Karlsruhe Institute of Technology

VM/EPT: A Virtualisation-based Isolation Backend for FlexOS

Master's Thesis
submitted by

cand. inform. Sebastian Rauch

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Jun.-Prof. Dr. Christian Wressnegger
Advisor:	Dr.-Ing. Marc Rittinghaus

1. September 2021 – 28. February 2022

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, February 28, 2022

Abstract

Modern operating systems fix a set of protection and isolation mechanisms at design time, making changes to those mechanisms costly in terms of engineering effort. Flexible isolation, as pursued by the FlexOS project, seeks to allow specialisation of operating systems in the dimension of security by allowing users to flexibly define protection domains at the level of individual libraries while also giving users the choice over the mechanism by which isolation between these domains is enforced.

This thesis explores the use of high-guarantee isolation mechanisms for compartmentalisation at the granularity of individual libraries. For this purpose, we perform a systematic analysis of existing isolation mechanisms, motivate for VM-based isolation, before designing, implementing, and evaluating a prototype in the FlexOS framework. We show that, while VM-based isolation comes with significant performance overhead compared to lightweight isolation mechanisms when considering raw domain crossing costs, our prototype achieves reasonable slowdown with real-world workloads (for SQLite roughly 50% compared to lightweight isolation and comparable to Linux), making it a practicable mechanism when security is the primary concern.

Acknowledgements

I would like to thank my supervisor, Dr. Marc Rittinghaus, for the fruitful discussions we had and for his insightful and detailed feedback on early versions of this thesis.

Furthermore, I would like to thank Hugo Lefeuvre for letting me participate in his research without which this thesis would not have been possible. We had numerous discussions throughout all stages of this work and I learned a lot from them. I would also like to express my sincere gratitude to him for taking the time to review my work and provide extensive feedback that led to countless improvements of this thesis.

Contents

Abstract **v**

Acknowledgements **vii**

Contents **1**

1 Introduction **5**

2 Background **7**
- 2.1 Virtualisation 7
  - 2.1.1 Virtual Machines: Concepts and Challenges 8
  - 2.1.2 Virtualisation Techniques 10
  - 2.1.3 Virtual Memory in the Context of VMs 11
  - 2.1.4 Intel VT-x and EPT 12
- 2.2 Unikernels 15
  - 2.2.1 Comparison with Containers 16
  - 2.2.2 Unikernel Use Cases and Examples 17
- 2.3 FlexOS: Specialisation Towards Security 18
  - 2.3.1 Existing Compartmentalisation Approaches 20
  - 2.3.2 FlexOS Architecture 22
  - 2.3.3 Compartmentalisation 23
  - 2.3.4 Porting of Applications 23
  - 2.3.5 The Build Process 25
  - 2.3.6 The MPK Isolation Backend 26

3 Analysis **29**
- 3.1 Threat Model 29
- 3.2 Isolation Mechanisms 30
  - 3.2.1 Kernel-enforced Address Spaces 30
  - 3.2.2 Capability-based Isolation 31
  - 3.2.3 Protection Keys 34

3.2.4 VM-based Isolation 36
3.3 Virtualisation-based Isolation 38
3.3.1 The Argument for VM-based Isolation 38
3.3.2 A Substrate for Protection Domains 40
3.3.3 Challenges of VM-based Isolation 40
3.4 Summary 42

4 Design 43
4.1 Cross-Compartment Function Calls 43
4.1.1 Preserving the Function Call Semantics 44
4.1.2 Requirements for the RPC Mechanism 44
4.1.3 Control Flow between Compartments 45
4.2 The inter-VM IPC Mechanism 49
4.2.1 Sharing Data between Compartments 52
4.3 Summary of Components 52
4.3.1 The RPC Server 52
4.3.2 RPC Threads 53
4.3.3 Private Communication Channels 53
4.3.4 VM/EPT Call Gates 54
4.4 Trusted Computing Base 55
4.5 Addressing Race Conditions on Shared Memory 56
4.5.1 Automatic Copies on Cross-Compartment Calls 56
4.5.2 Selective Write-Protection of Shared Memory 57
4.6 Summary 59

5 Implementation 61
5.1 The VM/EPT Call Gates 61
5.1.1 Private Communication Channels 62
5.1.2 The Implementation of VM/EPT Gates 64
5.2 The RPC Server 65
5.2.1 Sub-Components of the RPC Server 65
5.2.2 The Implementation of the RPC Server 66
5.3 The RPC Threads 67
5.4 Optimisations 68
5.5 Modifications to the FlexOS Toolchain 69
5.5.1 Generating Function IDs 70
5.6 Summary 72

6 Evaluation 73
6.1 Methodology and Evaluation Setup 73
6.2 Microbenchmarks 74

6.2.1 Measurements with the TSC 74
6.2.2 Gate Latency Measurements 76
6.2.3 Possibility for Optimisation of VM/EPT Gates 78
6.3 Network Throughput (iPerf) 81
6.4 File System Intensive Workloads (SQLite) 82
6.5 Discussion of Security 84
6.5.1 Private Communication between Compartments 84
6.5.2 Shared Memory Allocation 85
6.5.3 Addressing Race Conditions on Shared Memory 85
6.6 Summary 85

7 Conclusion **87**
7.1 Future Work 87

A Reproducibility **89**

Bibliography **91**

Chapter 1

Introduction

Modern operating systems build in a fixed set of protection and isolation mechanisms at design time. Changes to those mechanisms require significant engineering effort. The FlexOS [58] project explores the specialisation of operating systems in the security dimension by allowing users to define protection domains at the granularity of individual libraries as well as the mechanism by which isolation between these domains is enforced.

The prototype implementation of FlexOS comes with an intra-address-space isolation backend based on Intel's memory protection keys (MPK) [46, Vol. 3]. Unfortunately, this backend only covers a small part of the design-space, focusing on high performance at the cost of weaker security guarantees, suffering from the absence of execution protection, precluding the enforcement of inter-compartment control-flow integrity (CFI), as well as known weaknesses [29, 42, 84].

The objective of this thesis is to explore the use of high-guarantee isolation mechanisms for compartmentalisation at the granularity of a library. For this purpose, we perform a systematic analysis of existing isolation mechanisms, motivate for isolation based on virtual machines (VMs), before designing, implementing, and evaluating a prototype in the FlexOS framework.

We evaluate the runtime performance of our prototype implementation under several benchmark scenarios. We measure the latency of function calls across protection domains to determine the raw cost of crossing domain boundaries. Furthermore, we isolate core components of several applications to measure the overhead incurred by our isolation mechanism in real-world scenarios. Specifically, we isolate the file system from an SQLite application to evaluate the performance of our prototype on file system focused workloads, and we isolate the network stack from an iPerf server to measure network throughput.

Our contribution is the design, implementation, and evaluation of a novel

VM-based isolation mechanism. We place each protection domain in its own VM to achieve strong isolation enforced at the hardware level by the virtual machine monitor. To support cross-domain function calls we design and implement a remote procedure call mechanism relying on inter-VM communication. In the evaluation of our prototype implementation, we show that, while the radical approach of VM-based isolation comes at the cost of high latency for cross-domain function calls, adequate performance is still obtained in several benchmark scenarios. We also show that the latency of cross-domain function calls is less than double that of Linux system calls.

The rest of this thesis is organised as follows. In Chapter 2, we provide background information about virtualisation, operating system specialisation on the example of unikernels, and the FlexOS project in which our work integrates. We motivate the choice of VM-based isolation in Chapter 3 after reviewing multiple isolation and protection paradigms and assessing their suitability for our goal of providing strong, flexible isolation at the level of individual libraries. In Chapter 4, we describe the design of our VM-based isolation mechanism before giving details of our prototype implementation in Chapter 5. Next, in Chapter 6, we evaluate the runtime performance of our prototype implementation before concluding with Chapter 7.

Chapter 2

Background

In this thesis, we explore the use of virtual machines (VMs) to enforce isolation between different software components. Therefore, Section 2.1 first introduces the notion of virtualisation. We motivate its use in modern cloud computing environments and explain different concepts and techniques that allow for the efficient virtualisation of modern hardware. Section 2.2 then gives an introduction to the unikernel approach, its relation to virtualisation and operating system (OS) specialisation. We explain how specialisation of unikernels to individual use cases and applications can improve performance and resource utilisation compared to general-purpose operating systems. We also give an overview of previous research projects that explored the use of unikernels in various contexts. Finally, in Section 2.3 we take a look at the FlexOS project that explores the specialisation of OSes towards security. We motivate the specialisation in the security dimension and give an overview of the architecture of FlexOS. We explain key concepts employed by FlexOS to achieve the goal of flexible isolation.

2.1 Virtualisation

Historically, hardware virtualisation has been used to divide the resources of mainframe computers between multiple applications, often used by different people [82, p. 17-18]. With the advent of multi-user operating systems the need for virtualisation declined. However, in the context of cloud computing it became increasingly relevant again. Providers of cloud-computing services are faced with various challenges which are well-suited for VMs.

Customers may have vastly different needs, both in terms of required resources and the desired OS. For example, one customer might want to run a small web shop hosted on a Windows system while another would like to host a server capable of handling hundreds of requests per second on a Linux system. VMs offer the flexibility required to account for a large range of use cases as well as the opportunity to provide customers with different operating systems and resource configurations.

Naturally, providers of cloud computing services are interested in optimally utilising their physical hardware resources. Thus, it is often impractical to rent out entire physical servers, usually powerful and expensive hardware, to customers. Instead, one physical server can host multiple VMs, leading to better hardware utilisation and lower cost. However, when running VMs under the control of different, mutually distrusting customers on the same physical hardware, isolation is of paramount importance. A VM rented out to one customer must not affect other systems hosted on the same hardware, independent of the behaviour of the software running inside the VM. Therefore, the strong isolation between VMs enforced by the hypervisor is necessary to enable the consolidation of multiple VMs on the same physical hardware.

2.1.1 Virtual Machines: Concepts and Challenges

The term virtual machine refers to a virtual system environment provided by virtualisation software, usually called hypervisor or virtual machine monitor (VMM), in which an OS can execute [82, p. 11]. The VMM provides the guest OS inside the VM with virtual resources such as a CPU, memory, and optionally I/O-devices. Therefore, the VMM is conceptually placed at the boundary between the instruction set architecture (ISA) of the real machine hosting the VM and the guest OS which it provides with a virtual ISA [82, p. 11]. This virtual ISA may match that of the real machine or it may be a different one. In the former case, it is desirable to execute as many instructions as possible of the virtualised environment directly on the native hardware to lower the performance overhead incurred by virtualisation. In the case of differing ISAs, instructions of the virtual ISA must be either simulated by interpretation or translated to sequences of instructions of the native ISA via binary translation [82, p. 14]. The virtual ISA presented to the host OS consists of a user part and a system part, referred to as user ISA and system ISA respectively. The user ISA encompasses all non-privileged instructions that are directly available to application code. The system ISA on the other hand comprises all privileged instructions. These are only available in privileged mode and thus should only appear in the kernel of a guest OS.

The VMM controls the access of the guest OS to the physical hardware and is able to enforce restrictions concerning the use of those hardware resources. There are two basic types of VMMs: native and hosted ones [39]. Native VMMs, also called bare metal or type-1 hypervisors, run directly on the hardware of the real machine. This means that all device drivers need to be installed directly on the VMM, potentially reducing the selection of usable hardware devices. Some type-1 VMMs allow a special VM direct access to the hardware in order to provide device drivers. An example of this is the Dom0 domain in Xen [19] which can provide device drivers for other domains [20]. Installing a native VMM on a machine replaces existing OSes, which might not be suitable for every user. Examples for type-1 VMMs are Xen [19], Microsoft Hyper-V [6], and VMWare ESXi [17].

Hosted VMMs, also referred to as type-2 hypervisors, run on top of an OS. This comes with the advantage that the hosted VMM can make use of the device drivers already available for the host OS, essentially providing the same hardware compatibility as the latter. It also is a suitable solution for users that want to run applications on their native OS in addition to others on a different OS inside a VM. Some hosted VMM available for common OSes include VMware Workstation Player [18], Oracle VM VirtualBox [9] and QEMU [10]. Figure 2.1 illustrates the two VMM types as well as a native system for comparison.

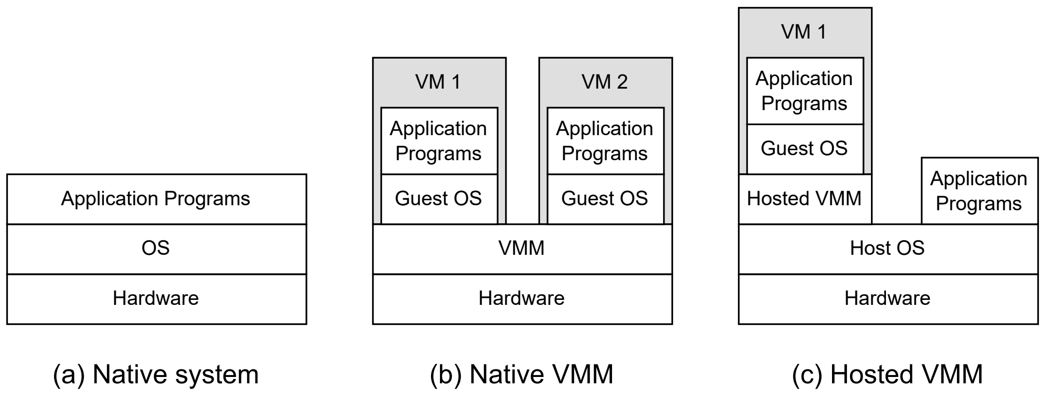


Figure 2.1: Conceptual representation of (a) a native system; (b) a type-1 VMM hosting two VMs; and (c) a type-2 VMM hosting one VM while running alongside other application programs on the host OS.

In order to be able to exercise control over the guest OS, the VMM needs to be protected from it. Since the guest OS itself expects to run in privileged mode, not all instructions can be directly executed on the real hardware, even if the ISAs match. Instead, sensitive instructions, those that configure system resources or whose behaviour depends on the configuration of system resources, must be emulated [82, p. 386]. Therefore, the state of virtual system resources of the VM is kept separate from the actual state of the real hardware and the guest OS only has direct control over this virtualised state. For example, when the guest OS tries to write to the machine status word (bits 0 through 15 of CR0) on a (virtual) x86 machine, it cannot be allowed to directly change the corresponding register in the real processor as this would allow it to take permanent control over the physical machine. The execution of this instruction must therefore be intercepted by the VMM which then can take appropriate action to emulate the effects of this instruction on the virtualised state of the VM. This paradigm of intercepting sensitive instructions in order to emulate their effect on the virtual environment in a manner that is safe for the underlying native system is known as trap and emulate [22]. It requires that all sensitive instructions trap when executed in a lower privilege level. Problems arise when an ISA contains instructions that are sensitive but not privileged, meaning that they do not trap when executed in non-privileged mode. A well-known example is the POPF instruction of the IA-32 architecture, which pops a word from the stack into the EFLAGS register. When executed in privileged mode (CPL0), it also overwrites the interrupt flag (IF). However, when the same instruction is executed in non-privileged mode (and CPL is less than I/O privilege level), the interrupt flag in the EFLAGS register is unaffected [46, Vol. 2B]. Since the execution of this instruction does not trap in the latter case, a VMM simply relying on trap and emulate would not be able to interfere in this situation. The resulting behaviour would be unexpected for the guest OS.

2.1.2 Virtualisation Techniques

The example above shows that the ISA which is to be virtualised has an impact on the complexity of the VMM. As early as 1974, Popek and Goldberg investigated which properties an ISA must have in order to be virtualisable in the classical sense, meaning only by use of trap and emulate. They showed that an ISA in which all sensitive instructions are also privileged (they trap when executed in non-privileged mode) can be virtualised purely through the trap and emulate technique [76]. However, this does not mean that architectures which violate this criterion cannot be virtualised at all. An efficient VMM can still be constructed by leveraging more advanced techniques such as paravirtualisation or binary translation, or by relying on hardware support for virtualisation.

With paravirtualisation, the guest OS is aware that it is running in a virtualised environment. The source code of the guest OS is modified to present an interface to the VMM that is easier to virtualise [90]. This has the benefit of simplifying the implementation of the VMM and also has the potential to improve performance [27]. On the other hand, it comes at the cost of modifying the guest OS, which poses a problem in the case of a proprietary OS where the source code is not available. This also implies that only OSes that are modified accordingly are supported, which limits compatibility. A popular example of a VMM that uses paravirtualisation is Xen [27].

Binary translation refers to the technique of converting the instructions from the source ISA to sequences of instructions of the target ISA [82, p. 49]. This is not only useful in the case that the source and target ISA are different, but also to detect and resolve situations where the guest OS executes sensitive instructions. Binary translation can be used to insert traps to transfer control to the VMM that can then take appropriate action to emulate the problematic instruction on the state of the VM. As opposed to paravirtualisation where the source code of the guest OS is modified, binary translation operates on the machine code level and is fully automated. This eliminates the need for any manual modification of the guest and thus all OSes that are available for the particular ISA provided by the VMM can be hosted. On the other hand, binary translation can lead to many transitions to the VMM when sensitive instructions are executed frequently and degrade performance as a consequence.

Hardware-assisted virtualisation makes use of special features offered by the hardware specifically designed to aid virtualisation [22]. For example, the processor can provide an additional privilege level that is intended for the VMM and that allows it to control a guest OS even when it is running in privileged mode. This allows for the guest OS to remain at its original privilege level without the risk of compromising the VMM. Sensitive instructions executed inside the VM must then either trap to the VMM or their behaviour is changed to modify the state of the virtual environment, which the hardware is now aware of, instead of the state of the underlying system. This can reduce the overhead of virtualisation by decreasing the number of traps to the VMM.

