SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Unikernels as a Lightweight Compatibility Layer for Microkernels

Lukas Graber

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Unikernels as a Lightweight Compatibility Layer for Microkernels

Unikerne als leichtgewichtige Kompatibilitätsschicht für Mikrokerne

Author:	Lukas Graber
Supervisor:	Prof. Dr. Claudia Eckert
Advisors:	Johannes Wiesböck, M.Sc.
	Alexander Weidinger, M.Sc.
Submission Date:	22.05.2023

I confirm that this master’s thesis is my own work and I have documented all sources and material used.

Munich, 22.05.2023 Lukas Graber

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my thesis supervisor Prof. Dr. Claudia Eckert and the Chair of IT Security at TUM for giving me the opportunity to write my master's thesis at the Fraunhofer AISEC. I am extremely grateful to my two advisors Johannes Wiesböck and Alexander Weidinger for their help in finding a suitable research topic and their guidance throughout the project. Finally, I would like to thank my friends and family, in particular Ludovica, for their perseverance and support over the years and particularly over these last few months.

Abstract

Monolithic operating systems have a large Trusted Computing Base (TCB) as they run all OS services in kernel space. Microkernels on the other hand push most services to user space and only run the absolute minimum in the kernel itself. Capabilities are used by microkernels to enforce a very fine-grained access control scheme to kernel resources. This way, a high level of isolation between software components can be realized that is attractive for mixed criticality workloads. The focus on minimality comes at a high price for microkernels as they require a significant porting effort to support general-purpose workloads such as Linux/POSIX applications. To circumvent this problem, microkernels can be used as hypervisors to run the required workload in a virtual machine. This approach negates many microkernel-related benefits and prevents more widespread microkernel adoption. The thesis proposes a design that utilizes unikernels as a lightweight compatibility layer to run real-world workloads on top of microkernels. A thin interface was designed that abstracts over hardware resources with the help of basic microkernel mechanisms to provide necessary low-level functionality to the unikernel. This interface then allows unikernels to be executed as normal microkernel processes, highly isolated through the utilization of capabilities provided by the microkernel. The actual compatibility to Linux/POSIX applications is realized by the unikernel and mostly independent of the designed interface. A prototype system was implemented based on the *Unikraft* unikernel and the *seL4* microkernel and evaluated for several performance benchmarks. The results compare positively to unikernels on other platforms and confirm that unikernels can be used as a lightweight approach to provide compatibility for microkernels. The presented system is an attractive target for real-world mixed criticality use cases and aims to further expand widespread microkernel adoption.

Kurzfassung

Monolithische Betriebssysteme sind oft sehr umfangreich in ihrer Funktionalität, da sie sämtliche Dienste direkt im Betriebssystemkern ausführen. Mikrokerne führen viele dieser Dienste als normale Anwendungsprogramme aus, um die Menge an privilegierten Diensten im Betriebssystemkern minimal zu halten. Zugriffsrechte zu Mikrokern-Ressourcen sind über so genannte Capabilities geregelt. Diese ermöglichen eine hohe Isolation zwischen verschiedenen Software-Komponenten, welche speziell für Mixed Criticality Szenarien von Bedeutung ist. Dieser Fokus auf Minimalität in Mikrokernen hat Auswirkungen bezüglich der Nutzbarkeit, da sie generell einen großen Portierungsaufwand benötigen, um komplexere Anwendungen, wie zum Beispiel Linux-/POSIX-Applikationen, zu unterstützen. Mikrokerne können als Hypervisor verwendet werden, um die nötigen Applikationen zusammen mit einem separaten Betriebssystem in einer virtuellen Maschine auszuführen. Dieses Vorgehen verhindert grundlegende Mikrokernvorteile, welches den Einsatz von Mikrokernen einschränkt. Diese Arbeit stellt einen Entwurf vor, der es ermöglicht Unikerne als leichtgewichtige Kompatibilitätsschicht für Mikrokerne zu verwenden. Der Entwurf beruht auf einer schmalen Schnittstelle, welche die grundlegenden Mikrokernmechanismen einsetzt, um über die nötigen Hardware-Ressourcen zu abstrahieren und systemnahe Funktionalität dem Unikern zur Verfügung zu stellen. Die Unikerne werden als normale Anwendungsprozesse auf einem Mikrokern ausgeführt, wobei der Mikrokern eine hohe Isolation auf der Softwareebene ermöglicht. Die eigentliche Kompatibilität zu den Linux-/POSIX-Anwendungen wird durch den Unikern bereit gestellt und ist relativ unabhängig von der eigentlichen Schnittstelle zwischen Unikern und Mikrokern. Ein Prototypsystem wurde basierend auf dem Entwurf für den *Unikraft* Unikern und den *seL4* Mikrokern implementiert. Das System wurde eingehend evaluiert bezüglich mehrerer relevanter Performanz-Kennzahlen und zeigt vielversprechende Ergebnisse verglichen mit Unikernen auf anderen Platformen. Diese Erkenntnisse bestätigen, dass Unikerne eine leichtgewichtige Variante darstellen, um Kompatibilität für Mikrokerne zu ermöglichen. Die Prototypimplementierung ist ein aussichtsreicher Kandidat für mehrere Mixed Criticality Szenarien und zielt damit darauf ab den allgemeinen Einsatz von Mikrokernen auszuweiten.

Contents

1 Introduction

Traditional operating systems (OS) such as Linux and Windows are monolithic software constructs that run the entire operating system functionality in kernel space. These operating systems typically provide a rich ecosystem of applications and have a big community of developers that maintain and improve this ecosystem. At the same time, running all OS services in kernel space leads to millions of lines of code being executed at the highest privilege level. This is a big security concern since one bug in the kernel can potentially compromise the entire system [1].

Microkernels were developed as a response to the growing complexity in monolithic kernels. A microkernel is the minimum amount of software that can provide the mechanisms needed to implement an operating system. Typical OS services such as device drivers, protocol stacks and filesystems are all run in user space. Consequently, microkernels have a much smaller Trusted Computing Base (TCB) than their monolithic counterparts. Modern microkernels often use capabilities to enforce a very fine-grained access control over system resources [2]. Despite all the microkernel-based benefits, implementing services directly on top of the microkernel itself can be a challenging task since the microkernel only provides a minimal set of OS functionality. This significantly reduces the usability of microkernels and prevents widespread microkernel adoption. To tackle this problem, real-world workloads such as applications known from popular operating systems need to be supported by the microkernel.

We look specifically at applications developed for UNIX-like operating systems. These applications typically follow the POSIX standard to ensure application portability across operating systems. Linux is arguably the most popular free and open-source UNIX-based operating system kernel. It is the predominant player in nearly every industry and deployed on a wide variety of computing systems, such as embedded devices, mobile devices, personal computers, servers, mainframes and supercomputers [3]. By focusing on applications developed specifically for UNIX-based operating systems, i.e. Linux/POSIX applications, we cover a realistic subset of available software encountered in real-world systems.

Manually porting these applications to the microkernel is unrealistic as the microkernel only provides a minimal set of OS functionality. Instead, the microkernel can be used as a hypervisor to run the application together with a corresponding (monolithic) operating system in a Virtual Machine (VM) [4]. This approach solves the compatibility problem of microkernels but relies on (hardware) virtualization techniques to operate the microkernel as a hypervisor. Since an entire operating system must be included in the virtual machine, this approach is also extremely resource-heavy in terms of memory utilization. Considering the fact that microkernels are often executed on small form factor hardware [5], we run into resource limitations for the virtualization case.

Virtual Machines provide a good level of security and isolation but come with a high overhead in terms of performance, memory consumption and image size. To tackle the described problems, the concept of containers was introduced. Containers are sandboxed applications that share the same underlying OS kernel. Compared to virtual machines, these containers exhibit a much lower resource utilization. At the same time, containers lack when it comes to isolation, as all containers share the same underlying kernel. There is a big incentive to find a solution that on the one hand offers the security and isolation properties of virtual machines but at the same time exhibits the minimal resource overhead that containers provide. Unikernels are aimed to bridge the gap between virtual machines and containers to provide the best of both worlds [6].

Unikernels are the most recent incarnation of library operating systems (libOS) [7], an OS construct pioneered by exokernels in the 1990s [8]. In a libOS, protection boundaries are pushed to the lowest hardware layers, enabling applications to access hardware resources directly without having to make repeated privilege transitions to move data between user space and kernel space. Unikernels revisit the concept of library operating systems with a new approach based on virtualization [9]. An application is (statically) linked together with a library of operating system components into a single flat address space, creating a standalone binary image that is bootable directly on (virtual) hardware [10]. This makes unikernels an optimal fit for cloud environments, where often only a single application is run inside a VM. Of specific interest to us are backwards-compatible unikernels, specifically the ones that target Linux/POSIX applications [7, 10].

This type of POSIX-like unikernels shall be used to provide a lightweight compatibility layer to real-world workloads on top of microkernels. Microkernels are often executed in environments with small form factor hardware that might not come with the necessary hardware virtualization technologies. The idea is to run unikernels as microkernel processes in user space and apply a fine-grained access control scheme based on capabilities to enforce resource isolation. This concept was first explored for the

Rumprun unikernel port on the seL4 microkernel [11, 12]. The thesis builds on these findings and extends research to unikernels and microkernels in general. Most notably, the thesis provides the following research contributions:

1. Design an interface to enable running a unikernel on top of a microkernel.
2. Optimize the design in terms of performance and resource requirements.
3. Select a suitable combination of microkernel and unikernel for a prototype implementation.
4. Evaluate possible use cases using real-world workloads.

In the following chapters, we will first present related work in chapter 2 and necessary background information in chapter 3. Chapter 4 identifies security-related threats for monolithic kernels that our design and prototype implementation shall protect against. We elaborate on certain design requirements in chapter 5, before providing a general interface between unikernel and microkernel in chapter 6. Chapter 7 describes the prototype implementation that utilizes the *Unikraft* unikernel in order to provide support for Linux/POSIX applications on top of the *seL4* microkernel. Several real-world use cases for the prototype system are presented in chapter 9. The design and prototype implementation are critically reflected on in chapter 10 and several potential future projects are shown in chapter 11. Finally, a short summary and some concluding remarks are stated in chapter 12.

2 Related Work

The goal of the project is to use unikernels as a lightweight compatibility layer for Linux/POSIX applications on top of microkernels as normal user space processes. In this chapter, I present related work and explain how my approach distinguishes from existing research.

2.1 Linux/POSIX Compatibility Systems

Tsai et al. [13] present several major Linux system APIs that must be supported by a Linux compatibility system. A similar observation was performed by Atlidakis et al. [14] for POSIX applications and how modern workloads relate to POSIX. Olivier et al. [15, 16] further present the necessary mechanisms to support Linux binary compatibility. This information helps us in understanding how unikernels can provide Linux/POSIX compatibility and the limitations of these approaches for our microkernel-based design.

2.2 Unikernels

2.2.1 Language-based Unikernels

Language-based unikernels are library OS environments that are tied to a specific programming language runtime and libraries, such as MirageOS [9, 17] for OCaml, IncludeOS [18] for C++, Clive [19] for Go, HalVM [20] for Haskell, runtime.js [21] for JavaScript, LING [22] for Erlang, ClickOS [23] for Click router rules, and RustyHermit [24] for Rust. This type of unikernel is not further considered as they do not aim for Linux/POSIX compatibility.

2.2.2 Backwards-Compatible Unikernels

Several unikernels and library operating systems aim specifically at providing backwards compatibility to applications developed for monolithic operating systems. This type of unikernel shall be used to provide a lightweight compatibility layer to legacy applications on microkernels.

The majority of unikernels are implemented from scratch, optimized for the unikernel environment. *Gramine* [25], formerly known as *Graphene* [26], is a libOS that runs Linux multi-process applications with Intel SGX support. *OSv* [8] is a modular unikernel designed to run single unmodified Linux applications on top of a hypervisor. Olivier et al. [15, 16] present *HermiTux*, a Linux binary compatible unikernel that is an extension of the *HermitCore* [27] unikernel. Some research [28] replicated the work provided by *HermiTux* to the *RustyHermit* [24] unikernel written in Rust. Another POSIX-compliant and Linux binary compatible unikernel was presented by Kuenzer et al. [29] with the *Unikraft* unikernel.

Another approach is to take an existing monolithic kernel and optimize the environment to resemble a unikernel. *Rump* kernels [30] are subsets of the NetBSD kernel and can be used to bring legacy POSIX support to applications without having to re-engineer or implement large code bases. The *Rumprun* unikernel is one specific rump kernel configuration [11]. Kuo et al. [7] present *Lupine Linux*, effectively configuring a regular Linux kernel to behave as a unikernel. *Unikernel Linux (UKL)* [10, 31] builds on the previous findings and transforms the Linux kernel into an actual unikernel. Backwards-compatibility was also achieved for Windows applications with *Drawbridge* [32], which runs native Windows applications as a libOS.

2.2.3 Unikernels as Processes

Instead of running unikernels in a virtual environment on top of hypervisors, unikernels can also be executed as a normal user space process on top of a monolithic operating system. Unikernels that target the Linux kernel itself include *Gramine* [25, 26], *Nabla* containers [33] and the *Linuxu* platform in *Unikraft* [34]. Williams et al. [33] describe the mechanisms needed to run unikernels in general as user space processes on top of monolithic operating systems. These unikernels apply several techniques to improve the isolation level, including syscall whitelisting [33] or secure enclave technology [25, 26]. The thesis revisits this idea of running unikernels as user space processes, but utilizes a microkernel as the underlying platform. Our microkernel-based approach uses capabilities to enforce a high level of isolation for system resources.

2.3 Microkernels

Odun-Ayo et al. [35] present several popular microkernels and come to the conclusion that their overall implementation does not differ substantially. Members of the L4 microkernel family are considered to be state-of-the-art when it comes to microkernel research. They all originate from the same L4 microkernel and have a shared history of ongoing design evolution [36]. Consequently, these microkernels are often similar in their implementation and follow common design guidelines [36, 37]. Vemuri et al. [38] further present measures to improve the security of microkernel-based operating systems. Our design assumptions are heavily based on the microkernel characteristics outlined by these documents.

2.3.1 Linux/POSIX on Microkernels

Several microkernel projects aim to provide a POSIX compatibility layer in order to increase the usability of microkernels. Examples include the *libsel4osapi* library [39] for the seL4 microkernel or libc support (*uClibc* [40]) in the *L4Re* operating system framework for the Fiasco.OC microkernel [41]. There also exists the *Device Driver Environment (DDE)* that utilizes multiple L4 threads in order to provide a Linux environment to run (native) Linux drivers [42]. *GNU Hurd* is a UNIX-compatible microkernel operating system that is implemented on top of the *Mach 3.0* microkernel [35]. It represents a second-generation microkernel and consists of the order of 300 kLoC.

Another common approach utilized in microkernels is virtualization: *L4Linux* is a paravirtualized Linux kernel that allows to reuse Linux applications and components on L4/Fiasco.OC [43]. The *NOVA* micro-hypervisor enables to run unmodified commodity operating systems through the utilization of virtualization techniques on a small TCB [35, 43]. *Karma VMM* is implemented on the Fiasco microkernel and utilizes hardware virtualization to simplify paravirtualization and reduce the TCB of VMMs [43]. The seL4 microkernel can also be used as a hypervisor to run virtual machines on seL4, and inside the virtual machine a mainstream OS such as Linux [2, 44].

The Rumprun unikernel was ported to the seL4 microkernel to provide a compatibility layer for POSIX applications [11, 4, 12]. The unikernel is executed as a normal user space process and utilizes capabilities to enforce a high level of isolation for system resources. The thesis builds on these ideas and extends research to unikernels and microkernels in general. The focus is also more on Linux application support, compared to NetBSD components.

3 Background Material

3.1 OS Concepts

3.1.1 OS Kernel Design

Monolithic Kernel Traditional operating systems such as Linux and Windows are monolithic software constructs that run all operating system services entirely in kernel space. As a result, modern operating systems have a large TCB, including millions of lines of privileged code. These monolithic kernels became increasingly complex and difficult to maintain [1]. There are estimations that the Linux kernel alone contains tens of thousands of bugs [2].

Microkernel Microkernels were developed as a response to the growing complexity in monolithic kernels. A microkernel is the minimum amount of software that can provide the mechanisms needed to implement an operating system. As a result, most OS services such as device drivers, protocol stacks and filesystems are moved to user space. Consequently, microkernels are much smaller than their monolithic counterparts. This reduces the complexity of the kernel resulting in easier maintainability and a lower number of bugs, ultimately shrinking the TCB [1, 38].

Virtual Machine/Exokernel Virtual Machines and exokernels typically do not provide an extra hardware abstraction layer but duplicate or partition the hardware resources. This way, multiple operating systems can run next to each other with the illusion of having a private machine. A Virtual Machine Monitor (VMM)/hypervisor is responsible for the protection of resources and multiplexing hardware requests [38].

3.1.2 High-Level OS Services

Scheduling The scheduler is the part of the OS that is responsible for assigning available tasks to the CPU. Scheduling allows tasks to run seemingly simultaneously (multitasking). In general, there are two styles of scheduling: cooperative and preemptive scheduling. In a cooperative scheduling scheme, the tasks manage their own lifecycle and the scheduler's job is only to assign tasks to any worker that is free. In preemptive scheduling, the scheduler actively controls the amount of time a task is scheduled on a specific CPU [45, 46, 47].

Memory Management Memory Management provides a way for programs to allocate and free memory. It further includes hardware paging mechanisms to manage address spaces between different processes [48, 49].

Inter-Process Communication (IPC) Processes communicate with each other through IPC mechanisms in order to coordinate their activities. Typical IPC mechanisms applied in operating systems include Message Passing, Remote Procedure Call (RPC), Shared Memory, Signals, Socket, UNIX Pipes and Synchronization Primitives such as Mutexes, Semaphores and Spinlocks [50, 51].

Networking Networking is a fundamental building block in any modern computing system. Applications typically hook into a TCP/IP networking stack through the socket API. The TCP/IP stack itself uses a link layer, that is provided by a network device, e.g. an ethernet card, to process packets of network protocols such as IP, ARP, TCP, and UDP [52, 53].

Filesystems A filesystem provides a generalized structure over persistent storage, allowing the low-level structure of the actual hardware devices to be abstracted away. Generally speaking, the goal of a filesystem is allowing logical groups of data to be organized into files, which can be manipulated as a unit [54].

Device Drivers Device Drivers are the operating system's way to abstract over I/O devices. They make use of memory-related mechanisms such as I/O memory (MMIO), I/O ports and DMA regions as well as device interrupts to facilitate polling- or interrupt-driven communication with the actual device hardware [55, 56].

3.1.3 Low-Level OS services

System Calls Operating Systems expose a handful of functions to user space, so called kernel services. System calls are essentially a user space request to these kernel services. The standard C library (libc) provides wrapper code around system calls, i.e. the application typically never issues syscalls directly but goes through the libc. There are several different ways for user programs to make system calls and the low-level instructions for making a syscall vary among CPU architectures. On the *x86* architecture, we distinguish between legacy system calls and fast system calls: The first one is generated through the *int 0x80* instruction as a software interrupt, while fast syscalls rely on special instructions that allow for more efficient syscall handling [57]. We specifically focus on the *x86_64* architecture that supports fast system calls through the *syscall* and *sysret* instructions. Before invoking a system call, the user space program must prepare common registers that serve as arguments to the kernel services. Most importantly, it needs to specify which kernel service shall be invoked. The kernel manages a so called *syscall table* that associates a specific number with a corresponding kernel service function. When the *syscall* instruction is triggered, the processor performs a privilege switch to transition from user to kernel space. Subsequently, the kernel takes over and jumps to a system call handler function that is registered in the *MSR LSTAR* hardware register during startup. MSR registers are a collection of registers that allow configuration of OS-relevant features. They are accessed using special privileged instructions such as *RDMSR* / *WRMSR* [58, 59, 60]. The kernel's syscall handler function takes the syscall number and uses it as an index into the *syscall table* to execute the corresponding kernel service. After processing the system call, the result is placed in a specific register and we transition back to user space through the *sysret* instruction. Afterwards, program flow returns to the original caller [61].

Thread-Local Storage (TLS) Besides data on the stack, threads require thread-local storage (TLS). TLS is a storage model for variables that allows each thread to have a unique copy of a global variable. We distinguish between explicit TLS and implicit TLS [62]. The former one is used by the POSIX thread interface and allows objects to be stored differently for each thread. Implicit TLS on the other hand is a standardized TLS format for different architectures and implemented in many modern operating systems [63]. Operating systems typically reuse the *FS* segment register in the *x86* architecture for TLS handling. The address stored at this segment register points to the start of the TLS area of the currently executing thread. Thread-local variables can be referenced relative to the *FS* base address [64]. The TLS base must become part of the thread context to be included on every context switch by the scheduler. The *FS* base address can be changed through the *FSBase MSR* register [65]. User space runtimes and threading libraries must manage TLS on their own for proper thread handling. This is typically realized through a separate system call, e.g. *arch_prctl* in Linux [64]. New Intel and AMD CPUs introduced the *FSGSBASE* instruction family that allows access to *FS/GS* segment registers directly from user space. The latter option is preferred for user space since it avoids the overhead of a separate privilege switch associated with system calls [64]. The *FSGSBASE* instructions must be explicitly enabled by the kernel through the *Control Register CR4* during system startup [66, 64].

3.2 Linux/POSIX

3.2.1 Linux vs POSIX

The Portable Operating System Interface (POSIX) comprises a set of OS abstractions that aid application portability across UNIX-based operating systems. The standard was developed in the 80s and its basic set of abstractions has remained largely unchanged. At the same time applications have evolved drastically since its conception and are no longer being written to standardized POSIX interfaces [14]. POSIX is ultimately limited to basic OS abstractions and therefore does not provide support for graphics or audio applications [40].

Linux is probably the most popular UNIX-like OS and comes with a big user base [67]. Consequently, many modern applications are developed for Linux. Linux attempts to comply with the POSIX standard, but diverges from it in certain areas whenever it makes sense to the kernel developers [68]. This is the case for more powerful APIs to support multimedia applications, games and rich GUIs [40].

3.2.2 Linux Kernel Architecture

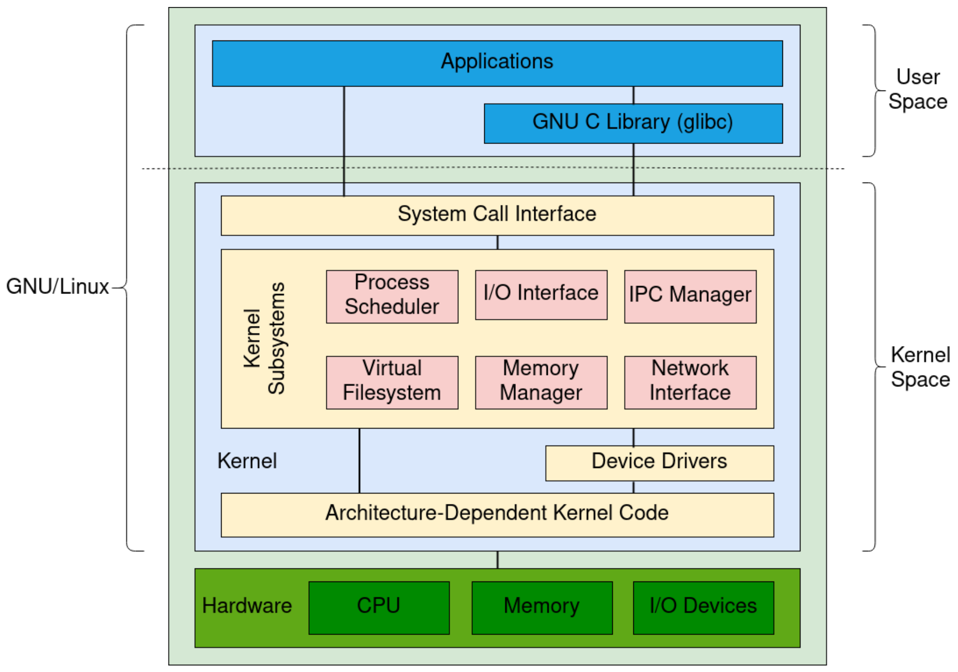


Figure 3.1: The Linux Software Stack.

The Linux kernel architecture is depicted in Figure 3.1. It comprises of several subsystems that implement OS-specific behavior such as Memory Management, Scheduling, I/O Handling, Filesystem, Network Stacks and various IPC mechanisms. These subsystems are heavily dependent on each other due to the non-modularity of the Linux kernel. They interact with hardware resources through device drivers or architecture-dependent code [69]. The Linux kernel exposes several kernel services to user space through the System Call Interface (SCI) and the GNU C library (glibc). Together, this is typically termed the Linux API. The Linux API has been kept stable over time which guarantees the portability of source code [67].

3.3 Microkernels

3.3.1 History

The history of microkernels had its start in the mid 80s with the Mach microkernel, generally referred to as the first ever microkernel. Research around these first-generation microkernels eventually stagnated due to overall bad IPC performance which resulted in a trend of moving core services back into the kernel [36, 70].

These IPC performance issues were firstly addressed by Jochen Liedtke's L4 microkernel in the mid 1990s. It was the successor of an earlier microkernel called the L3 microkernel which suffered from high IPC costs similarly to other microkernels of that time. Liedtke demonstrated with his new design an order-of-magnitude improvement of IPC costs, proving that microkernel-based systems can be fast [36, 71]. These second-generation microkernels focused specifically on performance and minimality, but had drawbacks in terms of access control and resource accounting [70].

The initial works of Liedtke and his L4 microkernel resulted in a long history of ongoing design evolution with multiple ABI revisions and from-scratch implementations [36]. Third-generation microkernels put an increasing focus on topics such as capability-based access control, formal verification and virtualization [70]. The L4 microkernel family is at the forefront of microkernel research and represents the state-of-the-art for third-generation microkernels [36, 1]. The most prominent members of the L4 microkernel family are the Fiasco.OC and the seL4 microkernel [36].

Odun-Ayo et al. [35] present additional microkernels and microkernel-based operating systems to show a more complete picture of available microkernel systems. They come to the conclusion that the overall implementation details of different microkernels does not differ substantially.

3.3.2 Characteristics

Minimality & Generality Microkernels move most functionality out of the kernel into user space. As a result, microkernels are mostly policy-free and only tolerate in the kernel what is deemed absolutely necessary. At the same time, this focus on minimality shall not prevent the microkernel from running general-purpose workloads [36, 37].

User-Level Servers As a result of the minimality principle, typical OS services such as device drivers, protocol stacks and filesystems are all implemented in user space. These user-level servers expose their services through common IPC mechanisms that can be used by other user space programs [1, 2, 36].

Capability-based Access Control A capability is a unique, unforgeable token that gives the possessor permission to access an entity or object in the system. It represents a pointer with access rights [72]. Third-generation microkernels utilize capabilities to enforce very fine-grained access control to system resources. The microkernel defines several data structures, also referred to as kernel objects, that abstract over common hardware resources. The kernel exposes a handful of services around these kernel objects to user space as the kernel API. Each invocation of a kernel service must be protected by a capability. As a result, user space must provide an appropriate capability to access a specific kernel service. This allows for very fine-grained access control and realizes a high level of resource isolation. Capability-based security is an enabler of the Principle of Least Authority (POLA) and serves as the fundamental mechanism to enforce isolation in the microkernel [37].

Root Task Many modern microkernels, specifically representatives of the L4 microkernel family, include an initial user-level task, also called the root task. It is given full access rights to all resources left over once the kernel has booted up and typically includes physical resources such as memory, I/O ports on x86, and interrupts. The root task is responsible for setting up other user-level tasks and pass necessary access rights to these tasks to build a complete system. These access rights are controlled by capabilities [5].

3.4 Unikernels

3.4.1 Characteristics

A single application is (statically) linked together with a library of operating system components into a single flat address space, creating a standalone binary image that is bootable directly on (virtual) hardware [10]. Due to the modular structure of the libOS design, the unikernel can be specialized for a specific application context [10]. The resulting images typically exhibit a small binary size, reduced memory consumption and provide good performance due to the specialization [73]. Unikernels are meant to run single address space applications, i.e. only a single process is supported [74]. They typically run at a single protection level and remove the user-to-kernel space separation. As a result, there is no need for expensive privilege switches [7].

3.4.2 Unikernel Types

We are specifically interested in unikernels that provide backwards-compatibility with legacy applications developed for monolithic operating systems. Our main focus lays on UNIX-based operating systems, specifically Linux. Applications developed for UNIX-based operating systems follow the POSIX standard. Unikernels aiming for POSIX compatibility are often called POSIX-like unikernels. Based on their approach to provide backwards compatibility, we distinguish between *clean-slate* and *strip-down* unikernels.

Clean-Slate Unikernels A modular OS is created from scratch and optimized for unikernel properties such as modularity and specialization. To provide compatibility to Linux/POSIX applications, a separate compatibility layer is constructed [75]. Examples of clean-slate unikernels include *Gramine*, *OSv*, *HermiTux*, and *Unikraft*.

Strip-Down Unikernels An existing UNIX-based OS is taken and optimized for unikernel-like properties through kernel specialization and syscall elimination [75, 7]. These unikernels provide Linux/POSIX compatibility by default but typically can not reach the same level of modularity and minimality as clean-slate variants [75]. Additionally, platform integration becomes more complex as these unikernels typically pose extensive dependencies on the underlying host [76]. Examples of strip-down unikernels include *Rumprun*, *Lupine Linux*, and *Unikernel Linux (UKL)*. Raza et al. [31] additionally distinguishes strip-down unikernels into forks of general-purpose OSes and incremental systems.

3.4.3 Architecture

Although implementation details might vary for different unikernels, on a conceptual level most Linux-/POSIX-compatible unikernels follow a similar structure. The unikernel architecture shown in Figure 3.2 serves as a blueprint for later design stages. A Linux/POSIX application is compiled into a single binary together with the unikernel

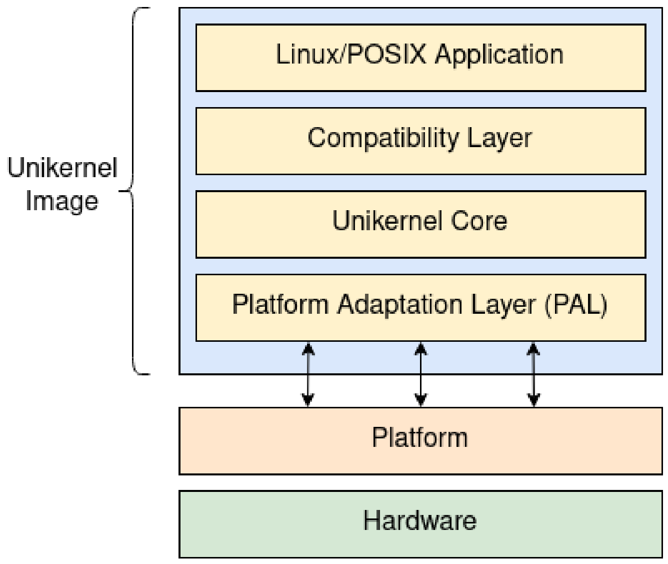


Figure 3.2: Unikernel Architecture.

sources. Unikernels implement a compatibility layer in order to map Linux/POSIX concepts to unikernel-internal implementations. In the core layer, unikernels provide their own lightweight implementations of common OS services. They often borrow existing implementations from other operating systems [29, 8]. Compared to the Linux kernel, the implemented services are often optimized for the unikernel environment and exhibit less interdependencies between each other. This is realized through the libOS design that enforces high modularity in the unikernel. A unikernel requires a handful of low-level functionality that must be provided by an underlying execution platform. Unikernels typically push the abstraction layer down to the hardware level to allow for easy platform integration [76]. For this purpose, they define a thin Platform Adaptation Layer (PAL) that abstracts over basic hardware resources. Unikernel platforms include hypervisors (Xen, KVM, QEMU and Firecracker [29]), unikernel monitors (Solo5, uHyve [33]), physical hardware (Rumprun [77]), and even Linux itself [34, 25].

4 Threat Model

Attacker Assumptions We assume that an attacker has taken complete control over a user space process without superuser privileges. The user space process is assumed to interact with OS services such as device drivers, filesystem and protocol stack, through the kernel's system call API. The attacker can further execute any code that is mapped into the address space of the compromised process. This can be exploited through Return-Oriented Programming (ROP) and Remote Code Execution (RCE).

Kernel Assumptions The system operates on a monolithic kernel that runs the majority of OS services in kernel space. We assume that the kernel is trusted and was implemented without any harmful intentions. The kernel is expected to be in a trusted and consistent state after booting. However, we assume that a monolithic kernel can not be formally verified [2] and therefore is expected to contain bugs [78, 79, 80].

Trusted Components It is assumed that hardware, firmware and any involved bootloaders can be trusted. The attacker shall not gain physical access to the hardware. We further exclude any side channels or microarchitectural attacks from the following security considerations.

4.1 Threat Analysis

4.1.1 Threat Vector 1: Privilege Escalation from user to kernel space

In a monolithic operating system, common OS functionality such as device drivers, protocol stacks and filesystems are all run in kernel space. A bug in any of these kernel services can potentially corrupt the entire system [81]. An attacker can exploit such a bug from the compromised user space process in order to let the vulnerable OS service execute some kernel-related functionality that requires kernel privileges. This way, the attacker effectively caused a privilege escalation from user to kernel space.

