. ClickOS. URL http://cnp.neclab.eu/clickos/. Accessed: 2017-12-01.

[24] Clive Project. Clive. URL https://lsub.org/ls/clive.html. Accessed: 2017-12-01.

[25] Cloudius Systems. Capstan project. GitHub. URL https://github.com/cloudius-systems/capstan. Accessed: 2017-12-01.

[26] J. Cormak. The Rump Kernel: A tool for driver development and a toolkit for applications. *AsiaBSDcon*, 2015.

[27] I. Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3C*. Intel Corporation, 2016.

[28] F. M. David, J. C. Carlyle, and R. H. Campbell. Context switch overheads for linux on arm platforms. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-751-3. doi: 10.1145/1281700.1281703. URL http://doi.acm.org/10.1145/1281700.1281703.

[29] Docker Inc. Docker. . URL https://www.docker.com/. Accessed: 2017-12-01.

[30] Docker Inc. Docker system documentation, . URL https://docs.docker.com/. Accessed: 2017-12-01.

[31] B. Duncan, A. Bratterud, and A. Happe. Enhancing cloud security and privacy: Time for a new approach? In *Innovative Computing Technology (INTECH), 2016 Sixth International Conference on*, 2016.

[32] B. Duncan, A. Happe, and A. Bratterud. Enterprise iot security and scalability: How unikernels can improve the status quo. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, UCC '16, pages 292–297, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4616-0. doi: 10.1145/2996890.3007875. URL http://doi.acm.org/10.1145/2996890.3007875.

[33] Eclipse Foundation. Mosquitto. URL https://mosquitto.org/. Accessed: 2017-12-01.

[34] EMC Inc. Unik project. GitHub. URL https://github.com/cf-unik/unik. Accessed: 2017-12-01.

[35] Erlang on XEN Project. Erlang on XEN. URL http://erlangonxen.org/. Accessed: 2017-12-01.

[36] R. T. Fielding, J. Gettys, and J. Mogul. Hypertext transfer protocol – http/1.1. RFC 2616, IETF, Juni 1999.

[37] E. Foundation. Californium. GitHub. URL https://github.com/eclipse/californium. Accessed: 2017-12-01.

[38] T. O. Foundation. Owasp iot top 10, 2014. URL https://www.owasp.org/index.php/Top_10_IoT_Vulnerabilities_(2014). Accessed: 2017-12-01.

[39] Galois Inc. Halvm. GitHub. URL https://github.com/GaloisInc/HaLVM. Accessed: 2017-12-01.

[40] M. Garg. Sysenter based system call mechanism in linux 2.6, 2006. URL http://articles.manugarg.com/systemcallinlinux2_6.html. Accessed: 2017-12-01.

[41] Gartner Inc. Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016. URL http://www.gartner.com/newsroom/id/3598917. Accessed: 2017-12-01.

[42] C. T. Gibson. Time-sharing in the ibm system/360: Model 67. In *Proceedings of the April 26-28, 1966, Spring Joint Computer Conference*, AFIPS '66 (Spring), pages 61–78, New York, NY, USA, 1966. ACM. doi: 10.1145/1464182.1464190. URL http://doi.acm.org/10.1145/1464182.1464190.

[43] Google Inc. Android things project. 2017. URL https://developer.android.com/things/index.html.

[44] A. Happe, B. Duncan, A. Bratterud, O. Gusikhin, V. M. Muñoz, F. Firouzi, D. Mønster, and V. Chang. Unikernels for cloud architectures: How single responsibility can reduce complexity, thus improving enterprise cloud security. *Proceedings of the 2nd International Conference on Complexity, Future Information Systems and Risk*, 2017.

[45] Httperf Project. Httperf. URL https://github.com/httperf/httperf. Accessed: 2017-12-01.

[46] C. Hunt. *TCP/IP*. O'Reilly Media, 2003.

[47] IEEE and The Open Group. POSIX.1-2008, 2016. URL http://pubs.opengroup.org/onlinepubs/9699919799/.

[48] Imagination Technologies. Mips virtualization, 2017. URL https://www.imgtec.com/mips/architectures/virtualization/.

[49] Include OS Project. Include OS. URL http://www.includeos.org/. Accessed: 2017-12-01.

[50] Information Sciences Institute University of Southern California. Transmission control protocol. RFC 793, 1981.

[51] Intel Inc. Quark processor series. URL https://www.intel.de/content/www/de/de/embedded/products/quark/x1000/overview.html. Accessed: 2017-12-01.

[52] R. Jones. Care and feeding of netperf. *GitHub*, 2012. URL https://github.com/HewlettPackard/netperf/blob/master/doc/netperf.pdf. Accessed: 2017-12-01.