2.1.3 Virtual Memory in the Context of VMs

Each VM represents a full system environment and as such the guest OS also has its own virtual and physical address space, the latter consisting of the virtualised memory provided to it by the VMM. In fact, when the guest is a traditional general-purpose OS, it provides each of its processes with its own virtual address space. Addresses belonging to a virtual address space of the guest are called guest-virtual and those belonging to its physical address space are referred to as guest-physical addresses. Therefore, memory accesses from inside the VM conceptually involve two mappings, first from guest-virtual to guest-physical and then from guest-physical to host-physical addresses. These are eventually used by the hardware to access the physical memory. In order to reduce the performance overhead caused by this additional indirection, the VMM sets up shadow page tables that map guest-virtual addresses directly to host-physical ones [82, p. 399-402]. These shadow page tables are used by the hardware to perform address translations, which has the added benefit that any hardware-managed translation lookaside buffer (TLB) automatically caches the correct host-physical addresses needed for memory accesses. However, the guest OS is still allowed control over its own paging structures which reflect mappings of guest-virtual to guest-physical addresses. The page table pointer of the VM is then virtualised and points to the paging structures controlled by the guest OS. Attempts of the guest to modify the page table pointer or its own page tables trap to the VMM which then updates the shadow page tables to reflect these changes. Figure 2.2 shows the relation between the page table maintained by the guest and the shadow page table. While the use of shadow page tables reduces the overhead for address translation, this comes at the cost of incurring traps to the VMM whenever the guest modifies its paging structures.

2.1.4 Intel VT-x and EPT

Intel's hardware support features for virtualisation of the IA-32 and Intel64 architectures are commonly known as Intel VT-x [71], also referred to as virtual machine extension (VMX) [46, Vol. 3]. It provides two new forms of CPU operation, called VMX root and VMX non-root, intended for the VMM and the guest respectively. This split in the mode of operation of the processor allows the guest OS to run at its intended privilege level while still being controlled by the VMM. Transition from VMX root to VMX non-root operation is achieved by a VM entry while the reverse transition is referred to as VM exit. In VMX non-root operation many instructions and events that may be relevant to a VMM when executed by a guest cause VM exits. To reflect the processor state in the two new operation modes, a so-called virtual machine control structure (VMCS) is introduced which holds information corresponding to the host state (VMX root) and the guest state (VMX non-root). The VMCS can be accessed via special VMREAD and VMWRITE instructions. On VM entry the processor state is stored to the host state part of the VMCS while the new processor state is loaded from the guest state. Correspondingly, on VM exit the state of the processor is stored to the guest state and the new processor state for root operation is restored from the host state. In addition to the processor state, the VMCS also contains virtual page table pointer

guest-virtual	guest-physical
1000	3000
2000	7000
5000	not mapped
6000	1000
8000	2000
....	....

(a) Page table of guest

hardware page table pointer

guest-virtual	host-physical
1000	7000
2000	not mapped
5000	not mapped
6000	8000
8000	3000
....	....

(b) Shadow page table

guest-physical	host-physical
1000	8000
2000	3000
3000	7000
7000	not mapped
....	....

(c) Mapping of VMM

virtual	physical
....	....
6000	8000
8000	3000
....	....

(d) TLB

Figure 2.2: A simplified example of address translation via shadow page tables. The virtual page table pointer of the guest points to its own page table. The real page table pointer points to the shadow page table. (a) the page table maintained by the guest; (b) the shadow page table maintained by the VMM; (c) the mapping of guest-virtual to host-physical addresses maintained by the VMM; (d) the TLB, currently only the entries for pages 6000 and 8000 are cached.

execution control fields that determine which instructions or events in non-root operation cause VM exits. For example, there are control fields that allow for the virtualisation of control register CR0 by specifying which bits the guest may modify freely and for which a VM exit is triggered [71]. Access to model specific registers (MSRs) can also be controlled in a similar manner and interrupt virtualisation is supported by additional controls in the VMCS.

A source of significant overhead is the virtualisation of address translation [23]. The maintenance of shadow paging structures in purely software-based virtualisation solutions can cause frequent VM exits depending on the workload. Intel's extended page tables (EPT) mechanism provides hardware support for the translation of guest-virtual addresses, in the following also referred to as linear addresses, to host-physical addresses without the need for the VMM to keep shadow page tables. The guest OS is allowed full control over the translation of linear addresses to guest-physical ones. However, when a linear address is used to access memory, the EPT mechanism in conjunction with the guest paging structures is used to automatically translate them to host-physical addresses. For this purpose, the VMCS contains an EPT pointer (EPTP) field that points to the base of the extended page table that is then used for the address translation. One translation step involving EPT is analogous to the multi-level address translation used for hardware-walked page tables on native machines of the IA-32 and Intel64 architecture. However, the translation of one linear address to the corresponding host-physical address involves multiple translations via the extended page table. The full translation steps, when both the host and guest use four levels for their paging structures, are as follows: The CR3 control register contains the linear address of the level 4 page map (PML4) which is translated to the corresponding host-physical address using the EPT mechanism. Then an entry from the PML4 is selected by bits [47:39] of the original linear address which in turn contains the guest-physical address of the page directory pointer table (PDP). This address is again translated by the EPT mechanism to obtain its host-physical counterpart. Bits [38:30] select an entry from the PDP which is the guest-physical address of the page directory (PD) and must again be translated via EPT. This results in the guest-physical address of the PD from which an entry is selected by bits [29:21] of the original linear address. This entry is the guest-physical address of the page table and thus must again be translated with the help of the extended page table to find the corresponding host-physical address. Bits [20:12] of the original linear address select an entry in the page table which must finally be translated to the host-physical page number. Together with the offset (bits [11:0]) of the original linear address this yields the host-physical address corresponding to the given linear address. Figure 2.3 gives a graphical representation of the address translation process with EPT.

In order to avoid excessive memory accesses, a TLB is used to cache translation results just as with normal address translation without virtualisation [23]. Since the use of EPT for address translation significantly increases the number of memory accesses in case of a TLB miss, a high hit-rate is crucial to achieve good performance. This is aggravated by the fact that the TLB is flushed on every VM exit or VM entry. To alleviate the effect of transitions between VMX root and VMX non-root operation, virtual processor identifiers (VPIDs) are used to tag each TLB entry with the virtual processor it corresponds to. A TLB entry is then only considered a match when the VPID also agrees with the one assigned to the (virtual) processor. This eliminates the need to flush the TLB on a VM exit or entry. Additional microarchitectural improvements such as caching the physical addresses of all levels of the paging structures (PML4, PDP, PD and PT) on the processor, can further reduce the cost of address translation even in cases of TLB misses [23].

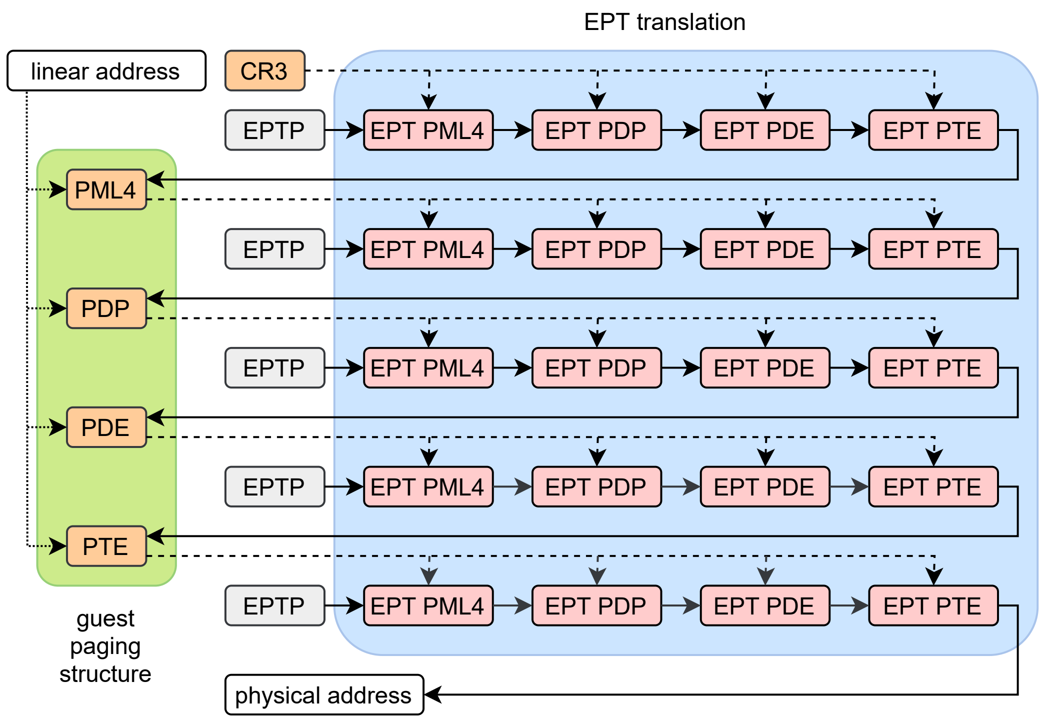


Figure 2.3: Address translation via the EPT mechanism. The paging structures of the guest are shown in the green area. The part of the translation that involves EPT is shown in the area highlighted in blue. Based on [23].

2.2 Unikernels

General-purpose OSes provide a rich system environment suitable for a wide variety of applications running concurrently on the system. Therefore, they implement abstractions and protection mechanisms to prevent different applications from interfering with each other. For example, the process abstraction provides each application program with its own virtual address space and the illusion of ownership of the central processing unit (CPU). In addition, the kernel usually provides a broad system call (syscall) application programming interface (API) with support for process and thread management, network communication, file system access, and inter-process communication (IPC) [69]. The OS is responsible for mapping the different address spaces to physical memory as well as multiplexing the CPU and other resources between the processes. When considering the more specialised use case of single purpose appliances hosted in the cloud, where each VM only executes one application, many of these abstractions and protection mechanisms are redundant. For example, there is no need to separate the address space of the kernel and the application for protection as there are no mutually distrusting applications to protect from each other. Consequently, there is no need to protect the OS from the application in a virtual environment since isolation between the VMs hosted on the same physical machine is enforced by the VMM at a lower level. The observation that general-purpose OSes as guests duplicate parts of the functionality of the VMM leads to the question of how a guest OS should be designed to provide better efficiency for such use cases.

Unikernels take a different approach compared to general-purpose OSes. Instead of providing an extensive system environment that is suitable for a wide variety of use cases, they employ specialisation to trade universality for performance. By sharing the same address space and privilege level between the kernel, libraries, and applications, the overhead associated with address space transitions incurred by syscalls of monolithic kernels is removed. As early as 1995, the Exokernel [34] project showed that allowing applications the freedom to choose which OS-level abstractions they want to use and how to implement them can lead to significant improvements in performance. Unikernels take the library OS (libOS) approach of compiling system libraries, language runtimes, and applications into a single bootable image [67]. This allows for compile-time specialisations such as including only libraries that are actually used by the application and in turn enables significant reduction in boot times and memory consumption of the resulting VM [55]. Unikernels also benefit from being hosted in a VM because this removes the need to provide device drivers for a large range of hardware devices. Those are provided by the VMM which presents the guest with simpler interfaces to the virtualised devices. The ability to provide OS functionality by libraries that can be included and configured at compile-time also offers applications the flexibility to choose the implementations that fit their requirements best [55]. For example, an application with needs for only modest network throughput could utilise a simple socket interface for ease of implementation while another application that requires much higher throughput can make use of a specialised high-performance network library. This can be achieved by including the suitable library at compile-time and programming the application against its interface.

2.2.1 Comparison with Containers

Containers provide virtualisation at the OS level. They use kernel facilities such as Linux namespaces and control groups to provide isolation between container instances. A namespace [48] is an abstraction of a system resource that gives a process or group of processes the illusion of having their own isolated instance of that resource. Changes to this resource are only visible to members of that namespace. Control groups [47] allow monitoring and limiting the usage of resources such as CPU time and memory. Containers are lightweight compared to classical full system virtualisation solutions exhibiting faster boot times and lower memory footprints [69]. This makes them attractive for providers of cloud services because it allows for higher degrees of server consolidation than traditional heavyweight VMs. On the other hand, containers have been shown to provide weaker resource isolation compared to full system virtualisation [80]. They are also plagued by vulnerabilities allowing malicious applications to break their isolation and threaten the security of the hosting systems and consequently other containers supported by it [63]. This is in part due to the large and complex syscall API provided by modern general-purpose OSes. For example, Linux has over 400 syscalls [52], many with multiple parameters and overlapping functionality [69]. Furthermore, since containers rely on the OS for isolation, the entire kernel is part of the trusted computing base (TCB) which is typically much larger compared to VMMs, thus offering a broader attack surface. Unikernels can provide similar benefits to containers while enjoying the strong isolation guarantees of a VMM making them a suitable alternative, especially in scenarios where isolation is of paramount importance.

2.2.2 Unikernel Use Cases and Examples

In the following, some of the unikernel concepts explored by various research projects are presented along with their respective goals and core findings.

Madhavapeddy et al. [67] explore the unikernel approach with their Mirage compiler that compiles and links OCaml code into bootable Xen VM images. The use of the type-safe OCaml language reduces the risk of vulnerabilities and thus increases security. VM images built by Mirage additionally have the ability to be sealed via a special hypercall to Xen which prevents the injection of code at runtime. They show that the unikernel approach significantly reduces the size of VM images compared to applications built on top of Linux while outperforming them by as much as 45% for network-focused applications.

Subsequent work optimises the Xen tool-stack to further reduce boot times and resource overhead in order to launch unikernels in response to network requests with low latency [66]. As guest they use MirageOS [7], a library OS built according to the unikernel approach. Together with the support of Xen for hardware-assisted virtualisation of ARM devices, this enables the deployment of cloud services on embedded systems near users which often provide a favourable trade-off between price, performance, and energy consumption. The resulting system, dubbed Jitsu, achieves boot latencies of around 350ms on ARM and around 30ms on x86 and thus is suitable for VMs instantiation in response to network requests.

HermiTux [74] addresses the issue of porting existing applications to unikernels. For many applications this poses a major problem because of missing kernel functionality or libraries not being available for the target platform. This hinders the adoption of unikernels in real-world scenarios despite their potential benefits. HermiTux is a unikernel based on HermitCore [56] that provides binary compatibility with the Linux application binary interface (ABI) while still retaining the benefits of high performance, low memory footprint and fast boot times. This is achieved by emulation of OS interfaces according to the Linux ABI and binary rewriting to transform system calls into ordinary function calls. The result is a binary compatible unikernel with significantly reduced syscall latency and negligible to acceptable runtime overhead compared to native Linux across several benchmark applications.

Unikraft [55] is a highly modular micro-library OS that allows to build high-performance unikernels specialised to the needs of an application. OS primitives such as schedulers, memory allocators, and network stacks are provided by micro-libraries that can be composed and configured at compile-time to fit the requirements of the application. APIs are also provided by micro-libraries which allows to easily add or remove functionality as needed. A syscall shim layer micro-library is used to transform system calls, required by some libraries such as the C standard library implementation musl, to function calls. The Unikraft build system provides a Kconfig-based menu to select and configure the libraries included in the final unikernel image. Evaluation on applications such as Redis [11] and Nginx [8] show that Unikraft achieves 30%–80% higher throughput compared to Docker containers and even 10%–60% increased throughput over native Linux with memory footprints smaller or equal to that of a corresponding Docker container.

2.3 FlexOS: Specialisation Towards Security

Contemporary OSes fix a set of protection mechanisms at design-time and integrate them into the kernel. Changes to protection mechanisms require significant engineering effort and are therefore rare. This is problematic in several ways. First, if a mechanism built in at design-time fails, e.g. the address space separation with Meltdown [64], it breaks the security of the whole system without an easy way to repair the vulnerability. Either the underlying mechanism has to be repaired or, should this not be possible, the entire system has to be redesigned around a suitable mechanism. When there are multiple protection mechanisms available that can be exchanged simply by reconfiguration and rebuilding, this process is much simpler and quicker. Second, applications have different performance and security requirements. Generally, protection mechanisms incur some performance overhead as demonstrated by the traditional privilege and address space separation [83] as well as various research projects that explore different concepts like type-safe languages [68], formal verification [54], or novel hardware mechanisms [78]. Therefore, there is a trade-off between the security achieved by a set of protection mechanisms and the resulting performance overhead.

Fixing protection mechanisms at design-time prevents applications from choosing the optimal configuration for their requirements. Also, an application may use untrusted libraries and therefore require fine-grained isolation to protect itself from those untrusted components. For example, an application designed for confidential communication may want to isolate itself and the cryptography library from an untrusted network stack. In case of a bug in the network library, the functionality of the application may be compromised, but no confidential information is disclosed because of the employed isolation. This is an example of a use case that is not easily supported by classical, coarse-grained protection mechanisms commonly used in general-purpose OSes but is well-suited for novel approaches such as intra-address space isolation via memory protection keys (MPK) [78]. Furthermore, different protection mechanisms can be employed to achieve the same security guarantees. For example, formal verification of software components can eliminate the need to isolate them via hardware mechanisms. This leads to the question of which set of protection mechanisms satisfies an application's security requirements with the least performance overhead. It is therefore desirable to leave this choice to the application instead of fixing it for a whole system at design-time. Finally, hardware is becoming increasingly heterogeneous, providing a wide variety of protection and isolation mechanisms such as Intel's Software Guard Extension (SGX) [70], MPK [46, Vol. 3], ARM TrustZone [25], or CHERI [88]. Allowing for the flexible employment of those mechanisms is thus important to enable the optimal use of the available hardware to build secure systems without sacrificing performance.

FlexOS [58] is a libOS designed to enable the specialisation of OSes towards security. It allows the flexible, fine-grained isolation of software components without committing to a specific mechanism to enforce it. This allows for the easy exploration of novel hardware and software-based protection and isolation mechanisms while providing applications the flexibility to choose the ones that suit their needs best. Postponing the choice of protection mechanisms to build-time also enables the exploration of the vast design space spanned by the dimensions of performance and security.

2.3.1 Existing Compartmentalisation Approaches

An early example of an attempt to modularise OSes is the Flux OSKit [37] project. Its goal is to provide OS primitives as libraries to make them reusable for the development of new OSes. Existing code such as device drivers and network stacks taken from different OSes, e.g. Linux and BSD, is also wrapped in libraries for simpler reusability. The separation into individual libraries with well-defined interfaces results in a highly modular framework with minimal interdependencies. Flux OSKit enabled various research projects, for example on programming languages that require tight integration with OS primitives, without the need to build a custom OS from scratch. On the other hand, it also demonstrates that gluing together existing code from different donor OSes with the component object model (COM) abstraction for compatibility generally results in suboptimal performance.