In the past, there have been explicit privilege escalation kernel exploits in the Linux kernel that utilize a bug in a kernel module. Specifically device driver modules are popular targets for attackers as they are error-prone and often contain bugs [79]. A well studied problem are stack overflows in the kernel. The buggy driver module might expose a function that gives control over an unbounded buffer on the stack to the user/attacker [82]. The oldest and most popular attack for this type of vulnerability is *Return-to-user* (*ret2usr*), which leverages the fact that user space processes cannot access kernel space, but kernel space can access user space [83]. A certain payload, e.g. shellcode, ROP payload or tampered-with data structures, is placed in user space of the compromised process. The vulnerability in the device driver module is exploited by the attacker to redirect program flow to the placed payload in user space. The payload is effectively executed in user space with ring 0 privilege. The *ret2usr* attack is a kernel exploitation technique that facilitates privilege escalation and allows for arbitrary code execution [82, 84, 85]. Modern processors/operating systems apply several mitigation techniques for this type of vulnerability. It was shown that all these security controls can be bypassed [86, 87]:

- **Kernel Stack Cookies/Canaries**: Values that are added to binaries during compilation to protect critical stack values like the return pointer against buffer overflow attacks [88]. The attacker can simply leak the stack cookie since the driver module allows arbitrary stack read [82].
- **Supervisor Mode Execution Protection (SMEP)**: This feature marks all the userland pages in the page table as non-executable when the process is in kernel mode [82]. To circumvent SMEP, the attacker can construct a kernel ROP chain that does not rely on payload from user space. A stack pivot can be used in case overflow is limited on the stack [89].
- **Supervisor Mode Access Prevention (SMAP)**: This feature marks all the userland pages in the page table as non-accessible when the process is in kernel mode [82]. Similar to SMEP, the attacker can bypass SMAP by only relying on kernel payloads. Stack pivots are not possible for SMAP; in these cases a *ret2dir*-style attack might help [90].
- **Kernel Page Table Isolation (KPTI)**: This feature separates the user space and kernel space page tables entirely. The user space page table contains a copy of user space and a minimal set of kernel space addresses [82]. KPTI can be circumvented through techniques such as KPTI trampoline or signal handler [89].
- **Kernel Address Space Layout Randomization (KASLR)**: It randomizes the base address where the kernel is loaded each time the system is booted [82]. For normal KASLR, a single leak of a *.text* address is sufficient [86]. KASLR can also be applied on a per-function level granularity (FG-KASLR). This feature can still be exploited, since certain regions in the kernel never get randomized [86, 91].

As demonstrated all kernel security mitigations can be bypassed through ROP and some additional technique. This ultimately shows that although mitigations can be developed for specific attacks, the underlying problem for monolithic kernels persists: Monolithic kernels have a large TCB and compromising a single bug in the kernel can potentially compromise the entire system [81].

4.1.2 Threat Vector 2: Privilege Escalation from a low-privileged to a high-privileged process

An OS does not only distinguish between user and kernel space, but user space processes themselves can have different access rights associated with them. Monolithic operating systems typically have one privileged user account that has more privileges than ordinary users, e.g. the root account in UNIX/Linux and the administrator account under Windows. This superuser account effectively bypasses kernel checks and has the highest access rights in the system: it can read and write any files on the system and effectively perform operations as any other user. Programs that require additional access rights to system resources are usually owned by the root user, representing high-privileged processes at runtime. Programs owned by other accounts represent low-privileged processes [92]. In Linux, it is generally possible to configure the system in a more fine-grained way through Linux capabilities. In the majority of deployed Linux systems, the system is still configured to give extensive privileges to the superuser account [93].

Having full control over an unprivileged user space process, the attacker might target subsequently another process on the system that runs with superuser privileges. We assume the privileged process to be vulnerable, e.g. because of a zero-day vulnerability or an access right misconfiguration [94]. The attacker might use this vulnerability in order to compromise the high-privileged process, effectively elevating its privileges to root. Some applications consist of multiple processes with different access rights at runtime. One specific example are web servers, the two most popular ones being NGINX and Apache [95]. Web servers usually hook into the socket API to interact with the kernel's network stack. Web servers often require root privileges, e.g. to open certain ports (< 1024). Consequently, part of the web server must run with superuser privileges. For this purpose, they have a parent process that is owned by root and starts worker nodes with less privileges [96, 97, 98]. Assuming a zero-day vulnerability or misconfiguration in the web server, the attacker can gain full control over the parent process and thus escalate its privileges to root.

The attack scope can be extended for a process with root privileges. An attacker can utilize the *insmod* program to insert an attacker-controlled module into the Linux kernel [99]. This way, the attacker effectively caused a privilege escalation from user to kernel space. Common techniques to protect against exploiting the *insmod* feature include *Kernel Lockdown* and *Kernel Module Signing*. The *Kernel Lockdown* feature is designed to prevent both direct and indirect access to a running kernel image, but is disabled by default as it causes compatibility issues for some user space applications [100, 101, 102]. *Kernel Module Signing* is a kernel feature that when enabled only loads kernel modules that are digitally signed with the proper key. By default, a permissive approach is utilized that only applies a signature verification for kernel modules that provide a valid signature. For kernel modules that have no signature, signature verification is ignored [103]. As a result, we can assume that the root user can use *insmod* to load a module into the kernel. Therefore, the same holds true for an attacker that runs as root.

5 Analysis

A unikernel shall be used as a lightweight compatibility layer to run Linux/POSIX applications on top of a microkernel. For this purpose, an interface between a unikernel and a microkernel must be defined that fulfills the following design requirements:

- **Security**: The design shall protect against the threats identified by the threat model in chapter 4.
- **Compatibility**: The design must allow Linux/POSIX applications to execute on top of a microkernel.
- **Lightweight Integration**: The design should integrate the unikernel as a normal microkernel user space process without relying on additional hardware features.

Before presenting the actual design, we take a closer look at each of these design considerations.

5.1 Security Considerations

We take a bottom-up approach to the security problem to understand the reasons behind the identified threats presented in chapter 4. Consequently, we argument how microkernels provide the necessary mechanisms to prevent those threats.

5.1.1 Threat Vector 1: Privilege Escalation from user to kernel space

In threat vector 1, the attacker is able to cause a privilege escalation from user to kernel space by exploiting a bug in one of the OS components that runs in kernel space. The underlying problem is that monolithic kernels have a large TCB and run all OS services in the kernel itself; one bug in the kernel can therefore potentially compromise the entire system. Microkernels are a natural fit for this problem as they only run the minimum amount of OS services in the kernel and push the rest to user space. As a consequence, the kernel's attack surface shrinks considerably and exhibits a lower number of bugs. Exploiting vulnerable OS components that previously were running in kernel space, become less critical since the attacker is now restricted to user space.

5.1.2 Threat Vector 2: Privilege Escalation from a low-privileged to a high-privileged process

In threat vector 2, an attacker compromises another user space component that runs with root privileges. As root, the attacker gains extensive control over the system and might have an easier time to subsequently break into kernel space. The underlying problem is that monolithic kernels often implement a binary view on access control. Modern microkernels utilize capability-based security to enforce a very fine-grained access control to kernel resources. This way, they enforce POLA so that any component in the system only ever has the minimal privileges it needs to perform its job [2, 37]. Compromising a user space component in such a system only gives the attacker the rights associated with this component.

5.1.3 Summary

The underlying design of monolithic kernels leads to the threats identified by the threat model. As was explained, microkernels provide the necessary mechanisms to mitigate (threat vector 1) or at least drastically restrict (threat vector 2) the identified threats in the threat model. Thus, microkernels are used as the underlying technology for the system design.

5.2 Compatibility Considerations

We apply a top-down approach to the compatibility problem. We first take a look at what constitutes a general Linux/POSIX compatibility layer. Unikernels must implement their own compatibility layer to support these applications. We categorize different approaches typically applied in unikernels to better understand the limitations we can expect for our microkernel-based design.

5.2.1 Linux/POSIX compatibility systems

Tsai et al. [13] investigate the Linux API usage across all applications and libraries for a modern Ubuntu Linux distribution. The paper gives useful guidelines on what a compatibility system needs to provide in order to support a sufficient amount of Linux applications. They investigate the importance of several system APIs including system calls, pseudo-files, vDSO regions and libc functions. A similar study for the POSIX standard was performed by Atlidakis et al. [14].

System Calls One metric for compatibility is the number of supported system calls. Systems that claim Linux compatibility typically support only a fraction of system calls available in Linux. But plain system call counts do not accurately estimate the actual Linux application support. One reason for this is the fact that not all system APIs are equally important. Tsai et al. [13] identify 320 system calls for Linux kernel version 3.19. They show that 224 of those 320 system calls are indispensable, i.e. they are required by at least one application on every distribution. They further demonstrate that around 145 syscalls are needed to statistically support around half of all Linux applications and libraries. Some system calls such as *ioctl*, *fcntl*, and *prctl* essentially export a secondary system call table, using the first argument as an operation code. These vectored system calls significantly expand the system API, increasing the effort to realize full API compatibility [13]. Other sources claim that as little as 160 system calls are sufficient to run many complex Linux applications [104]. They argue that whenever a system call is missing, it is sufficient to either stub or fake an implementation. The key insight is that applications are resilient to a significant portion of unimplemented syscalls. Therefore, the actual number of required syscalls to correctly run Linux applications is significantly lower than what is expected by the output of the static analysis performed by Tsai et al. [13].

vDSO and vsyscall Invoking a system call is an expensive operation in Linux, because the processor must perform privilege switches on every invocation. The vDSO (virtual Dynamic Shared Object) mechanism represents a small shared library that the kernel automatically maps into the address space of all user space applications. This library allows a handful of regularly used system calls to be accessible from user space as normal function calls to avoid expensive privilege switches. We assume that calls to vDSO are mostly performed by the C library. This way, applications do not need to know the details of its implementation. A predecessor of vDSO is the virtual system call (vsyscall) mechanism that nowadays is considered to be obsolete [105, 106, 15].

Pseudo-Files In addition to the main system call table, Linux exports many additional APIs through pseudo-filesystems, such as */proc*, */dev*, and */sys*. Pseudo-filesystems are not backed by disk, but rather export the contents of kernel data structures to an application or administrator as if they were stored in a file. Most of these pseudo-files are used exclusively on the command line or in scripts, but some are routinely used by applications [13].

Libc Most applications do not interact with the kernel directly via system calls, but instead utilize more user-friendly APIs in the standard C library (libc). To support

Linux applications, the compatibility system therefore needs to provide their own libc implementation that emulates the glibc functionality of Linux [13].

POSIX POSIX defines several APIs for signals, streams, IPC, real-time, threads and sockets. To comply with the POSIX standard, a system must provide additional libraries that provide the necessary POSIX functionality [14].

5.2.2 Unikernels as Linux/POSIX Compatibility Systems

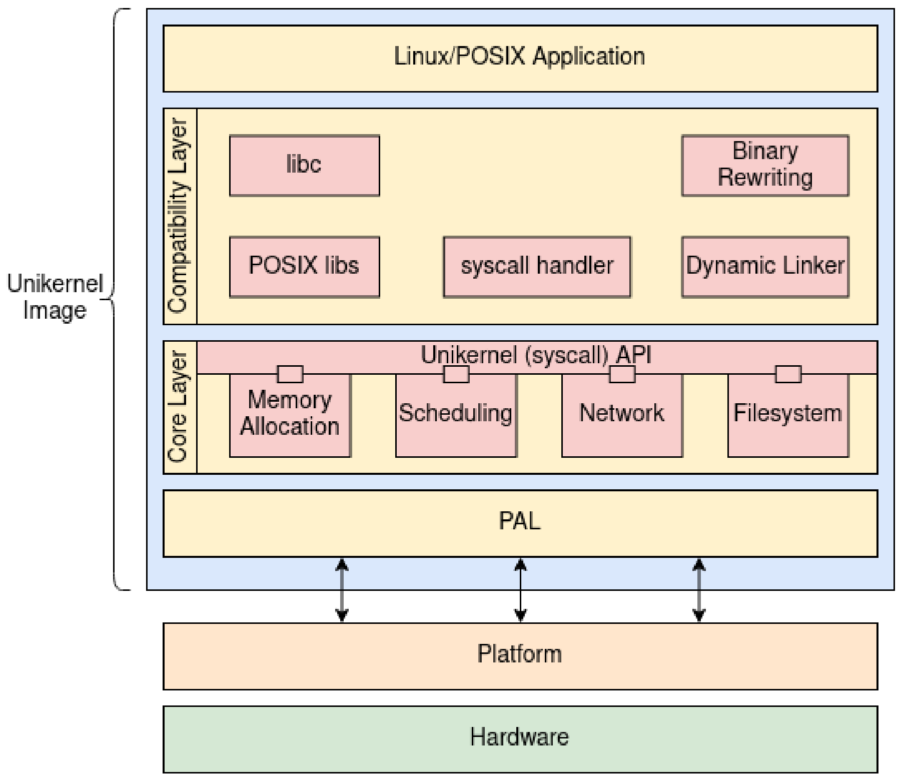


Figure 5.1: Unikernel Compatibility Layer.

In Figure 5.1, the general unikernel architecture is depicted and highlights how unikernels implement compatibility to Linux/POSIX applications. The compatibility layer must implement the presented system APIs in order to map the Linux/POSIX concepts to unikernel-internal OS service implementations. Compared to Linux, unikernels are based on a libOS design and therefore exhibit a much more modular and lightweight structure with little to no interdependencies between different OS components. Clean-slate unikernels implement these OS services from scratch and typically achieve a higher level of modularity than a strip-down unikernel that takes an existing operating system and modularizes it through kernel specialization. Due to the libOS design of unikernels, OS services are implemented as libraries and hidden behind common APIs. These interfaces effectively represent the unikernel-internal system call API through which higher software layers can access OS-specific functionality. This syscall interface typically only implements a fraction of available Linux syscalls. It is expected that strip-down unikernels have a more complete syscall API compared to clean-slate variants as they originate from the original monolithic kernel. In the unikernel environment, there is no need for a separate privilege switch to access kernel functionality as the OS services can be requested through normal function calls. This improves overall system performance.

Apart from the syscall interface, several other relevant Linux system APIs were presented such as libc and POSIX libraries, pseudo-files, and the vDSO segments. In the context of the thesis pseudo-files and vDSO have little importance for unikernels: Pseudo-files are mainly used by the command line or in scripts [13], while the vDSO mechanisms are often hidden behind the C library [107]. Unikernels provide their own implementation of common POSIX and libc libraries to provide the necessary abstractions for Linux/POSIX applications. These libraries internally use the unikernel core APIs to request OS services as normal function calls. This way, a compatibility layer can be constructed that maps Linux/POSIX concepts down to unikernel-internal implementations of common OS functionality.

API Compatibility

The presented system APIs aim for compatibility at the Linux/POSIX API level. This means that they effectively reduce the porting effort to bring existing Linux/POSIX applications to the unikernel framework. One requirement for API compatibility is the availability of source code.

Manual Porting In case the compatibility layer is still in an incomplete state, applications must be ported manually to the unikernel framework. This is a time-consuming task and requires the source code to be publicly available. The typical approach includes integrating the application into the unikernel build system and applying the necessary patches. It was shown by Kuenzer et al. [29] that the porting effort is diminishing over time as the compatibility layer becomes more complete.

API Compatibility at the *libc* level (Autoporting) At some point, the unikernel compatibility layer provides enough abstractions to support Linux/POSIX applications without applying additional patches. Porting efforts mainly revolve around integrating the project into the unikernel build system. This type of compatibilty is the natural evolution step of the *Manual Porting* approach. At this point, the native build system of the corresponding project can be used. The application is compiled against a C standard library that is also supported by the unikernel. The resulting application object files can be linked against the unikernel port of the libc library. This library implementation internally often uses function calls instead of actual system calls to trigger the unikernel-internal system call implementation [15, 8, 29]. The advantage of this approach is that the application's native build system can be used, which reduces porting efforts dramatically. At the same time, applications still need to be open-source. The application sources are further expected to not issue any syscalls directly but instead use functions provided by the libc. The Unikraft unikernel termed this approach *Autoporting* [104].

API Compatibility at the *syscall* level The previous approaches assume that all system calls are issued through the libc functions. In order to provide full API compatibility, a unikernel must also provide compatibility at the *syscall* level. Unikernels can register their own syscall handler function in the (virtual) *MSR LSTAR* register during startup. Whenever a system call is issued at runtime, the unikernel simply jumps to the registered syscall handler function and executes the corresponding unikernel-internal implementation. This way, unikernels such as Unikraft [29] and HermiTux [15] are fully API compatible at runtime. In case the unikernel is executed as a (Linux) user space process, there is no need to register a separate syscall handler function as the *syscall* invocation would operate on the syscall API of the underlying kernel. This is the case for example for the *Linuxu* platform of Unikraft [34].

ABI Compatibility

If a system is Linux ABI compatible, i.e. binary compatible, a native Linux binary can be taken and executed on the underlying system without additional porting efforts. This makes it possible to also execute applications where there is no access to the source code. A binary compatible system must emulate the Linux ABI at load- and run-time for both static and dynamic ELF binaries [15].

Static ELF Application sources are statically linked into the final executable together with depending libraries. At load time, an ELF loader that is part of the unikernel itself reads the executable segments and loads them accordingly into memory. Linux binaries can be compiled to be position-independent (PIE) in order to run correctly independent of the address at which the binary is loaded. Such a static loader is implemented by Unikraft [29] and HermiTux [15]. Installing a syscall handler function during startup as explained before is key to achieve runtime binary compatibility. Whenever a system call is issued at runtime, program flow simply jumps to the registered handler function in order to process a corresponding unikernel-internal implementation. When executed as a (Linux) user space process, there is no need for a separate syscall handler function as the underlying (Linux) syscall API will be used.

Dynamic ELF Dynamic libraries are not physically linked into the final executable. Instead, references to these libraries are stored together with the application sources. These references must be resolved in later stages. A dynamic loader loads the Linux application itself and takes care of resolving references to dynamic libraries. Unikernels can utilize an unmodified version of a regular Linux dynamic loader that is part of the Linux binary and its library dependencies [15]. In these cases, a unikernel-specific ELF loader executes the Linux dynamic loader which in turn loads the Linux ELF file and its dependencies. Such an approach is followed by HermiTux [15, 16] and Unikraft [108]. The dynamic loader resolves dynamic library dependencies at runtime. The way syscalls are handled for dynamic libraries is the same as for static libraries.

Optimizations Utilizing the *syscall* instruction results in a privilege switch on every invocation. Since unikernels do not distinguish between user and kernel space, this extra privilege switch is not needed anymore. The unikernel syscall implementations can be called as normal function calls. This improves overall system performance. Unikernels apply the following optimization techniques to avoid the expensive *syscall* instruction:

- **Fast Calls**: Olivier et al. [15, 109] describe a binary patching mechanism for static binaries that they call *fast calls* in order to replace the *syscall* instruction with a corresponding function call. They go to certain lengths to circumvent the issue that the *syscall* instruction is limited to two bytes.
- **Library Substitution**: Most unikernels that claim Linux binary compatibility, are compatible at the libc-level. They use a technique called *Library Substitution* for dynamically linked libraries [15, 16]. The general idea behind this approach is to use a dynamic linker that resolves calls to dependent libraries at runtime to corresponding unikernel implementations. These unikernel-specific library implementations internally use normal function calls to request OS functionality and thus do not require a separate privilege switch. OSv achieves this level

of compatibility by implementing an ELF dynamic linker that accepts standard dynamically-linked ELF compiled for Linux. When this code calls functions from the Linux ABI, i.e. functions provided by the glibc library, these calls are resolved by the dynamic linker to functions implemented by the OSv kernel [8]. Similarly, HermiTux [15, 16] links dynamic binaries at load time against a unikernel-aware C standard library, in which all the system calls are replaced by function calls to the unikernel. Compared to OSv, HermiTux does not write a libc from scratch but adapts an existing one using a code transformation tool. *Library Substitution* makes the assumption that all system calls are made through the standard C library, which does not hold true for all applications [15]. In case the actual *syscall* is issued by the application itself, we still require a syscall handling mechanism as was described before.

5.2.3 Design Limitations

Based on the information given above, we can now derive relevant limitations for our microkernel-based design in terms of compatibility.

Incomplete syscall API Unikernels generally only implement a fraction of all available Linux system calls. It was shown that a subset of system calls need to be implemented by a compatibility layer to provide sufficient support for many complex Linux/POSIX applications [104]. Any execution platform, including microkernels, can only support the amount of syscalls that is also supported by the unikernel itself.

Single Process Applications Unikernels typically run single applications in their address space and do not provide support for functions such as *fork* or *exec* [7]. Therefore, we can only expect compatibility to single process Linux/POSIX applications for our microkernel design.

Binary Compatibility As was shown, unikernels require to be executed in a virtual environment or directly in hardware in order to be able to register their own syscall handler function. This makes sure that runtime binary compatibility can be achieved at the syscall level. The only exception are unikernels that utilize the Linux kernel itself as the underlying platform [34, 33]. In these cases, there is no need to register a separate syscall handling function as applications can simply utilize the Linux syscall API. Since unikernels run at a single protection level, they typically apply optimization techniques, e.g. *fast calls* or *library substitution*, to avoid the expensive *syscall* instruction. Since we aim to run the unikernel as a microkernel user space process, we can not register a unikernel-specific syscall handler function, but are restricted to the (limited) syscall API of the microkernel. By utilizing unikernels that apply the described optimization techniques, it should still be possible to support native Linux binaries on top of the microkernel.

5.3 Integration Considerations

Unikernels require a handful of low-level functionality that must be provided by the underlying platform layer. If we wish to support a unikernel on top of a microkernel, it is important to understand the type of low-level functionality that must be provided in the PAL. This will give us valuable insights on how to design a generic interface between a unikernel and a microkernel. It is important that this interface design does not rely on expensive hardware features as is typically the case for unikernel virtualization.

5.3.1 Unikernel Platforms

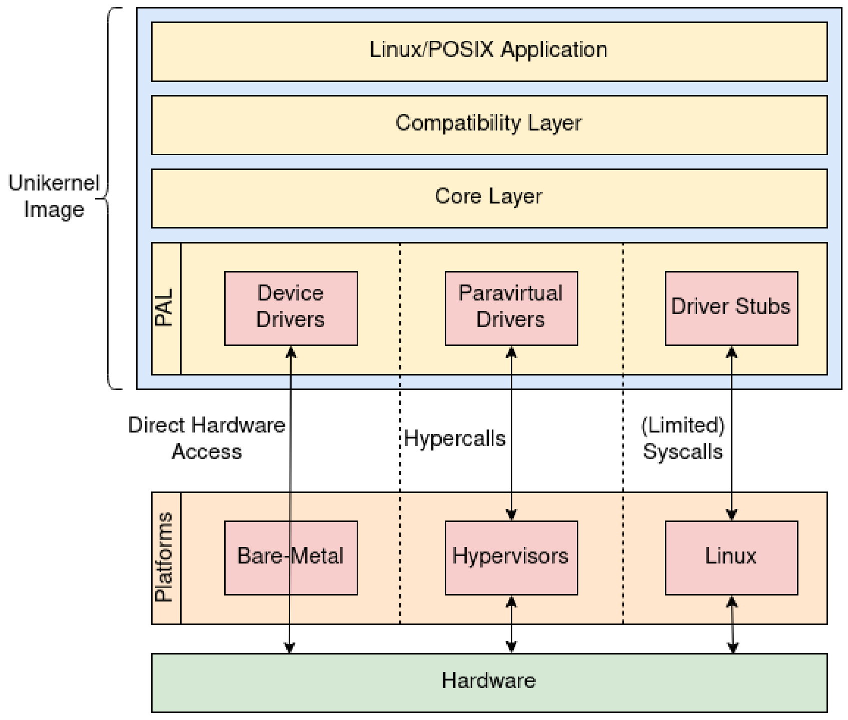


Figure 5.2: Unikernel Platforms.

Before taking a closer look at the functionality implemented in the platform layer, we investigate the different types of unikernel platforms as shown in Figure 5.2. Strip-down unikernels often pose extensive dependencies on the underlying host which complicates platform integration, compared to clean-slate unikernels [76].

Platform: Bare-Metal

In case the unikernel provides physical device drivers, it can operate directly on physical hardware. This is attractive for embedded systems and utilized by unikernels such as Rumprun [77]. The unikernel assumes to be the sole operator of the hardware, so there is no need for elaborate isolation and hardware sharing mechanisms.

Platform: Hypervisors

Unikernels are typically executed in a virtual environment on top of a hypervisor, also referred to as a Virtual Machine Monitor (VMM). These hypervisors utilize virtualization techniques either in software or hardware to provide necessary hardware resource abstractions to implement low-level functionality in the PAL. VMMs are able to emulate entire machines including CPU, MMU, memory and devices, effectively realizing multiple (virtual) computers in one. Virtual Machines can be either totally isolated or connected through virtual network interfaces and shared filesystems [38]. Unikernel monitors such as *ukvm* and *solo5* are lightweight VMMs specialized to the unikernel context [33]. Hypervisors and unikernel monitors utilize virtualization hardware, e.g. Intel VT-x, for isolation. The unikernel exits to the monitor via so called hypercalls, typically to perform some I/O operation. Hypercalls are similar to system calls, but instead of exiting to the kernel, hypercalls result in a world switch to the underlying hypervisor. Compared to the Linux syscall interface, unikernel monitors and hypervisors only have a handful of available hypercalls. This results in a higher level of isolation [33].

Hardware-based virtualization relies on hardware extensions that might not be readily available on all hardware types. If the hardware is not available, the emulation system relies on software virtualization which typically comes at a performance drawback. Additionally, hypercalls are needed to exit to the underlying monitor. It was shown that these hypercalls are less performant than corresponding system call implementations [33].

Platform: Linux

To tackle the problem of hardware availability and performance drawbacks, there have been attempts to run unikernels directly as user space processes on top of a monolithic kernel, specifically Linux. The PAL of the unikernel is implemented by utilizing the available syscall API of the underlying Linux kernel. This approach is realized by unikernels such as *Nabla* containers [33], the *Gramine* libOS [25] and the *Linuxu* platform in Unikraft [34].

Modern operating systems provide two mechanisms to enforce a high level of isolation for user space processes: (a) system call whitelisting policies, while mapping the hypercalls to a handful of whitelisted system calls (*Nabla* containers), and (b) utilizing secure enclave technology such as Intel SGX (*Gramine*).

Platform: Microkernels

As was already shown, unikernels can be executed as (Linux) user space processes. This way, unikernels can stay performant and are less reliant on hardware virtualization extensions. The idea of this project is to take this concept and utilize a microkernel as the underlying platform. This approach was so far only attempted by porting the Rumprun unikernel to the seL4 microkernel [11, 12]. The thesis extends these considerations to a generic interface design between a unikernel and a microkernel.

Microkernel Processes are similar to processes known from other operating systems: they consist of an address space realized by a page table and a number of associated threads [71]. These processes are physically protected/isolated from each other by the Memory Management Unit (MMU). If a process illegally tries to access memory of another process, this is detected by the MMU and an exception is raised. This exception is caught by the kernel which can take action to redeem it [38].

Compared to monolithic kernels, microkernels are much more minimal and rely on user space to implement most OS functionality. This also means that the microkernel must provide the necessary mechanisms to access physical resources from user space. Microkernels define so called kernel objects to abstract over hardware resources. They expose services around these kernel objects to user space through the microkernel API. To enforce isolation, each invocation of a kernel object is protected by a capability. The capability-based approach is enforced by the microkernel and does not involve any hardware feature. Since the microkernel API consists of only a handful of system calls, each protected by capabilities, there is also no need for a separate syscall whitelisting mechanism for microkernels [71].

In order to run a unikernel on top of a microkernel as a user space process, we have to provide the PAL functionality based on the microkernel API. Since user space is much closer to actual hardware resources in microkernels, the PAL can be constructed straight at the hardware level. This allows to integrate the microkernel at the lowest abstraction layer. As a result, porting efforts are minimized as the unikernel provides most of the OS features.

5.3.2 Abstraction Level: Hardware

Unikernels push the platform abstraction layer down to the hardware level to allow for easy integration with different execution platforms. The PAL is a thin layer around the typical hardware resources such as CPU, Events, Memory and I/O devices. It requires a handful of low-level functionality that must be provided by the underlying platform. We will identify these low-level functions and present the level of hardware access available for different unikernel platforms.

Resource: CPU

The CPU implements different protection rings that are utilized by an operating system to distinguish between user (ring 3) and kernel mode (ring 0). In user mode, a program is restricted to user ISA, while kernel mode has access to additional machine instructions and hardware registers, the so called system ISA. Execution of a privileged instruction from user space results in a hardware-generated exception [66]. We are specifically interested in the following system ISA-related functionality:

- **Global Interrupt Control**: The *FLAGS* register in modern x86 architecture allows to enable/disable interrupt delivery for the entire processor [60].
- **Halting the CPU**: The *HLT* instruction halts the entire CPU until the next interrupt is received [60].
- **Syscall Handling**: Syscall Handling requires a corresponding syscall handler function to be registered in the *MSR LSTAR* register. Access to this register is controlled by the privileged instructions *RDMSR/WRMSR* [60].
- **TLS Handling**: TLS Handling requires control over the FS/GS segment registers. The base address can be controlled either through a specific MSR register or a privileged control register must be configured to allow user space to modify the FS/GS base address [60].

There might be additional system ISA-related functionality implemented in the PAL. But for the purpose of this thesis, we restrict our considerations to this selected sample of low-level functionality.

Unikernel View Unikernels remove the user-to-kernel space separation and operate at a single protection level [33]. As a result, platforms are typically expected to have access to system ISA to provide the presented low-level functionality in the PAL. Hypervisors utilize several CPU virtualization techniques, e.g. a mixture of binary translation and direct execution, paravirtualization, trap-and-emulate, or hardware-assisted virtualization that introduces an additional privilege ring [110, 111].

Microkernel View A thread is an operating system's way to abstract over code execution. For this purpose, a microkernel defines one Thread Control Block (TCB) kernel object for each available thread on the system. This kernel object stores a thread's userland state while it is not running on the CPU. User CPU state that is stored in the TCB typically includes the instruction pointer, stack pointer, user-level registers and flags [112]. Since we plan to run the unikernel from user space, we are restricted to the user ISA. As a result, several privileged instructions and registers can not be executed without triggering an exception from the microkernel. Therefore, we need to find workarounds to implement/emulate the presented CPU-related functionality.

Resource: Events

The x86 architecture has 255 distinct hardware events. The general idea is that an interrupt descriptor table (IDT) stores for each event a specific interrupt service routine (ISR) that is triggered by a corresponding hardware event. Parts of the IDT are reserved for the system ISA [113]. We distinguish between three different types of hardware events:

- **Traps**: A trap occurs when program execution results in a switch to the highest privilege level. Typical traps include breakpoints, system calls and software interrupts [113].
- **Faults**: A fault occurs when an instruction results in a faulty system state, which must be resolved before execution can continue. Typical faults include exceptions, page faults and general protection faults [113].
- **Interrupts**: Interrupts are typically generated by a hardware device when an asynchronous event occurs. The interrupt is reported to the CPU, which then stops execution of the current program and executes the corresponding interrupt

service routine. We typically distinguish between maskable interrupts, e.g. from an I/O device, and non-maskable interrupts that must be handled immediately, e.g. from a timer [113].

Unikernel View Event Handling is the sole responsibility of the platform layer in unikernels. For device interrupts, unikernels typically use event loops with a blocking polling call to the underlying system for asynchronous I/O. As a result, they often do not even need to install interrupt handlers or use interrupts for I/O [33]. Some unikernels on the other hand might still need some form of interrupt handling, specifically when they rely on interrupt-driven device drivers. A hypervisor virtualizes interrupt delivery to virtual processors, that is, each virtual processor has a local interrupt controller instance [114, 115].

Microkernel View Due to the minimal design of microkernels, user space must be given (partial) rights to manage event handling. For our considerations, it is safe to assume that traps such as microkernel system calls are handled in the kernel. Faults and exceptions are either handled by the microkernel itself [38] or in some instances a user space fault handler can be applied [116]. In order to handle interrupts, microkernels typically implement an interrupt controller driver in the kernel that distributes interrupts to user-level processes [71]. Interrupts are delivered to user space based on underlying (asynchronous) IPC mechanisms, i.e. notifications. The kernel defines an IRQ kernel object and maps hardware IRQs to these data structures. To reduce the amount of policies in the kernel, often exactly one waiter (thread) per IRQ object is allowed [42].

Resource: Memory

Page Tables are used to manage address spaces of processes and encode the mapping of virtual addresses (VA) to physical addresses (PA). Traversing the page table from virtual to physical address is termed a page table walk. A page table must be setup and managed for each process on the system. The Memory Management Unit (MMU) and the Translation Lookaside Buffer (TLB) is the hardware responsible for translating virtual addresses to physical addresses [117].

Unikernel View A running unikernel is conceptually a single process that runs the same code for its lifetime. Thus, there is typically no need for the unikernel to manage page tables after initial setup [33]. Hypervisors virtualize the MMU to provide an additional level of indirection that first maps a guest virtual address (GVA) to a guest physical address (GPA), before mapping the GPA to a host physical address (HPA)

[118]. A Shadow Page Table (SPT) is kept for each process in the guest that resolves GVA to HPA, which is used in the hardware TLB for quick resolution of HPA [111]. Intel supports this technique in hardware through a hardware feature called Extended Page Table (EPT) [118].

Microkernel View Odun-Ayo et al. [35] state that memory management is the responsibility of user space in microkernels either through a separate pager process or actual user space control over page tables. An original idea in the L4 microkernel was to have an initial address space which receives a mapping of all free frames left over after the kernel has booted and apply a recursive page mapping mechanism. This approach is still used with microkernels such as Fiasco.OC and NOVA. The seL4 and OKL4 microkernels on the other hand originate mappings straight from physical frames [71]. Many modern microkernels such as NOVA, Fiasco.OC and OKL4 apply kernel memory quotas, i.e. the combination of a per-process kernel heap and a mechanism to donate extra memory to the kernel on exhaustion [71]. The seL4 microkernel sticks out in terms of kernel memory management as they remove kernel heaps all together and make all kernel objects explicit and subject to capability-based access control. As a result, after booting the kernel never allocates memory, and any remaining memory is passed to an initial user-mode process. In modern microkernels, often an initial user-level task (root task) is given full rights to all resources left over once the kernel has booted up, i.e. physical memory, I/O ports and interrupts. This root task is then responsible for setting up other tasks, including their address spaces [5].