[53] A. Kantee. *The Design and Implementation of the Anykernel and Rumpkernels*. 2016. URL http://www.fixup.fi/misc/rumpkernel-book/.

[54] A. Kantee and J. Cormak. Rump kernels - no os? no problem! *Login - The Usenix Magazine*, 39(5), October, 2014 2014.

[55] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: The Linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[56] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, 2014. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity.

[57] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. DDoS in the IoT: Mirai and other botnets. *Computer*, 50:80–84, 01 2017.

[58] B. Krebs. Hacked cameras, dvrs powered today's massive internet outage. *Krebs on Security*, 2016. URL https://krebsonsecurity.com/2016/10/hacked-cameras-dvrs-powered-todays-massive-internet-outage/. Accessed: 2017-12-01.

[59] B. Krebs. Reaper: Calm before the iot security storm, October 2017. URL https://krebsonsecurity.com/2017/10/reaper-calm-before-the-iot-security-storm/. Accessed: 2017-12-01.

[60] R. Love. *Linux-Kernel-Handbuch: Leitfaden zu Design und Implementierung von Kernel 2.6*. Open source library. Pearson Deutschland, 2005. ISBN 9783827322043.

[61] I. Lunden. Amazon Launches AWS IoT — A Platform For Building, Managing And Analyzing The Internet Of Things. *Techcrunch*, 10 2015. URL https://techcrunch.com/2015/10/08/amazon-announces-aws-iot-a-platform-for-building-managing-and-analyzing-the-internet-of-things/. Accessed: 2017-12-01.

[62] A. Madhavapeddy and D. J. Scott. Unikernels: The rise of the virtual library operating system. *Communications of the ACM*, 2014.

[63] A. Madhavapeddy, R. Mortier, C. Rotsos, and D. Scott. Unikernels: Library operating systems for the cloud. *ASPLOS '13 Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems Pages 461-472*, 2013.

[64] W. Mauerer. *Professional Linux Kernel Architecture*. Wiley, 2010. ISBN 9781118079911.

[65] Microsoft. Windows 10 iot core. URL `https://developer.microsoft.com/en-us/windows/iot`. Accessed: 2017-12-01.

[66] Microsoft Inc. Drawbridge. URL `https://www.microsoft.com/en-us/research/project/drawbridge/`. Accessed: 2017-12-01.

[67] R. Mijat and A. Nightingale. Virtualization is coming to a platform near you.

[68] Mirage OS Project. Technical background of MirageOS. URL `https://mirage.io/wiki/technical-background`. Accessed: 2017-12-01.

[69] MirageOS Project. MirageOS. URL `https://mirage.io/`. Accessed: 2017-12-01.

[70] R. Morabito. Virtualization on Internet of Things Edge Devices with Container Technologies: a Performance Evaluation. *IEEE Access*, PP, 05 2017.

[71] C. Moratelli, S. Johann, M. Neves, and F. Hessel. Embedded virtualization for the design of secure iot applications. In *Proceedings of the 27th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype*, RSP '16, pages 2–6, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4535-4. doi: 10.1145/2990299.2990301. URL `http://doi.acm.org/10.1145/2990299.2990301`.

[72] C. Moratelli, S. Johann, M. Neves, and F. Hessel. Embedded virtualization for the design of secure iot applications. In *Proceedings of the 27th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype*, RSP '16, pages 2–6, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4535-4. doi: 10.1145/2990299.2990301. URL `http://doi.acm.org/10.1145/2990299.2990301`.

[73] A. Mouat. *Docker: Software entwickeln und deployen mit Containern*. dpunkt.verlag, 2016. ISBN 9783960880370.

[74] J. MSV. Amazon Makes Foray Into Edge Computing With AWS Greengrass. *Forbes*, 2017. URL `https://www.forbes.com/sites/janakirammsv/2017/06/07/amazon-makes-foray-into-edge-computing-with-aws-greengrass/#416ec2603298`. Accessed: 2017-12-01.

[75] Netlab 360. IoT reaper: A Rappid Spreading New IoT Botnet, October 2017. URL http://blog.netlab.360.com/iot_reaper-a-rappid-spreading-new-iot-botnet-en/. Accessed: 2017-12-01.

[76] Netperf Project. Netperf. URL https://github.com/HewlettPackard/netperf. Accessed: 2017-12-01.

[77] G. O'Regan. *A Brief History of Computing*. SpringerLink : Bücher. Springer London, 2012. ISBN 9781447123590.

[78] OSv Project. OSv. URL http://www.osv.io. Accessed: 2017-12-01.

[79] OSv Project. OSv:Netperf Tests, May 2014. URL http://osv.io/netperf-benchmarks/. Accessed: 2017-12-01.