Microkernels reduce the functionality provided directly by the kernel to a minimum. Instead, device drivers and OS functionality is provided by userspace processes, sometimes called servers, that communicate via IPC. The SawMill [38] multiserver approach essentially re-examines a form of microkernel architecture for the isolation of OS components. An OS is decomposed into isolated servers that implement OS functionality such as resource management, networking, or file system access. Function calls crossing server boundaries are realised by IPC which inherently incurs some overhead because of the necessary context switch, marshalling and unmarshalling of parameters, and copying of data. Therefore, an efficient IPC mechanism that keeps the IPC frequency as well as overhead per IPC operation to a minimum is a key requirement for good performance. The SawMill prototype implements user-level servers on top of the L4 [62] microkernel. Benchmarks show that the SawMill prototype achieves better throughput for file system operations than L4Linux [41] but still lacks behind the monolithic Linux, confirming that system decomposition without hampering performance is extremely difficult.

LibrettOS [73] explores a hybrid approach between microkernels and libOSes. It can act as a microkernel providing applications with servers for interaction with hardware devices such as network interface controllers (NICs) or non-volatile memory (NVM). This allows for better isolation between device drivers and the kernel which in turn increases security through reduction of the TCB. At the same time, this enables the recovery from failures of individual components without the need to reboot the whole system. For applications with high performance needs LibrettOS can act similarly to a libOS, granting them direct access to virtual hardware resources. Furthermore, it allows applications to dynamically switch between the two modes at runtime to enable adaptation to the workload over time.

While microkernels allow for a natural decomposition of an OS into different components, they lack the support for fine-grained isolation and have historically suffered from lower performance compared to monolithic kernels [38]. In the past, there have been several attempts to provide fine-grained isolation based on protection mechanisms that rely on hardware extensions.

Mondrian memory protection (MMP) [92] associates protection domains with permission tables that specify the access rights for each address in the address space. Several alternative representations such as segments denoted by start address and size or multi-level permission tables indexed by the address of the target memory location are examined. On memory access the hardware performs checks against this permission table to determine the validity of the access. The hardware is extended by a permission lookaside buffer that caches entries from the memory resident protection table in a similar manner to a TLB for traditional address translation. Subsequent work uses MMP to enforce isolation between Linux kernel modules. Since no actual hardware architecture with the postulated properties exists, the resulting Mondrix [93] prototype could only be evaluated on machine simulators.

The code-centric memory domains (CODOMs) [85] approach explores the idea of using the instruction pointer as capability to control memory access. Each memory page is associated with a tag, all pages sharing the same tag form a protection domain. Code pages are additionally associated with access protection lists (APLs) that govern the access to protection domains. Again, the processor hardware is extended to allow for the representations of APLs and automated checks against the access rights specified by them. MMP and CODOMs both suffer from the fact that they rely on hypothetical hardware extensions that eventually were not adopted.

CubicleOS [78] also employs hardware extensions to allow for fine-grained isolation of components. In contrast to the ones presented above, the architectural extension relied upon by CubicleOS, i.e. Intel's MPK feature, is fully implemented in some commercially available Intel CPUs. Isolation is enforced at the granularity of dynamic libraries which are segregated into so-called cubicles. Each cubicle is associated with a protection key tag. Cross-cubicle function calls require switching of the current cubicle which amounts to manipulation of the protection-key rights for user space (PKRU) register. The toolchain automatically generates the code that performs this cubicle switch for each function of a libraries interface. For temporary sharing of data between cubicles the concept of windows is introduced. A window essentially modifies the protection key associated with the corresponding memory pages to allow the sharing of data with other cubicles, for example for the duration of a cross-cubicle call.

2.3.2 FlexOS Architecture

FlexOS is based on the Unikraft [55] unikernel framework and thus is highly modular. The basic architecture of FlexOS matches that of Unikraft from which it is descended, therefore the following description of the architecture also applies to both.

OS primitives are provided by libraries that can be individually included and configured at build-time. There are two basic types of libraries: core libraries and external libraries [14]. The former provide OS functionality such as memory allocation and scheduling. To increase flexibility, core libraries generally separate their interfaces from the implementation. For example, the alloc library defines the interface for memory allocators. At build-time a suitable implementation, such as the buddy allocator (allocbuddy), can then be chosen depending on the expected allocation pattern of the application. External libraries provide functionality not directly related to the OS, such as a C standard library. Besides those that implement general functionality, there are two special classes of libraries. Platform libraries provide support for the corresponding platform such as Linux Kernel-based Virtual Machine (KVM) or Xen. Architecture libraries implement architecture specific formats and operations like paging structures, cache line size, or access to specific registers. The flexos-core library defines and implements the isolation mechanisms offered by FlexOS. Figure 2.4 gives an overview of the organisation of internal libraries.

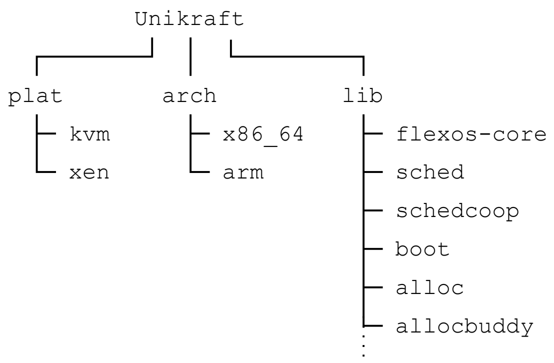


Figure 2.4: Organisation of FlexOS/Unikraft core libraries into platform libraries (plat), architecture libraries (arch), and others (lib). The flexos-core library is specific to FlexOS.

2.3.3 Compartmentalisation

Compartmentalisation decomposes software into isolated, collaborating components to restrict the impact of potential vulnerabilities [40]. In FlexOS isolation is enforced between compartments which are defined at the granularity of libraries with each compartment representing a protection domain. This has the advantage that existing library interfaces can be used as natural interfaces between compartments [58]. The process of compartmentalising an application amounts to replacing cross-library function calls with abstract call gates and annotating shared data accordingly. The required modifications to the source code are made by the developer during the process of porting an application to FlexOS and are facilitated by the assistance of tools (see Section 2.3.4).

Note that compartmentalisation does not impose a fixed assignment of libraries to compartments. Rather, it refers to the separation of individual libraries via call gates to allow arbitrary compartmentalisation configurations, the assignment of libraries to compartments, at build-time. The call gates represent the interface between compartments. They provide the necessary information for the toolchain to instantiate different compartmentalisation configurations according to a configuration file during the build process. For example, a call gate denotes the library to which the target function belongs in order to enable the toolchain, at build-time, to identify cross-compartment calls. Figure 2.5 gives a simplified example of how abstract call gates can be used to obtain different compartmentalisation configurations. Similarly, the annotations for shared data allow the toolchain to allocate the corresponding variables in the correct memory regions so that they can be accessed by all the compartments that use them.

Isolation backends implement the mechanism that enforces the segregation of compartments and can employ various software and hardware techniques such as protection keys [61, 84] or trusted execution environments (TEEs) [25, 30]. Once an application is compartmentalised, any available isolation backend can be utilised without further need for manual code modifications. The toolchain automatically replaces the abstract call gates with the appropriate implementations during the build process. As of now, the only supported isolation backend is based on Intel's MPK extension.

2.3.4 Porting of Applications

Since FlexOS is descended from Unikraft, porting an application or library to it first requires porting to Unikraft. Unikraft already supports a wide range of applications such as Nginx [8], Redis [11], and SQLite [12] in addition

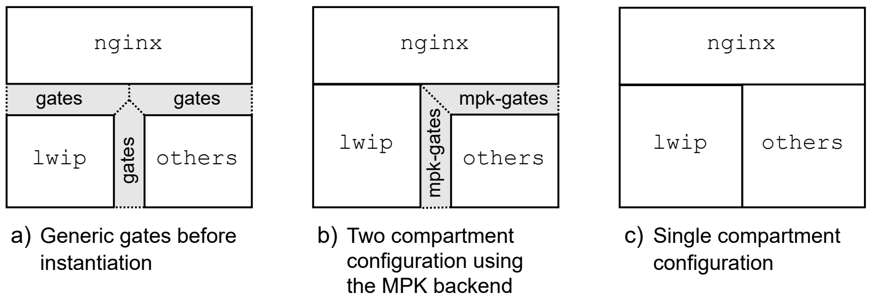

nginx
gates
gates
lwip
gates
others
a) Generic gates before instantiation
nginx
mpk-gates
lwip
mpk-gates
others
b) Two compartment configuration using the MPK backend
nginx
lwip
others
c) Single compartment configuration

Figure 2.5: Simplified example of different compartmentalisation configurations. a) Generic call gates between all libraries allow arbitrary assignment of libraries to compartments. b) Configuration with two compartments. The nginx application and lwip network stack occupy one compartment while all other libraries (summarised by others) are assigned to a different one. The remaining gates are implemented by the MPK isolation backend. c) The nginx application and all libraries are placed in the same compartment. All call gates are replaced with function calls.

to various programming languages and runtime environments [55]. Porting an application to Unikraft requires the creation of a Unikraft makefile (Makefile.uk) that is used to compile the source code [13]. For example, the Makefile.uk specifies include paths for header files and the source files to be compiled. A Config.uk file is used to define the dependencies of the application as well as possible configuration options [13]. Generally, the application needs to be modified to use the API provided by Unikraft. The porting effort is reduced by the fact that Unikraft offers partial POSIX support [55]. Therefore, applications relying on supported parts of the POSIX interface, for example for memory allocation or network sockets, do not need to be adapted.

Porting an application to FlexOS additionally requires the replacement of cross-library function calls with call gates and the annotation of shared data [58]. To facilitate this process, FlexOS comes with a tool that can perform these replacements automatically. This tool makes use of Cscope [2] to identify functions that are defined in other libraries. This information is then used to generate replacement rules used by Coccinelle [1] to make the appropriate replacements. Some cases like indirect function calls via pointers might not be handled correctly by the tool. The function instrumentation [16] feature of GCC can be used to locate such instances. Data that is shared between libraries has to be annotated so that the toolchain can place it in the appropriate memory regions.

2.3.5 The Build Process

Since FlexOS is based on Unikraft, their build processes are similar. Only the parts of the build process that are concerned with the instantiation of a particular compartmentalisation configuration are specific to FlexOS. The following description of the FlexOS build process reflects this by being split into two parts. The first one outlines the part that overlaps with the Unikraft build process while the second one focuses on details specific to FlexOS. Note that the description of the Unikraft build process is simplified for the sake of brevity.

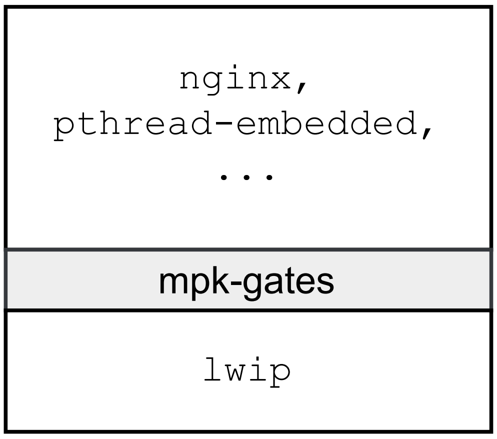
Each library or application provides a Unikraft makefile (Makefile.uk) and a configuration file (Config.uk) [13]. The former specifies include paths and the sources to build and registers the library with the build system. The Config.uk file defines the configuration options offered by the library in a similar manner to the Kconfig system used by Linux. These files are used to populate a graphical configuration menu that can be displayed via make menuconfig. This allows the user to select and configure libraries. For example, the user can specify the desired platform (e.g. kvm), architecture, system libraries such as allocators and a scheduler as well as other libraries like a C standard library implementation. Compiler options such as the optimisation level can also be specified here. The configuration choices are loaded from and stored to a .config file. Each configuration setting is used to define a macro that can be accessed in the source code.

FlexOS uses a modified version of the kraft [4] build tool. Therefore, the application additionally provides a kraft.yaml file. This is used to specify Unikraft related information such as the Unikraft version and dependencies. This file is then used by kraft to download the source code of the dependencies and to configure and build the unikernel image. For FlexOS this file is extended to also specify the isolation backend as well as to define the compartments and assign the libraries to them. Figure 2.6 shows a simplified example of a kraft.yaml file as well as the resulting compartmentalisation configuration. It is also possible to declare a compartment as default which has the effect that all libraries not explicitly assigned to any compartment will automatically be part of this default one. The toolchain then uses the description of the compartmentalisation configuration provided by the kraft.yaml file to instantiate the call gates accordingly. This is done by replacing the abstract call gates that are inserted as part of the porting process to FlexOS in one of two ways. Gates between libraries in the same compartment are replaced with function calls since no compartment switch is necessary. Gates that cross compartment boundaries are replaced with code that implements a compartment switch according to the isolation mechanism utilised by the

the chosen backend. The necessary code transformations are performed with the help of the Coccinelle [1] source-to-source transformation tool for C. Therefore, Coccinelle replacement rules are generated by the toolchain and subsequently used by Coccinelle to replace the abstract call gates with their appropriate implementations.

```
name: nginx
compartments:
  - name: comp1
     mechanism: intel-pku
     default: true
  - name: comp2
     mechanism: intel-pku
libraries:
  lwip:
    compartment: comp2
  nginx:
    compartment: comp1
  pthread-embedded:
    compartment: comp1
  [...]
```

(a) Configuration file (kraft.yaml)



(b) Resulting compartmentalisation configuration

Figure 2.6: A simplified example of (a) a configuration file and (b) the resulting compartmentalisation configuration. Two compartments, comp1 and comp2, are defined. The lwip network library is placed into comp2 while all other libraries, including the nginx application, are placed in the first compartment. The isolation backend is MPK (named intel-pku in the kraft.yaml), cross-compartment calls are implemented by it.

2.3.6 The MPK Isolation Backend

Intel's memory protection keys [46, Vol. 3], also known as protection keys for userspace (PKU), offer a mechanism for intra-address space isolation. They use four previously reserved bits in a page table entry to associate memory pages with one of 16 protection keys. These reside in the 32 bit wide PKRU register which holds two bits per key. The write disable (WD) bit controls write access to the page. If set, write access to pages associated with this key is disabled, resulting in page faults when attempted. Similarly, the access disable (AD) bit prevents read and write access to corresponding pages when set. MPK does not provide a way to prevent instruction fetch, a form of execute disable is not supported by this mechanism. Checks against the access rights set in the page table and TLB entries are performed independent of MPK, thus protection keys cannot allow accesses that are forbidden by page table entries. Special instruction (RDPKRU and WRPKRU) are provided to access the PKRU register.

The MPK isolation backend uses the protection keys after which it is named to isolate compartments. Each compartment is assigned a protection domain by associating it with its own protection key. The PKRU register is set up to disable access to pages tagged with protection keys of other compartments. On a cross-compartment call via an MPK-gate, the protection domain is switched to that of the target compartment. This involves saving the current domain's registers, clearing them, and loading arguments of the called function. Then the PKRU register is loaded with the protection rights for the target compartment before the stack pointer is switched to that of the thread in the target compartment and the call instruction is executed [58].

Since any code can directly modify the PKRU register, unauthorised writes to it must be prevented. This is achieved by scanning binaries for the WRPKRU instruction to ensure that it only appears in the code implementing MPK-gates. Should the WRPKRU instruction occur elsewhere, either intended by the programmer or unintentionally as part of another instruction or across the bounds of multiple instructions, additional measures have to be taken. For example, hardware watchpoints can be used to monitor and prevent the execution of these instructions [42].

Chapter 3

Analysis

In the previous chapter, we introduced the concept of flexible isolation and discussed its use cases. The goal of this work is to design and implement a novel isolation mechanism that is suitable for the flexible, fine-grained isolation of software components and that is able to provide strong isolation between protection domains. Therefore, we first define the threat model our isolation mechanism should be able to protect against in Section 3.1. Next, in Section 3.2, we examine different mechanisms that can be leveraged to provide isolation. We assess their practicability in the context of a framework that allows for easy exchangeability of isolation mechanisms and their ability to provide fine-grained isolation. Finally, we motivate the choice of a virtualisation-based approach over other options and discuss challenges inherent to it in Section 3.3.

3.1 Threat Model

Our goal is to provide strong isolation between protection domains. We assume that an attacker succeeds in corrupting a subset of the protection domains. These corrupted domains are then considered to be fully controlled by the attacker to the extent that arbitrary instructions may be executed. Even in the presence of a subset of potentially colluding adversarial protection domains, the attacker should only gain access to information that (1) is accessible via memory that is shared with at least one corrupted domain or (2) is accessible via interfaces exposed to the union of all corrupted domains. However, we assume that corruption of a domain may only happen after correct initialisation.

Note that we are only concerned with disclosure of private information of uncorrupted domains and not with system availability. This is because generally any domain can be assumed to be essential to the functioning of the overall system as otherwise this domain would be redundant and could be trivially removed. Therefore, a corrupted protection domain must be assumed to disrupt the functionality of the whole system.

3.2 Isolation Mechanisms

There exist many mechanisms to enforce isolation between software components. In the following, we want to consider the most common mechanisms and assess their adequacy in the context of flexible, fine-grained isolation.

3.2.1 Kernel-enforced Address Spaces

The most common form of isolation of software components is the separation of address spaces usually found in the process abstraction of general-purpose OSes. Each process is given its own address space consisting of a contiguous range of virtual addresses. The memory management unit (MMU) maps virtual addresses to the single physical address space according to paging structures set up by the kernel [81, p. 353-355]. Thus, processes are conceptually prevented from even naming addresses outside of their own address space. Privilege separation between the kernel and user processes protects the kernel from unauthorised access to its address space or direct manipulation of the hardware. This form of isolation has the advantage of being very flexible in the general scenario of running multiple distrusting applications concurrently on the same system. It is therefore not surprising that most general-purpose OSes adopt this mechanism. However, when considering fine-grained isolation at the library level and the coexistence and exchangeability with other isolation mechanism, this approach comes with some drawbacks.