Resource: I/O (Devices)

Software requires access to interrupts, device memory and I/O ports (in x86) to interact with a hardware device. Device drivers are the part of an operating system that interacts with actual device hardware. A device driver constitutes of the driver logic and the actual hardware access.

Unikernel View Unikernels often operate device drivers in polling mode and use event loops for asynchronous I/O [33]. As a result, there might not even be a need to implement separate interrupt handling in unikernels. For completeness, we should consider at least some basic form of interrupt handling. Hypervisors apply different virtualization techniques to emulate devices, e.g. hypervisor-based device emulation, paravirtual (PV) drivers, and device passthrough [119, 120].

Microkernel View Microkernels apply user-level device drivers and therefore must provide actual hardware access to user space in the form of interrupts, device memory and I/O Ports. As was already explained, interrupts are represented as kernel objects that have asynchronous IPC mechanisms attached to them. I/O Ports are an x86-specific feature and define their own I/O address space that can be accessed through special processor instructions. In order to restrict access to these I/O Ports from user space, L4 microkernels apply a per-task I/O bitmap that allows a fine-granular grant/deny of I/O Port access. I/O memory is treated like physical memory and must be mapped into the address space of the driver process just like normal RAM [42]. A small number of drivers are still best kept in the kernel: This typically includes a timer driver, that is used for preempting user processes at the end of their time slice, and an interrupt controller, which is required to safely distribute interrupts to user-level processes [71]. A security concern in microkernels is the fact that the component with hardware access control has the ability to corrupt the entire system [38].

5.3.3 Abstraction Level: OS Services

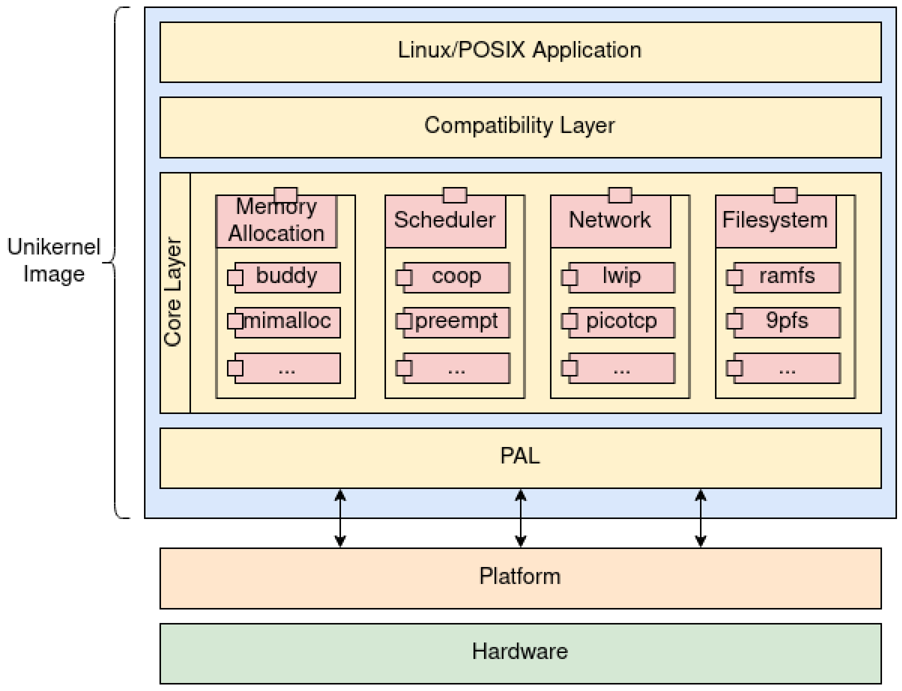


Figure 5.3: Unikernel Core Architecture.

Unikernels provide their own implementations of common OS services in a modular structure. OS functionality is realized as different library implementations that are hidden behind a common API as is demonstrated in Figure 5.3. Compared to the

Linux kernel subsystems, these unikernel library modules are typically less complex and exhibit little to no interdependencies between each other. This simplicity of the unikernel environment ultimately eases platform integration. We will explore the way common OS functionality is implemented in unikernels to better understand how these functions map down to the platform layer. Although microkernels export most functionality into user space, some policies must still be implemented in the kernel itself. When executed as a user space process, we have to assume that both the unikernel and the underlying microkernel have their own implementations of common OS functionality. In case these implementations collide, we need to extend our considerations for the interface design.

Scheduling

Unikernel View Unikernels typically run single process applications. At the same time, they support multiple threads that share the same address space. Unikernels therefore implement their own scheduler that manages these unikernel threads. A thread context switch is typically lightweight as there is no need to change address spaces during context switching. Unikernels often use cooperative scheduling [33]. This way, threads must voluntarily yield control or when logically blocked on a resource.

Microkernel View Scheduling is one of the OS functions that microkernels still implement in the kernel itself [71]. They typically use a (preemptive) priority-based round-robin scheduler [71, 35]. Microkernel threads can be located in different address spaces and thus a context switch requires additional effort.

Memory Allocation (Heap)

Unikernel View Apart from setting up the initial address space, a unikernel requires heap space to perform memory allocation that must be allocated by the underlying platform [33]. Unikernel-specific allocator libraries utilize this heap memory region to provide malloc/free functionality for higher libc layers.

Microkernel View Since memory management is typically the responsibility of user space in microkernels, physical memory regions can be mapped into the address space of a process in order to serve as the heap area.

IPC

Unikernel View Since unikernels realize single process applications, they typically do not implement IPC mechanisms. When run as a user space process, the unikernel potentially must interact with other processes. The underlying platform must then provide the necessary IPC mechanisms; these are typically transparent to the unikernel.

Microkernel View Modern microkernels implement their own IPC mechanisms in the form of message passing and provide these through the microkernel syscall API. They typically offer both synchronous as well as asynchronous IPC mechanisms. A simple form of asynchronous IPC are notifications that allow non-blocking senders and the receiver can block or poll asynchronously on the notification. For long messages shared buffers can be used to improve throughput for bulk data transfers. The send and receive system calls can be combined into a single system call to implement RPC-like behavior [71].

Filesystem

Unikernel View Unikernels, similar to other monolothic kernels, implement filesystem functionality in a layered approach. Different filesystem implementations are hidden behind a virtual filesystem abstraction that can be used by applications. Eventually, the filesystem stack interacts with a storage driver to implement persistent storage.

Microkernel View Filesystems are the responsibility of user space in microkernels. We want to utilize a unikernel to provide filesystem-related functionality for the microkernel.

Networking

Unikernel View Similar to the filesystem stack, the network stack is typically also implemented in a layered fashion, integrating the ISO/OSI network model conceptually speaking [121]. Higher software layers interact with the network stack via the POSIX socket API which hooks into a TCP/IP stack. The TCP/IP stack is responsible for providing networking functionality on layer 3 and 4. Eventually, the TCP/IP stack interacts with a network driver to implement layer 2 functionality.

Microkernel View Network Stacks are the responsibility of user space in microkernels. We want to utilize a unikernel to provide network-related functionality for the microkernel.

5.3.4 PAL Functionality

Unikernels push the platform layer down to the hardware level and require a handful of low-level functionality that must be implemented by the underlying execution platform. In order to integrate the unikernel as a microkernel process, we need to implement these functions for the microkernel. We explored for both unikernel and microkernel their view on the hardware and OS service abstraction level. The collected information helps us to define the requirements for the PAL interface between unikernel and microkernel.

CPU-related Functionality We need to provide a threading model that unites the unikernel and microkernel view on scheduling. The microkernel platform must provide the necessary mechanisms for event handling. We further need to consider how to deal with system ISA-related functionality.

Memory-related Functionality We need to explain how the paging mechanisms are utilized by the microkernel to setup the address space for the unikernel process. The microkernel platform further must provide a consecutive memory region that will be used by the unikernel for memory allocation.

I/O-related Functionality Device Drivers are used for actual I/O device access. We need to explain how unikernel device drivers can be used on the microkernel and how available microkernel user space drivers can be used in the PAL. Higher unikernel layers such as the filesystem and network stack rely on the platform layer to provide the necessary device drivers for their operation.

6 Design

Based on the previous considerations, we will define an interface that allows a unikernel to run as a microkernel user space process. The unikernel then serves as a lightweight compatibility layer to real-world workloads such as Linux/POSIX applications. As was shown before, a unikernel defines a platform layer that abstracts over common hardware resources in order to provide a handful of low-level functionality. During the design, we will complete this platform layer for a microkernel in order to use the microkernel as an execution platform. The design mainly focuses on the *x86_64* architecture, although the concepts should be applicable to other architectures as well.

6.1 CPU Model

6.1.1 Threading Model

Unikernels typically implement a cooperative scheduler. Context switching is simpler than in regular multitasking operating systems as all unikernel threads share the same address space. The microkernel on the other hand implements a preemptive scheduler, where threads can also be located in different address spaces. Figure 6.1 demonstrates how the unikernel and microkernel threading models can be united. It utilizes *user-level threading*, also referred to as *green threading* [122], in order to map N unikernel threads to one microkernel thread [47]. The unikernel runtime effectively hides all unikernel threads behind this one microkernel (control) thread. We assume the unikernel thread context to be a subset of the microkernel thread context. As a result, the microkernel (control) thread mirrors at any point in time the thread context of the currently scheduled unikernel thread. Unikernel and microkernel scheduler operate mostly transparent to each other. This approach is similar to what was implemented for the Rumprun unikernel port on the seL4 microkernel [12].

6.1.2 Event Handling

We identified three types of hardware events: traps, faults and interrupts. If unikernels are executed on (virtual) hardware, the underlying platform typically implements necessary event handling mechanisms. When executed as a user space process on top

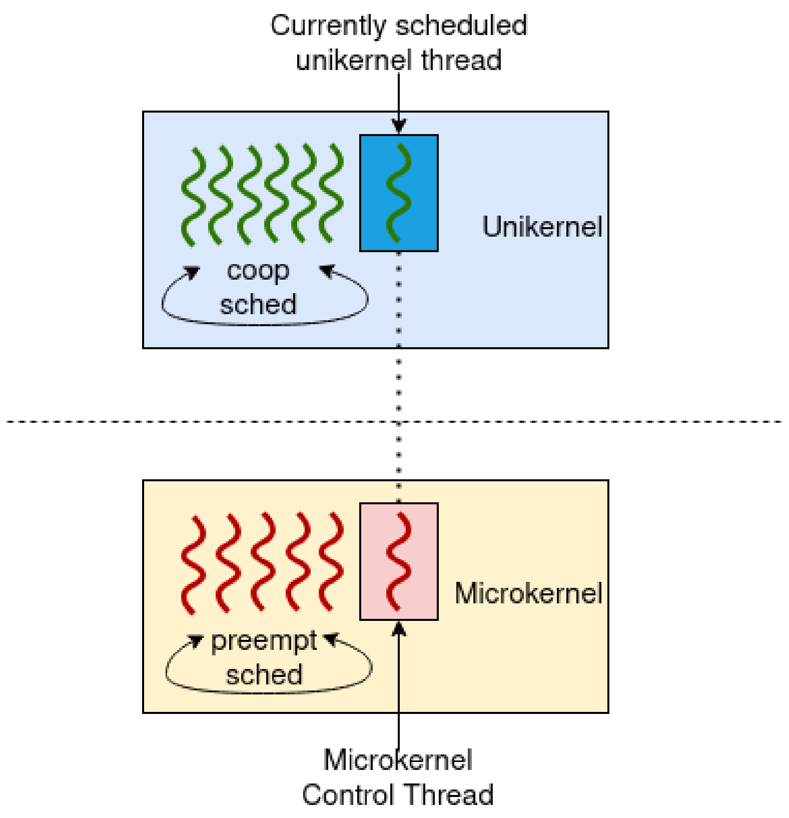


Figure 6.1: Threading Model.

of a monolithic kernel, unikernels rely on the underlying kernel to deal with these hardware events. Depending on the minimality of the microkernel, user space must implement some form of event handling.

Traps A trap occurs when the execution of a certain instruction causes a switch to the highest privilege level synchronous to program flow. Trapping instructions include system calls, software interrupts, and breakpoints. We assume the microkernel to implement the necessary policies to deal with these trapping instructions. In case a unikernel thread, i.e. the microkernel control thread, executes a trapping instruction, this will cause a trap event and trigger a microkernel-specific trap response as shown in Figure 6.2.

Faults A fault occurs when the execution of a certain instruction results in a faulty system state. Faults must be corrected before execution can continue. Typical faults

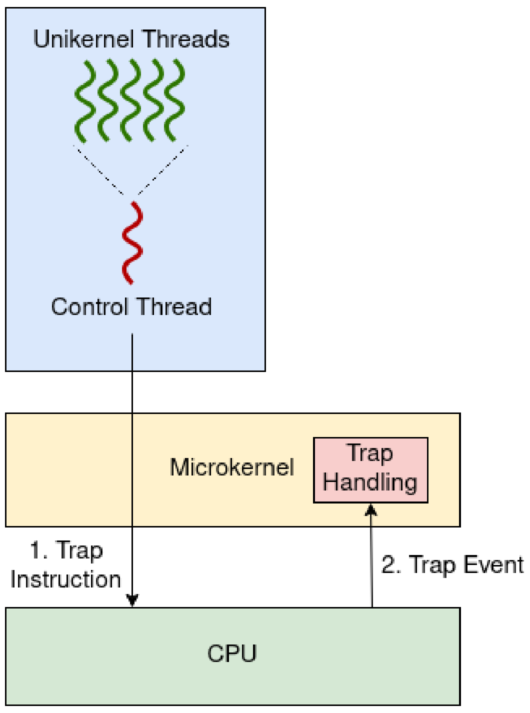


Figure 6.2: Trap Handling.

include exceptions, page faults and general protection faults. Some microkernels forward fault messages to a user space fault handler thread that is responsible to correct faulty behavior [116]. Such a situation is depicted in Figure 6.3. The CPU generates a hardware fault based on the processor state that leads the microkernel to block the control thread of the unikernel process. A separate thread in the same address space, i.e. the fault handler thread, is notified and must resolve the fault before execution of the original thread can be continued. In order to resolve the fault, the microkernel provides access to required system resources and capabilities of the faulting (control) thread to the fault handler thread.

Interrupts We extend our threading model to implement interrupt handling in the unikernel as shown in Figure 6.4. At least one additional microkernel thread is introduced that is responsible to handle registered interrupts. This interrupt handler thread must have capabilities to all required hardware interrupts to interact with corresponding kernel objects. When a hardware interrupt arrives, the microkernel notifies the interrupt handler thread that holds the capability to the corresponding interrupt kernel object. The interrupt handler thread processes the interrupt and must

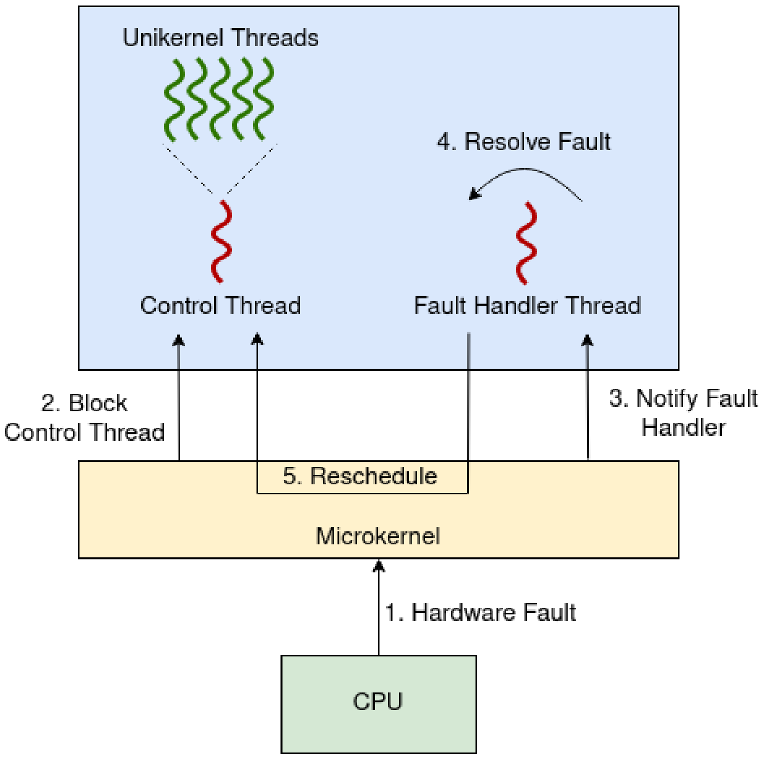


Figure 6.3: Fault Handling.

synchronize with the microkernel control thread to deliver interrupt events to the actual unikernel threads. After processing the interrupt, the interrupt handler thread must instruct the kernel to acknowledge the interrupt to enable successive interrupt delivery to this microkernel thread. For non-maskable interrupts, the interrupt handler thread must immediately process the interrupt. The way the microkernel interrupt handler thread delivers interrupts to the unikernel very much depends on a specific unikernel implementation. In case the unikernel has simple event loops with a polling call to asynchronous I/O, interrupt handling might not even be needed in the first place.

6.1.3 ISA Functionality

For the unikernel process, we are restricted to user space and therefore can not execute privileged instructions without causing an exception. We identified the most relevant ISA-related functionality in the PAL that requires privileged execution. We need to find workarounds for these functions for the microkernel environment.

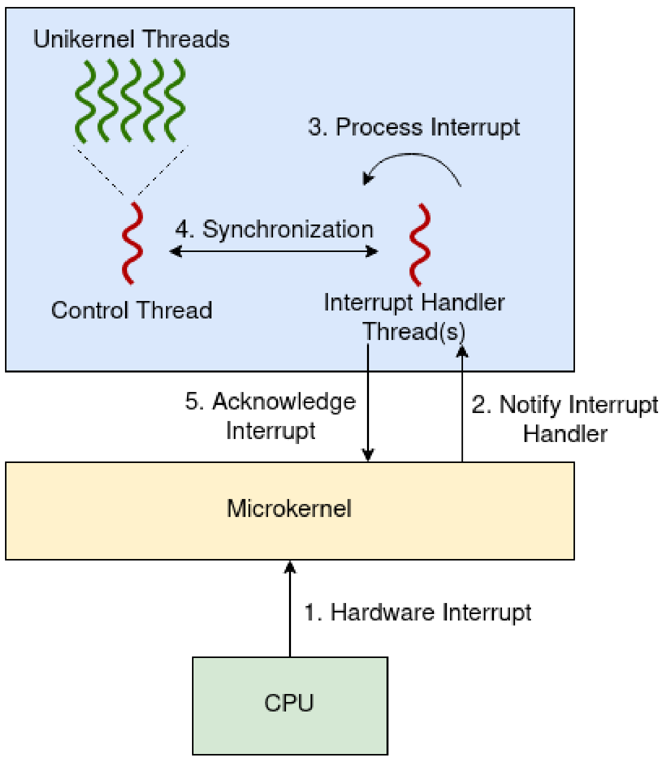


Figure 6.4: Interrupt Handling.

Global Interrupt Control

Unikernels require control over global interrupt delivery. They utilize the privileged *FLAGS* register to enable/disable (maskable) interrupts for the entire processor. When executed as a normal user space process, we can not globally control interrupt delivery, since the underlying CPU is shared with other processes. In the microkernel interface we solve this problem through a simple workaround: We do not actually enable/disable global interrupt delivery, but simply control whether or not interrupts that arrive on the interrupt handler thread shall be delivered to the unikernel threads. The decision can be based on a unikernel-wide switch that higher unikernel layers can set if they want to disable interrupt delivery.

Halting the CPU

The *HLT* instruction is another privileged operation that halts the entire CPU until an interrupt arrives. Similar to the observation before, user space can not explicitly execute this instruction without triggering an exception. The microkernel implements another workaround in the platform layer: The microkernel control thread waits on a specific signal and therefore blocks all unikernel threads. When an interrupt is registered in the interrupt handler thread, it unblocks the waiting unikernel threads.

Syscall Handling

Unikernels that run on (virtual) hardware can register a unikernel-specific syscall handler function in the *MSR LSTAR* register during startup. When the *syscall* instruction is issued, program flow jumps to the registered syscall handler function to execute corresponding unikernel functionality. This way, unikernels achieve Linux runtime binary compatibility. The *MSR LSTAR* register must be set through the privileged *WRMSR* instruction and therefore can not be utilized for microkernels in user space. Additionally, we would overwrite the microkernel API. In case unikernels are utilized that implement any of the optimization techniques presented in subsubsection 5.2.2, i.e. effectively replacing/avoiding the *syscall* instruction all together, our microkernel-based approach is expected to run native Linux binaries as long as it can load them into memory. For unikernels that do not apply these optimizations, system calls will actually call into the microkernel syscall API since we can not register a separate unikernel-specific syscall handler. Consequently, Linux binary compatibility would not be possible for these unikernels under the current microkernel design.

TLS Handling

In paragraph 3.1.3, TLS was presented. It was shown that operating systems typically utilize the FS/GS segment registers to manage TLS. We assume that this approach is also utilized by unikernels and microkernels in order to implement TLS. When executed directly on (virtual) hardware, unikernels typically utilize the privileged *WRMSR* instruction to set the corresponding base addresses. When executed on top of Linux, unikernels can utilize the *arch_prctl* syscall of the Linux API to manage TLS. When utilizing microkernels as the underlying execution platform for unikernels, we do not have access to these mechanisms. Instead, we will utilize the *FSGSBASE* instructions in order to let unikernels manage TLS from user space. We assume that the microkernel enables these instructions in the corresponding control register during system startup. As a result, TLS handling becomes part of the thread context for the presented threading model.

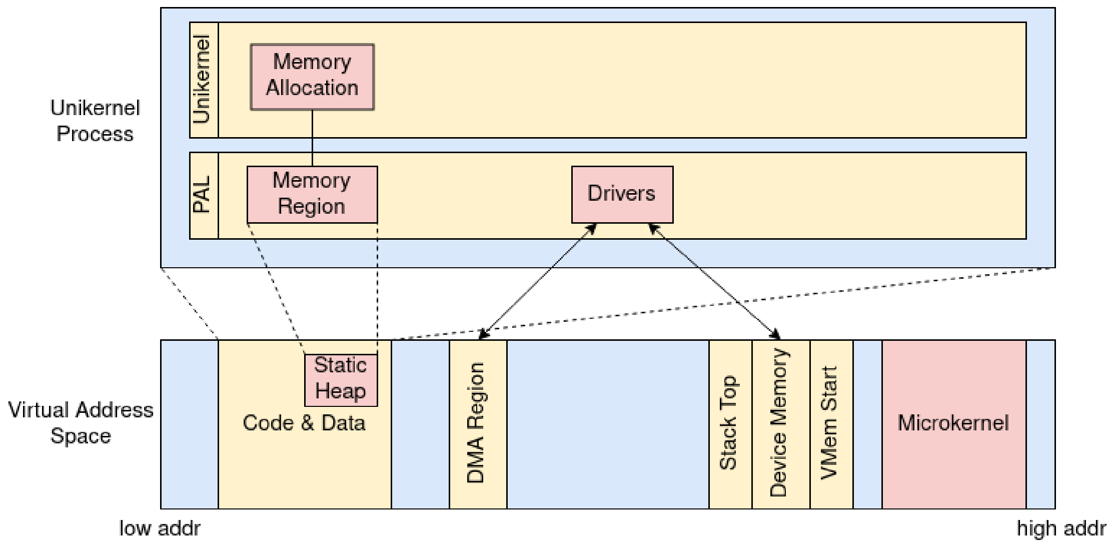


Figure 6.5: Memory Model.

6.2 Memory Model

6.2.1 Paging

A running unikernel is conceptually a single process that runs the same code for its lifetime [33]. As a result, there is usually no need for it to manage page tables after initial address space setup. Consequently, we can use a static system design for our microkernel setup, i.e. all user space processes are known at startup. As was explained before, modern microkernels often have an initial root task that is responsible for setting up the user-level environment. This root task is handed all system resources and corresponding capabilities from the kernel at startup and sets up the page tables for all microkernel processes, including our unikernel process. The root task must be instructed to load unikernel code together with device-related information and corresponding capabilities into the unikernel process address space.

An exemplary address space layout of the unikernel process is shown in Figure 6.5. We assume that the microkernel itself is mapped into the high address part of the virtual address space of every user space process [123, 124, 125]. In the lower part, unikernel code and data is mapped together with memory-related device information and DMA regions. These can be used by device drivers to implement communication with the actual device hardware.

6.2.2 Memory Allocation (Heap)

The unikernel must be provided a consecutive memory region from the platform layer in order to implement memory allocation for heap-like behavior. The easiest solution for any platform is to have a static character array (red box) in the data section of the process address map as shown in Figure 6.5. For dynamic heap behavior, the platform layer must allocate a memory region that can be dynamically scaled at runtime.

6.3 I/O Model

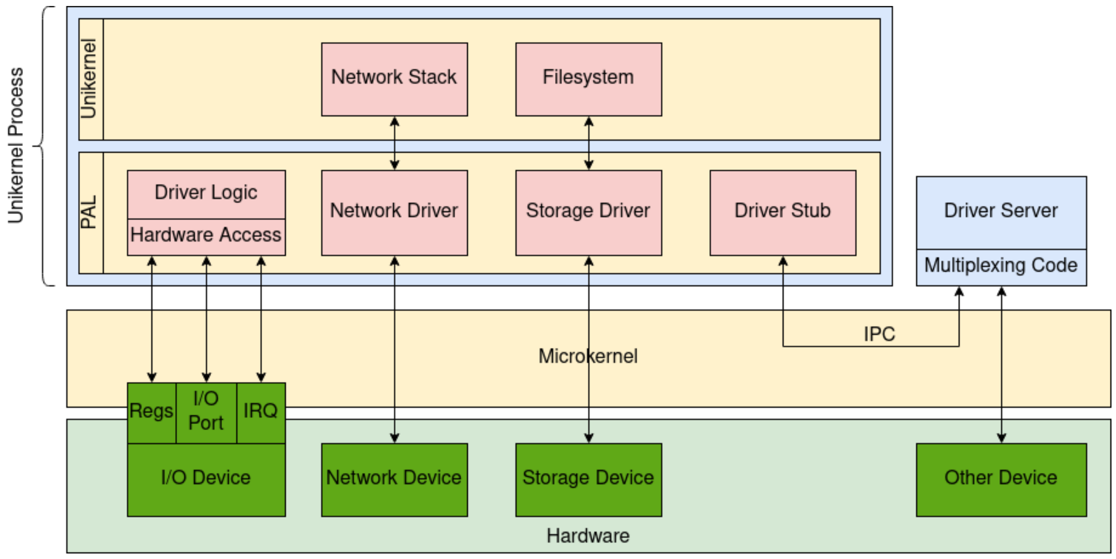


Figure 6.6: I/O Model.

6.3.1 Device Drivers

Microkernels implement device drivers in user space. This is realized by giving user space access to device-related hardware features including I/O memory, I/O ports (on x86) and hardware interrupts. The microkernel implements respective kernel services that can be accessed by user space with the appropriate capability.

Figure 6.6 shows how device drivers can be supported in the platform layer for an underlying microkernel. In case the unikernel already provides appropriate device drivers, they can be theoretically reused for the microkernel environment.

We differentiate a device driver into its respective driver logic and the actual hardware access. To reuse an existing unikernel device driver in the platform layer, the microkernel must implement device-specific hardware access based on the microkernel mechanisms. As a result, the actual driver logic can be reused for the microkernel. In case the microkernel already comes with the required device drivers, we can also utilize these and adapt them to the unikernel environment. Device Drivers in the unikernel are hidden behind a common API that can be utilized by higher OS components such as filesytems and network stacks.

Another common solution in a microkernel-based system is to run device drivers as their own servers in a separate address space. Clients connect to these user space servers via the described microkernel IPC mechanisms. These device driver servers often implement some form of multiplexing code in order to support multiple connecting clients. This approach can be used to extract device drivers from the PAL into their own user space processes and implement a simple driver stub that connects the unikernel process with the external driver interface. This increases overall isolation as the unikernel process can no longer access device-specific hardware information from its address space.

6.4 Summary

During the design, we constructed an interface between unikernel and microkernel based on the observations taken in previous chapters. For each hardware resource, we provided a corresponding resource model that connects unikernel with microkernel concepts. The CPU Model unites the threading models of both unikernel and microkernel, as well as providing mechanisms for event handling. Additionally, workarounds were demonstrated for ISA-related user space restrictions. In the Memory Model, we showed how the root task can be utilized to set up the address space for a unikernel process. The microkernel further provides a consecutive memory region to the unikernel to realize memory allocation. In the I/O Model, we demonstrated how unikernel device drivers can be reused for the microkernel as well as how existing microkernel drivers can be integrated into the PAL. External device driver servers can be used to provide better isolation and solve the issue of hardware resource sharing on a process level. By implementing these resource models, a unikernel can be executed as a user space process on top of a microkernel.

7 Implementation

In this chapter a prototype system shall be implemented that is based on the design choices stated in the previous chapter. We first select a suitable microkernel and unikernel for the prototype implementation. Afterwards, the implementation approach is introduced which lays out the porting steps that were needed to support the selected unikernel on top of the microkernel. We specifically focus on how the different resource models derived in the design stage were implemented and what kind of obstacles existed.

7.1 Microkernel Selection

7.1.1 seL4

Specifically members of the L4 microkernel family are of interest for the implementation since they represent a time-tested microkernel approach with decades of ongoing evolution in this field [36]. Out of all microkernels, the seL4 microkernel is arguably the fastest and most secure microkernel at the time of writing [2]. It is formally verified, utilizes a capability-based access control approach, and supports mixed criticality workloads. Therefore, it is the microkernel of choice for this project.

7.1.2 CAmkES

CAmkES provides a software development and runtime framework to quickly and reliably build static component-based multiserver (operating) systems on top of microkernels. Although primarily used for the seL4 microkernel, it is not strictly linked to any specific microkernel implementation [126]. A CAmkES system is specified in the CAmkES Architecture Description Language (ADL), which contains a precise description of the components, their interfaces and the connectors that link them up. The CAmkES promise to the system designer is that no interactions are possible beyond those specified by the CAmkES ADL [2]. The CAmkES specification gets translated into the seL4 low-level mechanisms and sets up the required capabilities with the help of the CapDL Loader app. We choose CAmkES for the implementation since it abstracts over all the low-level seL4 mechanisms and sets up a system as intended by the kernel developers. This reduces porting efforts and allows us to focus on the interface implementation.

7.2 Unikernel Selection

We decide for a specific unikernel based on the following criteria:

- **Platform Integration**: The unikernel has little dependencies on the underlying platform to ease porting efforts.
- **Linux/POSIX Compatibility**: The unikernel provides an extensive compatibility layer to support many Linux/POSIX applications.

Most existing unikernels are only of limited use for this project. This includes all the language-based unikernels. Since we strive for Linux/POSIX compatibility, especially POSIX-like unikernels are of interest:

- **Strip-Down Unikernels**: Rumprun [30], Lupine Linux [7], and Unikernel Linux (UKL) [10, 31]
- **Clean-Slate Unikernels**: Gramine (Graphene) [26, 25], OSv [8], HermiTux [15, 16], and Unikraft [29]

Although strip-down unikernels provide a high level of compatibility, they typically can't achieve the same level of minimality as clean-slate unikernels [75]. Additionally, they often pose extensive dependencies on the underlying platform which increases porting efforts. The Linux-based unikernels *Lupine* and *UKL* specialize a typical Linux kernel and optimize it for unikernel properties. Porting effort is expected to be substantial to realize a microkernel as the underlying execution platform for these unikernels. The *Rumprun* unikernel allows to reuse unmodified NetBSD kernel components. Compared to Linux-based unikernels, this approach allows for easier portability as the NetBSD kernel is more structured than the Linux kernel [12]. The *Rumprun* unikernel was already ported to the seL4 microkernel and mostly focuses on NetBSD components [11, 12].

Clean-Slate unikernels are optimized for the unikernel environment and typically provide a clearly defined platform interface that allows for easy platform integration. A Linux compatibility layer is constructed that maps down to unikernel-specific implementations. *Gramine* is a lightweight library OS that supports multi-process Linux applications with minimal host requirements. It currently only supports Linux itself and Intel SGX enclaves on Linux as the underlying platforms [25]. Platform integration becomes a problem for a microkernel, as the platform layer of Gramine requires high-level Linux concepts that are not provided by a basic microkernel. *OSv* is another clean-slate unikernel approach that aims to run unmodified Linux applications [8]. Microkernel integration might become a problem as the unikernel is specifically targeted for hypervisor environments. Additionally, the *OSv* core was written in C++ [8], which might pose problems when integrating with a C-based microkernel project. *HermiTux* is another unikernel approach to support native Linux applications. It implements several mechanisms to support a Linux binary compatibility layer. The *HermiTux* unikernel is no longer actively maintained and explicitly points to *Unikraft* as its successor [109].

7.2.1 Unikraft

Unikraft is a clean-slate unikernel that not only provides a Linux/POSIX compatibility layer but also easy portability by defining a thin platform layer. An extensive benchmarking study was performed that evaluates the *Unikraft* unikernel against other popular unikernels [73]. It was shown that *Unikraft* surpasses its competition in terms of image size, memory consumption, boot times and actual application performance. Compared to other unikernels [127, 128], *Unikraft* takes a proactive approach for common security vulnerabilities [129]. It provides a highly modular system architecture and is actively maintained. Based on these considerations, the prototype implementation will utilize *Unikraft* as the unikernel of choice.

7.3 Overview

7.3.1 General Architecture

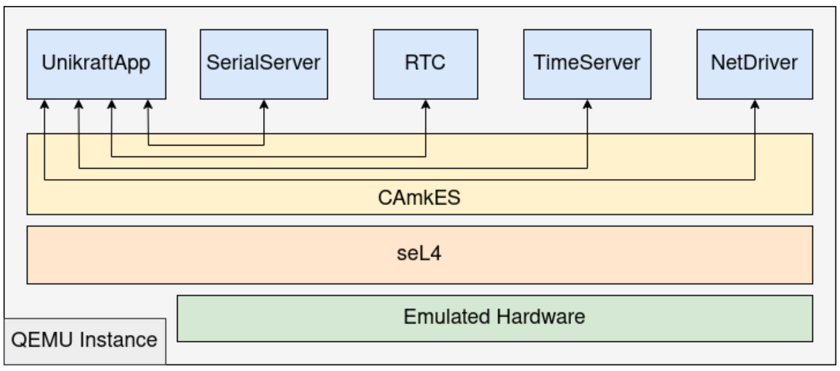


Figure 7.1: General System Architecture.