[80] J. Penninkhof. Mosquitto unikernel using osv and capstan, August 2015. URL https://www.penninkhof.com/2015/08/mosquitto-unikernel-using-osv-and-capstan/.

[81] Pivotal Software. Spring boot. URL https://projects.spring.io/spring-boot/. Accessed: 2017-12-01.

[82] M. Plauth, L. Feinbube, and A. Polze. A performance evaluation of lightweight approaches to virtualization. *CLOUD COMPUTING 2017*, page 14, 2017.

[83] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974. ISSN 0001-0782. doi: 10.1145/361011.361073. URL http://doi.acm.org/10.1145/361011.361073.

[84] D. E. Porter, S. Boyd-Wickizer, and J. Howell. Rethinking the library os from the top down. *ASPLOS XVI Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems Pages 291-304*, 2011.

[85] J. Postel. User datagram protocol, 1980. URL https://tools.ietf.org/html/rfc768.

[86] O. Project. POSIX API Support. GitHub, 2014. URL https://github.com/cloudius-systems/osv/wiki/POSIX-API-support. Accessed: 2017-12-01.

[87] PRPL Foundation. PRPL Foundation Unveils the First Open Source Hypervisor for the Internet of Things, July 2016. URL https://prplfoundation.org/2016/07/12/prpl-foundation-unveils-the-first-open-source-hypervisor-for-the-internet-of-things/. Accessed: 2017-12-01.

[88] Resin.io Corp. Resin.io. URL https://resin.io/how-it-works/. Accessed: 2017-12-01.

[89] Rumprun Project. Rumprun. GitHub. URL https://github.com/rumpkernel/rumprun. Accessed: 2017-12-01.

[90] Runtime.js Project. Runtime.js. URL http://runtimejs.org/. Accessed: 2017-12-01.

[91] D. A. Rusling. The linux documentation project, 1999. URL http://www.tldp.org/LDP/tlk/mm/memory.html. Accessed: 2017-12-01.

[92] R. Russell. Virtio: Towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008. ISSN 0163-5980. doi: 10.1145/1400097.1400108. URL http://doi.acm.org/10.1145/1400097.1400108.

[93] J. Saleem, B. Adebisi, R. Ande, and M. Hammoudeh. A state of the art survey - impact of cyber attacks on sme's. In *Proceedings of the International Conference on Future Networks and Distributed Systems*, ICFNDS '17, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4844-7. doi: 10.1145/3102304.3109812. URL http://doi.acm.org/10.1145/3102304.3109812.

[94] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252, IETF, June 2014.

[95] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition, 2008. ISBN 0136006329, 9780136006329.

[96] T. Suzuki. MQTT Bench. GitHub. URL https://github.com/takanorig/mqtt-bench. Accessed: 2017-12-01.

[97] The Apache Foundation. Spark framework. URL http://sparkjava.com/. Accessed: 2017-12-01.

[98] The Chromium Project. Filesystem autoupdates. URL https://www.chromium.org/chromium-os/chromiumos-design-docs/filesystem-autoupdate. Accessed: 2017-12-01.

[99] The OWASP Foundation. Owasp top 10, 2010. URL https://www.owasp.org/index.php/Top_10_2010-Main. Accessed: 2017-12-01.

[100] S. Thielman. Can we secure the Internet of Things in time to prevent another cyber-attack? The Guardian, October 2016. URL https://www.theguardian.com/technology/2016/oct/25/ddos-cyber-attack-dyn-internet-of-things. Accessed: 2017-12-01.

[101] H.-L. Truong, G. Copil, S. Dustdar, D.-H. Le, D. Moldovan, and S. Nastic. icomot – a toolset for managing iot cloud systems. In *Proceedings of the 2015 16th IEEE International Conference on Mobile Data Management - Volume 01*, MDM '15, pages 299–302, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-9972-9. doi: 10.1109/MDM.2015.65. URL http://dx.doi.org/10.1109/MDM.2015.65.

[102] Verizon Threat Research Advisory Center. Data breach digest 2017: Cybercrime case studies. URL http://www.verizonenterprise.com/verizon-insights-lab/data-breach-digest/2017/. Accessed: 2017-12-01.

[103] S. Wicki. The Rumprun Unikernel. *pkgSrcCon*, 2016.

[104] D. Williams. *Virtualization with Xen(tm): Including XenEnterprise, XenServer, and XenExpress*. Elsevier Science, 2007. ISBN 9780080553931.

[105] C. Wolf and E. Halter. *Virtualization: From the Desktop to the Enterprise*. Books for Professionals by Professionals. Apress, 2006. ISBN 9781430200277.