Since the kernel must enforce separate address spaces, it needs to be protected from applications. Most commonly this is implemented by leveraging privilege rings[1] and the MMU provided by modern hardware. A context switch is required whenever control is transferred from one process to another which incurs significant overhead for the mode switch, the switching of paging structures, and the storing and restoring of the processor state. This incurs significant overhead [57] which is particularly relevant when considering that frequent context switches may be required by protection domain transitions.

[1]While there are approaches to achieving separation of the kernel without resorting to privilege rings [32], these require special design of the kernel and a form of MMU virtualisation.

Therefore, strict separation of the kernel hinders specialisation to a specific application including direct access to (virtualised) hardware. Maintaining separate address spaces also requires the kernel to keep track of the mapping of multiple virtual address spaces to a single physical one. In the scenario of a single application running in a virtual environment this duplicates the functionality of the VMM which already provides the guest with virtualised memory.

In the light of flexible, fine-grained isolation, kernel-enforced address spaces pose the additional challenge of requiring significant changes to the kernel. When considering the coexistence with intra-address space isolation mechanisms such as protection keys [78], this further complicates the kernel. Therefore, exchanging the isolation mechanism would essentially require multiple different kernel designs. This further adds to the TCB and broadens the potential attack surface, a problem that is further exacerbated by hardware specific vulnerabilities [64] that require the kernel to implement equally specific mitigations.

3.2.2 Capability-based Isolation

Conceptually, a capability is a token that gives its holder permission to access the referenced object in the way encoded by the capability [60, p. 3]. Therefore, a capability contains two logically distinct parts, a unique identifier of the referenced object, such as a pointer in case of memory references, and the access rights that are granted by the capability. The unification of reference and access rights has the advantage that sharing of resources is as easy as passing the corresponding capabilities. Consider, for example, the case that an object consisting of several memory pages needs to be shared between two processes. The owner of the object can simply pass a capability with the appropriate access rights to the other process which subsequently uses the obtained capability to access the object. The second process can further delegate the obtained capability as needed.

To allow control over the type of access permitted by a capability, usually a mechanism is provided that allows to derive more restrictive capabilities from ones with broader access rights. This enables the delegation of capabilities according to the principle of least privilege [77], which states that each component should be granted the minimum access privileges that are necessary for it to function correctly. For example, the owner of an object may hold a capability that allows full access, including the ability to destroy it in order to free the system resources associated with this object. If this object is to be shared with another process, access should be restricted to the required minimum. In case that the other process only needs to read from the shared object, this requires a capability with read-only access. Therefore, the owner of the object can derive from its capability with full access a capability that permits only read operations. This capability can then simply be passed to the other process to allow it the desired read access to the object without granting any additional access rights.

Capabilities allow for more flexible definition of protection domains than the traditional process abstraction. A protection domain can simply be defined as a context of execution together with a set of capabilities that restrict the access to specific resources. The context of execution can be a process [79], thread [28], or procedure [43] and does not necessarily come with the notion of a private address space.

In order for capabilities to be useful for protection, a system must (1) ensure that capabilities cannot be forged; and (2) provide a mechanism to enforce the access rights associated with any capability. There are two broad types of capability systems, those that are implemented purely in software and do not require any specialised hardware, and those that rely on hardware architectural support.

Software Capability Systems

Purely software-based capability systems such as EROS [79], Mungi [43], and Opal [28] rely on the kernel to protect and enforce the access rights conveyed by capabilities. For protection from unauthorised manipulation of capabilities several mechanisms can be employed. In software capability systems two approaches are prevalent: (1) segregation of capabilities from data and (2) password capabilities [24].

The first method separates capabilities from data by keeping them in special memory pages that are only directly accessible to the kernel. Thus, capabilities are protected from being freely manipulated as data. Capabilities are referred to by handles from user processes analogous to Unix file descriptors. Operations to derive new capabilities from extant ones are privileged and thus also involve intervention from the kernel.

The second method are password capabilities [24] which provide security only in a probabilistic sense. A password capability conceptually consists of a reference part and a password that associates certain access rights with the reference. The password is usually chosen at random by the system when a capability is created, thus there is no systematic relationship between the password and the encoded access rights. This has the advantage that capabilities are simple values that can be treated as data. The security achieved by this approach is only probabilistic because in theory it is possible to guess a correct password for a given reference, thus obtaining access to an object that was never intended. However, with sufficiently long passwords, this is unlikely.

In order to enforce the access rights defined by capabilities, privilege separation between kernel and user processes or threads (or other abstractions of program execution) is employed. To enable efficient checking of memory accesses, the kernel translates the access rights specified by capabilities to the available hardware mechanisms such as page-based protection [79].

Hardware-supported Capability Systems

Historically, some hardware has been designed specifically to support the use of capabilities [33, 91]. This comes with the advantage that hardware facilities are used to enforce the protection of capabilities as well as the access rights conveyed by them. Such systems provide capability registers that are able to store capabilities and use them for access to the referenced object. Capabilities are protected by either storing them in separate memory regions or by extending memory cells by tags that indicate whether a capability or ordinary data is stored [35]. Special instructions are provided to store capabilities to memory and load them into capability registers. Instructions for loading data from or storing data to memory operate on capabilities instead of plain addresses as for traditional hardware.

More recently, hybrid approaches have been explored that permit the use of standard page-based protection and additionally offer capability-based protection for intra-address space isolation. This provides an opportunity for fine-grained isolation within an address space and thus is a particularly interesting concept for compartmentalisation.

The CODOMs [85] approach uses capabilities to facilitate the sharing of data between protection domains. It relies on a hypothetical extension of the hardware architecture, for example, the introduction of capability registers and augmentation of page tables with tags indicating whether pages store capabilities or ordinary data.

Capability Hardware Enhanced RISC Instructions (CHERI) [87] is an ISA extension that allows the use of capabilities for intra-address space isolation. Similar to CODOMs, CHERI follows a hybrid approach by only extending existing ISAs with support for capabilities; traditional page-based protection mechanisms are still supported. To allow the secure storage of capabilities, each capability-aligned memory location is associated with a tag. When a valid capability is stored to such an aligned memory location via special instructions, the tag indicates that a valid capability is present. When any memory overlapping with the location storing a valid capability is written to via normal data operations, the tag is cleared and a valid capability is no longer stored there. This prevents the manipulation of capabilities via data-level operations. Special instructions are provided for controlled manipulation of capabilities.

Capabilities for Compartmentalisation

Generally, capabilities are well suited for providing isolation, both within a single address space as well as in the presence of private address spaces. Therefore, they are a good candidate for a mechanism to implement fine-grained compartmentalisation. However, purely software-based capability systems rely heavily on specially designed kernels for enforcement of protection. As already discussed in Section 3.2.1, this is problematic when considering the goal of making isolation mechanisms interchangeable as it requires special kernel design and thus significant modification to a kernel that also provides other mechanisms.

Hardware-supported capability extensions, specifically CHERI [87], are a promising candidate for the implementation of an isolation mechanism because the employed hardware extensions significantly reduce the reliance on a kernel to enforce protection of capabilities. Unfortunately, this comes at the expense of generality since architectural extensions are required that are not (yet) widely available in commodity hardware.

3.2.3 Protection Keys

Memory protection keys [26, 44–46] extend paging structures by a tag that associates each page with a protection key. All pages with the same tag form a group for which the access rights can be specified in a central location such as a special register. This allows for the easy modification of access rights to a whole page group without the need to modify the corresponding page table entries. Protection domains can then be associated with page groups and domain switches only require the modification of the access rights in the central location where they are stored. While protection keys are often not designed as a security feature capable of protection against malicious access but rather to reduce the chance of accidental invalid memory accesses [29], they have been used to build isolation mechanisms [42, 78, 84].

Implementations of protection keys vary in the specific access rights that they allow to enforce and whether they restrict their modification to privileged mode. User-mode accessibility of instructions to modify access rights, e.g. with Intel's memory protection keys (MPK) [46, Vol. 3] extension, usually comes with the benefit of low overhead for domain switches which is desirable for fine-grained isolation. Generally, additional measures [42, 84] to prevent unauthorised manipulation of access rights are required. In the following, we give a brief overview of some protection key implementations in commercially available hardware.

The PA-RISC [44] architecture developed by Hewlett-Packard defines protection identifiers (PIDs) that are (at least) 15 bits wide, allowing for a large number of protection domains. However, at any given moment only four PIDs can be loaded into the four corresponding control registers, which are only accessible in privileged mode. The large number of PIDs makes it infeasible to specify the access rights associated with each PID in a single register as this would require (at least) 2^{15} bits. Therefore, one write disable (WD) bit is attached to each PID which allows or denies write access to pages tagged with this PID. On each write access, the PID tag of the corresponding page table entry is compared with the four control registers and only if a matching PID with cleared WD bit is present, the write access is allowed.

The ARM 32-bit architecture supports a form of protection keys called memory domains [26]. Translation table entries include a four bit tag that designates a protection domain, allowing up to 16 domains. The domain access control register (DACR) contains two bits per key that control access protection which is only accessible in privileged mode (PL1 or higher). Depending on the values encoded in these two bits, access is either denied, allowed according to the access permissions recorded in the page table entry (PTE), or allowed regardless of the access rights specified by the corresponding PTE. Therefore, protection domains cannot only be used to restrict access rights set in the address translation structures but also to circumvent them. Memory domains are not available in 64-bit ARM architectures.

The IBM Power ISA [45] provides 32 protection keys, called virtual page class keys, with which memory pages can be tagged. The authority mask register (AMR) and instruction authority mask register (IAMR) specify the access rights for each page group and are accessible in privileged mode. The AMR holds two bits for each key to allow or deny read and write access whereas the IAMR specifies whether instruction fetch is permitted. Access rights specified by those keys apply in addition to those set in page table entries (and by other protection mechanisms such as secure memory).

Intel's MPK architectural extension associates each page with one of 16 protection keys. The protection keys index a special PKRU register which contains two bits per key. This allows to either disable write access or all data access to the corresponding page group. MPK does not provide a way to prevent instruction fetch from a page. Instructions to modify the PKRU register are accessible in user mode. For a more detailed explanation, including how they can be used to build protection domains, see Section 2.3.6.

While protection keys are a promising mechanism for intra-address space isolation, they also come with some problems. Most obviously, the use of protection keys is specific to the hardware, restricting the portability across platforms. This is not necessarily a problem when multiple isolation mechanisms are available in the context of flexible isolation. It is conceivable to provide different implementations for different architectures. However, this does not provide a solution for architectures that do not support any form of protection keys. Therefore, a universal solution that is usable across a wide range of platforms is desirable.

Furthermore, the protection provided by mechanisms relying on protection keys such as CubicleOS [78] or the MPK backend of FlexOS [58] is relatively weak. Should an adversary gain access to instructions that allow the manipulation of access rights for page groups (or that allow to outright circumvent this mechanism via access to control registers), isolation breaks down entirely. The access rights specifiable by some implementations such as Intel's MPK are also lacking the option for restricting instruction fetch, thus only data can be protected but not code.

Connor et al. [29] demonstrated that it is challenging to build a secure isolation mechanism on the basis of Intel's MPK extension. This can partly be attributed to the fact that this feature was not designed to protect against malicious software components.

3.2.4 VM-based Isolation

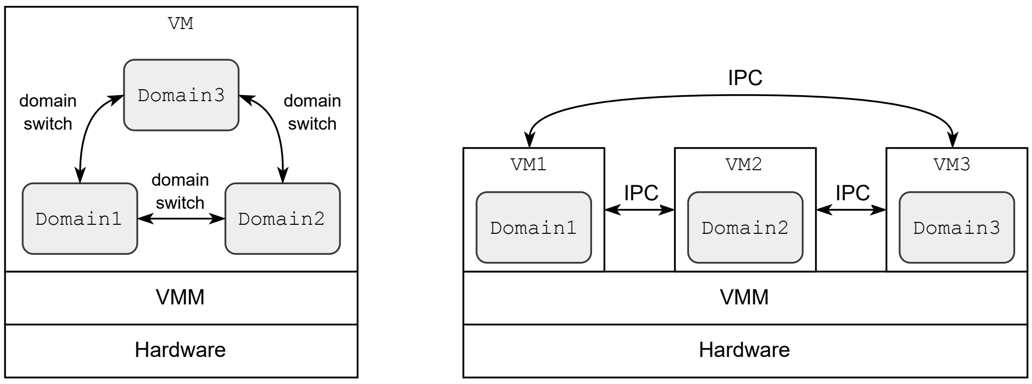
One rather radical approach to isolation is to house each protection domain in its own VM. This enforces strong isolation between domains, similar to the private address spaces of the process abstraction. However, in contrast to kernel-enforced address spaces, here the VMM is responsible for guaranteeing isolation between the VMs. This allows for a simpler kernel design because it no longer needs to protect itself from user software in order to be able to uphold isolation.

With the VM-based approach inter-domain calls are implemented via inter-VM IPC mechanisms. Domains only need to share memory where it is explicitly required (shared variables) or to implement the IPC mechanism. This allows for strong enforcement of inter-domain control-flow integrity (CFI)[2] since each domain provides a set of well-defined entry points only accessible via the IPC mechanism.

While this approach might seem prohibitively expensive at first, the use of lightweight unikernels makes it a viable option worth exploring. The mini-

[2]CFI refers to the concept that the execution of code must follow a control-flow graph that is determined at build-time [21].

malistic nature of unikernels drastically reduces the memory footprint of each VM which is critical for the viability of the VM-based isolation approach. For example, Unikraft [55] allows for VMs requiring as little as 2MB of RAM.



(a) Isolation within a single (virtual) machine

(b) VM-based isolation

Figure 3.1: Comparison of VM-based isolation to other approaches. In the case of isolation within a single (virtual) machine (a), the isolation mechanism is not further specified. Protection domains can be established via processes, capabilities, or intra-address space mechanisms and domain switches are implemented appropriately. In the case of VM-based isolation (b), protection domains are implemented by individual VMs with inter-VM IPC for cross-domain calls.

Previous research has successfully used virtualisation to solve related problems. LeVasseur et al. [59] use virtualisation to reuse device drivers across different OSes. Device drivers are executed in their expected host OS inside of a VM and a virtual device interface is exported to the client OS. The result is not only reusability of device driver code but also increased dependability and stability due to the isolation introduced between drivers in different virtualised environments.

VirtuOS [72] explores the use of virtualisation for decomposition of the kernel into vertical slices. The main motivation is again the isolation of drivers from the rest of the kernel to increase system stability. The VirtuOS project demonstrates the feasibility of distributing protection domains to different VMs while also showing that the overhead entailed by frequent domain crossings is a major challenge to this approach.

Kylinx [94] re-introduces the process abstraction in the context of unikernels to provide compatibility with legacy software expecting to execute in a general-purpose OS environment. Instead of burdening a unikernel with the implementation of processes, the concept of process-like VMs (pVMs) is used. Each process executes in its own VM, supported by a minimal unikernel. They show that the use of specialised unikernels as the basis for pVMs is highly effective in reducing the memory footprint, bringing it down even below that of Docker containers.

3.3 Virtualisation-based Isolation

Previously, we discussed different mechanisms that can be employed to achieve fine-grained isolation. In the following, we want to motivate the choice of the VM-based approach over any of the other mechanisms. We also discuss challenges inherent to this approach and examine the Unikraft unikernel framework as a potential substrate for the implementation of a VM-based isolation mechanism.

3.3.1 The Argument for VM-based Isolation

Specifically when considering flexible, fine-grained isolation, the VM-based approach offers some unique advantages over the other mechanisms discussed previously. Kernel-enforced address spaces and capability-based mechanisms require a separation of the kernel from user software, since the kernel is responsible for providing isolation. Clearly, kernel-enforced address spaces require the kernel to be protected from user code in order to prevent accidental or malicious changes to the data structures used to map virtual to physical pages. Software capability mechanisms that do not rely on specialised hardware also depend on the kernel to prevent the manipulation of capabilities stored in memory as well as the access rights specified by capabilities. Architectural extensions such as CHERI [87] can reduce the reliance on the kernel to enforce capability-based isolation, but they are confined to specific hardware and thus limit compatibility. While the separation of the kernel is standard in general-purpose OSes it has some disadvantages in the context of flexible isolation.

When considering flexible isolation with the option of multiple interchangeable mechanisms, requiring separation of the kernel is an additional challenge, at least when coexistence with intra-address space isolation mechanisms is desired. This is because a switch of the isolation mechanism would then come with drastic changes to core kernel components. With separation of the kernel a system call mechanism must be provided to access privileged functionality, an abstraction for the execution of code (e.g. a process) has to be implemented, and the protection of different virtual address spaces must be enforced. This is complicated even further by the fact that hardware specific vulnerabilities [64] must then be accounted for by the kernel. When switching to an intra-address space isolation mechanism such as protection keys, system calls and mode switches are no longer required and private virtual address spaces are abandoned. This makes approaches requiring separation of the kernel more difficult to coexist with those that do without it.

The VM-based approach on the other hand relies on the VMM to enforce isolation at the level of the (virtual) hardware. Therefore, no special precautions have to be taken to protect the kernel from untrusted code in the same domain. This keeps the required changes to the kernel to a minimum and thus only minimally increases the total TCB across all available isolation mechanisms when regarding them as interchangeable.

Furthermore, the VM-based approach is able to provide strong isolation, albeit at a higher performance overhead. This allows for the exploration of a vastly different point in the kernel design space that is spanned by the dimensions of security and performance. With the VM-based approach the primary focus is on security and strong isolation as opposed to low performance overhead prioritised by lightweight intra-address space isolation explored by previous research [42, 58, 78, 84].

Compared to other mechanisms, with VM-based isolation an attacker could be allowed unrestricted arbitrary code execution in one protection domain and still only be able to access other domains via their public interfaces. This is not the case with protection key based isolation since it must be ensured that instructions that allow the manipulation of access rights to protection domains are not accessible by untrusted code [42, 84]. Also, with VM-based isolation, a stronger form of inter-domain CFI can be enforced than with existing protection key based isolation mechanisms [58,78], because inter-domain calls are translated to inter-VM IPC operations. While kernel-enforced address spaces can ensure protection when an attacker achieves arbitrary code execution in user mode, the complexity of the system call interface increases the chance of vulnerabilities [69].