The overall system architecture is shown in Figure 7.1. The system consists of several CAmkES components that are connected through the CAmkES framework that abstracts over the low-level primitives of the seL4 microkernel. CAmkES components represent microkernel processes that run in separate address spaces with associated threads and can interact with each other through high-level IPC mechanisms such as Shared Memory, Remote Procedure Calls (RPC) and Notifications [130]. The Unikraft application and framework run inside the *UnikraftApp* CAmkES component. It is connected to several other driver components through the described IPC mechanisms. For testing purposes, the entire system is executed inside a QEMU instance that provides the emulated hardware.

UnikraftApp Component

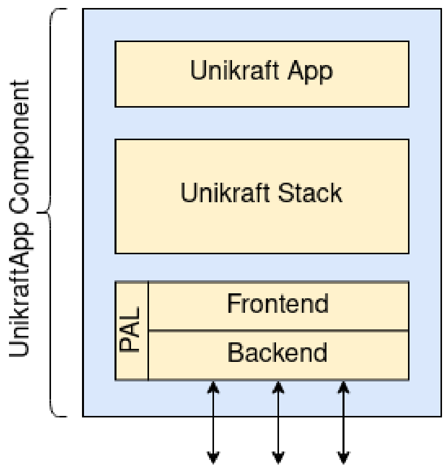


Figure 7.2: *UnikraftApp* Component.

The *UnikraftApp* CAmkES component as shown in Figure 7.2 is effectively Unikraft as a static library together with some wrapping code to connect Unikraft with seL4/CAmkES concepts. It simulates the design of a VM by splitting the PAL into frontend and backend implementation in order to provide different contexts for the build system. The frontend is a thin wrapper that has access to everything Unikraft provides and has no concept of seL4/CAmkES definitions except for the backend API. The functions of the backend API are defined as *extern* symbols and can be called as normal function calls. This way, the Unikraft build system can build a static library without any dependency on seL4. The backend implements this backend API and has access to everything that seL4/CAmkES provides.

7.3.2 Build System

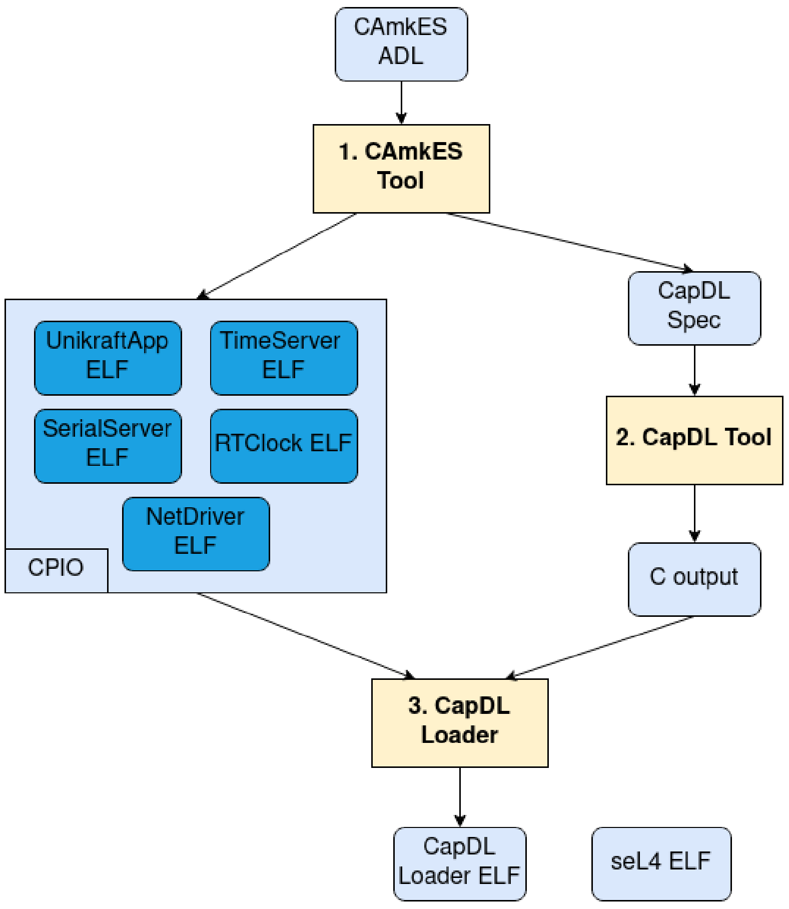


Figure 7.3: General Build System.

The presented system architecture in Figure 7.1 with all its connections and interfaces is specified through the CAmkES ADL. The overall build system is shown in Figure 7.3:

1. **CAmkES Tool**: The CAmkES tool first compiles the different CAmkES components into separate binaries together with generated glue code. The binaries are put into a CPIO archive for later use. Afterwards, the CAmkES tool generates a CapDL spec describing the capability distribution of the entire system [131].

2. **CapDL Tool/Translator**: This program transforms a CapDL spec into several formats, most importantly a C file for linking against the CapDL Loader to create a program that instantiates the system described by the CapDL spec [132].

3. **CapDL Loader**: The CapDL Loader must be linked against the C output of the CapDL Tool and the CPIO archive containing ELF images of all components [132].

Besides the CapDL Loader app, the build system also generates a separate binary for the seL4 microkernel.

UnikraftApp Component Build

The seL4 microkernel uses a CMake-based build system together with the *Ninja* build tool [133]. Unikraft on the other hand uses classic GNU *make* that can be configured with *make menuconfig*. In order to integrate Unikraft into a CAmkES component, we need to unite the build systems between Unikraft and seL4/CAmkES . In the project this is done according to Figure 7.4.

In order to integrate Unikraft into the seL4/CAmkES build system, we trigger a Unikraft build in CMake as an external project (step 1). This allows us to use the original Unikraft build system. The Unikraft build system must be configured manually before the build is triggered (step 0). The linker script in the Unikraft platform layer (*Linker.uk*) typically generates an ELF binary. It is modified for the seL4 platform in order to generate two by-products:

- **Static Library - uklib.a**: The generated object files of the Unikraft build system are put into a static library *uklib.a* instead of linking them all together into an ELF executable. Functionality that must be provided by the seL4 backend is marked as an *extern*-defined symbol. This way, Unikraft can generate object files and put them into a static library without the knowledge of any seL4-/CAmkES-specific symbols. In order to avoid symbol clashes with the provided CAmkES environment in the final link step, we had to rename certain symbols.
- **Linker File - uklib.lds**: Depending on the configuration, some Unikraft libraries come with additional linker information that is dynamically collected in the *uklib.lds* linker file depending on the Unikraft configuration.

The linker file *uklib.lds* is attached to the default CAmkES-specific linker script (*linker.lds*) that is put into the build directory of every CAmkES component (step 2). Afterwards, the static library *uklib.a* is linked against the *UnikraftApp* CAmkES component (step 3). Extern-defined symbols in the object files of the static library are resolved by the seL4/CAmkES linker.

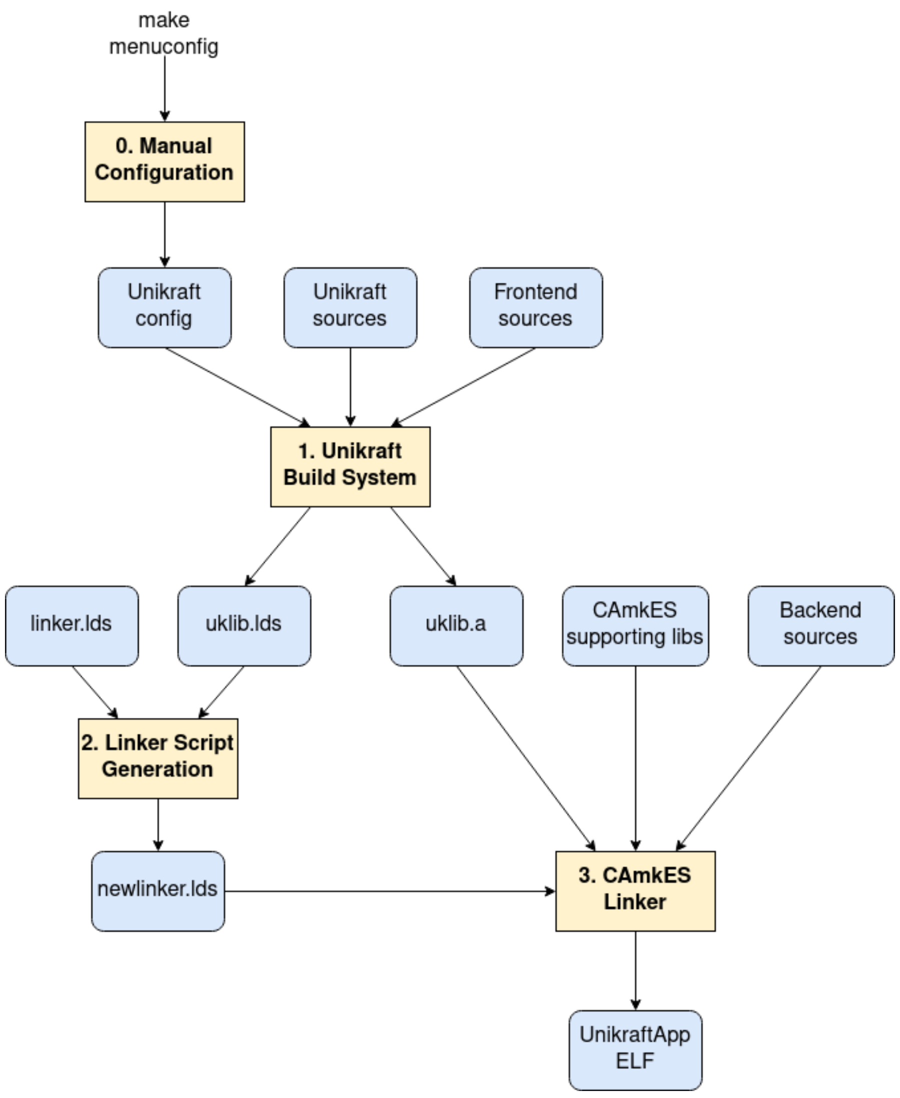


Figure 7.4: *UnikraftApp* Component Build.

7.3.3 System Startup

At startup, the seL4 microkernel ELF is loaded into memory by an initial bootloader, e.g. GRUB. The seL4 microkernel creates a minimal boot environment for the CapDL Loader app that is loaded into RAM and runs as the initial user-level task, i.e. the root task. It is given full access rights to all system resources left over by the kernel, including physical memory, I/O ports and interrupts [134, 5]. The CapDL Loader app was linked against the CapDL spec, that describes our system, and a CPIO archive, containing the ELF binaries of each CAmkES component. It initializes the seL4 user-level environment, creating kernel objects and distributing capabilities according to the CapDL spec. It further loads programs from the provided ELF binaries in the CPIO archive to memory locations specified by the CapDL spec. Finally, it starts all the threads it created and sleeps forever [132]. Each CAmkES thread first jumps into the *main* function, i.e. the C entry point of a CAmkES component. Afterwards, generated code initializes the component and synchronizes all component threads. A CAmkES component contains at least one active (control) thread, one (debug) fault handler thread and another thread for each interface invocation. Active CAmkES components, e.g. the *UnikraftApp* component, eventually call the CAmkES entry point *run* that contains user-defined code [135].

UnikraftApp Component Startup

As described, the control thread of the *UnikraftApp* component drops into the CAmkES entry point *run* after initial component setup. This thread then jumps into the Unikraft entry point through an *extern*-defined symbol. Subsequently, Unikraft code starts its own environment setup before reaching the *main* function of the Unikraft application. The boot sequence of Unikraft typically calls into the platform layer in order to initialize hardware. Since we run in a user space process, there is no need for this kind of initialization, reducing the boot time compared to a Unikraft guest that is started in a virtual environment from scratch.

7.4 CPU Model

7.4.1 Threading Model

Unikraft implements a cooperative scheduler, while the seL4 microkernel uses a preemptive, priority-based round-robin scheduler [136]. For each CAmkES component, the CAmkES framework typically creates one active control thread, one thread per interface invocation and a separate fault handler thread [135]. In order to realize the user-level threading idea, all Unikraft threads run on the control thread of the *UnikraftApp* component. Each unikernel thread consists of a thread-specific context including user space registers, stack pointer, and instruction pointer. Additionally, Unikraft utilizes the *FSGSBASE* instructions to reset the TLS base address on every context switch. The CAmkES control thread only sees the thread context of the currently executing Unikraft thread. It was documented that "the kernel will consider the register used for the TLS base and all thread identifier registers that can be written to/from user mode to be general-purpose registers and saves and restores them with all other registers upon trap and context switch" [137]. This means during scheduling of the microkernel thread, it will store and restore the currently executing Unikraft thread context. The Unikraft threads will only run when the CAmkES control thread is scheduled by the seL4 scheduler.

7.4.2 Event Handling

Traps

The most common usage of traps include the system calls. Similar to other kernels, the microkernel implements their own syscall handler function that gets triggered every time user space executes a *syscall* instruction. This way, the seL4 microkernel exposes a syscall API to user space to provide common kernel services [134].

Faults

CAmkES typically generates by default for each CAmkES component a debug fault handler thread that will print fault-specific information when triggered [138]. This does not fully implement the fault handler model presented in subsection 6.1.2, as it does not necessarily correct the fault and resume the faulting thread. But when a Unikraft thread causes a fault, the corresponding control thread is blocked and the fault handler thread gets notified. This way, the user gets at least some basic debugging information about the fault. For the scope of the thesis, this basic fault handling is sufficient. Additional user-level threads can theoretically be created through the use of CAmkES templates that then listen on the fault handler endpoint to implement a more sophisticated fault handling scheme [135].

Interrupts

In Unikraft, the PAL allows to register callback functions for specific interrupt numbers. When a specific interrupt arrives, the underlying platform must execute the registered callback function. This way, Unikraft realizes interrupt-driven device drivers. For non-maskable interrupts, e.g. the timer, the handler function must be executed immediately. For maskable interrupts, interrupt processing can be deferred to a later point in time through the *ukplat_lcpu_irqs_handle_pending* function. The cooperative scheduler calls this function on every reschedule. In the PAL, we can utilize seL4 notification objects to asynchronously receive device interrupts [134]. The CAmkES framework creates separate threads to asynchronously receive these interrupts or notifications independent of the *UnikraftApp* control thread [135]. At the moment, we only implement a rudimentary interrupt handling scheme in the PAL that relies on the external *TimeServer* component to provide a continuous tick for timer-related behavior. Since a timer represents a non-maskable interrupt, we immediately call the corresponding handling function from the timer interrupt handler thread on every notification event. In case actual device interrupts shall be received, at least one additional device interrupt thread is needed. The corresponding handler functions can be processed by the *ukplat_lcpu_irqs_handle_pending* function for maskable device interrupts. This way, interrupt delivery may get synchronized between the device interrupt handler thread and the unikernel threads that run on the *UnikraftApp* control thread.

7.4.3 ISA Functionality

Global Interrupt Control

Control over system-wide interrupts is highly discouraged and not possible on seL4 [139]. This mainly boils down to the user space restriction as the *UnikraftApp* component runs together with other CAmkES components on the same physical processor. We implement a software switch in the PAL backend that dictates, whether arriving interrupts shall be forwarded to the frontend or not. When disabled, all interrupt handler threads will simply ignore incoming interrupts and therefore interrupt delivery to the control thread is blocked. For the unikernel threads it looks like interrupts are disabled globally. This feature is mostly relevant for maskable interrupts.

Halting the CPU

Halting the entire CPU is also not possible from user space, since other seL4 threads might need to be scheduled on this CPU. Unikraft halts the CPU either indefinitely or the CPU is halted for a specific time interval. Both cases get unblocked when an interrupt arrives (as long as global interrupts are enabled). In the PAL, we utilize a CAmkES event object [135] to simulate the halting behavior. An event can either be received by a callback function, through a blocking wait or a synchronous polling operation. Every interrupt handler thread of the *UnikraftApp* component emits an event to signal that an interrupt/notification has arrived. When the unikernel requires to halt the CPU indefinitely, the control thread simply waits on the event object until a signal is consumed. In case the CPU shall be halted for a specific time interval, the control thread loops until either the time interval has passed or the event was signalled. Instead of waiting on the event object, the control thread uses the polling option provided by the CAmkES event object.

Syscall Handling

Unikraft typically registers its own syscall handler function from the *syscall shim* layer in the MSR LSTAR register during startup. On every *syscall* invocation, program flow would be directed to the registered syscall handler function. We can not use this approach for the microkernel since the unikernel process is restricted to user space and therefore can not use the privileged instructions necessary to modify MSR registers. The seL4 microkernel can theoretically be configured to expose these instructions to user space, but it is considered dangerous and therefore will not be used in the prototype system [140]. Modifying the MSR register would also overwrite the seL4-internal syscall handler function, rendering the microkernel API useless. In case Unikraft is executed as a Linux user space process (*Linuxu* [34]), it utilizes the actual Linux syscall API for syscall handling. Since we run Unikraft as a seL4/CAmkES process, we are restricted to the seL4 syscall API from user space. The seL4 microkernel implements its own set of system calls and generates an exception for unknown syscalls [2]. As a result, when the *UnikraftApp* component issues a Linux system call, it will actually trigger an exception in the microkernel. Based on these assumptions, it was not possible to support Linux runtime binary compatibility in the prototype system.

TLS Handling

The seL4 microkernel can be configured to enable the *FSGSBASE* instructions during startup. This setting is actually part of the verified configuration [141] for the *x86_64* architecture on seL4 [142]. The *UnikraftApp* component relies on this feature to provide proper TLS handling for Unikraft threads from user space. We utilize the *sel4runtime_set_tls_base* and *sel4runtime_get_tls_base* functions to set and get the TLS base address. These functions map internally to the *wrfsbase* and *rdfsbase* instructions. One problem with the presented threading model is that it doesn't account for the fact that both microkernel and unikernel environment may have their own thread-specific view on data. But the seL4 microkernel expects the address of the IPC buffer to be part of the TLS region [137]. The IPC buffer is required for the thread to perform any invocations or system calls, i.e. basically to interact with kernel or other user space processes. To solve this problem, Unikraft threads must be made aware of the IPC buffer on their TLS region. We reuse the original IPC buffer of the control thread for all Unikraft threads that run on this CAmkES thread. Before resetting the TLS base address on every Unikraft thread context switch, we read the IPC buffer of the previously scheduled Unikraft thread through the *seL4_GetIPCBuffer* function. After changing the TLS base address to the newly scheduled Unikraft thread, the IPC buffer address gets copied into the TLS region through the *seL4_SetIPCBuffer* function.

7.5 Memory Model

7.5.1 Paging

The CapDL Loader, i.e. our root task, is responsible for setting up the address space of each CAmkES component. It is given the necessary system resources and capabilities from the kernel at startup. The address space layout can be freely constructed according to user-level policy, the only restriction is that the kernel itself must be mapped into the high part of the virtual address space [123]. This way, the root task creates a similar address space layout as described during the design [124]. Since device drivers are running in separate CAmkES components, and thus separate address spaces, the *UnikraftApp* component does not have any knowledge or required capabilities to access corresponding device-related information.

7.5.2 Memory Allocation (Heap)

The PAL must provide a contiguous memory region for Unikraft's allocator libraries to implement memory allocation. Each CAmkES component gets by default a separate copy of the seL4 *musl libc* library compiled into the address space [143]. This library comes with a predefined heap memory region with a default size of 1 MiB [144, 145] that is typically reused as the component's heap [146]. This heap area is implemented as a static character array located in the data section of the address space [147] and can be resized in the CAmkES specification through the *heap_size* attribute. The dimensions of the CAmkES heap area can be accessed through the variables *morecore_area* and *morecore_size* [135]. The PAL uses this information to provide the required heap information for the Unikraft allocators.

7.6 I/O Model

As was described in the design phase, the I/O model implements device drivers in user space either inside the unikernel process itself or they run in an external component. Since Unikraft is mainly targeted for hypervisor platforms and to increase the overall isolation of the *UnikraftApp* component, we use existing device driver components that were already part of the CAmkES framework. As a result, the *UnikraftApp* component can not access device hardware directly, but instead relies on these external components to provide the required services. This makes the overall system architecture more secure as the CAmkES components are completely isolated, except for the few interactions permitted by the CAmkES IPC mechanisms. We focus on the following I/O devices to provide the necessary device abstractions in the PAL for higher Unikraft layers:

- Timer: *TimeServer* component
- Real-Time Clock (RTC): *RTC* component
- Console: *SerialServer* component
- Network Device: *NetDriver* component
- Storage Device: not implemented

7.6.1 Timer: *TimeServer* component

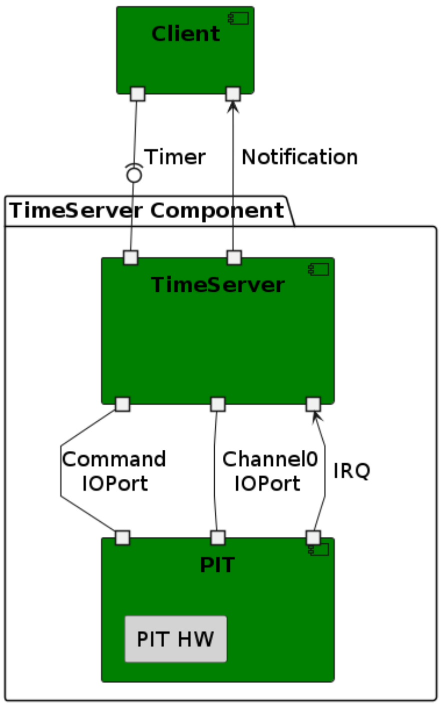


Figure 7.5: *TimeServer* Component.

The *TimeServer*[1] CAmkES component allows multiple clients to connect to the Programmable Interrupt Timer (PIT) [148] as is shown in Figure 7.5. It consists of a driver component (TimeServer) and the actual device hardware (PIT). The driver component interacts with the PIT hardware through I/O ports, one for *Channel 0* and the other to control the *Command* register. The PIT can be programmed to generate a hardware

[1]`https://github.com/seL4/global-components/tree/master/components/TimeServer`

interrupt either as a one-off event or periodically [148]. The driver component exposes the *Timer* interface together with a notification object to other CAmkES components. The *Timer* interface allows connecting clients to request common timer functionality through RPC calls. These include setting one-off or periodic timer events, as well as getting timer ticks. The notification object notifies the connecting client when a programmed timer event has passed.

PAL Integration The *UnikraftApp* component connects to the *TimeServer* component through the described interfaces to provide the following functionality:

- **Monotonic Clock**: A Monotonic Clock is a constantly increasing timer which guarantees that it will never go backwards in time. The absolute value of the timer is of little use, but monotonic clocks can be used to measure durations with high certainty [149]. The *UnikraftApp* component requests the PIT monotonic clock through the *Timer* interface.
- **Timer**: Unikraft initializes a periodic tick timer at system startup. For this purpose, the *UnikraftApp* component registers a periodic timeout in the *TimeServer* component and then gets notified on a separate handler thread on every timer tick event.

7.6.2 RTC: *RTC* component

The *RTC*[2] CAmkES component allows multiple clients to connect to the Real Time Clock (RTC) as is shown in Figure 7.6. Internally, 64 bytes of CMOS RAM are used to provide wall clock time [150]. The CAmkES component consists of a driver component (RTC) and the actual device hardware (CMOS). The driver component interacts with the CMOS through two I/O Ports, one for the data and the other for the address [151]. The driver component exposes the *RTC* interface through which connecting clients can get wall clock time.

PAL Integration The *UnikraftApp* component connects to the *RTC* component through the *RTC* interface to provide the following functionality:

- **Wall Clock Time**: Wall Clock time is counted as the number of time units since a specific epoch in the past [149]. The *UnikraftApp* component requests the wall clock time directly through the *RTC* interface.

[2]`https://github.com/seL4/global-components/tree/master/components/RTC`

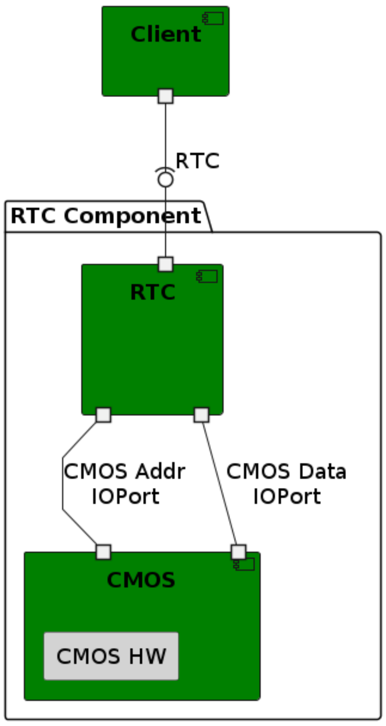


Figure 7.6: *RTC* Component.

7.6.3 Console: *SerialServer* component

The *SerialServer*[3] CAmkES component allows multiple clients to connect to a serial hardware device to write on the console as is shown in Figure 7.7. The CAmkES component consists of a driver component (SerialServer) and the actual device hardware (Serial). The serial device generates a hardware interrupt and can be controlled through a separate I/O Port. The driver component exposes three different interfaces to connecting clients:

- **PutChar**: This interface allows to put a single character on the console.
- **GetChar**: This interface is stubbed. Clients can read characters from the *GetChar* memory region that is shared with connecting clients.
- **Batch**: This interface allows clients to put multiple characters stored in the *Batch* memory region onto the console at once.

[3]`https://github.com/seL4/global-components/tree/master/components/SerialServer`

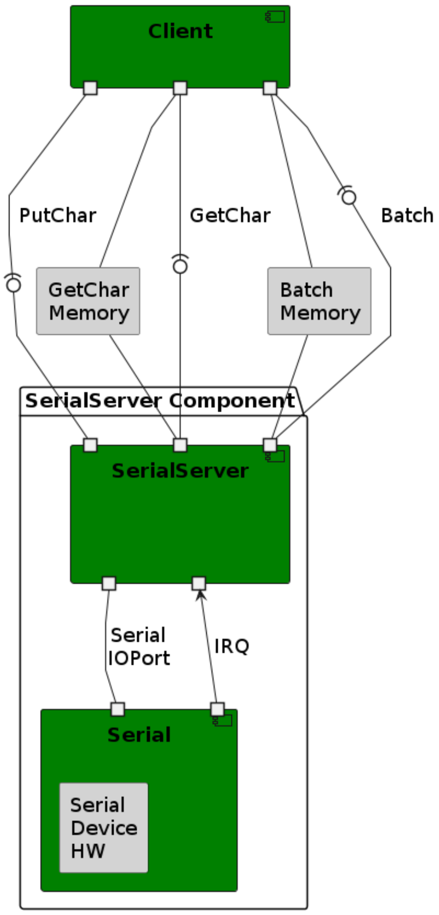


Figure 7.7: *SerialServer* Component.

PAL Integration The *UnikraftApp* component connects to the *SerialServer* component through the described interfaces to provide the following functionality:

- **Console Output**: The *UnikraftApp* component utilizes the *Batch* memory to write multiple characters at once to the console. This is more efficient than the simple *PutChar* interface as it only involves a single RPC call to transfer multiple characters.
- **Console Input**: The *UnikraftApp* component utilizes the *GetChar* memory region to get user input.

7.6.4 Network: *NetDriver* component

It was not possible to utilize the *Ethdriver*[4] CAmkES component in QEMU. Consequently, a virtio-based network driver was implemented in the *NetDriver* CAmkES component that allows for a single connecting client as shown in Figure 7.8. I was provided an existing seL4 network driver and repurposed it for the CAmkES framework. The *NetDriver* component itself consists of two parts: the *NetDriver* component that implements the driver logic and the *NetDriverHW* hardware component that abstracts over the (emulated) ethernet device. More precisely, QEMU exposes a PCI-based virtual network device to the system that the network driver can interact with through the virtio standard [152, 153, 154]. PCI acts as the transport layer between the driver and the ethernet device and consists of three different I/O Ports: Config Data Port, Config Address Port, and I/O Space Port [155]. Additional memory regions must be defined in the CAmkES ADL for PCI and ethernet I/O memory. The *NetDriver* CAmkES component exposes the *NetDriverInterface* interface to the connecting client as well as two dataports for the client to read and write ethernet frames. QEMU effectively forwards incoming and outgoing frames through a tap interface to an external network bridge [156]. External components such as a DHCP server or networking scripts can connect to this network bridge in order to communicate with the QEMU network device, and thus ultimately with the network stack of Unikraft.

PAL Integration

The *UnikraftApp* component connects to the *NetDriver* component through the described interface to send and receive ethernet frames as well as requesting the MAC address. A thin device driver stub is placed in the PAL that connects the Unikraft-internal *uknetdev* networking API to the external device driver API. Higher Unikraft software layers such as the network stack utilize this network-related API to implement network functionality.

Limitations

The current network driver implementation is far from ideal: It is polling-driven and only allows for a single connecting client. Additionally, the network driver represents a virtio-based driver that makes use of an emulated ethernet device in QEMU. These limitations must be addressed in future work.

[4]`https://github.com/seL4/global-components/tree/master/components/Ethdriver`

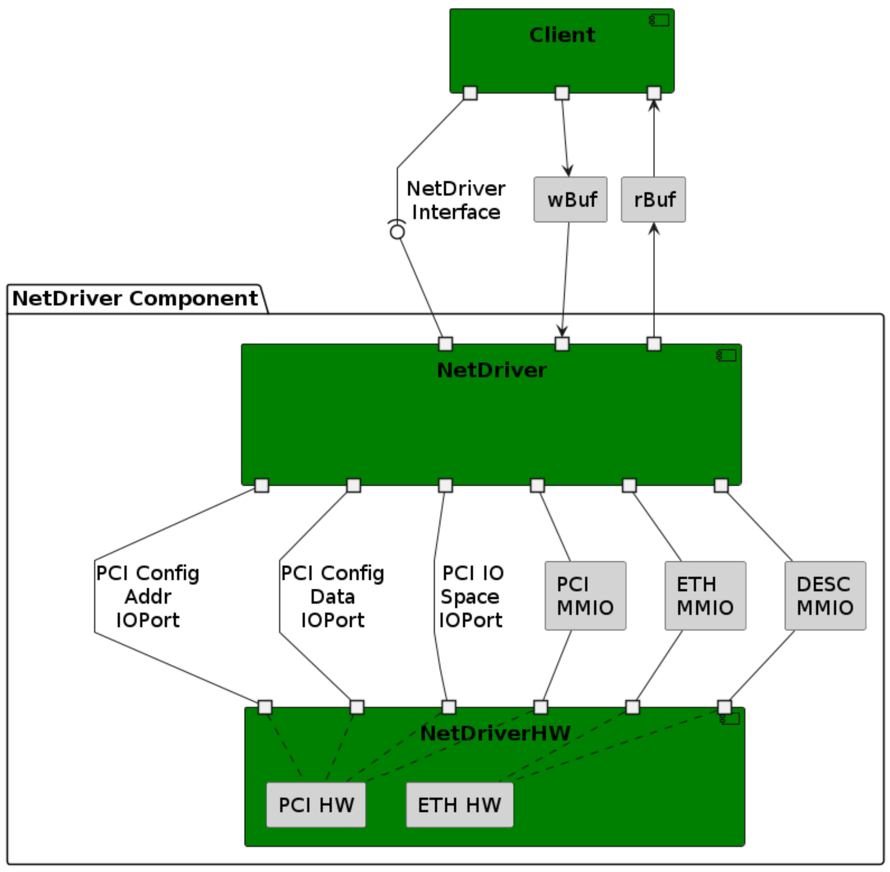


Figure 7.8: *NetDriver* Component.

7.6.5 Storage

There was no time to implement a separate storage device driver. Therefore, it is not possible to provide persistent storage functionality to the unikernel. Such a storage driver would normally be utilized to implement the *ukblkdev* API of Unikraft that is used by higher software layers such as the filesystem implementations.

7.7 Obstacles & Workarounds

Several obstacles appeared during the implementation that the design does not account for. We present workarounds that were implemented to circumvent these obstacles and their limitations.

7.7.1 Obstacle 1: No Storage Driver

Unikraft implements a block device driver API (*ukblkdev*) through which higher software layers can interact with a storage device in the PAL. The most prominent example are filesystems that require storage drivers to implement persistent storage. Since we do not have a block device driver, we need to find a workaround for filesystem-related functionality in Unikraft.

Workaround One solution that does not require physical storage driver support is to use the RAM filesystem (*RAMFS*) of Unikraft. It simply operates on the assigned heap memory region provided in the PAL. Unikraft extends this concept with the *InitRD* filesystem that essentially extracts a CPIO archive into RAM during startup and then operates on the contents in RAM as a normal RAM filesystem. The advantage of this approach is that we can actually pass existing files and directories from the outside to Unikraft. This is useful if applications require to operate on existing files and directories. The corresponding CPIO archive is normally passed as the InitRD flag in Unikraft when starting QEMU. In CAmkES, this flag is already used to pass the CapDL Loader as a boot module. A workaround is to actually integrate the CPIO archive into the *UnikraftApp* component binary and simply define a separate InitRD memory region in the PAL that reads the CPIO archive memory. The Unikraft internal *ukcpio* library is used by the *InitRD* filesystem to load contents of the provided CPIO archive into memory and setting up the filesystem in RAM.

Limitation Although the *RAMFS* and *InitRD* filesystem of Unikraft allow us to use filesystems and thus the *vfscore* API of Unikraft, we do not support filesystems that require persistent storage to disk. One such filesystem is the *9pfs* filesystem that utilizes the *ukblkdev* API to interact with an actual storage device. The underlying issue is the missing block device driver in the PAL.

7.7.2 Obstacle 2: Symbol Clashing

Since we statically link the Unikraft sources into the CAmkES component, we need to make sure that existing symbols are not clashing with Unikraft-related symbols. Although basic seL4 processes do not dictate a specific C runtime [142], the CAmkES environment comes by default with several supporting libraries, most notably the *musl libc* library [143]. A separate copy of this library is included in every CAmkES component by default. There is no easy way to remove the seL4 *musl libc* library from a CAmkES environment without meddling with the CAmkES build system. Unikraft comes with their own implementation of popular libc libraries, such as *musl* and *newlib*.

The problem is we can not have two symbols of the same name in the component address space without causing the linker to complain.

Workaround 1 The first workaround uses the seL4 *musl libc* library also for Unikraft itself. This is implemented by bringing the seL4 *musl libc* library headers into the Unikraft build system as a separate Unikraft library (*seL4libc*). The Unikraft build system is configured to use these headers instead of their own libc implementations and the resulting object files reference these missing symbols. When linking into the CAmkES component the missing symbols are resolved by the CAmkES linker. In some occasions, it was necessary to overwrite the seL4 default implementation of certain *musl* symbols to make it work with the Unikraft framework. In the final link step, the Unikraft-internal implementation overwrites the default seL4 *musl libc* implementation.

Workaround 2 The second workaround uses the actual Unikraft *libc* libraries. Since the CAmkES framework provides each CAmkES component with a separate copy of the *musl libc* library by default, we need to rename all libc-related symbols in the *UnikraftApp* component to avoid clashing in the final link step. This potentially doubles the amount of libc symbols in the component's address space. Since we do not explicitly optimize for memory consumption, the applied technique represents a viable solution for our purposes.

Limitation Although the first approach allows using a single libc implementation in the component address space, it comes with a certain porting cost to the Unikraft framework as the seL4 *musl libc* library is not optimized for the Unikraft environment. The second workaround potentially doubles the number of libc symbols and requires a renaming scheme for Unikraft-related symbols to avoid symbol clashing during linking.

7.7.3 Obstacle 3: Section Naming & Placement

The CAmkES environment typically sets up a runtime environment for each CAmkES component [135, 157]. Similar to the problem before, Unikraft defines sections for the final ELF that are already present in this CAmkES environment, e.g. constructor and destructor tables for the C runtime [157]. Unikraft might also require additional section definitions depending on the library configuration that must be dynamically included in the final CAmkES linker script.

Workaround In case Unikraft defines a section that is used by the CAmkES environment and the linker complains about double section placement, we had to rename the corresponding section in code and linker scripts. For certain Unikraft libraries specific linker snippets are defined that must be added dynamically to the linker script depending on the Unikraft configuration. The Unikraft *Linker.uk* script is instructed to collect all enabled linker snippets and other required section definitions and place them all in a separate linker script *uklib.lds*. This linker script is part of the Unikraft build system output as was explained before. It is concatenated to the CAmkES-specific linker script *linker.lds* in order to define the required Unikraft-specific sections in the final ELF.

Limitation The placement of each section in the final ELF and which sections need to be renamed might need to be revised in the future. For certain section definitions, I for example had to change the relative location in the component address space to avoid clashing into CAmkES-specific linker bounds.

7.8 Status

PAL In order to support higher Unikraft layers, it was necessary to implement the low-level functions of the PAL for the seL4 microkernel. The presented resource models behave mostly as described in the design phase.

OS Core The Unikraft OS core libraries implement basic OS functionality similar to the Linux subsystems, but in a much more modular fashion:

- **Scheduler**: We only support a cooperative scheduler (*ukschedcoop*), since it is also the one selected by default.
- **Memory**: Unikraft relies on memory allocators to implement basic heap management through the use of the provided memory regions from the PAL. The seL4 microkernel was tested with the binary buddy allocator (*bbuddy*), the region allocator (*region*) and the *tlsf* allocator.
- **Filesystem**: The seL4 microkernel supports the *RAMFS* and *InitRD* filesystem of Unikraft, and potentially any filesystem that does not require actual physical storage support. Since no storage driver is provided, we can not use the *9pfs* filesystem. Higher software layers interact with the selected filesystem through the *vfscore* API.
- **Network**: Network functionality is typically provided via the *posix-socket* API that abstracts over a specific network stack. The network stack uses the functions

provided by a network device via the *uknetdev* API. We support the *lwip* network stack for the seL4 microkernel. This network stack supports DHCP auto configuration.

Since Unikraft runs single-process applications, there is no need for it to implement extensive IPC mechanisms. The CAmkES framework makes sure that different user space processes are properly connected.

Compatibility Layer Unikraft provides several layers to implement compatibility with POSIX-like applications. These include:

- **POSIX Libraries**: POSIX functionality that was required by the tested applications were all sufficiently supported for seL4.
- **Libc Libraries**: Unikraft provides *musl* and *newlib* as native libc ports. Through a simple renaming of symbols, it was possible to use these Unikraft libc implementations. Additionally, a separate seL4-related libc library was provided that mainly contains the headers of the seL4 *musl libc* library (*seL4libc*). This library brings advantages in terms of final binary size but requires to be optimized for Unikraft first.

Application Support The current system provides support for the following applications: *nginx*, *redis*, *sqlite*, *lua*, and *micropython*. Other applications might also be supported but were not tested on the prototype system. Since we support the *musl* libc implementation of Unikraft, we can utilize the autoporting feature. This way, porting efforts can be drastically reduced. The Linux runtime binary compatibility feature of Unikraft is not supported for the prototype implementation as it requires registering a syscall handler function.

Limitations The current implementation and workarounds lead to several limitations that require follow-up work. These include:

- **Persistent Storage**: The PAL does not support a block device driver. As a result, we do not have persistent storage. Therefore, filesystems that require physical storage drivers are not supported. Adding a block device driver in the platform layer should resolve this problem.
- **Network Driver Design**: The *NetDriver* component only supports a virtio-based network device driver that relies on virtual hardware provided by QEMU and operates in polling mode. In order to execute the system on top of actual

hardware, the network driver in the *NetDriver* component must be replaced with a real physical device driver. To improve performance an interrupt-driven approach must be implemented.

- **Binary Compatibility**: The PAL does not support Linux runtime binary compatibility since a syscall handler function can not be registered from user space and Unikraft does not apply any optimizations to avoid the *syscall* instruction. The Unikraft ELF loader can successfully load a position-independent Linux static ELF into memory, but causes an seL4 exception on every Linux syscall invocation.

7.9 Summary

We saw that it was possible to implement the presented resource models for the seL4 microkernel without the need for expensive virtualization solutions (except for the virtual network driver). We relied on the CAmkES framework to statically instruct the root task (CapDL Loader) to setup a component-based system architecture. The seL4 microkernel utilizes capabilities to enforce isolation between resources on the software side. By focusing implementation efforts on the low-level functionality represented by the PAL of Unikraft, it was possible to support the majority of Unikraft features. Most importantly, we support Unikraft's autoporting feature. This way, real-world applications such as Linux/POSIX applications can be supported on the seL4 microkernel with minimal porting efforts. Several limitations of the current prototype implementation were presented that must be addressed in the future.

8 Evaluation

In this chapter, we evaluate the prototype system against reference implementations along four dimensions:

- **Performance**: How does the prototype system perform compared to unikernels on other platforms?
- **Compatibility**: How much Linux/POSIX compatibility does the prototype system provide?
- **Lightweight Integration**: How does the prototype system compare to VM and container solutions in terms of resource utilization and isolation?
- **Security**: Does the prototype system successfully protect against the threats from the threat model in chapter 4?

The prototype implementation represents our Target of Evaluation (TOE). It will be compared against related solutions, including virtual machines, containers, other unikernels as well as Linux itself.

8.1 Environment

Hardware The evaluation was performed on a *Lenovo ThinkPad E495*. It features an *AMD Ryzen 7 3700U* processor with *Radeon Vega Mobile Graphics*. This processor comprises of 1 socket with 4 physical cores, reaching 8 cores through hyperthreading, each running between 1.4 and 2.3 GHz. The processor features a 3-level cache hierarchy with 384 kiB L1 cache (128 kiB dL1, 256 kiB iL1), 2 MiB L2 cache and 4 MiB L3 cache. System Memory is 16 GiB with 2 x 8 GiB SODIMM DDR4. A 512 GB Samsung SSD PM991 (MZALQ512HALU) is used as disk.

Operating System The measurements were performed on an Ubuntu 20.04.5 LTS (*Focal Fossa*) operating on the Linux kernel version 5.15.0.

Target of Evaluation (TOE) The TOE uses the seL4 microkernel v12.1.0 and Unikraft v0.11.0 (Janus).

8.1.1 Notes on Benchmarking

The evaluation, especially performance-related benchmarking, will be compared to the results provided by the Unikraft team at the *EuroSys* conference 2021 [158, 73]. It is therefore important to work out the differences in the evaluation environments between my measurements and the Unikraft benchmarking suite.

Reference System The Unikraft benchmarking suite states clear requirements in terms of hardware and kernel configuration. They utilize three different physical hosts for their evaluation [159]. All benchmarking tests were run on *Shuttle SH370R6* boxes with an Intel i7 9700K 3.6 GHz (4.9 Ghz with Turbo Boost, 8 cores) and 32GB of RAM. For their measurements, they further disabled hyperthreading (*noht*), isolated 4 CPU cores (*isolcpus*), switched off the IOMMU (*intel_iommu=off*) and disabled IPv6 (*ipv6.disable=1*) by setting kernel boot parameters through the GRUB command line [158].

Target of Evaluation (ToE) For my measurements, I try to adhere as much as possible to these hardware, kernel and Unikraft configuration requirements and state differences when required. The measurements are performed on QEMU with KVM acceleration enabled and utilizing the host CPU. On the hardware side, there are obvious differences: While the Unikraft team ran their experiments on an Intel i7 9700K with 3.6 GHz processor speed, the TOE utilizes an AMD Ryzen 7 3700U with a base clock speed of 2.3 GHz. As a result, CPU-heavy tasks can only be compared in a limited way. Unikraft artifacts rely on Debian Buster with Linux kernel version 4.19, while my measurements were performed on an Ubuntu Focal Fossa with Linux kernel version 5.15.0. I doubt that these kernel differences have any significant effect on the overall evaluation results. In order to have a common denominator, the Linux kernel was configured in GRUB as described for Unikraft. For newer kernel versions, it is recommended to utilize the *nosmt* directive to switch off hyperthreading [160]. The *taskset*[1] Linux command makes sure that the experiments are run on a specific isolated CPU set. In addition to isolating physical CPUs, we have to make sure that the applied CPU frequency is constant for the experiments. This is achieved by setting the maximum CPU frequency of 2.3 GHz with the *cpupower*[2] Linux utility. We want to further disable CPU frequency scaling, i.e. the AMD equivalent to Intel Turbo Boost [161]. The last aspect that needs to be considered during benchmarking is the Unikraft configuration itself and the libraries it uses. Different libc implementations have an impact on the final image size as well as the memory requirements, while different allocators can have measurable effects on performance-related tasks such as boot time and application performance.

[1]`https://man7.org/linux/man-pages/man1/taskset.1.html`
[2]`https://linux.die.net/man/1/cpupower`

8.2 Performance Evaluation

Performance-related benchmarks take a look at the image size, boot time, memory consumption and actual application performance for several popular (cloud) applications:

- **Redis**: An in-memory key-value store [73].
- **NGINX**: A web server, reverse proxy, mail proxy and HTTP cache [162].
- **SQLite**: A self-contained, serverless, zero-configuration, transactional SQL database engine [163].

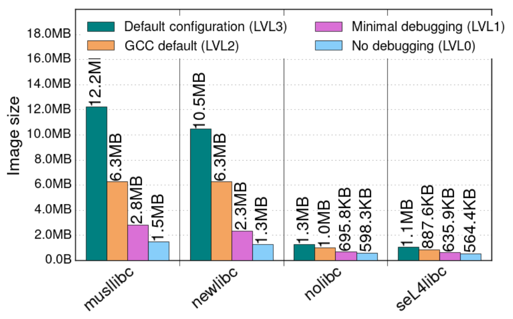
These applications together can build the foundation of a custom technology stack [164]. The measurements further include a minimal *helloworld* application serving as a baseline. The benchmarking techniques as well as the resulting diagrams are heavily based on the official Unikraft benchmarking suite [73]. We focus specifically on the *UnikraftApp* CAmkES component and how this unikernel process compares to the performance results of the Unikraft benchmarking suite. Other CAmkES components in the system, i.e. the driver servers, are not further considered for the evaluation and are expected to exhibit (constant) one-off costs in terms of image size, boot time and memory consumption.

8.2.1 Image Size

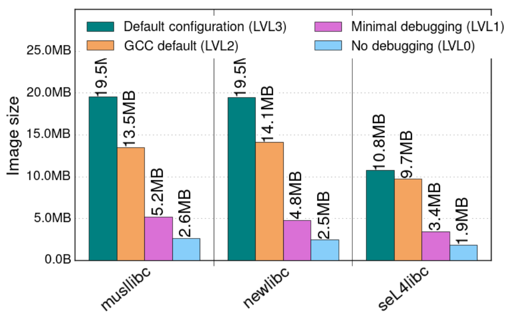
Image size is a good indicator on the amount of included code and hint towards the level of specialization that was performed for the unikernel. Unikraft provides four different debugging levels, increasingly removing debugging information from the final image:

- **Level 3 (-g3)**: The final image includes extra information, such as all the macro definitions present in the program.
- **Level 2 (-g2)**: This represents gcc's default setting.
- **Level 1 (-g1)**: The final image contains minimal debugging information. This includes descriptions of functions and external variables, but no information about local variables and no line numbers.
- **Level 0 (-g0)**: The final image contains no debugging information.

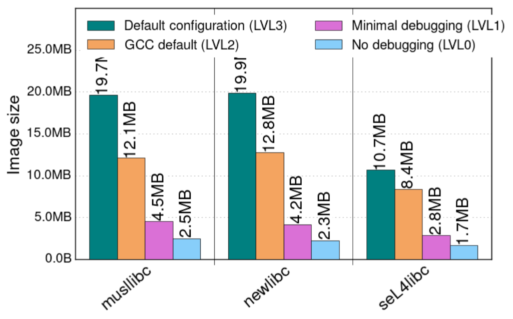
In Figure 8.1, the final image sizes for each of the benchmarking applications is shown depending on the selected libc implementation and debugging level. It demonstrates that using the seL4 *musl libc* headers (*seL4libc*) results in the least amount of code included in the final binary. Using a Unikraft-internal libc implementation, e.g. *musl*, *newlib* or *nolib*, means that extra code will be included in the final image on top of the seL4 headers that are part of each CAmkES component by default.



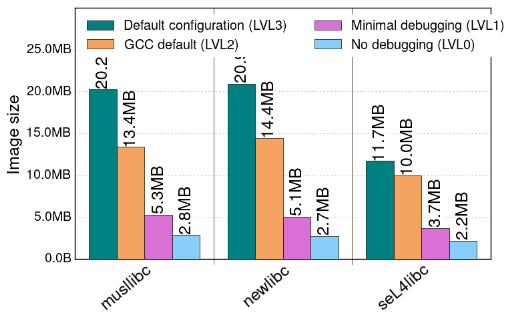
(a) *helloworld*



(b) *sqlite*



(c) *nginx*



(d) *redis*

Figure 8.1: Image Size for different libc implementations and debugging levels.

For the comparison with other unikernels, only stripped images were considered (debug level 0). For each application the smallest of these stripped images was chosen, i.e. the ones that use the seL4 musl headers (*seL4libc*). Figure 8.2 demonstrates that the image sizes of the *UnikraftApp* CAmkES component are in the same ballpark as most other unikernels. The *UnikraftApp* component image is slightly bigger than their Unikraft equivalent. This can be explained through the following observations:

- **CAmkES-related code**: The CAmkES framework includes additional libraries, most importantly a separate copy of the seL4 *musl libc* port, into each CAmkES component by default. This inflates the overall binary size by several hundred kiB. If the Unikraft *newlib* or *musl* implementation are compiled into the *UnikraftApp* component image, potentially twice the amount of libc symbols will be included in the final executable.

- **CPIO Archive**: Since we currently lack a proper storage driver in the prototype system, we have to provide external files through a CPIO archive which needs to be linked into the final component image.

An advantage of the TOE is that device drivers are implemented as their own CAmkES components. As a result, the CAmkES image hosting the Unikraft application only includes minimal driver stubs. This concept can theoretically be extended to other OS services running in the *UnikraftApp* process such as the filesystem and network stack. It is expected that this way the resulting image size for the Unikraft component can be further minimized. For now, we only exclude device drivers and this effect is outweighed by the aforementioned TOE drawbacks. Image size could be further reduced through optimizations such as Dead-Code Elimination (DCE) and Link-Time Optimization (LTO) [73].

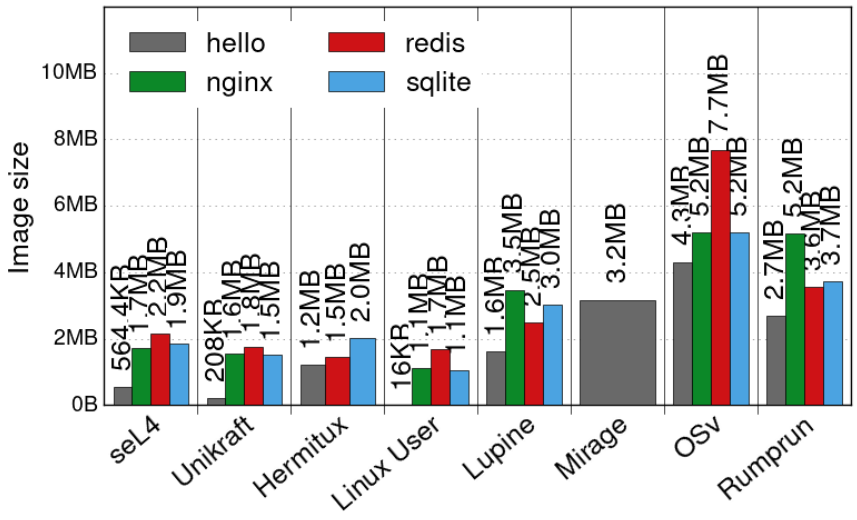


Figure 8.2: Image Size Comparison.

Unikernels	VMM	Guest Boot Time (in ms)
MirageOS	Solo5	1-2
OSv	Firecracker	4-5
Rump	Solo5	14-15
HermiTux	uHyve	30-32
Lupine	Firecracker	70 (18)
Alpine Linux	Firecracker	330

Table 8.1: Guest boot times for different unikernels and VMMs according to the Unikraft benchmarks [73].

8.2.2 Boot Time

Boot Time is an important metric to understand how fast a VM can be instantiated. The Unikraft benchmarks [158] looked at both the time taken by the VMM (Firecracker, QEMU, Solo5) and the boot time of the actual unikernel guest, measured from when the first guest instruction is run until *main()* is invoked [73]. It was shown that Unikraft images boot extremely quickly, ranging anywhere from 3 ms (Firecracker, Solo5) to around 40 ms (QEMU) depending on the VMM used. The total boot time is mostly dominated by the VMM itself. The actual Unikraft guest only takes a fraction of that, ranging from tens (no NIC) to hundreds of microseconds (one NIC). Table 8.1 shows the guest boot time for other unikernels and VMMs, demonstrating that Unikraft images boot extremely quickly.

The TOE runs the Unikraft application as a normal user space process on top of the microkernel. Instead of measuring the boot time of a VMM, we measure the boot time the kernel takes to get to the *UnikraftApp* CAmkES entry point. And instead of measuring the guest start time, we measure the process start time, i.e. the time taken from the CAmkES C entry point to the Unikraft *main* function. In Figure 8.3, the boot time of the TOE is plotted against the Unikraft reference measurements. The overall boot time of around 300 ms is substantially longer compared to the Unikraft results. One reason for this effect is that the seL4 microkernel and CapDL Loader must instantiate all the CAmkES components in the system. At the same time, our measurements were performed on a significantly lower CPU speed (2.3 GHz) compared to the Unikraft benchmarks (3.5 GHz). These factors contribute to the fact that the total boot time of the TOE is longer than the Unikraft reference implementations.

Looking at the startup times of the Unikraft guests paint a different picture. The

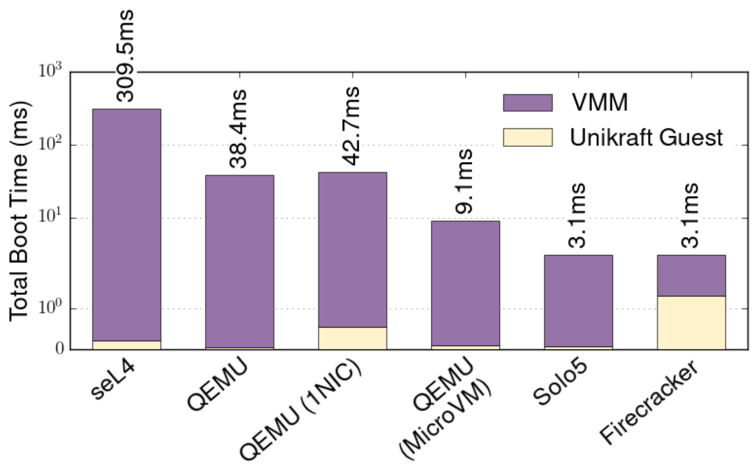


Figure 8.3: Boot Time Comparison.

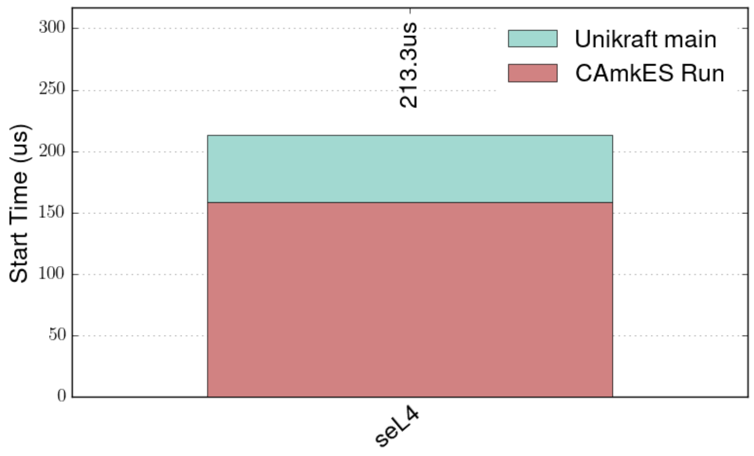


Figure 8.4: Start Time of *UnikraftApp* CAmkES component.

average startup time for the *UnikraftApp* CAmkES component sits at around 200 us. This compares well to other unikernels. For the TOE, the component startup time is split into CAmkES-related code, i.e. the time taken from the CAmkES C entry point to the CAmkES *run* method, and Unikraft-related code, i.e. the time taken from the CAmkES *run* method to the Unikraft *main* method. Figure 8.4 shows that the majority of the startup time is spent in CAmkES-related startup code (around 150 us) and only a fraction in the actual Unikraft code (around 50 us). Any device driver related code is extracted into a separate CAmkES component and thus not required for Unikraft instantiation. As a result, the boot times will be effectively the same, independent of the fact whether a NIC is used or not.

8.2.3 Memory Consumption

Memory Consumption is an important metric for cloud computing scenarios. It helps to understand how many VMs can run simultaneously on a machine with a fixed amount of RAM. In order to find out how much memory each application requires on Unikraft, the Unikraft team applied a binary search algorithm to figure out the minimum amount of memory that needs to be given to the VM to successfully boot. They state that 2-7 MB of memory suffices for Unikraft guests [73].

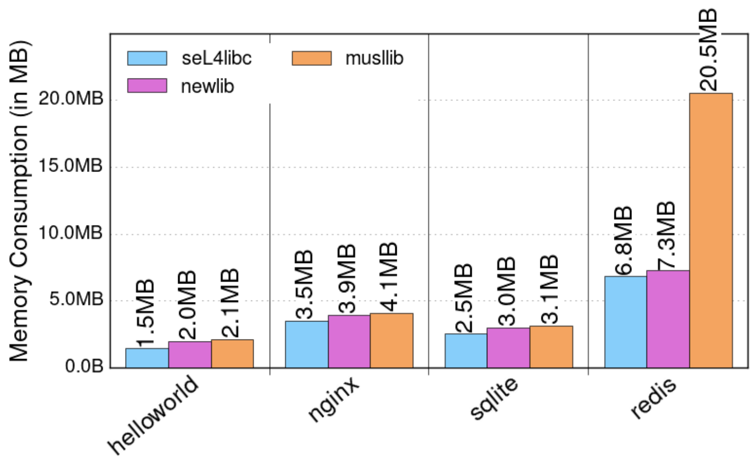


Figure 8.5: Memory Consumption for different applications and libc implementations.

Their method of discovering the minimal amount of memory required to run the application can not be properly applied to the TOE. This is due to the fact that we run multiple user space processes that all require memory. The binary search can only discover the minimum amount of memory required for the entire system. But we are mostly interested in the memory usage of the *UnikraftApp* CAmkES component. The CapDL Loader App is responsible for setting up the page tables of each CAmkES component statically during system startup [165]. This gives a hard upper bound on the amount of memory each CAmkES component consumes. The CapDL Loader can be instructed to print the amount of physical memory that gets mapped into each component's address space. Tracking the memory requirements of the *UnikraftApp* component this way for different applications and libc implementations results in Figure 8.5. It shows clearly that the seL4 musl libc headers (*seL4libc*) require the least amount of memory, mostly due to the reduced code size. For each application, Unikraft requires a differently sized heap: *helloworld* (1 MiB), *sqlite* (1 MiB), *nginx* (2 MiB) and *redis* (5 MiB). For the *musl* libc of Unikraft, we actually require around 18 MiB for the *redis* app. These values were determined through the Unikraft-internal allocation tracking system[3].

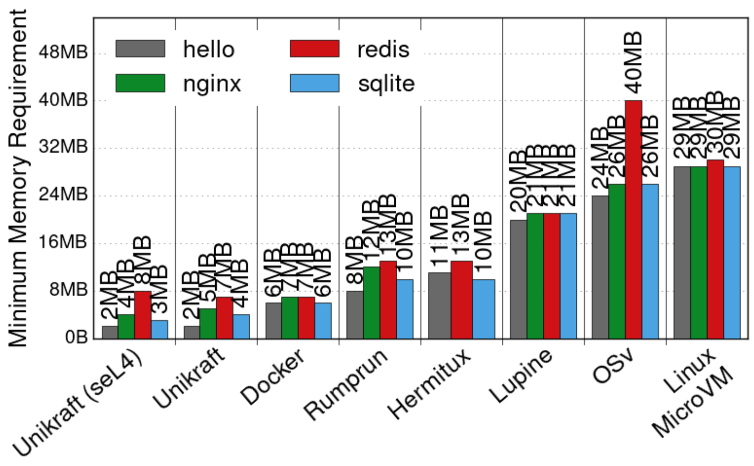


Figure 8.6: Memory Consumption Comparison.

In Figure 8.6, the TOE results were plotted against the official results from Unikraft. The *newlib* images were taken for comparison, since these are the ones used by Unikraft for benchmarking. We can see that the TOE exhibits a similar memory consumption as Unikraft guests. The prototype system has the advantage that only driver stubs are required in the *UnikraftApp* component since the actual drivers are implemented in separate CAmkES components. At the same time, similar to the image size considerations, there are several reasons that additionally increase the memory requirement for the TOE, including CAmkES-related code and the CPIO archive that must be kept in memory. One thing I noticed is that the Unikraft benchmarks do not test for actual application workload but simply check for successful startup in the case of *helloworld* and *sqlite* or ping the network stack in case of *nginx* and *redis* [166]. If pinging alone is enough to evaluate for memory consumption, 2 MiB of heap would suffice for network-related applications on the TOE, shrinking the overall amount of

[3]`https://github.com/unikraft/unikraft/blob/staging/lib/ukalloc/Config.uk#L20-L43`

physical frames that need to be allocated for the *UnikraftApp* component. At the same time, I believe that testing the actual application workload shows a more realistic picture of runtime memory consumption.

8.2.4 Application Performance

Unikraft provides different benchmarks for *nginx*, *redis* and *sqlite*. I try to adhere as much as possible to the configuration used in the official Unikraft benchmarks. In case a different configuration is needed for the TOE, it will be addressed.

NGINX

The Unikraft benchmark[4] uses the *wrk*[5] performance suite to measure the NGINX performance for 1 minute using 14 threads, 30 connections, and a static 612B HTML page. They perform the measurements on one isolated CPU core using the *mimalloc* memory allocator. At the time of writing, the TOE is missing proper *mimalloc* support due to requiring at least 256 MiB of memory [167]. An alternative allocator library is *tlsf*, which performs similarly if not better than *mimalloc* [168]. In Figure 8.7, the performance of the TOE is plotted against the Unikraft results. Compared to other unikernels, the TOE only allows for around 1380 requests/sec, i.e. one to two orders of magnitude less than other unikernels. The actual throughput was also measured with the *iperf*[6] utility to be around 2.35 MBit/s.

Redis

The Unikraft benchmark[7] uses the *redis-benchmark*[8] utility to measure Redis performance. The performance test utilizes 30 concurrent connections doing 100k requests of type *get* and *set* with a pipelining level of 16. Similar to the NGINX benchmark, instead of the *mimalloc* allocator the *tlsf* allocator is used for the prototype system. In Figure 8.8, the results for the TOE are plotted against other unikernels. The results show that the TOE performs two to three orders of magnitudes worse than other unikernels.

[4] `https://github.com/unikraft/eurosys21-artifacts/tree/master/experiments/fig_13_nginx-perf`

[5] `https://github.com/wg/wrk`

[6] `https://iperf.fr/`

[7] `https://github.com/unikraft/eurosys21-artifacts/tree/master/experiments/fig_12_redis-perf`

[8] `https://redis.io/docs/management/optimization/benchmarks/`

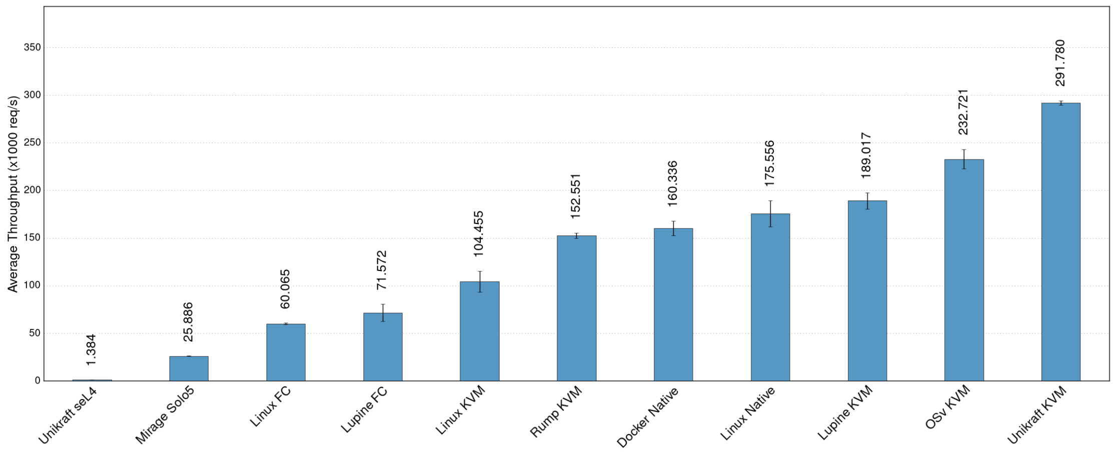


Figure 8.7: *nginx* throughput tested with *wrk* (1 minute, 14 threads, 30 conns, static 612B HTML page).

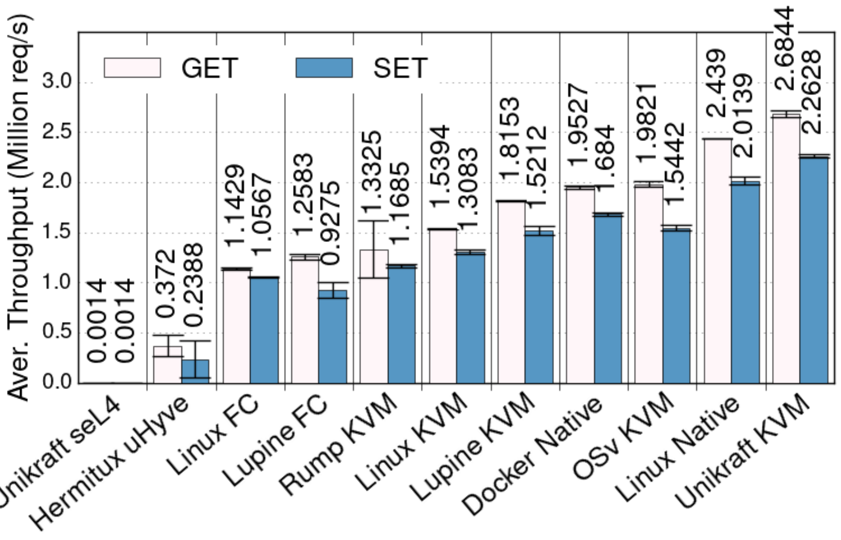


Figure 8.8: *redis* throughput tested with the *redis-benchmark* (30 connections, 100k requests, pipelining level of 16).

SQLite

The last Unikraft benchmarking suite[9] measures the time it takes for 60k SQLite insertions comparing different libc implementations. The results of the TOE are plotted together with the official Unikraft benchmarking results in Figure 8.9. The TOE takes around 6.5 seconds to perform all 60k SQLite insertions, thus taking substantially longer than reference implementations. Similarly to the official Unikraft results, *musl* performs slightly better than *newlib*.