Another factor to consider is the availability of hardware support when relying on architectural extensions. Intel's MPK extension is only available to Intel Xeon Scalable server processors and 10th or later generation Intel Core processors [46, Vol. 1], restricting compatibility with other hardware. Hardware support for capabilities [87] is planned for some ARM processors [86], but not yet widely available. On the other hand, the VM-based approach offers a universal isolation mechanism since hardware support for virtualisation is common on modern hardware. This also makes it a good candidate as a default mechanism to fall back to if more hardware specific solutions are not available.

3.3.2 A Substrate for Protection Domains

Since protection domains are implemented by individual VMs, each of them needs a basic execution environment. This includes boot code to initialise the domain, scheduling in multithreaded scenarios, and the implementation of inter-VM IPC channels. To reduce the additional memory footprint that results from the duplication of this basic execution environment per domain, a minimal kernel is desirable. As already discussed in Section 3.3, unikernels exhibit these properties. They can be specialised to only provide the needed functionality to minimise the resource overhead.

The Unikraft [55] unikernel framework is highly modular and can be easily specialised to the requirements of VM-based isolation. The ability to build VM images containing a selected set of libraries facilitates the compartmentalisation. Libraries can be assigned to protection domains and subsequently distributed between the corresponding VM images via the Unikraft toolchain. Also, multiple compartmentalisation frameworks [58, 78] utilising protection key based isolation mechanisms have been successfully built on top of Unikraft.

3.3.3 Challenges of VM-based Isolation

Implementing protection domains as individual VMs comes with some challenges. Since each protection domain is placed in its own VM, cross-domain function calls have to be implemented by some form of IPC between the involved VMs. This requires making the parameters accessible to the domain of the callee as well as passing any potential return value back to the caller domain on return. While a call is being executed in the callee domain, the thread that initiated the call cannot make progress and must be paused until control flow for this thread returns to the calling domain. This can happen either because of a return from the function or because the called function leads to another function being called in the first protection domain, resulting in a nested call inside the first domain. Note that a nested call need not originate from the domain that was originally called, as it can in turn make calls to other domains. This essentially results in the call stack that would be observed in a single execution environment being distributed over all the involved domains.

In order to enforce CFI for inter-domain calls the translation of those calls to inter-VM IPC must ensure that control is only transferred to valid entry points. These entry points are the starting addresses of all functions that are part of the public interface exposed by a protection domain. This restricts the access gained by an attacker in case of corruption of a domain to the public interfaces of and memory shared with other domains.

The implementation of the inter-VM IPC channel between two VMs has to be guarded against access from any domains other than the two communication partners. Failure to secure these communication channels would allow attackers that successfully corrupted one domain to observe or even manipulate the communication between other domains, including parameters and return values from function calls. This opens up further possibilities to corrupt other domains or disclose information private to them and violates our notion of security (see Section 3.1). Therefore, securing private communication is essential to providing strong isolation between protection domains.

To allow for the sharing of data across protection domains, a mechanism must be provided to allocate memory in a way such that it can be accessed by all domains that need to access the shared data. It must be ensured that pointers to this shared data remain valid when they are passed as arguments to inter-domain function calls.

Race Conditions on Shared Memory

One problem inherent to the VM-based isolation approach relates to the fact that the resulting system is inherently distributed. Since protection domains are implemented as separate VMs, even a single thread is split into activities in different execution environments. This may have unexpected consequences for libraries operating on memory shared between different protection domains. When a library expects to execute in a single-threaded context, it does not need to consider the possibility that a region of shared memory might be manipulated concurrently by different activities. In fact, even code that is used in multi-threaded environments frequently does not consider this problem sufficiently [89]. The resulting race conditions often lead to security relevant bugs that are hard to detect.

When considering the compartmentalisation of a system to increase security, such race conditions could potentially enable an attacker that successfully corrupted one protection domain to gain control over further domains, significantly weakening the isolation gained by compartmentalisation. Therefore, the possibility of race conditions on shared memory should be taken into account when decomposing a system into multiple protection domains. Especially for inherently distributed mechanisms such as VM-based isolation, mitigations should be provided to reduce the reliance on individual software components to explicitly address the issue of possible race conditions. The inter-VM IPC mechanism used to implement cross-domain calls must be designed to be resistant against potential race conditions on shared memory.

3.4 Summary

In this chapter, we defined the notion of security we want to achieve and the thread model that we want to protect against with our isolation mechanism. We also discussed different protection mechanisms in the context of flexible, fine-grained isolation. We argued that the VM-based approach has several advantages over the other options and allows for exploration of an isolation mechanism with strong guarantees at the cost of higher overhead for domain transitions. Finally, we discussed challenges to this approach, including the necessity for secure inter-VM IPC and the problem of race conditions on shared memory inherent to a distributed system.

Chapter 4

Design

In the previous chapter, we defined our threat model and motivated the exploration of a VM-based isolation mechanism. In this chapter, we present the design of a VM-based isolation mechanism that supports our threat model previously defined in Section 3.1. In Section 4.1, we begin by explaining how control is transferred between compartments as the result of cross-compartment function calls. Next, in Section 4.2, we explain the design of the inter-VM inter-process communication (IPC) mechanism on top of which cross-compartment function calls are implemented. In Section 4.3, we give an overview over all the components introduced over the course of the previous sections. Next, in Section 4.4, we complement the threat model defined in Section 3.1 with a description of the trusted computing base. Finally, in Section 4.5, we address the problem of race conditions on shared memory which we previously identified in Section 3.3.3 before summarising in Section 4.6.

4.1 Cross-Compartment Function Calls

Cross-compartment function calls are at the heart of our VM-based isolation mechanism. Therefore, we focus much of our effort on the design of an efficient remote procedure call (RPC) mechanism. Note that we distinguish the RPC mechanism from the underlying inter-VM IPC mechanism. The former is responsible for implementing cross-compartment function calls, whereas the latter only provides primitives for communication between individual VMs. Therefore, the inter-VM IPC mechanism is much simpler and can be understood as the substrate on top of which the more complex RPC mechanism is implemented.

4.1.1 Preserving the Function Call Semantics

Since we are concerned with flexible isolation, we need to make sure that we can support arbitrary compartmentalisation configurations. This implies that cross-library function calls are either direct function calls in the case that both libraries are placed in the same compartment or cross-compartment function calls otherwise. Therefore, cross-compartment function calls must preserve the function call semantics to ensure compatibility between the case of a call within the local compartment and the case where a compartment boundary is crossed.

This includes the transfer of the active control flow to the target compartment on cross-compartment function calls and the correct handling of nested calls. We also have to ensure that cross-compartment function calls behave according to the calling convention of the underlying system. Our design is oblivious to the employed calling convention although our implementation is specific to the System V AMD64 ABI calling convention (see Section 5.1).

4.1.2 Requirements for the RPC Mechanism

Before going into the details of our design, we first list the requirements that our RPC mechanism must meet in order to provide an adequate mechanism for cross-compartment function calls. The following functionality is required:

(F1) *Signal control flow between compartments.* When a cross-compartment function call is executed, control flow must be transferred between the compartments of the caller and the callee. Therefore, the IPC mechanism must be able to signal events that transfer the active control flow between compartments. These events are either of type `CALL` to indicate that control flow is transferred from the caller to the callee, or of type `RETURN` to signal control flow in the opposite direction.

(F2) *Identify thread of execution.* We have to allow any thread in any compartment to issue cross-compartment calls. Therefore, the IPC mechanism must be able to signal from which thread a cross-compartment call originates. The callee compartment then uses this information to execute the requested function in the context of the corresponding handling thread.

(F3) *Transmit required information.* In order to complete a cross-compartment function call, the caller must indicate which function to call and provide the arguments to the callee compartment. On return from a cross-compartment call any potential return value must be handed back to the caller compartment.

(F4) *Preserve function call semantics.* As discussed in Section 4.1.1, cross-compartment, function calls must preserve the semantics of function calls as defined by the calling convention. Our design is not specific to any calling convention and the implementation is responsible for ensuring that cross-compartment function calls follow the calling convention of the underlying system.

In addition to the functional requirements listed above, our RPC mechanism must meet the following non-functional requirements to ensure security and efficiency.

(S1) *Private communication channels.* As discussed in Section 3.3.3, private communication channels between compartments must be provided. This is necessary in order to protect the communication between two compartments from being intercepted or manipulated by any compartment other than the two communication partners.

(S2) *Inter-compartment control-flow integrity.* We must ensure that code execution initiated by cross-compartment function calls can only begin at well-defined entry points. These entry points are the starting addresses of all functions that are part of a compartment's public interface, which in turn is the set of all functions that make up the public interfaces of all libraries placed in that compartment.

(SC) *Scalability.* We must ensure that the RPC mechanism does not perform any complex and time-consuming operations and that performance does not degrade with increasing levels of concurrency. Therefore, we have to design all data structures in a way such that operations on them that occur with every cross-compartment call have constant runtime, regardless of the number of threads concurrently issuing calls.

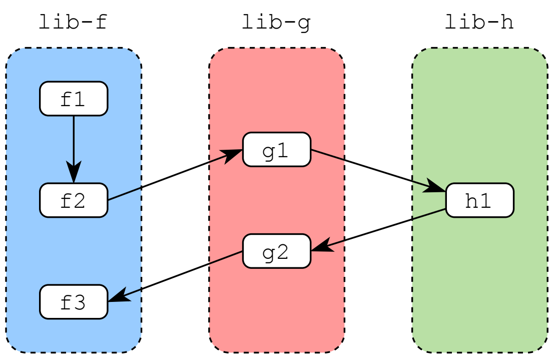
4.1.3 Control Flow between Compartments

Cross-compartment function calls conceptually transfer the active control flow from a thread in the compartment of the caller to a thread in the callee compartment. In order to manage this control flow between compartments, we introduce a special thread, the so-called RPC server, in each compartment. The RPC server of a compartment is responsible for handling incoming requests for cross-compartment function calls. For this purpose, the inter-VM IPC mechanism must be able to signal two types of events: `CALL`s and `RETURN`s. An event of type `CALL` indicates that a function is requested to be called in the compartment to which the event is signalled. On the other hand, an event of type RETURN indicates that a function completed execution and is signalled to the compartment that originally requested the execution of that function.

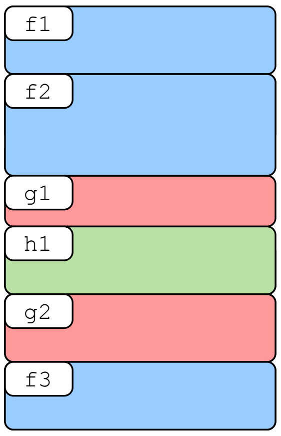
Distributing Call Stacks between Compartments

To understand how cross-compartment function calls are handled, we first compare a sequence of local function calls to one that contains cross-compartment calls. Consider, for example, the case of three libraries lib-f, lib-g, and lib-h shown in Figure 4.1 together with the corresponding call graph. When all libraries are placed in the same compartment, the call stack of a thread executing f1 at the point where f3 is called is depicted in Figure 4.1b (note that we depict call-stacks as growing downwards in line with the behaviour on x86 architectures). However, when each library is placed in its own compartment, some of the function calls must cross compartment boundaries. This essentially results in the same call stack being distributed between three threads, one in each compartment involved in the call sequence. The resulting state of the distributed call stack is shown in Figure 4.1c. The unmarked grey frames inserted between cross-compartment calls represent additional logic needed to correctly route the control flow and is part of the RPC server.

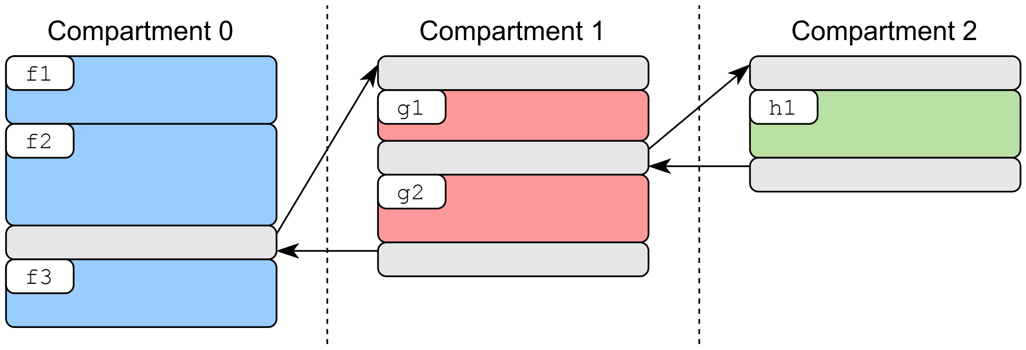
We can see that, in order to handle cross-compartment calls initiated from a thread in compartment 0, we need one handler thread per other compartment that is involved in the call sequence. Furthermore, we observe that we need to define a strategy to assign handling threads to incoming requests in a way that correctly handles nested calls. When the cross-compartment call to compartment 1 is made to request the execution of g1, a thread is first assigned to handle that request. Later, when compartment 2 requests to call g2 as part of the same call sequence, instead of assigning a new thread to handle the request, the function must be executed in the context of the thread that is already handling the request for g1. This is necessary in order to maintain information about active function calls, local as well as across compartments, in an efficient manner. Therefore, we get essentially the same call stack as in the case without isolation, with stack frames in the same order, but distributed between the individual compartments. The top of this distributed stack can be found by following the stack frames downwards for local function calls and along the arrows shown in Figure 4.1 on cross-compartment calls (whenever one of the unmarked grey blocks is encountered). Similarly, the bottom of the distributed call stack can be found by tracing calls back in the reverse direction.



(a) Libraries and corresponding call graph



(b) Call stack without isolation



(c) Distributed call stack with isolation into three compartments

Figure 4.1: Comparison between local function calls and cross-compartment calls. (a) shows the three libraries, the functions they provide and the corresponding call graph (starting at function f1). (b) depicts the call stack that is resulting from a call to f1 at the point just after f3 is called. (c) shows the corresponding call stack in the case that each library is placed in a separate compartment. The arrows indicate control flow between compartments in response to cross-compartment function calls. The grey blocks indicate additional information related to cross-compartment function calls stored on the stack.

Assignment of Threads Handling RPC Requests

In order to explain how cross-compartment function calls are handled, we make a distinction between the threads that exclusively handle RPC requests and those that perform general tasks. In the following, we refer to threads that exclusively handle RPC request as *RPC threads* whereas all others are referred to as *regular threads*.

To ensure that cross-compartment function calls are handled by the correct RPC thread, we first need a way to uniquely identify the *origin* of each cross-compartment call. Consider, for example, the distributed call stack depicted in Figure 4.1c. The first cross-compartment call in the sequence of active function calls comes from the thread in compartment 0. Therefore, the origin of this cross-compartment call as well as any one that appears after it in the active call sequence is the thread in compartment 0 that contains the stack frame of function f2.

Formally, we define the origin of a cross-compartment call as the thread that initiated the first cross-compartment call in the sequence of active function calls. Note that the origin is a property of the sequence of currently active function calls and therefore is identical for all cross-compartment calls within that sequence. Conceptually, this thread can be identified by tracing the distributed call stack back to the very first stack frame. The call stack of this thread must also contain the first active cross-compartment call (if there is an active cross-compartment call, otherwise the distributed call stack is identical to the local call stack).

Since the thread IDs (TIDs) across compartments are not unique, we define the origin of any cross-compartment function call as the pair (cid,tid) where tid is the TID of the originating thread and cid the ID of the compartment to which it belongs. Note that the origin of any cross-compartment call is necessarily a regular thread since RPC threads only become active in response to cross-compartment calls and therefore can never contain the first stack frame on the distributed call stack. In practice, the origin is passed as a hidden argument to any cross-compartment call and the individual compartments need not keep track of the state of the call stacks of threads in other compartments.

Now that we know how to associate each cross-compartment call with a unique origin, we can define how RPC threads are assigned to handle incoming requests. The RPC server of each compartment keeps a mapping from origins to so-called *RPC entries*. An RPC entry indicates the handling thread assigned to requests from a specific origin. Additionally, an RPC entry contains an activation counter to keep track of the number of active function calls (stack frames) in the context of the RPC thread. Initially, each origin is mapped to an empty entry indicating no RPC thread is assigned to it and the activation counter is zero. When the RPC server receives a request for a cross-compartment function call, it first finds the RPC entry corresponding to the origin of the call. In the case that the RPC entry indicates that an RPC thread is already assigned to that origin, the activation counter is incremented and the RPC thread is dispatched to handle the request. However, in the case that the RPC entry indicates no assigned RPC thread to handle the request, the RPC server assigns one from the pool of idle RPC threads it manages. Then it proceeds, analogous to the previous case, by incrementing the activation counter (which previously was zero) and dispatching the newly assigned RPC thread to handle the request.

On return from a cross-compartment call we need to revert the steps outlined above. This means that the activation counter of the corresponding RPC entry is decremented and in the case that it reaches zero, the RPC thread is deassigned and returned to the pool of idle RPC threads. The steps necessary to manage the assignment of RPC threads to incoming requests is illustrated by Figure 4.2.