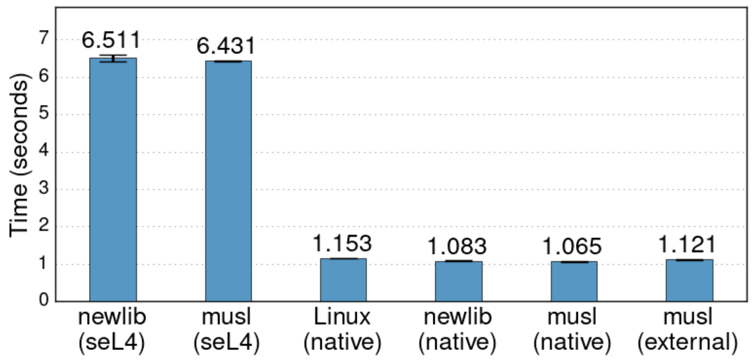


Figure 8.9: Time for 60k *sqlite* insertions.

TOE Limitations

These results show that the current prototype system does not compare well to other platforms when it comes to actual application performance. There exist several reasons for the poor performance of the prototype system that must be addressed in the future: The virtual network driver in the *NetDriver* component is not optimized for performance and therefore network-related benchmarks perform poorly. The *UnikraftApp* component further interacts frequently with other driver components through RPC. Any such invocation incurs some additional cost that might add up over time for throughput-heavy workloads. It is a natural effect of running all device drivers in separate user space components to provide a high level of isolation. Lastly, the TOE is operated at a lower CPU speed compared to the Unikraft measurements.

[9] `https://github.com/unikraft/eurosys21-artifacts/tree/master/experiments/fig_17_unikraft-sqlite-libc`

8.2.5 Summary

The experiments showed good results for the prototype system in terms of image size, boot time and memory consumption. For each metric, there is an expected overhead due to the CAmkES framework. The experiments further demonstrated that the current system implementation does not perform well when it comes to actual application performance. This must be addressed in the future.

8.3 Compatibility Evaluation

We want to show that our system provides sufficient support to run Linux/POSIX applications. We investigate the current system call support in the prototype system and the expected porting effort for Linux/POSIX applications.

8.3.1 System Calls

One of the main indicators for Linux compatibility is the amount of system calls that are supported. Although the TOE does not support Unikraft's runtime binary compatibility feature, the underlying system call support is available and is used by Unikraft's libc implementations internally as normal function calls. Unikraft provides more than 160 system calls and supports several complex applications and languages [104]. The TOE uses Unikraft as a lightweight compatibility layer for Linux/POSIX applications by providing a corresponding PAL implementation for the seL4 microkernel. This way, at least in theory, the TOE supports all system calls that are implemented by Unikraft.

To support this claim, I implemented a syscall tracing mechanism in Unikraft that collects all system calls that are triggered during an application run. The results of the syscall tracing for the four reference applications (*helloworld*, *sqlite*, *nginx* and *redis*) are shown in Figure 8.10. For each application, the triggered system calls (green) are ordered according to the official Linux system call API[10]. Linux officially supports around 360 system calls for the *x*86_64 architecture at the time of writing. Linux system call numbers range anywhere from 0 to 450, although the range from 335 to 423 is not implemented. All four applications in total require around 49 distinct system calls to run a typical workload, e.g. running the previously used benchmarks. At the top of the diagram, the total amount of implemented (and stubbed) syscalls in Unikraft is plotted.

Figure 8.11 visualizes these results in a matrix plot. Out of the 360 Linux system calls (grey boxes), Unikraft implements around 160 syscalls (green boxes), 16 of those are

[10]`https://marcin.juszkiewicz.com.pl/download/tables/syscalls.html`

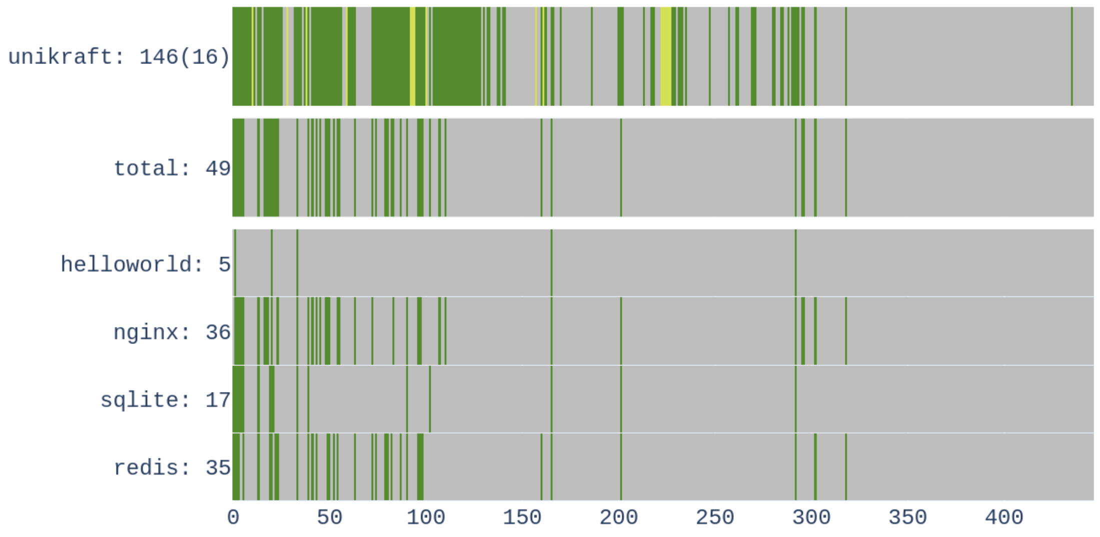


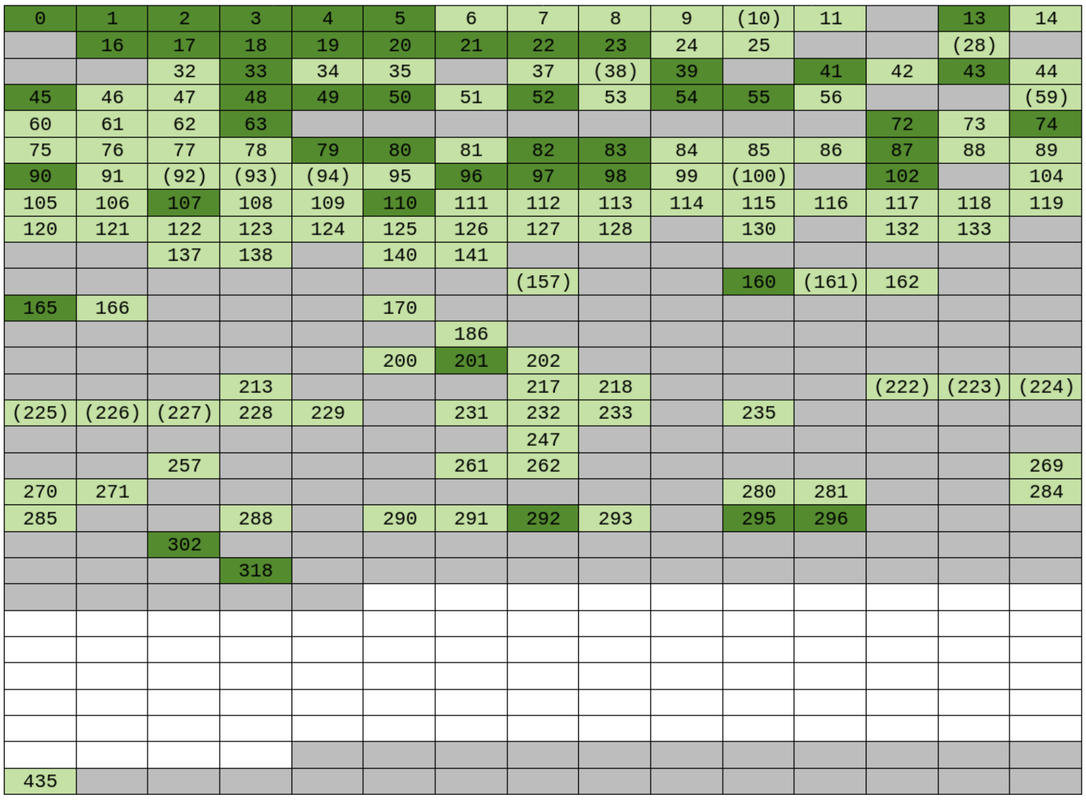
Figure 8.10: (Runtime) Syscall Tracing for Unikraft applications.

stubbed (in brackets). The majority of implemented system calls are located in the first half of the syscall matrix. Unsupported Linux system calls are shown through the white boxes. The four reference applications trigger 49 distinct system calls (dark green boxes). These results show that the TOE supports these 49 system calls, and potentially all 160 Unikraft syscall implementations. It further demonstrates that many complex Linux applications only ever require a fraction of the official Linux system call API.

8.3.2 Portability

Another indicator for compatibility is the effort that is required to port existing Linux/POSIX applications. So far, we only looked at Unikraft native ports of corresponding applications, i.e. application code is manually integrated into the Unikraft build system, potentially applying necessary patches. Unikraft-internal studies showed that the general porting effort reduces over time due to increasing (library) support in Unikraft itself [29]. Unikraft provides two more mechanisms to keep porting efforts low: Autoporting and (runtime) binary compatibility [104].

Autoporting Unikraft provides compatibility at the libc level, i.e. applications can be compiled with their native build system against the *musl* C standard library and link the resulting object files against Unikraft. For this purpose, Unikraft provides their own port of *musl* libc that resolves all undefined symbols of the application



0	1	2	3	4	5	6	7	8	9	(10)	11		13	14
	16	17	18	19	20	21	22	23	24	25			(28)	
		32	33	34	35		37	(38)	39		41	42	43	44
45	46	47	48	49	50	51	52	53	54	55	56			(59)
60	61	62	63									72	73	74
75	76	77	78	79	80	81	82	83	84	85	86	87	88	89
90	91	(92)	(93)	(94)	95	96	97	98	99	(100)		102		104
105	106	107	108	109	110	111	112	113	114	115	116	117	118	119
120	121	122	123	124	125	126	127	128		130		132	133	
		137	138		140	141								
							(157)			160	(161)	162		
165	166				170									
						186								
					200	201	202							
			213				217	218				(222)	(223)	(224)
(225)	(226)	(227)	228	229		231	232	233		235				
							247							
		257				261	262							269
270	271									280	281			284
285			288		290	291	292	293		295	296			
		302												
			318											
435														

Figure 8.11: Syscall Matrix View.

object files during link time. Internally, system calls are redirected to corresponding implementations as regular function calls. This approach only works if the application sources are available. The TOE supports the Unikraft *musl* libc port and therefore can make use of the *Autoporting* feature. As a result, existing applications are expected to be easily portable to the TOE as long as the application sources only issue system calls over the *musl* libc library [104].

(Runtime) Binary Compatibility Unikraft also supports binary compatibility, i.e. unmodified Linux ELF binaries can be run on top of Unikraft. The idea is to load a native Linux binary into memory with the help of an ELF loader [108] and then trap system calls in the *syscall shim* layer of Unikraft at runtime to map the Linux syscall to the corresponding Unikraft implementation. This approach is specifically useful when application sources are not openly available. The TOE does not support Unikraft's binary compatibility feature as it would require overwriting the seL4 microkernel's own syscall API. Additionally, the corresponding Unikraft syscall handler function would need to be registered in the *MSR LSTAR* register, which requires privileged instructions that the *UnikraftApp* component does not have access to from user space. Unfortunately, Unikraft does not implement any of the presented optimization mechanisms to avoid using the *syscall* instruction [169].

8.3.3 Summary

We demonstrated that the system provides sufficient support for major Linux/POSIX applications with Unikraft as the compatibility layer. In terms of system call support, the system supports in theory all 160 Unikraft system calls, although only around 50 system calls were covered through the tested applications. Several Unikraft native ports of popular applications are supported on the TOE. With the help of Unikraft's autoporting feature, porting existing applications to the seL4 microkernel becomes easy if not trivial as long as the application sources are publicly available and system calls are only issued through the *musl* libc library. The TOE does not support Unikraft's binary compatibility feature at the time of writing.

8.4 Integration Evaluation

Another design goal was to run Unikraft as a regular microkernel process with minimal hardware requirements (lightweight) yet isolated, ideally avoiding any type of virtualization. Unikernels were introduced as a concept that bridges the gap between VMs and containers by providing a high level of isolation and performance paired with minimal resource overhead. Since our microkernel process is based on a unikernel, we shall compare the prototype system to VMs and containers in terms of resource requirements and isolation.

8.4.1 System Characteristics

Minimal Resource Requirements (Lightweight)

In order to integrate Unikraft into a CAmkES process, the frontend and the backend of the Unikraft platform layer are located in the same process. What would be solved by hypercalls in a VM, the TOE implements as normal function calls from frontend to backend. By statically linking the Unikraft code into the CAmkES component, Unikraft becomes part of the component address space and enables the unikernel to run as a user space process. Apart from the actual application performance, the current design allows for very lightweight processes in terms of image size, memory footprint and boot time. The combination of a minimalistic microkernel (seL4) and a highly modular compatibility layer (Unikraft) allows for an extremely lightweight overall system design. One key to the lightweight characteristics of the design and the prototype system is the fact that it avoids virtualization techniques for the hardware resources at any cost to integrate the unikernel as a user space process: There is no need for CPU virtualization as we are able to run all Unikraft threads on one CAmkES control thread as explained by our threading model. Several workarounds were presented that circumvent the user space restrictions for system ISA-related functionality in the PAL. For memory management, the platform layer simply provides a static heap memory region for Unikraft. Device Drivers run all outside of the *UnikraftApp* component in separate processes and are multiplexed between different clients. This not only shrinks the size of the *UnikraftApp* component image, but also allows I/O devices to be shared among different processes without the need for virtualization. The only device that currently requires virtualization support is the virtio-based network driver in the *NetDriver* component. This is a temporary fix and must be replaced with a physical device driver in the future.

Isolation

The processes in the prototype system are first of all physically protected by the MMU that causes an exception if a process tries to access memory of another process [38]. The seL4 microkernel further utilizes capabilities to enforce a high level of isolation between system resources. The CAmkES framework ensures that only the absolute necessary rights are delegated to each CAmkES component [2]. The Unikraft process communicates with other CAmkES components over clearly defined IPC connections. Any potential communication point is protected by capabilities and enforced by the seL4 microkernel. Consequently, except for these few specific inter-connection points, the *UnikraftApp* CAmkES component is completely isolated from the rest of the system. By extracting device drivers into their own separate CAmkES components, the *UnikraftApp* component is further not able to interact with any hardware device directly.

8.4.2 Prototype System vs Containers

Containers are sandboxed process instances that share the same underlying kernel interface [170]. This makes them particularly lightweight while at the same time less secure when compared to virtual machines. The Linux kernel utilizes two kernel features to implement container sandboxes: *Control Groups (cgroups)* limit and distribute system resources among groups of processes through a hierarchical structure, while *Namespaces* provide an isolated view for processes on the system [171]. However, containers can still access some resources from the host, most importantly the kernel through the Linux syscall API. Containers apply several Linux security features to restrict system access for containers:

- **Linux Capabilities** are used to provide fine-grained access to kernel resources that was previously unavailable to unprivileged resources [172]. This way, a process running with root privileges can be limited to get only the minimal permissions it needs to perform its operation [173]. Compared to Linux capabilities that are more like access control lists with system call granularity, the seL4 microkernel applies object capabilities that are ultimately more powerful [2].
- The exposed Linux syscall API can be restricted through **Syscall Whitelisting Mechanisms** such as *seccomp* [174] in order to filter system calls to the kernel from a container. The default *seccomp* profile for Docker for example blocks 44 syscalls out of more than 300 available [173]. The number of blocked syscalls is a trade-off between isolation and application compatibility. Compared to containers, the seL4 kernel API only consists of a handful of system calls and each syscall invocation is protected by a capability. Therefore, the seL4 kernel API is more minimal and arguably more secure than what is available to Linux containers.
- **Linux Security Modules (LSM)** such as AppArmor [175] or Security-Enhanced Linux (SELinux) [176] further define policies to implement Mandatory Access Control (MAC). The SELinux technology is more complex than AppArmor and assigns labels to the system's files, processes and ports [177]. Ultimately, SELinux is an add-on to a fundamentally insecure operating system and suffers from the same problem as standard Linux: a large Trusted Computing Base, and correspondingly large attack surface. In contrast, seL4 provides strong isolation from the ground up [2].

Based on these explanations, we can assume that the seL4-based prototype system enforces a higher level of isolation for user space processes than what is the case for containers under Linux. This holds true even when Linux security features are enabled. Apart from the actual application performance, the TOE further exhibits similar if not better performance-related properties than containers, specifically in terms of image size, memory consumption and boot time. Therefore, the actual resource utilization is also more lightweight for the TOE than is the case for containers.

8.4.3 Prototype System vs Virtual Machines

Since the TOE exhibits a similar if not better resource utilization level than containers, the prototype system most definitely is more lightweight than a virtual machine. It is more difficult to compare the isolation levels of virtual machines to the ones provided by the *UnikraftApp* CAmkES component, i.e. an seL4 user space process. Relying on the formal correctness proofs of the seL4 microkernel and the corresponding kernel API [2, 134], we can assume that breaking out of a user space process into the seL4 kernel is impossible in a correctly configured seL4 system. Otherwise, the seL4 proof assumptions would be flawed. For virtual machines, there have been numerous VM breakouts for all major hypervisors (Xen [178], KVM [179], QEMU [180]) in the recent past. These vulnerabilities most definitely have been fixed by now, but there is no guarantee that a similar attack might not be discovered in the future. When the hypervisor gets compromised, then all guests are at risk. For the seL4 microkernel on the other hand, there exist no known kernel escape exploits at the time of writing. This leaves an attacker only with utilizing the basic CAmkES IPC connection points between components. Consequently, it can be assumed that CAmkES processes exhibit similar if not better isolation properties than virtual machines.

8.4.4 Summary

The TOE manages to run Linux applications as normal user space processes on top of the seL4 microkernel. It mainly avoids using any form of virtualization, while simultaneouly providing a high level of security and isolation based on underlying seL4 mechanisms. The combination of seL4 and Unikraft make for a lightweight overall system design. The TOE compares positively to containers and VMs in terms of resource utilization and provided isolation level.

8.5 Security Evaluation

8.5.1 System Guarantees

In terms of security, we want to show that our prototype system protects successfully against the threats identified by the threat model in chapter 4. Our assumptions are based on guarantees that are provided by the seL4 microkernel and the CAmkES framework.

seL4

Minimal Trusted Computing Base (TCB) The seL4 microkernel has an extremely small TCB due to its code size. Any functionality that is not needed in the kernel is pushed to user space. This shrinks the overall kernel to a point where formal verification becomes feasible [2].

Capability-based Security The seL4 microkernel defines separate data structures for kernel-related resources, also called kernel objects. Each kernel object is protected by a capability which must be enforced by the seL4 microkernel itself. Access to any of these kernel objects and related services is only possible by providing a corresponding capability. This enforces a high level of isolation realized fully in software [134].

Minimal kernel API User space can only interact with the kernel through interfaces that are related to these kernel objects. It implements its own system call API for this purpose. This API is also used to interact with other threads through the kernel's IPC mechanisms [134].

Formal verification The security and correctness claims of the seL4 microkernel stem from formal verification. Formal proofs are written against an abstract model that represents a formal specification of the kernel's behavior. The seL4 microkernel is functionally proven correct, i.e. the C implementation is bug-free with respect to its specification. The functional correctness proofs imply the absence of the following common programming errors: buffer overflow, null pointer dereferences, pointer errors in general, memory leaks, arithmetic overflows and exceptions, and undefined behavior [181]. Consequently, the functional correctness proofs enable the seL4 microkernel to not be susceptible to stack smashing, code injection or return-oriented programming (ROP) attacks [182, 2]. Depending on the used architecture (x86, ARM, RISC-V), additional formal proofs were implemented. For example the final binary can be formally verified to secure against buggy or malicious compilers. The seL4 microkernel has additional formal proofs for the security properties confidentiality, integrity and availability (CIA) [2]. Since the prototype system was implemented on the *x86_64* architecture, we can only rely on the functional correctness proofs and resulting security guarantees [183].

CAmkES

Minimal Access Rights Our system is built according to the CAmkES framework which sets up all kernel objects and corresponding capabilities precisely as stated in the CAmkES specification. This guarantees that only interactions as stated by the specification are possible [2].

Strong Isolation of user space processes Each user space process in the system is known at design time and represented as a CAmkES component. Each CAmkES component (unless colocated [184]) gets compiled into a separate binary and runs in its own address space. The only interaction points with other components have to be enforced by capabilities and are specified statically in the CAmkES description [135].

8.5.2 Threat 1: Privilege Escalation from user to kernel space

Similar to the threat model, we assume a vulnerability in one of the OS services in our system. Compared to a monolithic kernel, all OS services are pushed to user space. As a result, it is impossible for an attacker to gain access to the kernel by compromising any of these services since they effectively all run in user space. The kernel itself is formally verified to be functionally correct and bug-free in regards to its specification. Typical operating system attacks such as stack smashing, code injections and control flow highjacking are not possible for a properly configured seL4 microkernel [2]. The only interaction points between user space and kernel are limited to the few seL4 system calls available to user space. This system call API is verified itself as it's part of the kernel [185]. Relying on the formal correctness proofs of seL4, we can assume that there is no way for user space to corrupt kernel space. As a result, any privilege escalation from user to kernel space is prevented by our system.

8.5.3 Threat 2: Privilege Escalation from a low-privileged to a high-privileged process

In a monolithic kernel we distinguish between ordinary users and the superuser account. Processes are owned by specific users and inherit the corresponding privileges. Access control is very coarse-grained and having root privileges circumvents most kernel checks. In threat vector 2, we described a situation where a vulnerable user space process that runs under root privileges gets compromised allowing the attacker to gain full control over the system. The attacker was then able to extend the attack to kernel space by loading an attacker-controlled kernel module into the underlying kernel. In our system, access control is implemented by capabilities and thus much more fine-grained than with monolithic kernels. This means we have to distance ourselves from the binary notion of access control that is usually applied in monolithic kernels. Instead, user space processes have clearly defined access rights enforced by capabilities. User space processes can still be compromised but the impact is much more restricted. Exploiting any user space process only gives the attacker access rights as specified by the available capabilities in the compromised process [134].

8.5.4 Summary

We demonstrated that our system prevents successfully against threat vector 1 and drastically restricts the impact of threat vector 2. Our argumentation relies on the guarantees provided by the underlying seL4 microkernel and the CAmkES framework.

9 Use Cases

Modern computing systems are increasingly complex and consist of various parts that run at different levels of criticality. These applications have different requirements for safety, reliability and security and must be tested and certified depending on the criticality level [38]. Safety-critical code are applications that have a high level of safety/security requirements, while non-safety-critical applications, e.g. general-purpose workloads, have less stringent requirements. Mixed Criticality Systems (MCS) ensure that safety- and non-safety-critical workloads can coexist on the same system, without compromising safety or security [186].

Originally, parts with high criticality were physically separated from software components of lower criticality on separate physical processors. Due to the increasing complexity of modern computing systems, this approach does not scale well for an increasing number of software functions. As a response, multiple software components were consolidated on the same physical processor. Removing the physical barrier between the software components requires new computing paradigms to ensure a proper isolation level between software functions on the same physical hardware [81]. These solutions apply a combination of hardware and software techniques that provide the necessary isolation and separation between software components to ensure that workloads are prioritized according to their respective criticality level [186, 187]. On the hardware side, multicore CPUs, secure enclaves and virtualization technologies can be used to create isolated execution environments for different workloads. Software techniques include real-time operating systems (RTOS), Trusted Execution Environment (TEE) and hypervisors to manage the execution of mixed criticality workloads. Software-based solutions typically rely on hardware extensions themselves for efficient execution [186].

Mixed Criticality scenarios become increasingly important in an inter-connected world. Instead of processing computing workloads at one centralized location, e.g. a data center, distributed architectures increasingly move computing power to the network edge. This is done mainly for latency reasons, since in a highly dynamic environment the user requires fast responses to their requests. The overall system resembles a hierarchy, where heavy workloads are processed centrally in big data centers, while less computing intensive workloads are processed right at the network edge [188]. These edge devices often represent small form factor hardware that might not come with the necessary hardware features to enforce the isolation level needed for mixed criticality workloads [33].

Microkernels have been shown to be suitable candidates to run mixed criticality workloads [2]. They can enforce spatial and temporal isolation of kernel resources through capabilities completely in software without the need for expensive hardware features. In mixed criticality systems, general-purpose workloads are executed as code components with low criticality. Microkernels typically run as hypervisors in order to support these general-purpose workloads. This ultimately removes the benefits of the microkernel approach in the first place.

The presented design and prototype implementation add the missing piece to allow complex mixed criticality systems on small form factor hardware. It utilizes unikernels as a lightweight compatibility layer to run real-world workloads such as Linux/POSIX applications on top of a microkernel. It combines the lightweight footprint of unikernels with the isolation properties of capability-based access control to run Linux/POSIX applications as user space processes. The proof of concept implementation runs on the seL4 microkernel, and therefore is expected to be utilized in mixed criticality scenarios [2]. We focus specifically on mixed criticality use cases that utilize static workloads.

9.1 Use Case: In-Vehicle Infotainment (IVI)

In-Vehicle Infotainment systems are one such use case that runs workloads with different criticalities. Drivers are now exposed to massive infotainment screens that not only display entertainment and comfort settings, but also safety information generated by Advanced Driver Assistance Systems (ADAS). They must thus encompass information for both safety- and non-safety-critical interactions [189].

The capability-based access control of our microkernel design can enforce the necessary isolation level between software components of different criticality without the need for hardware virtualization support. The unikernel process can be used to run more general-purpose workloads, typically associated with media applications in entertainment systems. The microkernel is then responsible for handling and scheduling the mixed criticality workloads.

9.2 Use Case: Critical Infrastructure

Tedeschi et al. [190] list several scenarios where the fog and edge computing concepts described above are utilized for critical infrastructure scenarios. Examples include smart airports, smart ports and smart offshore oil and gas extraction sites. An increasing amount of more computing-heavy tasks such as AI workloads are moved to the network edge. The corresponding edge devices typically lack proper security hardenings and represent an easy target for an attacker. This makes it difficult to protect the intellectual property (IP) of sensitive workloads running on them [191]. Common mitigation techniques include encryption at rest, which does not properly solve the underlying problem, or hardware features such as secure enclaves or virtualization extensions that might not be available for edge devices [191]. At the same time, many software projects make use of 3rd party code that can easily introduce new vulnerabilities to the code base [192].

In these scenarios, the presented microkernel design can provide the necessary protection mechanisms fully in software through the capability-based security approach of modern microkernels. Consequently, less powerful edge devices can be better protected against untrusted 3rd party code, while actual sensitive workloads are less prone to IP loss.

10 Discussion

10.1 Static vs Dynamic Systems

During the design and prototype implementation, we made the assumption that unikernels exhibit rather static runtime behavior after initial process construction. For more dynamic workloads, it might still be useful to adapt the design to a more dynamic setup. Although utilizing microkernel primitives allows for full flexibility in system construction, building complex systems on these low-level mechanisms can be quite challenging. Consequently, microkernels come with supporting libraries and additional frameworks that help in creating complex systems by abstracting over the low-level microkernel mechanisms. These frameworks can either implement static or dynamic system behavior; we focus specifically on options for the seL4 microkernel.

CAmkES For the prototype system, we relied on the CAmkES framework that allows to build static component-based microkernel systems. It abstracts over the seL4 primitives and helps the system designer to reason about the system architecture at design time. The CAmkES framework instructs the CapDL loader, i.e. the root task, to set up the system precisely as stated in a corresponding system specification. CAmkES takes care of distributing all system resources and corresponding capabilities as expected by the seL4 kernel developers. This effectively helps us to keep porting efforts low for complex systems such as our prototype implementation.

The CAmkES approach comes with certain drawbacks: Being a static microkernel framework, all microkernel mechanisms are set up statically during system startup. It is discouraged to change any low-level seL4 mechanisms after the root task has setup the system. Another drawback of this approach is that each CAmkES component comes by default with supporting libraries including the *musl libc* library. Considering that unikernels also provide their own C runtime, unikernel integration becomes more difficult. We had to apply several workarounds for our prototype system to avoid symbol clashes during the final link stage. Although the seL4 runtime does not necessarily require the *musl libc* library to be included [142], the CAmkES framework typically sets up each component with a separate copy of the seL4 *musl libc* library by default [143].

seL4 Core Platform (sel4cp) The *seL4 Core Platform (sel4cp)* is an operating system personality for the seL4 microkernel [193]. The purpose of *sel4cp* is to enable system designers to create static software systems based on the seL4 microkernel [194]. The goal of *sel4cp* is to support systems with static architecture but fully dynamic implementation, i.e. a defined set of page directories, but fully dynamic implementation, including dynamic connections and a degree of dynamic allocation of memory regions to page directories [195]. The project is kept up-to-date but mainly supports the ARM and RISC-V architecture [193]. Since the focus of the implementation was on the *x86_64* architecture, I did not choose *sel4cp* for the prototype implementation.

***libsel4osapi* library** The *libsel4osapi* [39] library provides high-level system services for seL4 applications including system bootstrap and configuration, UNIX-like processes and threads, thread synchronization primitives, IP network interface support with socket-like API for UDP communication and read/write API for serial interfaces. It allows to construct seL4-based systems with dynamic system behavior [39]. The *libsel4osapi* library is outdated and was build on an old seL4 microkernel version [196]. Expected porting efforts to update the library to the latest seL4 microkernel version stopped me from using this library for the prototype implementation.

Genode Genode [197] is a multi-server OS framework that abstracts over several different microkernels, including the seL4 microkernel. It allows to implement dynamic system behavior to construct custom operating systems for the underlying platforms [197]. The combination of seL4 with Genode would enable seL4-based systems to scale to dynamic application domains [198]. Genode allows for much flexibility and comes with proofed system components to build highly-complex dynamic operating systems. The problem with Genode is that it supports seL4 kernel version 9.0.1 that still operates on the old *Kbuild* build system [199]. Starting from kernel version 10.0.0, seL4 switched to a CMake-based build system [200]. I expected porting efforts to be substantial to rapidly develop a seL4-based system under Genode for the x86 architecture.

Other Other operating system projects that build on top of the seL4 microkernel include Robigalia [201] and RefOS [202]. Both were not considered thoroughly for the implementation.

Conclusion The design and prototype system would need to be extended to allow for more dynamic unikernel behavior. Although there exist dynamic system frameworks for the seL4 microkernel, they all either are unmaintained or require a significant porting effort upfront before an actual system can be designed for the x86 architecture.

Implementing a system based on the low-level microkernel mechanisms provides good flexibility, but also increases porting effort. Therefore, building a static system based on the CAmkES framework seemed like the best option for the prototype system in terms of implementation effort.

10.2 Unikernels as Processes

The presented design and prototype implementation enables a unikernel to run as a normal user space process on top of a microkernel. Running unikernels as processes was so far only attempted by a few projects: *Nabla* containers [33], *Gramine* [25], the *Linuxu* platform in Unikraft [34] and the *Rumprun* unikernel port for the seL4 microkernel [11, 12].

The first three approaches are based on Linux itself and operate on the Linux syscall API to implement the PAL functionality. They utilize either a syscall whitelisting mechanism (*Nabla*) or are based on hardware enclaves (*Gramine*) to improve isolation for the unikernel process. Our microkernel-based design relies on the microkernel API which exposes only a handful of system calls to user space processes. Consequently, this API is arguably more minimal than the Linux API. It utilizes capabilities to enforce a very fine-grained access control scheme to kernel objects that abstract over hardware resources. This way, the microkernel is able to enforce a high level of isolation between software components. There is no need for separate syscall whitelisting mechanisms or hardware extensions. Consequently, microkernel processes are arguably more lightweight and isolated than Linux user space processes.

Our microkernel-based design extends the findings of the seL4 Rumprun unikernel port [12] to a generic interface between a unikernel and a microkernel. The implemented prototype system is based on Unikraft and the seL4 microkernel. Several reasons were presented at the beginning of the implementation phase on why Unikraft is the unikernel of choice for the prototype system. The Rumprun unikernel represents a strip-down unikernel that focuses on NetBSD OS components. The original port relies on basic seL4 mechanisms and supporting libraries. This allows for some flexibility that we can not achieve with the CAmkES approach. The Rumprun seL4 port was extended to the CAmkES framework by now and supports multiple Rumprun processes simultaneously [203]. Since the Rumprun unikernel is based on NetBSD components, the focus is mostly on POSIX compatibility. Unikraft not only supports POSIX libraries but also aims for Linux compatibility. It is a clean-slate unikernel and therefore exhibits better performance metrics than Rumprun [73]. While Unikraft is an actively maintained unikernel project, it is unclear whether the Rumprun unikernel is kept up-to-date [204], specifically for the *x86_64* architecture [205].

Conclusion The microkernel-based approach allows unikernels to be executed as a microkernel user space process that runs more isolated than on a Linux kernel. The prototype system was compared to the seL4 Rumprun unikernel port. Since both implementations are based on the same underlying microkernel, they essentially provide the same system guarantees to the unikernel.

10.3 Clean-Slate vs Strip-Down Unikernels

Although the design itself should be mostly independent of the actual unikernel type, our prototype system is explicitly based on a clean-slate unikernel (Unikraft). At the beginning of the implementation phase, we stated the reasons why Unikraft was the unikernel of choice for the prototype system. The main selection criteria were easy platform integration and available Linux/POSIX support. Unikraft explicitly defines a thin platform layer that eases platform integration and provides an extensive Linux/POSIX compatibility layer. In terms of compatibility, strip-down unikernels might have advantages over clean-slate versions as they originate from the actual monolithic kernel. At the same time, optimizing an existing kernel can not reach the same level of minimality and modularity that can be achieved by a clean-slate unikernel [75]. But the big deciding factor against strip-down unikernels for the prototype system were the expected porting effort: This type of unikernels often comes with extensive dependencies on the underlying host complicating platform integration [37]. This would have required to explicitly define a platform layer for the microkernel, which resembles more a technique called paravirtualization in which the guest OS is modified to interact with the underlying hypervisor. The goal of the microkernel port was to be as little intrusive as possible. The Rumprun unikernel is an exception for strip-down unikernels as it is based on NetBSD and therefore exhibits a more modular structure compared to Linux [12]. It defines an explicit platform layer to ease platform integration [77]. This unikernel was already successfully ported to the seL4 microkernel as was explained before.

Conclusion Plainly from the compatibility standpoint, strip-down POSIX-like unikernels might provide a higher level of compatibility to Linux/POSIX applications compared to clean-slate unikernels. At the same time, integrating a new platform might be a challenge as this type of unikernels poses extensive dependencies on the underlying host. Clean-slate unikernels are usually also better optimized for the unikernel environment. Unikraft was chosen as the unikernel of choice for the prototype system as it provides an extensive Linux/POSIX compatibility layer and allows for easy platform integration through a clearly defined platform layer.

10.4 Linux Binary Compatibility

The microkernel design and prototype system do not support Linux (runtime) binary compatibility as we are restricted to the microkernel API and can not register a unikernel-specific syscall handler function from user space. We presented several optimizations that unikernels can apply in order to circumvent the *syscall* instruction. This way, native Linux binaries can theoretically be executed on top of the microkernel, while the compatibility layer must replace the *syscall* instruction at runtime. As it turns out, Unikraft does not apply the presented optimizations to avoid the *syscall* instruction, therefore the prototype system does not support Linux binary compatibility at the time of writing. In hindsight, unikernels that apply these optimization techniques might have been a viable alternative to Unikraft in terms of binary compatibility. At the same time, these unikernels also have disadvantages compared to Unikraft as was presented in the unikernel selection process.

Conclusion The microkernel design does not support Linux binary compatibility based on a unikernel-specific syscall handler function. It is expected that a system based on a unikernel that applies optimizations to avoid the *syscall* instruction, might still be able to support native Linux binaries. The prototype system is based on Unikraft and does not implement such optimization techniques. Therefore, Linux binary compatibility is not supported for the current prototype system. Other unikernels might have been a viable alternative to Unikraft in regards to Linux binary compatibility.

11 Future Work

11.1 OS-Level Integration

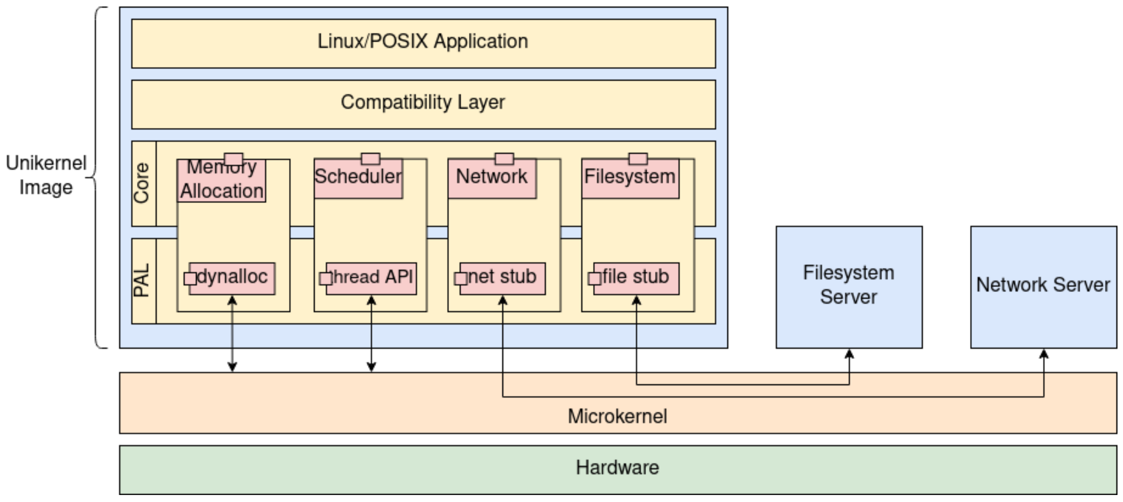


Figure 11.1: Unikernel OS-Level Integration.

During the design and prototype implementation, we assumed that the unikernel requires a handful of low-level functionality at the hardware level that must be provided by an underlying platform. Due to the modular libOS design of unikernels, the responsibility of the PAL can be extended to the OS service level. The idea is to provide microkernel-specific library implementations for common OS functionality, such as memory allocators, scheduling algorithms, network stacks, and filesystems as shown in Figure 11.1. Since OS service library implementations are hidden behind a common API, the changes are transparent to higher unikernel layers. Similar to device drivers, OS functions such as network stacks and filesystems can be run as their own user space processes. Existing unikernel implementations of these OS services can therefore be extracted into their own user space process. As a result, the overall system architecture becomes much more modular and the unikernel shrinks to a thin compatibility layer to Linux/POSIX applications. This approach is similar to a technique called *incremental cyber-retrofitting* [2], in which a monolithic system architecture is increasingly transformed into a modular multi-server operating system [40]. The scheduling API can be extended to create actual microkernel threads to enable a 1-to-1 mapping between unikernel and microkernel threads. This way, real multi-threading can be realized for the unikernel threads. The (preemptive) microkernel scheduler becomes responsible for scheduling the required threads of the unikernel process. The PAL must allow higher unikernel layers to create microkernel threads dynamically at runtime. Concerning memory allocation, a library can be implemented that makes use of the paging structures available to microkernel user space processes. This way, dynamic system behavior can be implemented for the unikernel.

The prototype system is based on Unikraft which exhibits a very modular system structure. Additional library implementations can easily be integrated into the overall build system. This allows to either extract an existing Unikraft OS service into its own CAmkES component or implement a new library in Unikraft that connects to an existing CAmkES-specific implementation. The described approach was applied for the *picoTCP*[1] network stack implementation in the seL4 microkernel. A library stub was placed in the PAL of Unikraft and connects straight to the abstract socket API that the application layer can use. The implementation is not complete yet and lacks some functionality required for applications such as NGINX. The prototype system must be completed for other OS functionality. This requires the microkernel to provide necessary OS-level abstractions. One such approach could involve the *libsel4osapi* library [196] updated to the latest seL4 microkernel version.

11.2 Dynamic System Design

During the interface design, we assumed that the general system architecture is static and does not change after the root task has set up all processes on the system. Based on these assumptions, we utilized CAmkES during the prototype implementation in order to create a static microkernel-based system. Ultimately, a static system design might prohibit more dynamic use cases. One idea to realize dynamic system behavior on microkernels, is to instruct the root task to manage the system more dynamically at runtime. Potential implementations can either be based on the basic seL4 mechanisms and supporting libraries or utilize additional frameworks such as the *libsel4osapi* library or Genode to implement dynamic system behavior. Even with CAmkES, we can realize some (limited) form of dynamic system behavior, e.g. by passing an untyped memory pool to the *UnikraftApp* component at startup that can be transformed into any other seL4 object dynamically at runtime [206].

[1] `https://github.com/tass-belgium/picotcp`

11.2.1 Unikernel Orchestration Framework

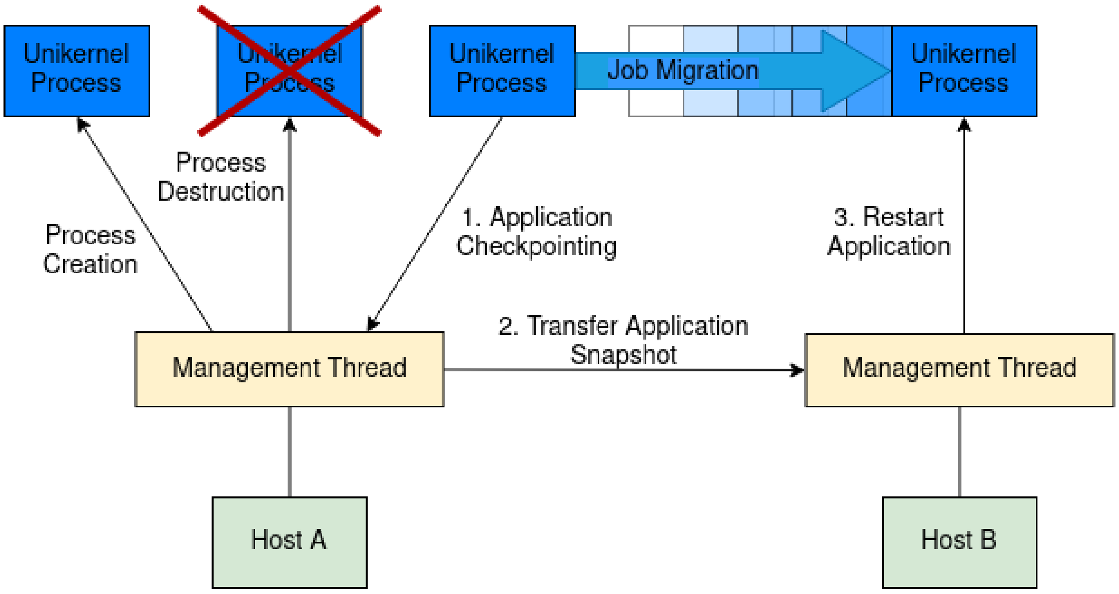


Figure 11.2: Unikernel Orchestration Framework.

The prototype implementation supports a single unikernel process. By utilizing multiplexed driver components, the prototype system can theoretically be extended to support multiple different unikernel processes. These processes share the device driver components through the described IPC mechanisms. By implementing a dynamic PAL, this concept can be extended to allow for dynamic unikernel orchestration. Such a situation is shown in Figure 11.2: A management thread, e.g. the root task, manages all unikernel processes. It allows for dynamic unikernel process creation and destruction. An application checkpointing mechanism can be used to save a snapshot of the currently running unikernel process and migrate this snapshot to another machine [207]. This resembles live migration utilized in modern container/VM orchestration tools [208].

11.2.2 Multi-Process Application Support

Unikernels are typically limited to a single-process application. Consequently, functions such as *fork* and *exec* are often not implemented. Unikernels that support multi-process applications include Lupine Linux [7], and the Gramine libOS [25]. Unikraft already provides *clone* system call support by creating a new child process in the same address space [209]. Since the prototype system with Unikraft runs as a normal user space process, it can theoretically be extended with a separate *fork* system call to create a new process. The PAL would need to be extended for dynamic memory management in order to create a new address space for the process. The Unikraft user space process can be given a block of physical memory, i.e. an untyped memory pool, and access to hardware paging structures by the root task at startup. When issuing the *clone* syscall, the PAL would need to setup a new address space through the utilization of provided paging structures. The unikernel scheduler either needs to be extended for multi-process support to change address spaces during process context switches, or we change the threading model and utilize a 1-to-1 mapping in order to let the seL4 microkernel deal with scheduling decisions.

11.3 Linux Binary Compatibility

The current design and prototype implementation do not support Linux binary compatibility. As stated by Olivier et. al. [15], a system must provide the necessary load- and run-time rules for Linux ABI compatibility. For the prototype system, we can theoretically utilize the Unikraft ELF loader to load static or dynamic Linux ELF binaries into memory [108]. We would probably need to define a separate memory region that is marked as executable in order to be able to execute the loaded Linux code. The bigger problem is most definitely the required syscall handling to realize runtime binary compatibility. Several unikernels provide optimization techniques that allow to circumvent the *syscall* instruction. These include a dynamic linker for dynamic ELFs (*library substitution*) and a syscall rewriting mechanism for static ELFs (*fast calls*). For the future, we could either create a new system implementation based on these unikernels or we port the compatibility mechanisms applied in these unikernels to Unikraft in order to support the optimizations for our existing prototype implementation. This way, it will be possible to utilize a native Linux binary without any porting effort on top of the seL4 microkernel. Apart from the presented optimizations, two additional techniques might be useful for our prototype system:

Trap & Execute The seL4 microkernel provides a limited amount of system calls and uses negative syscall numbers [210]. Any *syscall* invocation that uses a syscall number that is unknown to the seL4 microkernel results in an *Unknown Syscall* fault, which reports the faulting thread's register state to the fault handler thread of the *UnikraftApp* component [134]. This may allow the fault handler to emulate system call interfaces other than seL4 [211], e.g. the Linux syscall API. Since the seL4 kernel forwards the exception to the corresponding fault handler, multiple components can implement their own syscall handler function. A potential program flow is depicted in Figure 11.3. It resembles the *trap-and-emulate* approach many hypervisors take, except here we operate in user mode (*trap-and-execute*) [110]. Unfortunately, the seL4 system calls are stored in the *rdx* register, while Linux system calls use the *rax* register [210]. This means that in case the Linux system calls use *rdx* with a corresponding seL4 syscall value, then no exception is triggered and Linux syscall handling would not be possible.

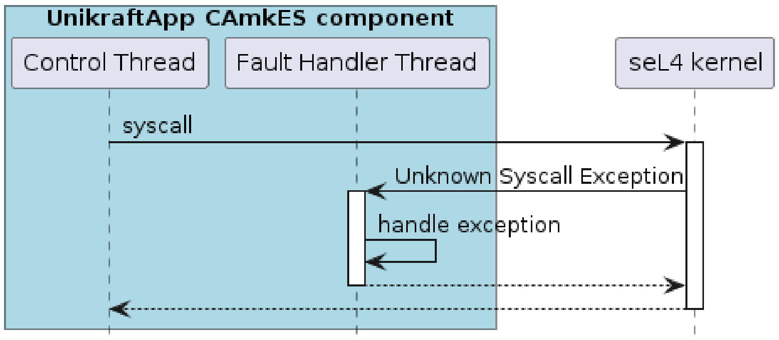


Figure 11.3: Trap-and-Execute mechanism.

Zpoline Another potential workaround is to use some form of binary rewriting to replace the *syscall* instructions. *Zpoline* is a system call hook mechanism that offers 100% coverage and is more performant than existing alternatives. It resolves the long-standing problem of replacing a small-sized instruction with a bigger instruction. The *syscall* instruction is only two bytes long and thus can not easily be replaced by a *jmp* or *call* instruction to jump to a user-defined system call hook [212]. *Zpoline* as is shown in Figure 11.4 resolves this problem by introducing a novel binary rewriting strategy and a special trampoline code. It effectively loads the binary into memory and then overwrites every occurence of the *syscall* instruction with *callq *%rax*, which effectively jumps to the address stored in the *rax* register that holds the system call number. Further, a trampoline code needs to be prepared that starts at virtual address 0 in order to redirect the execution to a user-defined system call hook. *Zpoline* applies several mitigations to deal with NULL pointer exceptions [213]. The address range between 0 and the maximum system call number is filled with single-byte *nop* instructions that effectively fall through to the following jump instruction that redirects program flow to the system call hook. After the syscall hook handled the system call, it returns to the original caller by removing the caller's address from the stack [212].

In order to use this concept for our prototype system, it is necessary to map a page at virtual address 0 into the page directory of the CAmkES component. The seL4 microkernel allows user space to have access to paging structures, and therefore

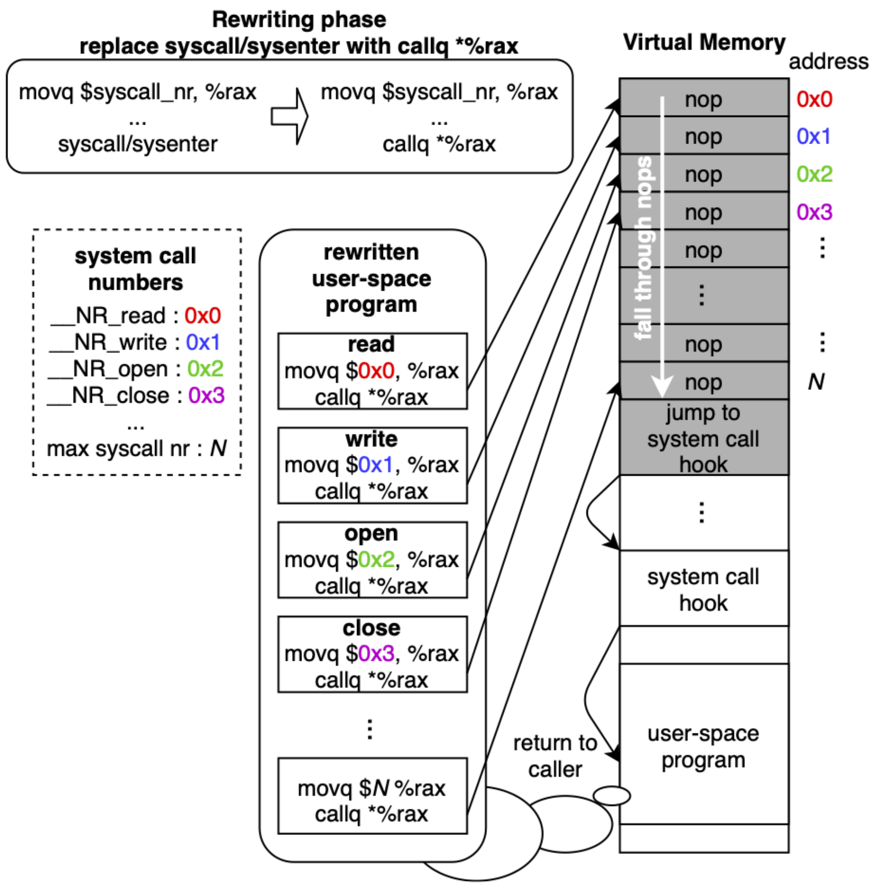


Figure 11.4: *Zpoline* taken from the official documentation [212].

manage the process address space. The only requirement is that the microkernel itself is mapped into the high part of the address space [123]. We could use the ELF-loading tool of Unikraft to load the Linux ELF into memory and then follow the approach outlined by *Zpoline*. As long as we can use virtual addresses starting from 0, this concept should be possible for the prototype system.

11.4 ARM-/RISC-V Support

The current implementation is focused on the *x86_64* architecture. It should be possible to translate the design ideas and presented workarounds to other architectures as well. Architecture-specific code is mostly the responsibility of the microkernel and therefore transparent to the PAL. By building the system for other architectures such as ARM or

RISC-V, the seL4/CAmkES build system should provide most architecture abstractions. This is especially true for user-level device drivers that run separate from the unikernel process.

11.5 Multi-Core Support

Multi-Core Support allows to run software on multiple cores. A popular approach is Symmetric Multiprocessing (SMP) that consists of one kernel/OS running on multiple cores that share the same physical memory and I/O bus. SMP is also known as tightly coupled multiprocessing and does not usually exceed 16 processors [214, 215]. The seL4 microkernel in particular implements SMP support for x86 and ARM platforms by using a big kernel lock approach that works well for cores that share the same L2 cache. It does not scale to many cores and does not offer verification support yet [5]. Each seL4 thread effectively has the CPU affinity as an attribute that specifies on which physical CPU the thread is scheduled [135]. In order to utilize multi-core support for the prototype system, we need to implement a 1-to-1 mapping between unikernel and microkernel threads. This way, the seL4 microkernel is responsible for scheduling unikernel threads on a corresponding CPU core.

11.6 Mixed Criticality Support (MCS)

Mixed Criticality Support similar to Multi-Core Support can be implemented transparent to the unikernel. The microkernel is then responsible for implementing MCS support to schedule microkernel threads accordingly for mixed criticality workloads. In our prototype system, the *UnikraftApp* component can be used as the low-criticality code component that runs general purpose workloads, while more safety-critical code is run at a higher criticality level. The seL4 microkernel provides MCS support to implement the kind of temporal isolation that is required for mixed criticality workloads [2, 5].

12 Conclusion

We began our observation by looking at modern systems that often run software at different criticality levels. Mixed criticality workloads require a high level of isolation for software components on the same physical processor. We saw that monolithic kernels can not provide the required isolation properties on their own. Instead a mix of hardware and software technologies are utilized that aid software separation. Modern microkernels can provide the necessary isolation enforcement strictly in software through the use of capability-based access control for space- and time-related system resources. This makes microkernels ideal candidates for mixed criticality workloads for hardware that does not provide the necessary virtualization solutions. As was stated at the beginning, microkernels are optimized for minimality and security, which makes it difficult to support complex applications known from other (monolithic) operating systems. This compatibility problem is typically solved by utilizing the microkernel as a hypervisor and running general-purpose workloads together with the corresponding monolithic operating system in a virtual machine. Unfortunately, this effectively removes the reason we use microkernels in the first place since the virtual machines come at a high cost in terms of resource utilization. We explored the virtualization space and saw that while containers focus on minimality and performance, they typically are much less isolated than virtual machines. Unikernels were introduced as a concept that bridges the gap between virtual machines and containers to provide good performance and minimality with a high focus on security and isolation. The general idea behind the thesis was to use unikernels as a lightweight compatibility layer to run real-world workloads such as Linux/POSIX applications on top of microkernels. Microkernel mechanisms shall be used to enforce the necessary isolation in software without the need for extensive usage of hardware technologies.

Research Question 1: Design an interface to enable running a unikernel on top of a microkernel. Unikernels are designed in such a way that they push the platform abstractions to the hardware level. They often rely on a handful of low-level functionality that must be provided by an underlying platform. For this purpose, a platform abstraction layer (PAL) is defined that represents a thin wrapper around hardware resources. We took a bottom-up approach from first principles to derive the required hardware resources and related low-level functionality that must be implemented in the

PAL for underlying platforms. The general hardware resources include CPU, Events, Memory and I/O. During the design, we presented several resource models that can provide the identified low-level functionality on top of a microkernel. This way, we can effectively integrate the microkernel as a separate platform into the PAL of the unikernel. As a result, a unikernel can be executed as a microkernel user space process.

Research Question 2: Optimize the design in terms of performance and resource requirements. The key to minimizing resource utilization is to run the unikernel as an ordinary microkernel user space process. This requires instructing the root task to set up the system accordingly during startup. The interface relies on basic microkernel mechanisms and their corresponding capabilities to enforce the required isolation fully in software. The microkernel provides abstractions close to the hardware level that can be accessed through the kernel interface by providing an appropriate capability. This way, there is no need for the use of hardware virtualization techniques. This keeps the overall system design lightweight and performant. By extracting device drivers into their own user space processes shrinks the unikernel process and hardens the overall design in terms of isolation. This concept can theoretically be extended to higher-level OS services such as protocol stacks and filesystems. This effectively increases the responsibility of the platform layer, i.e. the microkernel, and transforms the unikernel into a very thin Linux/POSIX compatibility layer.

Research Question 3: Select a suitable combination of microkernel and unikernel for a prototype implementation. Before implementing the design ideas on an actual prototype system, we had to decide for a specific unikernel and microkernel. For both sides there exist various choices, therefore a selection process was performed for both unikernels and microkernels prior to the implementation. For the unikernel side, we decided for Unikraft as it provides an extensive compatibility layer for Linux and POSIX applications and allows for easy platform integration. The seL4 microkernel was used as the microkernel of choice, mainly because it is known to be the most advanced microkernel to date. To further ease porting efforts, the CAmkES framework allows to abstract over the basic seL4 microkernel primitives and sets up the overall system in a static way at startup. For the prototype system, we implemented the resource models identified during the design stage in the platform layer of Unikraft in order to execute the unikernel as an ordinary seL4 user space process. Along the way, we had to implement several workarounds to deal with implementation-specific obstacles such as system ISA-related functionality, CAmkES-specific complications and IPC buffer-related TLS clashes. With the applied workarounds it was possible to support most Unikraft features and several complex applications. In later chapters, we explored whether a static system approach through the likes of CAmkES is the best choice for our endeavours. CAmkES was mainly chosen to ease implementation efforts and instruct the root task to set up the system as intended by the kernel developers. One problem is that dynamic behavior after startup is much harder to implement on CAmkES. Using the CAmkES framework further inflates the resulting component binaries as several supporting libraries are linked into each component's address space. This required the prototype implementation to implement corresponding workarounds. Several alternatives to the CAmkES approach were presented that allow for a more dynamic system behavior and might be a more suitable choice for highly dynamic workloads.

Research Question 4: Evaluate possible use cases using real-world workloads. The prototype implementation supports in theory all the applications and libraries that are supported by Unikraft. During the evaluation phase, we specifically looked at popular (cloud) applications (NGINX, Redis, SQLite) in order to compare the implemented system to unikernels on other platforms. We considered performance-related metrics such as image size, boot time, memory consumption, and application performance for these applications. Unikraft provides an extensive benchmarking suite around these core metrics for various unikernels. Except for the application performance, we saw that the prototype system compares positively and as expected to unikernels on other platforms for the presented metrics. The prototype system must be revised in the future to increase the application performance. Another focus during the evaluation was the original goal of supporting Linux/POSIX applications. In theory, we support all applications and libraries that are provided by the Unikraft ecosystem. To better evaluate the actual support for POSIX and especially Linux applications, we took a look at the actual triggered system calls for the evaluated applications. From a total of 160 possible system calls in Unikraft, only around 50 distinct system calls are triggered for the considered applications. This tells us that only a fraction of system calls is needed to support many complex Linux/POSIX applications. The prototype system can further utilize the autoporting feature of Unikraft to ease porting efforts. Unfortunately, the prototype system does not support Linux binary compatibility. By considering the mixed criticality support built into the seL4 microkernel, we extended our evaluation of the design to actual real-world use cases. Of specific interest for our design are mixed criticality scenarios that run static workloads. We argumented that our microkernel-based prototype system is attractive for scenarios such as In-Vehicle-Infotainment systems in the automotive industry or securing edge devices in critical infrastructure networks.

12.1 Concluding Remarks

The thesis contributes a design for a general interface between a unikernel and a microkernel. This allows a unikernel to run on top of a microkernel as a normal user space process. Consequently, the unikernel can be used to provide real-world workloads to the microkernel in the form of Linux/POSIX applications. Several ideas were presented to further optimize the design in terms of resource requirements and performance. A prototype system was constructed that implements the resource models identified during the design stage. The system was evaluated against several key performance metrics and generally compares positively to unikernels on other platforms. The design and prototype system demonstrate a lightweight approach to the usability problem of microkernels. The unikernel is responsible for the compatibility to general-purpose workloads, while the microkernel enforces a high level of isolation between the software components through the use of capabilities. This concept can be revisited in the future to build highly complex microkernel-based mixed criticality systems that run heavily isolated on the same physical machine. Therefore, this work can lay the foundation for more widespread microkernel adoption in real-world systems.

List of Figures

List of Tables

Acronyms

ADL	Architecture Description Language.
CAmkES	Component Architecture for microkernel-based Embedded Systems.
IDT	Interrupt Descriptor Table.
IPC	Inter-Process Communication.
MCS	Mixed Criticality System.
MMU	Memory Management Unit.
MSR	Model-Specific Register.
OS	Operating System.
PAL	Platform Adaptation Layer.
POLA	Principle of Least Authority.
RPC	Remote Procedure Call.
TCB	Thread Control Block.
TCB	Trusted Computing Base.
TLS	Thread-Local Storage.
TOE	Target of Evaluation.
vDSO	virtual Dynamic Shared Object.
VM	Virtual Machine.
VMM	Virtual Machine Monitor.

Bibliography

1. Wikipedia. Microkernel. Available from: `https://en.wikipedia.org/wiki/Microkernel` [Accessed on: 2023 May 11]
2. Heiser G. seL4 Whitepaper: The seL4 Microkernel - An Introduction. Available from: `https://sel4.systems/About/seL4-whitepaper.pdf` [Accessed on: 2023 May 11]
3. Wikipedia. Linux kernel. Available from: `https://en.wikipedia.org/wiki/Linux_kernel` [Accessed on: 2023 May 11]
4. Data61. Using Rump kernels to run unmodified NetBSD drivers on seL4. Available from: `https://research.csiro.au/tsblog/using-rump-kernels-to-run-unmodified-netbsd-drivers-on-sel4/` [Accessed on: 2023 Mar 28]
5. seL4. Frequently Asked Questions on seL4. Available from: `https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html` [Accessed on: 2023 Mar 31]
6. VMWare. Why use containers vs. VMs? Available from: `https://www.vmware.com/topics/glossary/content/vms-vs-containers.html` [Accessed on: 2022 Nov 2]
7. Kuo HC, Williams D, Koller R, and Mohan S. A Linux in Unikernel Clothing. *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. DOI: `10.1145/3342195.3387526`
8. Kivity A, Laor D, Costa G, Enberg P, Har'El N, Marti D, and Zolotarov V. OSv—Optimizing the Operating System for Virtual Machines. *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014 Jul :61–72
9. Madhavapeddy A and Scott DJ. Unikernels: Rise of the Virtual Library Operating System: What If All the Software Layers in a Virtual Appliance Were Compiled within the Same Safe, High-Level Language Framework? Queue 2013 Dec; 11:30–44. DOI: `10.1145/2557963.2566628`

10. Raza A, Sohal P, Cadden J, Appavoo J, Drepper U, Jones R, Krieger O, Mancuso R, and Woodman L. Unikernels: The Next Stage of Linux's Dominance. *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19. Bertinoro, Italy: Association for Computing Machinery, 2019 :7–13. DOI: 10.1145/3317550.3321445

11. Elphinstone K, Zarrabi A, Mcleod K, and Heiser G. A Performance Evaluation of Rump Kernels as a Multi-Server OS Building Block on SeL4. *Proceedings of the 8th Asia-Pacific Workshop on Systems*. APSys '17. Mumbai, India: Association for Computing Machinery, 2017. DOI: 10.1145/3124680.3124727

12. McLeod K. Usermode OS Components on seL4 with Rump Kernels. MA thesis. Sydney, Australia: School of Electrical Engineering and Telecommunication, 2016 Oct

13. Tsai CC, Jain B, Abdul NA, and Porter DE. A Study of Modern Linux API Usage and Compatibility: What to Support When You're Supporting. *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: Association for Computing Machinery, 2016. DOI: 10.1145/2901318.2901341

14. Atlidakis V, Andrus J, Geambasu R, Mitropoulos D, and Nieh J. POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing. *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: Association for Computing Machinery, 2016. DOI: 10.1145/2901318.2901350

15. Olivier P, Lefeuvre H, Chiba D, Lankes S, Min C, and Ravindran B. A Syscall-Level Binary-Compatible Unikernel. IEEE Transactions on Computers 2022; 71:2116–27. DOI: 10.1109/TC.2021.3122896

16. Olivier P, Chiba D, Lankes S, Min C, and Ravindran B. A Binary-Compatible Unikernel. *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE 2019. Providence, RI, USA: Association for Computing Machinery, 2019 :59–73. DOI: 10.1145/3313808.3313817

17. Madhavapeddy A, Mortier R, Rotsos C, Scott D, Singh B, Gazagnaire T, Smith S, Hand S, and Crowcroft J. Unikernels: Library Operating Systems for the Cloud. *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: Association for Computing Machinery, 2013 :461–72. DOI: 10.1145/2451116.2451167

18. Bratterud A, Walla AA, Haugerud H, Engelstad PE, and Begnum K. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 2015 :250–7. DOI: `10.1109/CloudCom.2015.89`

19. Ballesteros FJ. Clive. Available from: `https://lsub.org/clive/` [Accessed on: 2022 Nov 2]

20. Cheon J, Kim Y, Hur T, Byun S, and Woo G. An Analysis of Haskell Parallel Programming Model in the HaLVM. Journal of Physics: Conference Series 2020; 1566

21. Homepage runtime.js. runtime.js: JavaScript library operating system for the cloud. Available from: `http://runtimejs.org/` [Accessed on: 2022 Nov 3]

22. Software V. Erlang on Xen. Available from: `https://data-room-software.org/erlangonxen/` [Accessed on: 2022 Nov 3]

23. Martins J, Ahmed M, Raiciu C, Olteanu V, Honda M, Bifulco R, and Huici F. ClickOS and the Art of Network Function Virtualization. *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014 :459–73

24. Repo G. RustyHermit - A Rust-based, lightweight unikernel. Available from: `https://github.com/hermitcore/rusty-hermit` [Accessed on: 2022 Nov 3]

25. Homepage G. Gramine - a Library OS for Unmodified Applications. Available from: `https://gramineproject.io/` [Accessed on: 2022 Nov 3]

26. Tsai CC, Porter DE, and Vij M. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '17. Santa Clara, CA, USA: USENIX Association, 2017 :645–58

27. Lankes S, Pickartz S, and Breitbart J. HermitCore: A Unikernel for Extreme Scale Computing. *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*. ROSS '16. Kyoto, Japan: Association for Computing Machinery, 2016. DOI: `10.1145/2931088.2931093`

28. Densham C. Binary-Compatibility with Linux Application for a Unikernel Written in Rust. Available from: `https://raw.githubusercontent.com/wiki/ssrg-vt/hermitux/files/christopher-densham-project-report.pdf` [Accessed on: 2023 May 3]

29. Kuenzer S, Bădoiu VA, Lefeuvre H, Santhanam S, Jung A, Gain G, Soldani C, Lupu C, Teodorescu Ş, Răducanu C, Banu C, Mathy L, Deaconescu R, Raiciu C, and Huici F. Unikraft: Fast, Specialized Unikernels the Easy Way. *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021 :376–94. DOI: 10.1145/3447786.3456248

30. Kantee A and Cormack J. Rump Kernels: No OS? No Problem! Available from: https://www.usenix.org/publications/login/october-2014-vol-39-no-5/rump-kernels-no-os-no-problem [Accessed on: 2022 Nov 2]

31. Raza A, Unger T, Boyd M, Munson E, Sohal P, Drepper U, Jones R, Oliveira DB de, Woodman L, Mancuso R, Appavoo J, and Krieger O. Integrating Unikernel Optimizations in a General Purpose OS. 2022. arXiv: 2206.00789 [cs.OS]

32. Porter DE, Boyd-Wickizer S, Howell J, Olinsky R, and Hunt GC. Rethinking the Library OS from the Top Down. *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: Association for Computing Machinery, 2011 :291–304. DOI: 10.1145/1950365.1950399

33. Williams D, Koller R, Lucina M, and Prakash N. Unikernels as Processes. *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '18. Carlsbad, CA, USA: Association for Computing Machinery, 2018 :199–211. DOI: 10.1145/3267809.3267845