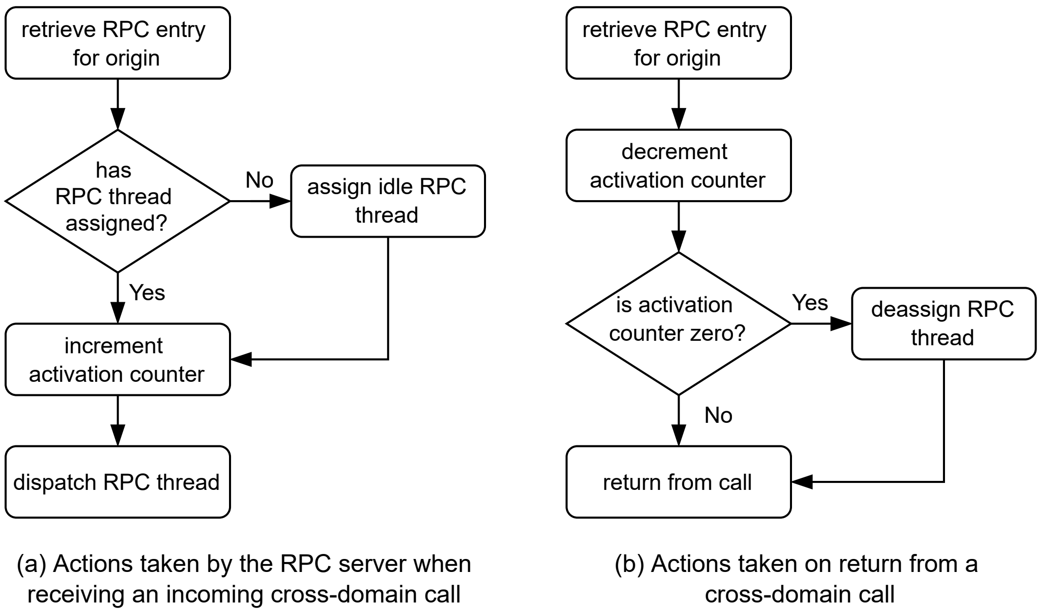

retrieve RPC entry for origin
has RPC thread assigned?
No
assign idle RPC thread
Yes
increment activation counter
dispatch RPC thread
(a) Actions taken by the RPC server when receiving an incoming cross-domain call
retrieve RPC entry for origin
decrement activation counter
is activation counter zero?
Yes
deassign RPC thread
No
return from call
(b) Actions taken on return from a cross-domain call

Figure 4.2: Actions taken when (a) a cross-compartment call is received by the RPC server and (b) a cross-compartment call returns.

4.2 The inter-VM IPC Mechanism

The inter-VM IPC mechanism is responsible for communication between the VMs which house the individual compartments. Our RPC mechanism we described earlier is responsible for implementing cross-compartment function calls and is built on top of the much simpler inter-VM IPC which consists of two parts.

First, each compartment has a message queue which is used to signal control flow between compartments and to identify the origin of a cross-compartment call. This message queue is accessible to all other compartments which can signal events to a compartment by posting messages to the corresponding message queue. The RPC server of each compartment retrieves messages from this message queue and dispatches RPC threads in response to requests for cross-compartment function calls.

Second, private communication channels, as required by (S1), are established by sections of memory that are shared between each pair of compartments. The maximum number of regular threads allowed per compartment is fixed (but configurable), therefore the number of possible origins of a cross-compartment call is also fixed (we remember that an origin is a pair `(cid,tid)` consisting of the TID of the originating thread and the ID of the compartment to which this thread belongs, see Section 4.1.3). For each possible origin, a slot in this shared memory section is reserved. This slot is called *call control block (CCB)* of the corresponding origin and is used to convey information needed to execute the function call in the target compartment. Note that a CCB is re-used for nested calls of the same origin between two compartments, e.g. in the example shown in Figure 4.1 the call from `h1` to `g2` re-uses the same CCB previously used for the call from `g1` to `h1`. The information conveyed in a CCB includes an identifier of the function to call as well as the arguments and return values. By identifying functions with abstract IDs we satisfy requirement (S2). This requires the RPC servers to translate the function IDs to the correct addresses before dispatching RPC threads. Figure 4.3 gives an overview of the components involved in the implementation of cross-compartment function calls and their interaction with each other.

The split between the shared message queues and the private communication channels has several benefits when considering the runtime overhead of a cross-compartment function call and addresses the scalability requirement (SC). By sharing the message queues between all compartments, each RPC server is only required to observe one queue, regardless of the number of compartments. Furthermore, by statically assigning each possible origin a CCB in the private communication channel, this removes the need for any dynamic allocation when a cross-compartment function call is issued. Instead, the origin, which is known to RPC threads at any time, is used to index the corresponding CCB in the private communication channel. The RPC threads can freely write to the CCBs without the need for synchronisation because they are dispatched by the RPC server of the corresponding compartment. This happens only in response to messages posted to the message queue of a compartment. Therefore, the only part of this system that requires syn-

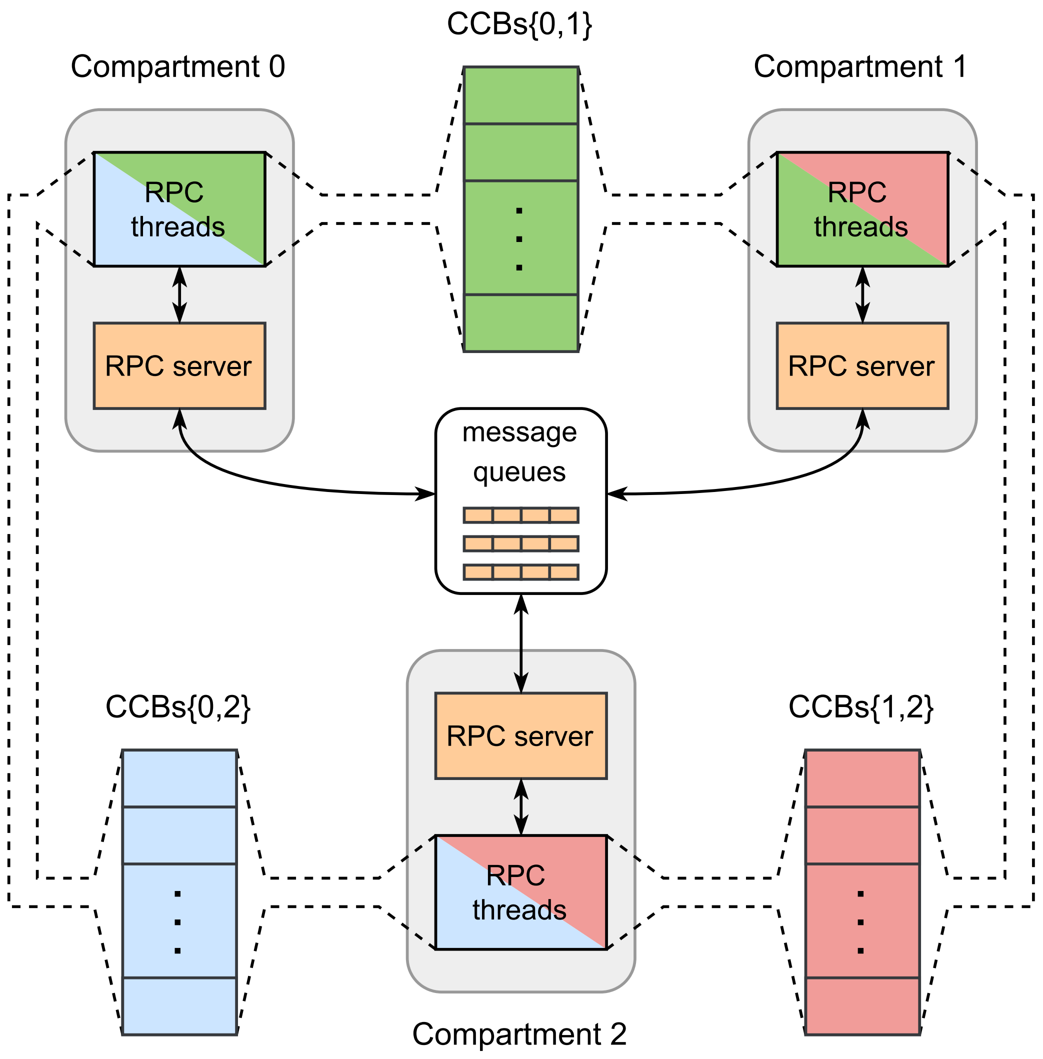


Figure 4.3: Overview of the inter-VM IPC mechanism and the components involved in the implementation of cross-compartment function calls. The RPC servers communicate via shared message queues. The dashed lines indicate the private communication channels between each pair of compartments. They contain the CCBs that are used by the RPC threads to convey the information needed for cross-compartment function calls. The memory regions for the private communication channels are only accessible by the two compartments that share a particular channel.

chronisation are the message queues that follow the *multiple producer single consumer* pattern. The RPC server of a compartment consumes messages from its message queue while all other RPC servers can post messages to it. Note that this requires careful implementation, specifically the introduction and careful use of a state variable in the CCBs, in order to prevent malicious compartments to spoof cross-compartment function calls between two other compartments (see Sections 5.1.2 and 5.3).

A drawback of the static allocation of CCBs is the high memory demand which is proportional to the product of the number of compartments and the maximum number of supported (regular) threads. In most scenarios it is plausible that only a small subset of all threads will engage in cross-compartment function calls, thus leading to most CCBs being unused. However, this problem is alleviated by the fact that the maximum number of threads can be easily configured to the needs of the application.

4.2.1 Sharing Data between Compartments

To allow the sharing of data between compartments, each compartment maintains a shared allocator that allows it to allocate memory that is shared between all compartments. Pointers to this shared memory can then be passed as arguments in cross-compartment function calls. Without any further precaution the potentially security-relevant problem of race conditions on shared memory already described in Section 3.3.3 arises. We address this problem later in Section 4.5.

4.3 Summary of Components

In the previous sections, we outlined our mechanism for implementing cross-compartment function calls and the inter-VM IPC mechanism on top of which this is implemented. We now want to give a brief summary of all the components as well as the requirements we have to consider when implementing them.

4.3.1 The RPC Server

One RPC server per compartment coordinates control flow between its compartment and others. It communicates with other RPC servers through a set of shared message queues and dispatches RPC threads in response to messages retrieved from its own message queue. It is also responsible for identifying the RPC thread in the context of which a requested function should be executed. Therefore, the RPC server provides the functionality (F1) and (F2) as required by Section 4.1.2. The RPC server requires several sub-components which are listed in the following. Figure 4.4 gives an overview of the interaction of the RPC server and its sub-components.

Message Queue

Each compartment's RPC server has its own message queue from which it retrieves messages. All RPC servers have access to all message queues since they need to send messages to the RPC servers of other compartments. This leads to the messages queues being accessed according to the *multiple producer single consumer* pattern.

RPC Entry Map

The RPC entry map is used to map the origin of any cross-compartment function call, which is indicated with each message received via the message queue, to an RPC entry. The RPC entry indicates the handling RPC thread assigned to an origin and the activation counter. Since the RPC entry map is consulted frequently by the RPC server, determining the RPC entry corresponding to a given origin must be fast (runtime $\mathcal{O}(1)$).

RPC Thread Pool

The RPC thread pool holds all idle RPC threads that are eligible to being assigned to handling an incoming cross-compartment function call. When empty, the RPC thread pool must be expanded by creating additional RPC threads to guarantee that there are RPC threads available to handle cross-compartment calls. Since RPC threads are frequently added and removed from the RPC thread pool, these operations must be fast (runtime $\mathcal{O}(1)$).

4.3.2 RPC Threads

RPC threads are assigned and dispatched by the RPC server in response to messages received via its message queue. Idle RPC threads are managed by the RPC thread pool.

4.3.3 Private Communication Channels

Private communication channels between each pair of compartments enable the secure exchange of information required for cross-compartment function calls. This information includes an identifier for the function to call, arguments, and potential return values and is organised into individual CCBs.

4.3.4 VM/EPT Call Gates

As discussed in Section 4.1.1, call gates implement cross-compartment function calls. They are responsible for transmitting the information required to perform a cross-compartment function call via the private communication channels while ensuring that the semantics of a function call are preserved. Therefore, they provide the functionality (F3) and (F4) as required by Section 4.1.2. Note that call gates should not be viewed as a separate component but rather a piece of code that is inlined at the call-site of any cross-compartment function call to initiate the necessary actions by communication with the RPC server.

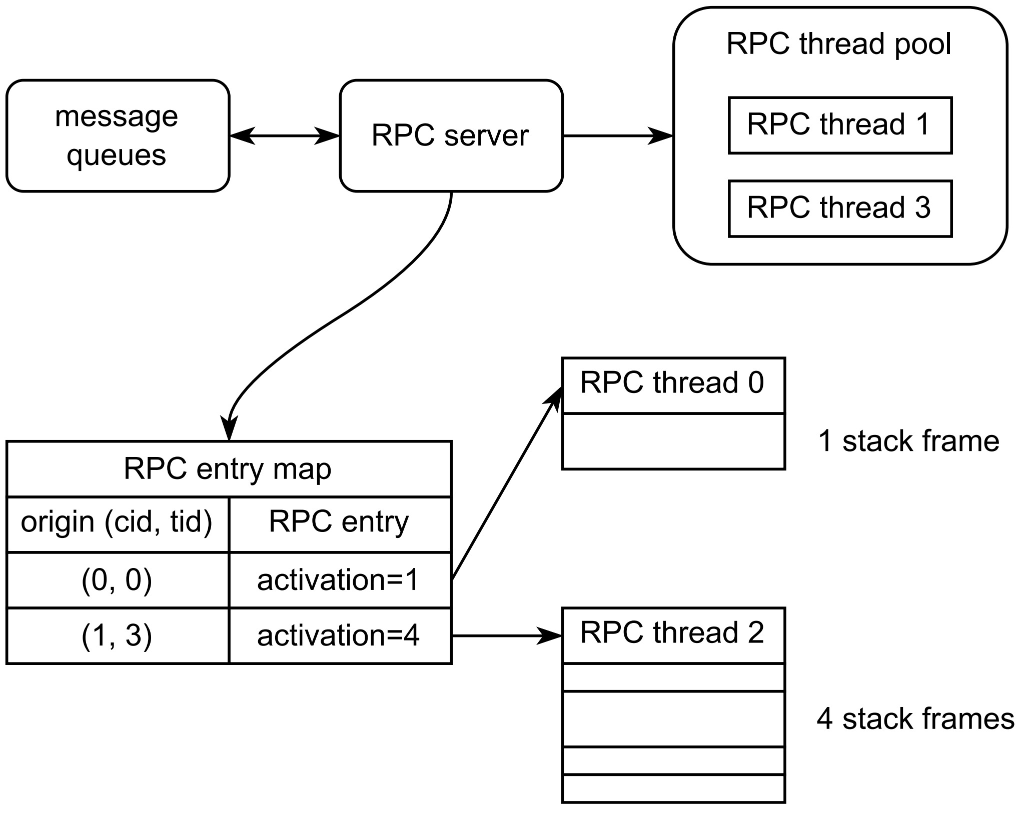


Figure 4.4: Interaction between the RPC server of a compartment and its sub-components. In the depicted scenario there are two active cross-compartment calls handled by separate RPC threads.

4.4 Trusted Computing Base

In Section 3.1, we defined the notion of security that our isolation mechanism should provide as well as the kind of adversary it should protect against. We recognise that some components are so deeply involved with the system that the defined notion of security cannot be achieved without considering these components trusted. Therefore, in the following, we want to complement the definition of our threat model by describing which components of the system must be included in the trusted computing base (TCB).

Clearly, the VMM must be trusted because it provides each VM it is hosting with virtual hardware resources and thus has unrestricted access to those resources. Therefore, the VMM is trusted to provide each compartment with a virtual execution environment that conforms to the respective ISA.

Furthermore, core system components must be considered trusted as they provide essential functionality that allows subverting the whole system when considered under adversarial control. We must assume the correctness of these components as clearly such a malicious core component replicated across all compartments results in the corruption of all individual compartments and therefore the system as a whole. Since these components are replicated in each compartment, including them in the TCB does not imply that they need to be specially protected in each compartment, since their corruption only affects the integrity of this compartment. Should an attacker gain control over a compartment this also includes the parts of the TCB of that particular compartment. The components we consider part of the TCB are:

1. Early boot code that must be trusted to ensure correct system initialisation.
2. The memory manager that is trusted to correctly maintain memory mappings.
3. The scheduler that is trusted to correctly initialise and restore threads when they are dispatched.
4. The first-level interrupt handler's context switch primitives are trusted to correctly transfer control to ISRs and restore the original state of the executing thread on return.
5. The implementation of the isolation mechanism must be trusted to enforce CFI between compartments and not disclose private data other than what is explicitly passed via cross-compartment calls. This includes all components listed in Section 4.3.

4.5 Addressing Race Conditions on Shared Memory

In Section 3.3.3 we introduced the problem of race conditions on memory that is shared between two or more compartments. In the following, we address this problem in two ways. We first present a simple solution that relies on automatic copies but is limited to scenarios where the program semantic allows the substitution of private copies for shared data. Next, we present a general solution that leverages the VMM to prevent the manipulation of shared data that is pointed to by arguments of cross-compartment function calls.

4.5.1 Automatic Copies on Cross-Compartment Calls

A simple way to eliminate race conditions on shared memory is to copy the data to a private memory region before performing operations on it that require it to be consistent. It could be argued that any library that is aware of the possibility of concurrent activities manipulating shared data is responsible for ensuring consistency in any case where this is relevant to the security of the system. Unfortunately, this is often neglected resulting in a wide range of vulnerabilities [89]. Furthermore, this problem is exacerbated by the distributed nature of our VM-based isolation mechanism.

Therefore, we suggest the introduction of simple code annotations that allow the toolchain to detect such situations and to emit code that automatically performs a copy of the shared data in question. These annotations are attached to the definition of the functions that take as arguments one or more pointers to memory shared with other libraries. The toolchain then uses these annotations to emit code (in the compartment to which the annotated function belongs) that copies the shared data to a private memory region and substitutes a pointer to that private copy for the original pointer to the shared data. On return from a cross-compartment function call that used this automatic copy mechanic, all memory allocated as part of the copy process must be freed.

Listing 4.1 gives multiple examples of such annotations. In the case of the function `foo`, a flat copy suffices. Therefore, it is enough to attach an annotation that indicates that the data pointed to by the argument has to be copied on cross-compartment calls. For more complex data structures a deep copy may be required. In this case, code to make a suitable copy can be provided. Furthermore, code to perform cleanup after the return of the function can be provided which is necessary to free any memory allocated as part of the custom copy process. An example of this is shown for the function bar in Listing 4.1.

```
@vmept_copy(x)
@vmept_copy(s)
void foo(int *x, struct simple_struct *s) { ... }

@vmept_copy(s,
        { /* code to perform custom copy */ },
        { /* code to perform cleanup */ } )
void bar(struct complex_struct *s) { ... }
```

Listing 4.1: Examples of annotations used to make automatic copies of shared data on cross-compartment function calls.