34. Wiki U. Linux Userspace. Available from: https://unikraft.org/docs/operations/plats/linuxu/ [Accessed on: 2022 Nov 10]

35. Isaac OA, Okokpujie K, Akinwumi H, Juwe J, Otunuya H, and Alagbe O. An Overview of Microkernel Based Operating Systems. IOP Conference Series: Materials Science and Engineering 2021 Apr; 1107:012052. DOI: 10.1088/1757-899X/1107/1/012052

36. Heiser G and Elphinstone K. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. ACM Trans. Comput. Syst. 2016 Apr; 34. DOI: 10.1145/2893177

37. Heiser G. seL4 Design Principles. Available from: https://microkerneldude.org/2020/03/11/sel4-design-principles/ [Accessed on: 2023 Feb 8]

38. Vemuri D and Al-Hamdani W. Measures to Improve Security in a Microkernel Operating System. *Proceedings of the 2011 Information Security Curriculum Development Conference*. InfoSecCD '11. Kennesaw, Georgia: Association for Computing Machinery, 2011 :25–33. DOI: 10.1145/2047456.2047460

39. GitHub. libsel4osapi Design. Available from: https://github.com/rticommunity/libsel4osapi/blob/master/docs/design.md [Accessed on: 2023 Mar 31]
40. OC F. Legacy Reuse. Available from: https://os.inf.tu-dresden.de/Studium/KMB/WS2018/09-Legacy-Reuse.pdf [Accessed on: 2023 Apr 6]
41. Kernkonzept. L4Re Operating System Framework. Available from: https://www.kernkonzept.com/l4re-operating-system-framework [Accessed on: 2022 Nov 2]
42. Dresden) MR(. Drivers. Available from: https://os.inf.tu-dresden.de/Studium/KMB/WS2018/06-Drivers.pdf [Accessed on: 2022 Nov 2]
43. OC F. Virtualization. Available from: https://os.inf.tu-dresden.de/Studium/KMB/WS2018/10-Virtualization.pdf [Accessed on: 2023 Apr 6]
44. Matos Ed and Ahvenjärvi M. seL4 Microkernel for Virtualization Use-Cases: Potential Directions towards a Standard VMM. Electronics 2022; 11. DOI: 10.3390/electronics11244201
45. OSDev. Scheduler Tutorial. Available from: https://wiki.osdev.org/User:Mariuszp/Scheduler_Tutorial [Accessed on: 2023 Apr 8]
46. OSDev. Scheduling Algorithms. Available from: https://wiki.osdev.org/Scheduling_Algorithms [Accessed on: 2023 Apr 8]
47. Priambodo B. Cooperative vs. Preemptive: a quest to maximize concurrency power. Available from: https://medium.com/traveloka-engineering/cooperative-vs-preemptive-a-quest-to-maximize-concurrency-power-3b10c5a920fe [Accessed on: 2023 Apr 6]
48. OSDev. Brendan's Memory Management Guide. Available from: https://wiki.osdev.org/Brendan%27s_Memory_Management_Guide [Accessed on: 2023 Apr 8]
49. OSDev. Memory Allocation. Available from: https://wiki.osdev.org/Memory_Allocation [Accessed on: 2023 Apr 8]
50. OSDev. Category:IPC. Available from: https://wiki.osdev.org/Category:IPC [Accessed on: 2023 Apr 8]
51. OSDev. Chapter 5 Interprocess Communication Mechanisms. Available from: https://tldp.org/LDP/tlk/ipc/ipc.html [Accessed on: 2023 Apr 8]
52. OSDev. Network Stack. Available from: https://wiki.osdev.org/Network_Stack [Accessed on: 2023 Apr 8]
53. Zephyr. Network Stack Architecture. Available from: https://docs.zephyrproject.org/3.2.0/connectivity/networking/net-stack-architecture.html [Accessed on: 2022 Dec 5]

54. OSDev. File Systems. Available from: https://wiki.osdev.org/File_Systems [Accessed on: 2023 Apr 8]
55. Singh R. Differences between Memory Mapped I/O and DMA mode of data transfer. Available from: https://rashandeepsingh.medium.com/differences-between-memory-mapped-i-o-and-dma-mode-of-data-transfer-471ddb5424f2 [Accessed on: 2022 Dec 8]
56. Jonathan Corbet Alessandro Rubini GKH. Communicating with Hardware. Available from: https://static.lwn.net/images/pdf/LDD3/ch09.pdf [Accessed on: 2022 Dec 8]
57. Damato J. The Definitive Guide to Linux System Calls. Available from: https://blog.packagecloud.io/the-definitive-guide-to-linux-system-calls/ [Accessed on: 2023 May 12]
58. OSDev.org. Model Specific Registers. Available from: https://wiki.osdev.org/Model_Specific_Registers [Accessed on: 2023 Feb 14]
59. Intel. Model Specific Registers and Functions. Available from: http://datasheets.chipdb.org/Intel/x86/Pentium/Embedded%20Pentium%AE%20Processor/MDELREGS.PDF [Accessed on: 2023 Feb 14]
60. Hasan MZ. X86 Architecture Basics: Privilege Levels and Registers. Available from: https://sites.google.com/site/masumzh/articles/x86-architecture-basics/x86-architecture-basics [Accessed on: 2023 Feb 14]
61. Wikipedia. System call. Available from: https://en.wikipedia.org/wiki/System_call [Accessed on: 2022 Nov 16]
62. Tic C. A Deep dive into (implicit) Thread Local Storage. Available from: https://chao-tic.github.io/blog/2018/12/25/tls [Accessed on: 2023 Apr 8]
63. Drepper U. ELF Handling For Thread-Local Storage. Available from: https://www.uclibc.org/docs/tls.pdf [Accessed on: 2023 Apr 8]
64. Kernel.org. 28.8. Using FS and GS segments in user space applications. Available from: https://www.kernel.org/doc/html/next/x86/x86_64/fsgs.html [Accessed on: 2023 Apr 8]
65. OSDev. SWAPGS. Available from: https://wiki.osdev.org/SWAPGS [Accessed on: 2023 Apr 8]
66. Duarte G. CPU Rings, Privilege, and Protection. Available from: https://zhihuicao.wordpress.com/2015/06/15/privilege-level-in-x86-architecture-instructions/ [Accessed on: 2023 Feb 14]

67. Wikipedia. Linux Kernel Interfaces. Available from: https://en.wikipedia.org/wiki/Linux_kernel_interfaces [Accessed on: 2022 Nov 7]
68. Wikipedia. Linux Standard Base. Available from: https://en.wikipedia.org/wiki/Linux_Standard_Base [Accessed on: 2022 Nov 7]
69. Entrup G, Herrmann F, Matter S, Entrup E, Jakob M, Eberhardt J, and Casselt M. Architecture of the Linux kernel. 2018. Available from: https://www.sra.uni-hannover.de/Lehre/WS17/S_AKSI/preview/document.pdf [Accessed on: 2023 May 7]
70. Bugaev S. The Three Generations of Microkernels. Available from: https://fediverse.blog/~/3542/the-three-generations-of-microkernels/ [Accessed on: 2022 Nov 2]
71. Huang J. Microkernel Evolution: Designs and Aspects. Available from: https://www.slideshare.net/jserv/microkernel-evolution [Accessed on: 2022 Nov 2]
72. seL4. Capabilities. Available from: https://docs.sel4.systems/Tutorials/capabilities.html [Accessed on: 2022 Nov 18]
73. (Website) U. Unikraft Performance. Available from: https://unikraft.org/docs/features/performance/ [Accessed on: 2023 Mar 20]
74. Cosbuc M. All About Unikernels: Part 1, What They Are, What They Do, and What's New. Available from: https://blog.container-solutions.com/all-about-unikernels-part-1-what-they-are [Accessed on: 2022 Nov 3]
75. Huici F. Unikraft vs. UKL: What's the Difference? Available from: https://unikraft.org/blog/2022-10-19-ukl-vs-unikraft/ [Accessed on: 2023 Feb 1]
76. Unikraft. Design Principles - Overview. Available from: https://unikraft.org/docs/concepts/design-principles/ [Accessed on: 2023 Feb 1]
77. GitHub. Rumprun Build Status. Available from: https://github.com/rumpkernel/rumprun [Accessed on: 2023 Apr 23]
78. Swift MM, Bershad BN, and Levy HM. Improving the Reliability of Commodity Operating Systems. SIGOPS Oper. Syst. Rev. 2003 Oct; 37:207–22. DOI: 10.1145/1165389.945466
79. Chou A, Yang J, Chelf B, Hallem S, and Engler D. An Empirical Study of Operating Systems Errors. SIGOPS Oper. Syst. Rev. 2001 Oct; 35:73–88. DOI: 10.1145/502059.502042

80. Ryzhyk L, Chubb P, Kuz I, and Heiser G. Dingo: Taming Device Drivers. *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. Nuremberg, Germany: Association for Computing Machinery, 2009 :275–88. DOI: `10.1145/1519065.1519095`
81. Corbet J. Mixed-criticality support in seL4. Available from: `https://lwn.net/Articles/745946/` [Accessed on: 2023 Mar 31]
82. Midas. Learning Linux Kernel Exploitation - Part 1. Available from: `https://lkmidas.github.io/posts/20210123-linux-kernel-pwn-part-1/` [Accessed on: 2023 Mar 23]
83. Mu D. ret2usr - 2018-core. Available from: `https://mudongliang.github.io/2022/07/11/ret2usr-%E5%BC%BA%E7%BD%91%E6%9D%AFCTF-core.html` [Accessed on: 2023 Mar 23]
84. Nikolenko V. Linux Kernel ROP - Ropping your way to # (Part 1). Available from: `https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/linux-kernel-rop-ropping-your-way-to-part-1/` [Accessed on: 2023 Mar 23]
85. Nikolenko V. Linux Kernel ROP - Ropping your way to # (Part 2). Available from: `https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/linux-kernel-rop-ropping-your-way-to-part-2/` [Accessed on: 2023 Mar 23]
86. Midas. Learning Linux Kernel Exploitation - Part 3. Available from: `https://lkmidas.github.io/posts/20210205-linux-kernel-pwn-part-3/` [Accessed on: 2023 Mar 23]
87. Kemerlis VP, Polychronakis M, and Keromytis AD. Ret2dir: Rethinking Kernel Isolation. *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, 2014 :957–72
88. Lemmens M. Stack Canaries - Gingerly Sidestepping the Cage. Available from: `https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/` [Accessed on: 2023 Mar 23]
89. Midas. Learning Linux Kernel Exploitation - Part 2. Available from: `https://lkmidas.github.io/posts/20210128-linux-kernel-pwn-part-2/` [Accessed on: 2023 Mar 23]
90. 0x434b. Learning Linux kernel exploitation - Part 1 - Laying the groundwork. Available from: `https://0x434b.dev/dabbling-with-linux-kernel-exploitation-ctf-challenges-to-learn-the-ropes/` [Accessed on: 2023 Mar 23]

91. Ornaghi D. The Return of Stack Overflows in the Linux Kernel. Available from: https://conference.hitb.org/hitbsecconf2023ams/materials/D2%20COMMSEC%20-%20The%20Return%20of%20Stack%20Overflows%20in%20the%20Linux%20Kernel%20-%20Davide%20Ornaghi.pdf [Accessed on: 2023 Mar 23]
92. Academy S. Root User Account and How to Root Phones. Available from: https://www.ssh.com/academy/iam/root-user-account [Accessed on: 2023 Feb 14]
93. StackOverflow. Super User mode - when exactly do I need it? Available from: https://stackoverflow.com/questions/22907594/super-user-mode-when-exactly-do-i-need-it [Accessed on: 2023 Mar 23]
94. Carson J. Privilege Escalation on Linux: When it's good and when it's a disaster (with examples). Available from: https://delinea.com/blog/linux-privilege-escalation [Accessed on: 2023 Mar 24]
95. Wikipedia. NGINX. Available from: https://en.wikipedia.org/wiki/Nginx [Accessed on: 2023 Feb 8]
96. packt. Controlling the Nginx service. Available from: https://subscription.packtpub.com/book/networking-and-servers/9781785280337/1/ch01lvl1sec11/controlling-the-nginx-service [Accessed on: 2023 Feb 8]
97. F5 OG of. Inside NGINX: How We Designed for Performance & Scale. Available from: https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/ [Accessed on: 2023 Feb 8]
98. Apache. Apache MPM Worker. Available from: https://httpd.apache.org/docs/2.4/mod/worker.html [Accessed on: 2023 Feb 8]
99. page L man. insmod. Available from: https://linux.die.net/man/8/insmod [Accessed on: 2023 Feb 15]
100. page L manual. kernel_lockdown. Available from: https://man7.org/linux/man-pages/man7/kernel_lockdown.7.html [Accessed on: 2023 Mar 24]
101. Loschwitz M. Lock It Up. Available from: https://www.linux-magazine.com/Issues/2020/239/Lockdown-Mode [Accessed on: 2023 Mar 24]
102. Loschwitz M. Using Lockdown Mode. Available from: https://www.linux-magazine.com/Issues/2020/239/Lockdown-Mode/(offset)/3 [Accessed on: 2023 Mar 25]
103. linux gentoo. Signed kernel module support. Available from: https://wiki.gentoo.org/wiki/Signed_kernel_module_support [Accessed on: 2023 Mar 25]
104. Unikraft. POSIX-Compatibility. Available from: https://unikraft.org/docs/features/posix-compatibility/ [Accessed on: 2022 Nov 7]

105. gitbook. vsyscalls and vDSO. Available from: https://0xax.gitbooks.io/linux-insides/content/SysCall/linux-syscall-3.html [Accessed on: 2022 Nov 16]

106. Corbet J. On vsyscalls and the vDSO. Available from: https://lwn.net/Articles/446528/ [Accessed on: 2022 Nov 16]

107. page L manual. vdso. Available from: https://man7.org/linux/man-pages/man7/vdso.7.html [Accessed on: 2022 Nov 16]

108. GitHub. Run App ELF Loader. Available from: https://github.com/unikraft/run-app-elfloader [Accessed on: 2023 May 12]

109. GitHub. HermiTux: a unikernel binary-compatible with Linux applications. Available from: https://github.com/ssrg-vt/hermitux [Accessed on: 2022 Nov 10]

110. Hasan MZ. CPU Virtualization. Available from: https://sites.google.com/site/masumzh/articles/hypervisor-based-virtualization/compute-virtualization [Accessed on: 2023 Feb 17]

111. Whitepaper V. Understanding Full Virtualization, Paravirtualization, and Hardware Assist. Available from: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf [Accessed on: 2023 Feb 17]

112. Dresden) MR(. Threads. Available from: https://os.inf.tu-dresden.de/Studium/KMB/WS2018/02-Threads.pdf [Accessed on: 2022 Nov 2]

113. Hasan MZ. X86 Architecture basics: Interrupts, Faults and Traps and IO. Available from: https://sites.google.com/site/masumzh/articles/x86-architecture-basics/interrupts-faults-and-traps [Accessed on: 2023 Feb 15]

114. Docs M. Virtual Interrupt Controller. Available from: https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/virtual-interrupts [Accessed on: 2022 Nov 24]

115. Hasan MZ. IO and Interrupt Virtualization. Available from: https://sites.google.com/site/masumzh/articles/hypervisor-based-virtualization/io-and-interrupt-virtualization?pli=1 [Accessed on: 2022 Nov 24]

116. seL4. Faults. Available from: https://docs.sel4.systems/Tutorials/fault-handlers.html [Accessed on: 2023 Feb 23]

117. Hasan MZ. X86 Architecture Basics: Memory Management. Available from: https://sites.google.com/site/masumzh/articles/x86-architecture-basics/x86-architecture-basics-memory-management [Accessed on: 2023 Feb 15]

118. Hasan MZ. Memory Virtualization. Available from: `https://sites.google.com/site/masumzh/articles/hypervisor-based-virtualization/memory-virtualization` [Accessed on: 2023 Feb 17]

119. Jones M. Linux virtualization and PCI passthrough. Available from: `https://developer.ibm.com/tutorials/l-pci-passthrough/` [Accessed on: 2022 Nov 13]

120. Jones M. Virtio: An I/O virtualization framework for Linux. Available from: `https://developer.ibm.com/articles/l-virtio/` [Accessed on: 2022 Nov 13]

121. ResearchGate. The logical mapping between OSI basic reference model and the TCP/IP stack. Available from: `https://www.researchgate.net/figure/The-logical-mapping-between-OSI-basic-reference-model-and-the-TCP-IP-stack_fig2_327483011` [Accessed on: 2022 Dec 7]

122. Priambodo B. Green Threads Explained. Available from: `https://c9x.me/articles/gthreads/intro.html` [Accessed on: 2023 Apr 6]

123. seL4. Mapping. Available from: `https://docs.sel4.systems/Tutorials/mapping.html` [Accessed on: 2023 Mar 24]

124. UNSW. M3: A virtual memory manager. Available from: `https://www.cse.unsw.edu.au/~cs9242/20/project/m3.shtml` [Accessed on: 2023 Mar 24]

125. OC F. Memory. Available from: `https://os.inf.tu-dresden.de/Studium/KMB/WS2018/04-Memory.pdf` [Accessed on: 2023 Apr 6]

126. seL4. CAmkES. Available from: `https://docs.sel4.systems/projects/camkes/` [Accessed on: 2022 Nov 28]

127. Michaels S and Dileo J. NCC Group Whitepaper - Accessing Unikernel Security. 2019 Apr. Available from: `https://research.nccgroup.com/wp-content/uploads/2020/07/ncc_group-assessing_unikernel_security.pdf` [Accessed on: 2023 May 3]

128. Talbot J, Pikula P, Sweetmore C, Rowe S, Hindy H, Tachtatzis C, Atkinson R, and Bellekens X. A Security Perspective on Unikernels. 2020 Jun :1–7. DOI: `10.1109/CyberSecurity49315.2020.9138883`

129. Unikraft. Unikraft Security. Available from: `https://unikraft.org/docs/features/security/` [Accessed on: 2022 Oct 6]

130. Eckl S. Easy component-based system design with CAmkES. Available from: `https://hensoldt-cyber.com/2021/07/02/easy-component-based-system-design-with-camkes/` [Accessed on: 2022 Nov 24]

131. Docs seL4. CAmkES Internals. Available from: `https://docs.sel4.systems/projects/camkes/internals` [Accessed on: 2023 Mar 24]
132. seL4. CapDL. Available from: `https://docs.sel4.systems/projects/capdl/` [Accessed on: 2023 Mar 20]
133. OpenGenus. CMake vs Ninja. Available from: `https://iq.opengenus.org/cmake-vs-ninja/` [Accessed on: 2023 Feb 3]
134. Systems T. seL4 Reference Manual, Version 12.1.0. Available from: `https://sel4.systems/Info/Docs/seL4-manual-latest.pdf` [Accessed on: 2023 May 11]
135. GitHub. CAmkES Documentation. Available from: `https://github.com/seL4/camkes-tool/blob/master/docs/index.md` [Accessed on: 2022 Nov 24]
136. seL4. Threads. Available from: `https://docs.sel4.systems/Tutorials/threads.html` [Accessed on: 2023 Feb 23]
137. seL4. Cross-platform thread-local storage support. Available from: `https://sel4.atlassian.net/browse/RFC-3` [Accessed on: 2023 Feb 23]
138. GitHub. CAmkES Debug Manual. Available from: `https://github.com/seL4/camkes-tool/blob/master/docs/DEBUG.md` [Accessed on: 2023 May 12]
139. discourse seL4. Interrupt in seL4. Available from: `https://lists.sel4.systems/hyperkitty/list/devel@sel4.systems/thread/7255K4AL7XEJV3JOGNGYFRPJPNE4OJOO/` [Accessed on: 2023 Mar 24]
140. GitHub. Dangerous MSR instructions. Available from: `https://github.com/seL4/seL4/blob/master/src/arch/x86/config.cmake#L285-L292` [Accessed on: 2023 Mar 24]
141. GitHub. X64 Verified Configuration. Available from: `https://github.com/seL4/seL4/blob/master/configs/X64_verified.cmake#L25` [Accessed on: 2023 Mar 24]
142. seL4. A dedicated C runtime for seL4. Available from: `https://sel4.atlassian.net/browse/RFC-2` [Accessed on: 2023 Feb 23]
143. GitHub. CAmkES comes with musl libc by default. Available from: `https://github.com/seL4/camkes-tool/blob/master/Findcamkes-tool.cmake#L18-L55` [Accessed on: 2023 May 12]
144. GitHub. seL4 musl libc default heap size. Available from: `https://github.com/seL4/seL4_libs/blob/master/libsel4muslcsys/CMakeLists.txt#L13-L17` [Accessed on: 2023 May 12]

145. GitHub. CAmkES Default Heap Size. Available from: `https://github.com/seL4/camkes-tool/blob/master/libsel4camkes/CMakeLists.txt#L13-L20` [Accessed on: 2023 May 12]

146. GitHub. CAmkES heap usage. Available from: `https://github.com/seL4/camkes-tool/blob/master/camkes/templates/component.common.c#L256-L262` [Accessed on: 2023 May 12]

147. GitHub. seL4 musl libc morecore heap. Available from: `https://github.com/seL4/seL4_libs/blob/master/libsel4muslcsys/src/sys_morecore.c#L33` [Accessed on: 2023 May 12]

148. OSDev.org. Programmable Interval Timer. Available from: `https://wiki.osdev.org/Programmable_Interval_Timer` [Accessed on: 2023 Feb 4]

149. Ahuja A. Clocks, Timers and Virtualization. Available from: `https://arush15june.github.io/posts/2020-07-12-clocks-timers-virtualization/` [Accessed on: 2022 Nov 24]

150. OSDev.org. RTC. Available from: `https://wiki.osdev.org/RTC` [Accessed on: 2023 Feb 4]

151. OSDev.org. IO Ports. Available from: `https://wiki.osdev.org/I/O_Ports` [Accessed on: 2023 Feb 5]

152. Adam A, Ilan A, and Nadeau T. Introduction to virtio-networking and vhost-net. Available from: `https://www.redhat.com/en/blog/introduction-virtio-networking-and-vhost-net` [Accessed on: 2023 May 8]

153. Martin EP. Virtio devices and drivers overview: The headjack and the phone. Available from: `https://www.redhat.com/en/blog/virtio-devices-and-drivers-overview-headjack-and-phone` [Accessed on: 2023 May 8]

154. Martin EP. Deep dive into Virtio-networking and vhost-net. Available from: `https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net` [Accessed on: 2023 May 8]

155. StackOverflow. Understanding PCI address mapping. Available from: `https://stackoverflow.com/questions/37901128/understanding-pci-address-mapping` [Accessed on: 2022 Dec 8]

156. Wiki Q. Documentation/Networking. Available from: `https://wiki.qemu.org/Documentation/Networking` [Accessed on: 2023 May 8]

157. seL4. The seL4 Run-time. Available from: `https://docs.sel4.systems/projects/sel4runtime/` [Accessed on: 2023 Feb 23]

158. GitHub. Unikraft EuroSys'21 Artifacts. Available from: `https://github.com/unikraft/eurosys21-artifacts` [Accessed on: 2023 Mar 21]

159. (GitHub) U. Unikraft EuroSys'21 Artifacts - Prerequisites. Available from: `https://github.com/unikraft/eurosys21-artifacts#3-prerequisites` [Accessed on: 2023 Mar 20]

160. Perez JL. Disable Hyperthreading From a Running Linux System. Available from: `https://www.baeldung.com/linux/disable-hyperthreading` [Accessed on: 2023 May 12]

161. StackExchange. Disabling AMD's equivalent (on a Zen-1 Epyc) of Intel's "turbo boost" at runtime? Available from: `https://askubuntu.com/questions/1294142/disabling-amds-equivalent-on-a-zen-1-epyc-of-intels-turbo-boost-at-runtime` [Accessed on: 2023 May 12]

162. F5 NS of. Introducing the LEMUR Stack and an Official NGINX Mascot. Available from: `https://www.nginx.com/blog/introducing-the-lemur-stack-and-an-official-nginx-mascot/` [Accessed on: 2023 Mar 20]

163. SQLite. About SQLite. Available from: `https://www.sqlite.org/about.html` [Accessed on: 2023 Mar 20]

164. Fingent. What are the most trending technology stacks of 2023? Available from: `https://www.fingent.com/blog/top-7-tech-stacks-that-reign-software-development/` [Accessed on: 2023 Mar 20]

165. seL4. Untyped. Available from: `https://docs.sel4.systems/Tutorials/untyped.html` [Accessed on: 2023 Mar 21]

166. GitHub. Unikraft Memory Consumption Benchmarks. Available from: `https://github.com/unikraft/eurosys21-artifacts/blob/master/experiments/fig_11_compare-min-mem/impl/unikraft.sh#L173` [Accessed on: 2023 Mar 21]

167. GitHub. Mimalloc for Unikraft. Available from: `https://github.com/unikraft/lib-mimalloc` [Accessed on: 2023 Mar 21]

168. GitHub. Unikraft Nginx TX with varying allocators. Available from: `https://github.com/unikraft/eurosys21-artifacts/tree/master/experiments/fig_15_unikraft-nginx-throughput` [Accessed on: 2023 Mar 21]

169. Unikraft. Session 7: Binary Compatibility. Available from: `https://unikraft.org/community/hackathons/usoc22/bincompat/` [Accessed on: 2022 Nov 7]

170. Osnat R. A Brief History of Containers: From the 1970s Till Now. Available from: `https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016` [Accessed on: 2022 Nov 13]

171. page L manual. Differences Between cgroups and Namespaces in Linux. Available from: https://www.baeldung.com/linux/cgroups-and-namespaces [Accessed on: 2023 May 7]

172. hackernoon. What's The Big Deal With Linux Capabilities? Available from: https://hackernoon.com/whats-the-big-deal-with-linux-capabilities [Accessed on: 2023 Feb 23]

173. RedHat. Chapter 8. Linux Capabilities and Seccomp. Available from: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/container_security_guide/linux_capabilities_and_seccomp [Accessed on: 2023 Feb 23]

174. Wiki M. Security/Sandbox/Seccomp. Available from: https://wiki.mozilla.org/Security/Sandbox/Seccomp [Accessed on: 2023 Feb 23]

175. AppArmor. AppArmor FAQ. Available from: https://gitlab.com/apparmor/apparmor/-/wikis/FAQ [Accessed on: 2023 Feb 23]

176. RedHat. What is SELinux? Available from: https://www.redhat.com/en/topics/linux/what-is-selinux [Accessed on: 2023 Feb 14]

177. Zivanov S. AppArmor vs. SELinux. Available from: https://phoenixnap.com/kb/apparmor-vs-selinux [Accessed on: 2023 Feb 14]

178. Constantin L. Xen hypervisor faces third highly critical VM escape bug in 10 months. Available from: https://www.csoonline.com/article/3193718/xen-hypervisor-faces-third-highly-critical-vm-escape-bug-in-10-months.html [Accessed on: 2023 Mar 25]

179. Wilhelm F. An EPYC escape: Case-study of a KVM breakout. Available from: https://googleprojectzero.blogspot.com/2021/06/an-epyc-escape-case-study-of-kvm.html [Accessed on: 2023 Mar 25]

180. Suse. qemu/KVM/Xen: floppy driver allows VM escape ("VENOM" vulnerability, CVE-2015-3456). Available from: https://www.suse.com/support/kb/doc/?id=000018540 [Accessed on: 2023 Mar 25]

181. seL4. The Proof. Available from: https://sel4.systems/Info/FAQ/proof.pml [Accessed on: 2023 Mar 24]

182. Kovacs E. Highly Secure Operating System seL4 Released as Open Source. Available from: https://www.securityweek.com/highly-secure-operating-system-sel4-released-open-source/ [Accessed on: 2023 Mar 24]

183. seL4. Verified Configurations. Available from: https://docs.sel4.systems/projects/sel4/verified-configurations.html [Accessed on: 2023 Mar 24]

184. GitHub. Single Address Space Components (Groups). Available from: https://github.com/seL4/camkes-tool/blob/master/docs/index.md#single-address-space-components-groups [Accessed on: 2023 Mar 24]

185. seL4. Inquiry to Verified Components. Available from: https://lists.sel4.systems/hyperkitty/list/devel@sel4.systems/thread/57YOH76G5UCAHVO6V6FGPZD7LUCKD637 [Accessed on: 2023 Mar 24]

186. Trick C. What is mixed criticality? Available from: https://www.trentonsystems.com/blog/what-is-mixed-criticality [Accessed on: 2023 Mar 31]

187. Paolino M, Chappuis K, Rigo A, Spyridakis A, Fanguede J, Lalov P, and Raho D. Building Trusted and Real Time ARM Guests Execution Environments for Mixed Criticality. 2016

188. Bigelow SJ. What is edge computing? Everything you need to know. Available from: https://www.techtarget.com/searchdatacenter/definition/edge-computing [Accessed on: 2023 Mar 31]

189. Dolan L. Navigating Mixed-Criticality Requirements of Integrated ADAS and Infotainment. Available from: https://embeddedcomputing.com/application/automotive/navigating-mixed-criticality-requirements-of-integrated-adas-and-infotainment [Accessed on: 2023 Mar 31]

190. Tedeschi P and Sciancalepore S. Edge and Fog Computing in Critical Infrastructures: Analysis, Security Threats, and Research Challenges. *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2019 :1–10. DOI: 10.1109/EuroSPW.2019.00007

191. Schaffner J. IoT Security and the Protection of IP at the Edge. Available from: https://embeddedcomputing.com/technology/security/network-security/iot-security-and-the-protection-of-ip-at-the-edge [Accessed on: 2023 Mar 31]

192. Eng C. Veracode: How Third-Party Code Impacts Software Security. Available from: https://thenewstack.io/veracode-how-third-party-code-impacts-software-security/ [Accessed on: 2023 Mar 31]

193. seL4. The seL4 Core Platform. Available from: https://sel4.atlassian.net/browse/RFC-5 [Accessed on: 2023 Mar 31]

194. GitHub. seL4 Core Platform. Available from: https://github.com/BreakawayConsulting/sel4cp [Accessed on: 2023 Mar 31]

195. Sydney U. The seL4 Core Platform. Available from: https://trustworthy.systems/projects/TS/sel4cp/ [Accessed on: 2023 Mar 31]

196. GitHub. libsel4osapi. Available from: https://github.com/rticommunity/libsel4osapi [Accessed on: 2023 Mar 31]

197. Genode. Introduction. Available from: https://genode.org/ [Accessed on: 2023 Mar 31]

198. Genode. Interactive and dynamic workloads on top of the seL4 kernel. Available from: https://genode.org/documentation/release-notes/16.08#Interactive_and_dynamic_workloads_on_top_of_the_seL4_kernel [Accessed on: 2023 Mar 31]

199. Genode. seL4 microkernel. Available from: https://genode.org/documentation/release-notes/19.05#seL4_microkernel [Accessed on: 2023 Mar 31]

200. GitHub. CHANGES. Available from: https://github.com/seL4/seL4/blob/master/CHANGES#L527-L530 [Accessed on: 2023 Mar 31]

201. Genode. Robigalia. Available from: https://robigalia.org/ [Accessed on: 2023 Mar 31]

202. GitHub. RefOS. Available from: https://github.com/seL4/refos [Accessed on: 2023 Mar 31]

203. GitHub. rumprun and camkes. Available from: https://103.230.158.80/hyperkitty/list/devel@sel4.systems/thread/EKT3NAGJKFVQIGAUW56L5E7RN5P6XTJU [Accessed on: 2023 Mar 31]

204. GitHub. Demote x86_64 - rumprun-netbsd target. Available from: https://github.com/rust-lang/rust/issues/81514 [Accessed on: 2023 May 14]

205. forum seL4 discussion. Sel4 rumprun has issues compiling. Available from: https://lists.sel4.systems/hyperkitty/list/devel@sel4.systems/thread/KTNQ2XUQ3NRJCLPVRBVU2DS6FFDMVCN6 [Accessed on: 2023 May 14]

206. GitHub. CAmkES untyped pool. Available from: https://github.com/seL4/camkes-tool/blob/master/camkes/templates/component.simple.c#L7-L11 [Accessed on: 2023 May 12]

207. LSF. Chapter 11. Checkpointing and Migration. Available from: https://tin6150.github.io/psg/3rdParty/lsf4_userGuide/11-checkpoint.html [Accessed on: 2023 Mar 31]

208. Hagoort N. Hot and Cold Migrations; Which Network is Used? Available from: https://blogs.vmware.com/vsphere/2019/12/hot-and-cold-migrations-which-network-is-used.html [Accessed on: 2023 Mar 31]

209. Foundation U. Unikraft Releases. Available from: https://unikraft.org/releases/ [Accessed on: 2023 Mar 20]

210. seL4. UnknownSyscall Exception Handler for non-seL4 syscall API. Available from: `https://sel4.discourse.group/t/unknownsyscall-exception-handler-for-non-sel4-syscall-api/672/1` [Accessed on: 2023 Mar 23]

211. Systems T. seL4 Reference Manual. Available from: `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=4a543594da389efc06ae6271acb3275bd5e1c73a` [Accessed on: 2023 May 11]

212. GitHub. Zpoline. Available from: `https://github.com/yasukata/zpoline/blob/master/Documentation/README.md` [Accessed on: 2023 Mar 23]

213. GitHub. Coping with NULL pointer exceptions. Available from: `https://github.com/yasukata/zpoline#coping-with-null-pointer-exceptions` [Accessed on: 2023 May 14]

214. NXP. Three Reasons Why Embedded Heterogeneous Systems Are More Efficient. Available from: `https://www.nxp.com/company/blog/three-reasons-why-embedded-heterogeneous-systems-are-more-efficient:BL-3-REASONS-EMBEDDED-SYSTEMS-EFFICIENT` [Accessed on: 2023 Mar 31]

215. Tutorialspoint. Symmetric Multiprocessing. Available from: `https://www.tutorialspoint.com/Symmetric-Multiprocessing` [Accessed on: 2023 Mar 31]