Limitations to the Automatic Copy Approach

While the automated copy approach is simple, its use is limited to scenarios where copies can be substituted for shared data without altering the program semantics. This excludes cases where shared data is used for bi-directional communication. In such cases a more general approach must be taken that is explained in the following. Note that simply copying back the data to shared memory before returning is not necessarily sufficient in cases where there are nested calls (e.g. back to the compartment that initiated the cross-compartment call).

4.5.2 Selective Write-Protection of Shared Memory

A more general approach to solve the problem of race conditions on shared memory involves an extension of the VMM. The basic idea is to provide a custom hypercall that grants exclusive write access to pages holding shared data to the callee compartment for the duration of a cross-compartment function call. In the following, we formalise this concept and show how the VMM can be extended to support this mechanism.

Ownership of Shared Memory Pages

Each compartment maintains a shared allocator that makes allocations from a memory region that is shared between all compartments. The memory regions from which shared allocators of different compartments make their allocations are must not overlap, therefore each shared memory page can be uniquely associated with an allocator. At any given time at most one compartment is the owner of a shared memory page. Initially, this is the compartment to which the allocator belongs that contains the page. Ownership of a page gives a compartment full (read and write) access to it while all other compartments only have permission to read the page. The VMM is responsible for enforcing the access permissions.

Transferring Ownership on Cross-Compartment Calls

When a cross-compartment function call is issued with a pointer to shared memory as an argument, the allocator is consulted to find the page numbers over which the shared data extends. The calling compartment then signals to the VMM, via hypercall, that it is willing to transfer ownership of these pages to the callee compartment. In response to that hypercall the VMM makes the appropriate changes to the memory mapping such that the compartment that issued the call no longer has ownership of (and therefore no write access to) the specified pages[1]. The VMM also needs to verify that the pointer indeed points to the beginning of the memory region consisting of the pages indicated by the caller compartment. Therefore, the pointer must also be passed in the hypercall and the VMM needs to have access to the shared allocators to verify that the allocation unit corresponding to the pointer is within the given pages. When the callee handles a cross-compartment call involving a pointer (that necessarily points to the shared memory region), it issues a hypercall to request ownership of the shared data pointed to. The VMM records the pointer and corresponding pages from the initial hypercall made by the caller compartment and therefore only needs to make sure that the pointer presented by the callee matches that indicated by the caller. At this point the transfer of ownership from the compartment of the caller to the callee compartment is considered complete. On return from a cross-compartment function call, ownership of the pages is transferred back to the caller compartment.

Note that this protocol of the caller authorising transfer of ownership and the callee explicitly requesting ownership is necessary to protect from misuse of this mechanism by malicious compartments. If the callee would not explicitly request ownership of the pages containing the shared data, a malicious caller could simply omit the transfer of ownership. Similarly, we cannot allow the callee to simply take ownership without prior authorisation by the caller as otherwise it could illegitimately take ownership of arbitrary pages.

[1]When EPT is used for address translation, the VMM must work with guest-physical page numbers as the guest fully controls the mapping of guest-virtual to guest-physical addresses.

Modifications to the Shared Allocator

Since ownership, and thus write access, is defined at the granularity of pages, the shared allocator must provide the option to make allocations from distinct pages. Otherwise, shared data that happens to be allocated on the same page could not be used independently. Therefore, the allocator is extended such that any allocation request can optionally include a tag. A tag is simply an integer and allocation requests with different tags are guaranteed to be satisfied from non-overlapping page ranges. This guarantees that allocation units corresponding to different tags can be used independently.

4.6 Summary

In this chapter, we designed a VM-based isolation mechanism that supports the threat model previously introduced in Section 3.1. We described how VM/EPT call gates implement cross-compartment function calls on top of an inter-VM IPC mechanism. We outlined all components necessary for implementing cross-compartment function calls, most notably the RPC server, RPC threads, and private communication channels between each pair of compartments. Finally, we complemented the thread model defined in Section 3.1 by a detailed description of the TCB and addressed the problem of race conditions on shared memory.

Chapter 5

Implementation

In the previous chapter, we presented the design of our VM/EPT isolation mechanism. In this chapter, we give details on our prototype implementation and its integration into the FlexOS [58] framework. We note that, while our design is oblivious to the underlying hardware architecture, our prototype implementation is specific to the x86-64 architecture. We start with explaining the implementation of VM/EPT call gates and the private communication channels between compartments in Section 5.1. Next, we give details about the RPC server in Section 5.2 and the RPC threads in Section 5.3. In Section 5.4, we explain optimisations we implemented and how those impact our original design. Finally, we describe the most important modifications to the FlexOS toolchain that were required to instantiate compartmentalisation configurations based on the VM/EPT mechanism before summarising in Section 5.6.

5.1 The VM/EPT Call Gates

Call gates allow FlexOS the flexible instantiation of compartmentalisation configurations by replacing them with the appropriate code, at build-time, according to the compartmentalisation configuration and chosen isolation backend. This entails replacing gates between libraries in the same compartment with simple function calls while implementing cross-compartment function calls according to the selected isolation mechanism. This source-to-source transformation is performed by the FlexOS toolchain with the help of the Coccinelle [1] tool. Listing 5.1 shows the general form of FlexOS call gates before replacement with backend-specific gates for cross-compartment calls by the toolchain. The `lib_to` parameter indicates the name of the library in which the target function `func` is located. In the second form the return value is assigned to the variable `ret` passed to the gate macro.

```
flexos_gate(lib_to, func, ...);
flexos_gate_r(lib_to, ret, func, ...);
```

Listing 5.1: Generic FlexOS call gates before instantiation.

For the VM/EPT isolation backend cross-compartment function calls are implemented by the two macros shown in Listing 5.2. The parameters comp_from and comp_to indicate the IDs of the caller and callee compartment respectively which are automatically inserted by the toolchain. The func_id parameter indicates the ID of the function to be called which is automatically derived during the build process.

```
flexos_vmept_gate(comp_from, comp_to, func_id, ...);
flexos_vmept_gate_r(comp_from, comp_to, ret, func_id, ...);
```

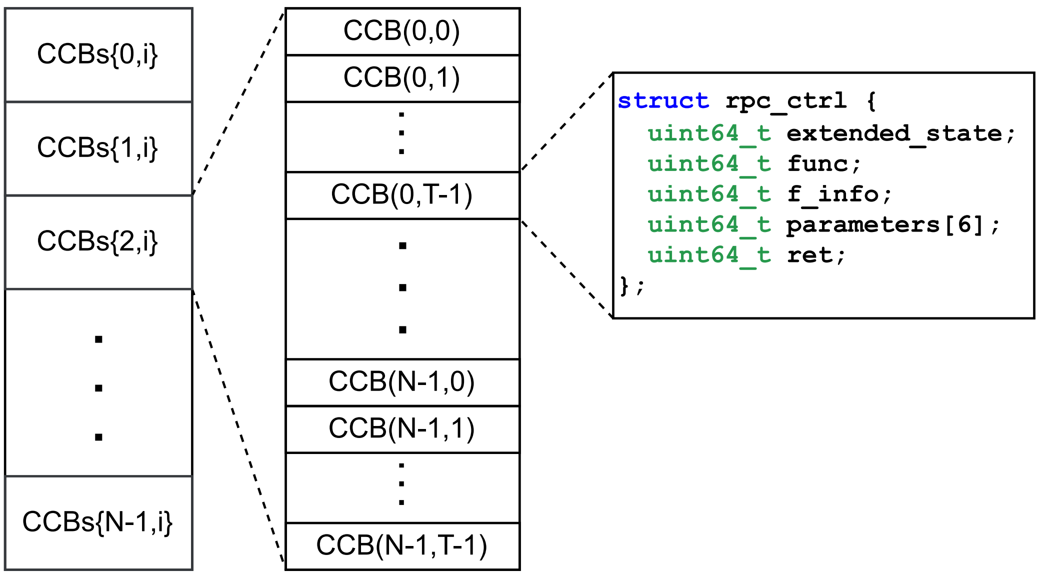
Listing 5.2: VM/EPT call gates after gate instantiation. Note that comp_from and comp_to are integer constants for any concrete instantiation.

The call gates take a variable number of arguments which are passed on to the function to be called. The prototype implementations of both the MPK isolation backend as well as our VM/EPT isolation backend only support up to six arguments of type INTEGER as defined by the System V AMD64 ABI [65] and a single optional return value of type INTEGER. This means that more complex types such as structs or floating point types cannot be passed or returned by value. Varidadic functions are only supported to the extent that they meet these restrictions. Integer types, including pointers, are cast to the uint64_t type by VM/EPT gates.

5.1.1 Private Communication Channels

As outlined in Section 4.2, private communication channels between each pair of compartments are central to our implementation of cross-compartment function calls. These communication channels are simply a region of memory that is shared between each pair of compartments. Figure 5.1 shows the memory layout used in the compartment with ID i when there is a total of N compartments and a maximum of T (regular) threads per compartment (note that compartment IDs and TIDs start at zero).

The channel CCBs{j,i} is used by compartment i for communication with compartment j. CCBs{i,i} is allocated to facilitate address calculations but unused since a compartment never performs cross-compartment function calls



```
struct rpc_ctrl {
  uint64_t extended_state;
  uint64_t func;
  uint64_t f_info;
  uint64_t parameters[6];
  uint64_t ret;
};
```

Figure 5.1: Memory layout used to implement private communication channels in the compartment with ID i. The total number of compartments is given by N while T indicates the maximum number of (regular) threads per compartment.

to itself. Each communication channel is organised into individual call control blocks (CCBs) which represent a struct containing all the information necessary for a cross-compartment function call.

The extended_state indicates the state of the CCB in bits [31:0], the ID of the calling compartment in bits [39:32], and the ID of the called compartment in bits [47:40]. The possible states are IDLE, indicating the CCB is inactive, CALLED, and RETURNED. The CALLED state indicates that the CCB holds valid information to request a function call whereas a state of RETURNED indicates that a call was completed. In both cases the calling and called compartments are indicated in bits [47:32]. These states are carefully set (see Sections 5.1.2 and 5.3) and subsequently checked by handling threads to ensure that partially setup CCBs cannot be used for cross-compartment function calls. This is necessary to guarantee security as otherwise spoofed messages sent by corrupted compartments could lead to the execution of cross-compartment function calls with CCBs in inconsistent states (e.g. only partially setup parameters). The func field is the unique ID of the function to be called and the f_info field indicates the number of parameters passed and whether a return value is expected. Up to six parameters are passed in the parameters fields and a return value can be passed back to the caller via the ret field.

5.1.2 The Implementation of VM/EPT Gates

VM/EPT call gates implement cross-compartment function calls according to the VM/EPT isolation mechanism. They are directly responsible for copying the arguments to the correct CCB before sending a message to the message queue of the receiving compartment's RPC server and performing a context switch to the RPC server of their own compartment. This is necessary to allow the RPC server to react to events signalled to it including a return from the cross-compartment call, nested calls back to the calling compartment, or unrelated cross-compartment calls from any other compartment.

In Section 4.1.3, we defined the abstract notion of the *origin* of a cross-compartment function call. Since this notion of origin, also referred to as rpc_index in our implementation, is important for the implementation of VM/EPT gates because it identifies the thread in the context of which a function is executed, we now show it is determined. For any *regular thread*, its rpc_index is computed as (current_comp $\ll$ MAX_THREAD_SHIFT) | tid where current_comp indicates the ID of the current compartment and tid is the TID of the thread executing the gate. The rpc_index is subsequently stored as part of the uk_thread struct, which contains general information required by a thread, such as its TID, a pointer to its stack, and an area where register values are stored on a context switch. The macro MAX_THREADS_SHIFT, taking values between 1 and 8 (default), is defined by the toolchain to make the maximum number of supported (regular) threads configurable. For any *RPC thread* the rpc_index is set by the RPC server before it is dispatched (see Section 5.2.2).

The implementation of VM/EPT gates consists of the following simple steps:

1. Retrieve the rpc_index of the current thread.
2. Locate the correct CCB. The rpc_index indicates the location of that CCB in the data structure representing the private communication channel CCBs{comp_to, current_comp} where comp_to is the ID of the callee compartment and current_comp is the current compartment's ID. Note that this is the same CCB as indicated by CCB(current_comp, tid) in Figure 5.1 with the same tid used to compute the rpc_index.
3. Set the state of the CCB to IDLE. This is necessary to prevent spoofed messages from a malicious compartment from triggering the execution of a function with arguments only partially set up by the caller.
4. Set the ID of the target function and copy the arguments to the parameters array in the CCB.

5. Set the state of the CCB to CALLED.
6. Post a message to the message queue of the callee compartment.
7. Perform a context switch to the RPC server thread [1]. The RPC server guarantees that a thread waiting for a cross-compartment call to finish is only resumed when an appropriate event is signalled by another compartment. Such an event is either a RETURN signalling the completion of the call or a nested CALL back to the original compartment that needs to be handled in the context of that thread.

5.2 The RPC Server

As outlined in Section 4.1.3, the RPC server is responsible for the communication between compartments and dispatches RPC threads in response to events signalled by other compartments. In Section 4.3, we identified three simple sub-components that are central to the functioning of the RPC server.

5.2.1 Sub-Components of the RPC Server

In the following, we describe the implementation of the three sub-components on which the RPC server relies, the *RPC thread pool*, the *RPC entry map*, and the *message queue*.

The RPC Thread Pool

The RPC thread pool is essentially a LIFO-queue of pointers to uk_thread structs representing the idle RPC threads. Whenever an RPC thread needs to be assigned to handle a request, the thread at the front of this queue is removed. Should the queue be empty, a new RPC thread is first created before being assigned to handle the request. Similarly, returning an RPC thread that became idle after completing a request for a cross-compartment function call is as simple as appending a pointer to this thread to the front of the queue of idle RPC threads. Therefore, RPC threads are never destroyed but idle RPC threads are recycled via the RPC thread pool.

The RPC Entry Map

The RPC entry map is a simple array, indexed by the rpc_index, mapping each possible rpc_index to an instance of a rpc_entry struct which is shown in

[1]In the course of optimising the RPC mechanism this was revised in order to save the overhead associated with context switches to and from the RPC server, see Section 5.4.

Listing 5.3. In the case that the rpc_index was already assigned a handling RPC thread, this is indicated by the thread pointer, otherwise the thread pointer is NULL. The activations counter indicates the number of active calls on the stack of that thread.

```
struct vmept_rpc_entry {
  struct uk_thread *thread;
  int activations;
};
```

Listing 5.3: An entry of the RPC entry map. The thread pointer indicates the assigned RPC thread and the activations counter keeps track of the active function calls in the context of this RPC thread.

The Message Queue

The message queue associated with each compartment's RPC server is a simple ring buffer which can hold a fixed number of messages. Each message consists of two bytes and contains the rpc_index corresponding to the origin of the cross-compartment function call the message refers to as well as the ID of the compartment from which the message was sent. The implementation of the ring buffer allows the RPC server to consume messages from its message queue without the need for synchronisation with the RPC servers of other compartments filling the queue. However, multiple RPC servers posting messages to the same message queue must synchronise their access to it by means of obtaining a write lock before inserting the message into the ring buffer and advancing the write pointer.

5.2.2 The Implementation of the RPC Server

With all the sub-components implemented, the RPC server can be described in terms of simple operations on those components. The RPC server thread continuously polls for the arrival of new messages in its message queue. If there are no messages to retrieve from its message queue, the RPC server thread calls uk_sched_yield to allow other regular threads in this compartment to be scheduled. Note that regular threads that are waiting for cross-compartment calls to finish are temporarily given the status of an RPC thread and are not considered for scheduling as they cannot make progress. When the RPC server retrieves a new message from its message queue, it extracts the rpc_index from that message and looks up the corresponding rpc_entry in the RPC entry map. In the case that this RPC entry indicates that no handling RPC thread is assigned, it removes an idle RPC thread from the RPC thread pool and assigns it to this RPC entry. In both cases the activations counter of the RPC entry is incremented and the rpc_index and the ID of the compartment that sent the message are saved to fields which were added to the uk_thread struct specifically for the purpose of conveying this information to the handling RPC thread. Finally, a context switch to the RPC thread is performed which handles the event at hand, either of type CALL or RETURN, depending on the extended_state set in the corresponding CCB.

5.3 The RPC Threads

As described in the previous section, a compartment's RPC server is responsible for assigning and dispatching RPC threads in response to incoming messages. It is then the responsibility of the RPC thread to execute the requested action. Therefore, the RPC thread uses the information relayed to it by the RPC server, specifically the ID of the compartment that sent the message to the RPC server and the rpc_index, to obtain the relevant CCB. The compartment ID can be understood as selecting the communication channel (CCBs{comp_id, i} in Figure 5.1) and the rpc_index subsequently selects the CCB within that channel. It must first be verified that the state of the CCB is CALLED in order to prevent spoofed messages from a malicious compartment to trigger the execution of a function with arguments only partially set up by the caller. The function ID is then extracted form that CCB and translated to the address of the corresponding function via a lookup table constructed during the build process (see Section 5.5.1). The arguments are then copied to the registers as required by the System V AMD64 ABI [65] calling convention before the function is called. After a return from that function, the return value, if available, is copied back to the corresponding field in the CCB before the state is set to RETURNED and a corresponding message is sent back to the compartment that requested the call. Finally, the activations counter in the RPC entry to which this thread is assigned is decremented, returning the RPC thread back to the pool of idle RPC threads should it reach zero, before a context switch back to the RPC server thread is performed.

In the following, we want to detail the step of calling the function with the arguments retrieved from the corresponding CCB. For that purpose we first introduce the relevant parts of the System V AMD64 ABI calling convention.

The System V AMD64 ABI Calling Convention

A calling convention defines how arguments are passed to functions. We follow a restricted version of the System V AMD64 ABI calling convention. We restrict parameters and return values to the INTEGER class of arguments which is defined as any integral type that fits into a single general-purpose register [65]. In terms of C types this means that parameters and return values must be integers comprised of at most eight bytes. The first six arguments of that type are passed to the callee via the general-purpose registers rdi, rsi, rdx, rcx, r8, and r9 while a potential return value of type INTEGER is passed back to the caller in rax.

Executing the Function Call

In order to call the requested function with the provided arguments we use the inline assembly feature [15] of the GCC compiler. This allows us to copy the given parameters into the registers as required by the System V AMD64 ABI calling convention as well as copying back the return value afterwards. Listing 5.4 shows an excerpt from the code that initiates the call to the requested function after translating the function ID indicated by the func field of the CCB (pointed to by ctrl) to the address of the corresponding function in the callee compartment.

5.4 Optimisations

In the previous sections, we described our prototype implementation of the design of the VM/EPT isolation mechanism as presented in Chapter 4. Early experiments showed that the strict separation of the RPC server in a dedicated thread, requiring a context switch to this thread whenever the RPC server needs to become active, is a source of significant overhead. To alleviate this we integrated the functionality of the RPC server into each RPC thread in order to eliminate the need for excessive context switches. This lead to a reduction of cross-compartment call latency upwards of 50% compared to the initial implementation.

Another optimisation is concerned with the dynamic assignment of RPC threads to handle incoming requests. Instead of returning idle RPC threads to the RPC thread pool we keep them assigned to their respective rpc_index. This effectively assigns each rpc_index, or origin in the more abstract terminology used to describe our initial design (see Section 4.1.3), a dedicated

```c
register uint64_t ret asm("rax") =
  (uint64_t) translate_func(ctrl->func);
switch (argc) {
  case 0: /* left out for brevity  */
  case 1: /* left out for brevity  */
  case 2:
    asm volatile (
      "movq 0(%[args]), %%rdi      \n\t"
      "movq 8(%[args]), %%rsi      \n\t"
      "call *%[ret]                \n\t"
       : /* outputs */
       [ret] "+&r" (ret)
       : /* inputs */
        [args] "r" (ctrl->parameters)
       : /* clobbers */
       "rdi", "rsi", "rdx", "rcx", "r8", "r9",
       "r10", "r11", "memory"
     );
    break;
  case 3: /* left out for brevity  */
  case 4: /* left out for brevity  */
  case 5: /* left out for brevity  */
  case 6: /* left out for brevity  */
  default: UK_CRASH("Invalid number of arguments\n");
}
```

Listing 5.4: Excerpt of the inline assembly code used to call the requested function. Note that the variable ret, placed in the rax register, holds the address of the function before it is called and also holds the potential return value after the call completes.

RPC thread. In early experiments we experienced a reduction in cross-compartment call latency of roughly 10% due to this optimisation.

5.5 Modifications to the FlexOS Toolchain

In order to integrate our VM/EPT isolation mechanism into the FlexOS framework we need to enable the toolchain to instantiate compartmentalisation configurations based on this isolation mechanism. The most significant difference between the VM/EPT isolation mechanism and the existing MPK backend is the separation of address spaces. While the MPK isolation backend provides intra-address space isolation, the VM/EPT separates compartments into different VMs. Therefore, we need to build one VM image per compartment.

Furthermore, we identify functions requested via cross-compartment calls by IDs in order to provide a form of inter-compartment CFI. This requires the toolchain to keep track of all functions that are called via gates, to automatically generate IDs for these functions, and to substitute these IDs for the function names in the VM/EPT gates.

5.5.1 Generating Function IDs

The FlexOS toolchain uses the Coccinelle [1] tool to replace generic call gates with call gates specific to the employed isolation mechanism (see Section 5.1). We use this tool to automatically generate IDs to identify functions in cross-compartment calls and substitute these IDs for the function names in the VM/EPT gates. For this purpose, we modified the rules responsible for the replacements of call gates to incorporate this additional functionality.

```
@gatereplacer_return@
identifier func;
expression list EL;
expression ret, lname;
fresh identifier func_id = "_RPC_ID_" ## func;
@@
- flexos_gate_r({{ lib_dest_name }}, ret, func, EL);
+ flexos_vmept_gate_r({{ comp_cur }}, {{ comp_dest }},
    ret, func_id, EL);

@script:python@
func_name << gatereplacer_return.func;
@@
add_func(func_name, {{ comp_dest }})
```

Listing 5.5: Example of a rule template for gate replacements according to VM/EPT isolation mechanism.

Listing 5.5 shows a (simplified) example of a rule template used to perform replacements according to the VM/EPT isolation backend. Placeholders of the form {{ var }} are replaced with the appropriate values before the rule is applied by the Coccinelle tool. Specifically, the name of the destination library is substituted for {{ lib_dest_name }} and the IDs of the current compartment and the destination compartment are substituted for the placeholders {{ comp_cur }} and {{ comp_dest }} respectively. The identifier func is matched by Coccinelle to the name of the function called via the gate. From that identifier we create a name that is later defined as a macro to hold the integer representing the function ID. We do this by adding the prefix

```c
#ifndef VMEPT_RPC_ID_H
#define VMEPT_RPC_ID_H
#define _RPC_ID_write  0
#define _RPC_ID_close  1
#define _RPC_ID_fchown 2
#define _RPC_ID_fcntl  3
/* [...]  */
#define VMEPT_ID_CNT 47
#endif VMEPT_RPC_ID_H
```

Listing 5.6: Automatically generated function IDs.

```c
#ifndef VMEPT_ADDR_TABLE_H
#define VMEPT_ADDR_TABLE_H
static const void*
  vmept_addr_table[] = {
     0x11df90, /* write */
     0x11d5e0, /* close */
     0x120c10, /* fchown */
     0x11f760, /* fcntl */
     /* [...]  */
  };
#endif VMEPT_ADDR_TABLE_H
```

Listing 5.7: Automatically generated address table.

_RPC_ID_ to the function name as show in line 5 of Listing 5.5. Whenever this rule is applied by Coccinelle, the Python script indicated by the @script:python@ tag in line 10 is executed. The variable func_name receives the name of the function used in the gate to which the replacement rule is applied which is subsequently added to the list of all functions called via VM/EPT gate as shown in line 13. From this list of function names we automatically generate a C header file that defines the IDs for all these functions. Simultaneously, we generate another header file that defines an array indexed by these function IDs which is later used by the target compartment to map a function ID to the address of the corresponding function. Simplified examples of these automatically generated header files are shown in Listings 5.6 and 5.7.

When first building the address table during the rewriting of the gates, we cannot determine the addresses of these functions. Therefore, we initialise the address table shown in Listing 5.7 with NULL pointers during the rewriting phase. After compilation we extract the real addresses from the binaries via the readelf [49] tool and insert the correct addresses into the address table before recompiling and relinking the VM images.

These address tables specific to each compartment are then used to translate function IDs to the corresponding addresses by the translate_func (inline) function shown in line 2 of Listing 5.4 before executing the function call.

5.6 Summary

In this chapter, we gave details about the implementation of the VM/EPT isolation mechanism. We explained how VM/EPT gates implement cross-compartment function calls on top of private communication channels between compartments. We gave implementation details of the RPC server and RPC threads that are responsible for handling cross-compartment function calls and explained how these components are used by VM/EPT gates to perform cross-compartment function calls. Next, we explained where we deviated from our original design for optimisation purposes, most notably by integrating the RPC server into RPC threads to reduce the number of context switches. Finally, we described the modifications to the toolchain that were required to integrate the VM/EPT isolation mechanism into the FlexOS framework, including the automatic generation of function IDs and the corresponding table used to translate these IDs back to addresses.

Chapter 6

Evaluation

In the following chapter, we evaluate the VM/EPT isolation backend. We start by explaining the evaluation methodology and evaluation setup in Section 6.1. Then we proceed with microbenchmarks, measuring the latency of cross-compartment calls in Section 6.2 and compare the results with the cross-compartment call latencies of the MPK backend as well as Linux system call latencies. Following this, we proceed with macrobenchmarks. In Section 6.3, we measure network throughput performance in a scenario where the application is isolated from the network stack. Next, we evaluate the performance of the VM/EPT backend on file system intensive workloads with SQLite in Section 6.4. Finally, in Section 6.5, we discuss security limitations of the current implementation of the VM/EPT isolation backend. We show that these limitations are not fundamental but due to the prototype nature of the VM/EPT implementation as well as the FlexOS framework as a whole and discuss how these limitations can be overcome before summarising the results of the evaluation in Section 6.6.

6.1 Methodology and Evaluation Setup

We evaluate the runtime performance of FlexOS with the VM/EPT isolation backend and compare the performance overhead with that of the much more lightweight MPK isolation backend as well as Unikraft without any isolation. We show that, while the strong isolation provided by the VM/EPT backend comes at the price of high latency for cross-compartment calls, reasonable performance is still achieved, especially when the frequency of cross-compartment calls is moderate.

All measurements are taken on a Debian 10 (Buster) system (kernel version 4.19.0-8) equipped with an Intel Xeon Silver 4110 CPU with a base frequency of 2.1GHz, hyperthreading disabled, and 32GB of RAM. The experiments are run inside of Docker containers to allow for easy reproducibility (see Appendix A). To minimise the overhead introduced by Docker, we run all containers with seccomp [50] disabled. We use the isolcpus [5] kernel command line parameter to isolate three cores from the kernel scheduler. The taskset [53] command is used to pin the QEMU guest processes to the isolated cores to minimise the disturbance of our measurements. The current FlexOS version is based on Unikraft 0.5. The Unikraft version used for comparison of results is also 0.5.

6.2 Microbenchmarks

To measure gate latency, regardless of the backend, we build an instance of FlexOS with two compartments. The first compartment contains the benchmarking code that repeatedly calls an empty function in the second one. In each case, including function calls and Linux system calls, we measure the elapsed CPU cycles with the time-stamp counter (TSC). Therefore, we first give an introduction to time measurements with the TSC and describe the exact methodology we use to obtain precise measurements.

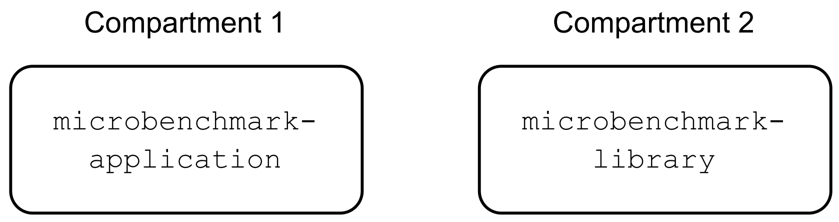


Figure 6.1: Compartmentalisation configuration for the microbenchmarks. The microbenchmark-library contains the function that is called. The measurements are performed in the microbenchmark-application.

6.2.1 Measurements with the TSC

The Intel64 and IA-32 architectures, starting with Pentium processors, provide a TSC [46, Vol. 3] that allows for high precision measurements of elapsed time. The read time-stamp counter (RDTSC) instruction can be used to obtain the current value of the TSC. However, this instruction is not serialising [46, Vol. 3], which means that it does not guarantee that modifications to flags, registers, or memory by previous instructions are completed before

```c
static inline uint64_t bench_start()
    __attribute__ ((always_inline)) {
    uint32_t tsc_low, tsc_high;
    asm volatile(
        "CPUID          \n\t"                        // serialise
        "RDTSC          \n\t"                        // read TSC
        "MOV %%edx, %0\n\t"
        "MOV %%eax, %1\n\t"
        : "=r" (tsc_high), "=r" (tsc_low)      // outputs
        : : "%rax", "%rbx", "%rcx", "%rdx");  // clobbers
    return ((uint64_t) tsc_high << 32) | tsc_low;
}
```

Listing 6.1: Code to read the TSC before executing the code under measurement.

the TSC is read. In order to prevent overlapping execution of instructions of the code under measurement and previous instructions, we insert the serialising CPU identification (CPUID) instruction immediately before obtaining the initial TSC value in each iteration of the benchmark loop. The resulting code is shown in Listing 6.1.

```c
static inline uint64_t bench_end()
__attribute__ ((always_inline)) {
uint32_t tsc_low, tsc_high;
asm volatile(
"RDTSCP         \n\t"                        // read TSC
"MOV %%edx, %0\n\t"
"MOV %%eax, %1\n\t"
"CPUID          \n\t"                        // serialise
: "=r" (tsc_high), "=r" (tsc_low)      // outputs
: : "%rax", "%rbx", "%rcx", "%rdx");  // clobbers
return ((uint64_t) tsc_high << 32) | tsc_low;
}
```

Listing 6.2: Code to read the TSC after executing the code under measurement.

While this allows us to obtain precise measurements of the TSC before executing the code to benchmark, it is recommended [75] to use a different method for reading the TSC after the code under measurement completes execution in order to reduce the impact of the serialising CPUID instruction on the measurements. For that purpose, we use the read time-stamp counter and processor ID (RDTSCP) instruction. While this instruction is not serialising, it waits until all previous instructions have executed and all previous loads are globally visible [46, Vol. 2B]. We then execute the CPUID instruction for serialisation after the RDTSCP instruction to prevent overlapping execution of any subsequent instructions with parts of instructions of the code under measurement. The resulting code is shown in Listing 6.2.

6.2.2 Gate Latency Measurements

Figure 6.2 shows the latencies of MPK and VM/EPT gates, Linux system calls, and a simple function call. The measurements are taken according to the methodology discussed previously. The results shown in Figure 6.2 are median values taken over 100.000 measurements. The functions called via MPK and VM/EPT gates as well as via direct function calls do not perform any computation and are of type `void func(void)` [1]. For the Linux system call the `syscall` [51] function was used with an invalid system call number to measure only the base overhead of the system call mechanism.

In Figure 6.2 we can see that the VM/EPT gates have about 8.3 times increased latency compared to the much more lightweight MPK gates, while the overhead of a Linux system call is about 56% of that of an cross-compartment call with VM/EPT.

The high overhead incurred by VM/EPT gates comes from two sources. First, the inter-VM IPC mechanism requires additional logic to ensure correct control flow between compartments. Incoming calls must be dispatched to the appropriate handler thread and nested calls must be handled correctly. Second, communication between compartments is implemented via shared memory. This includes the message queues in each compartment as well as the individual CCB for each thread. On a call, arguments must be copied to the CCB of the corresponding thread, the function identifier must be set, and the state must be updated. The receiving compartment then retrieves the message from its message queue before dispatching the handler thread. Then the function identifier is checked and translated to the address of the corresponding function and the arguments are copied from the CCB to the correct registers (as required by the System V AMD64 ABI calling convention), before the call is executed. Finally, after the call completes, potential return values are copied back to the CCB, the return from the call is indicated by setting the state of the CCB appropriately, and a message is posted to the message queue of the calling compartment.

The access to these shared memory locations contributes a large part

[1] In the case of VM/EPT gates, measurements with different numbers of arguments were performed but are not shown because no statistically relevant impact could be observed.

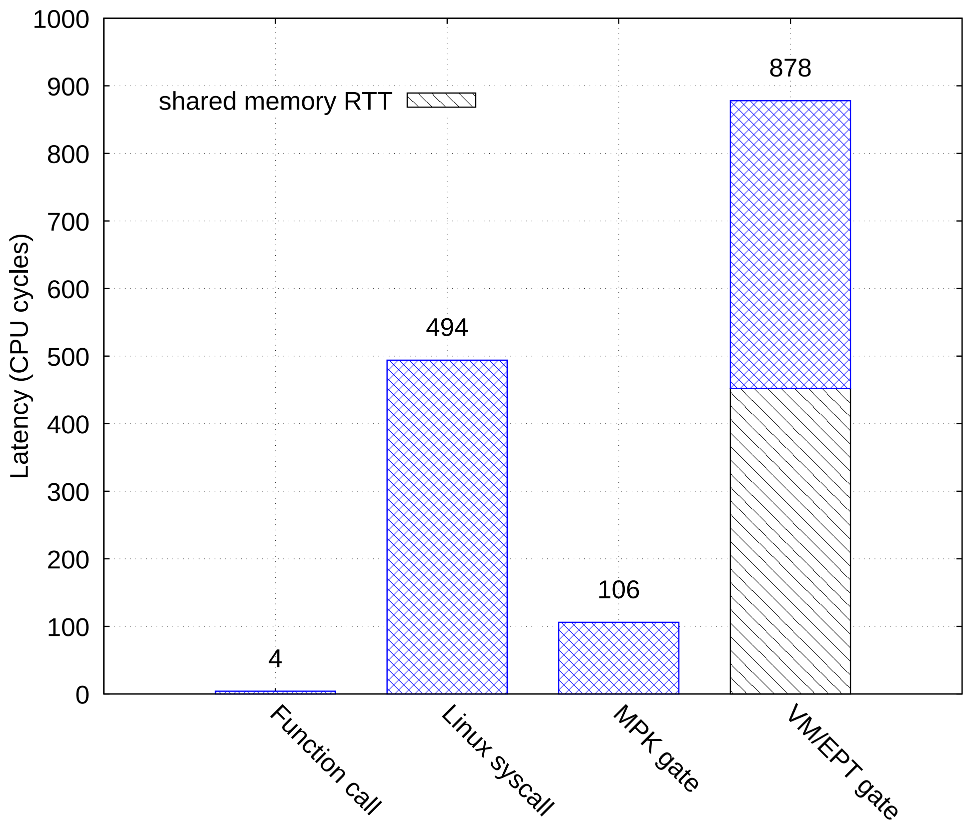


Figure 6.2: Latency measurements of function calls, Linux system calls, FlexOS MPK gates, and FlexOS VM/EPT gates. VM/EPT gates incur about 8.3 times increased latency compared to MPK gates and roughly 1.8 times increased latency compared to Linux system calls. At least 51% of the VM/EPT gate latency stems from frequent communication via shared memory.

of the overhead associated with VM/EPT. To determine the base overhead that is incurred only from setting the state part of the CCB, we measured the execution time of code that sets a shared integer variable in one compartment from an `IDLE` state (0) to `CALLED` (1) and waits for the second compartment to set it to `RETURNED` (2). The second compartment checks the value of the shared state variable in a loop and sets it to `RETURNED` as soon as it observes that it has the value of `CALLED`. The code measured to determine the minimum overhead is shown in Listing 6.3 and executed in the first compartment. The state variable is reset to `IDLE` after each iteration of the measurement loop (not shown in Listing 6.3). The second compartment executes the code shown in Listing 6.4.

The results of these measurements are indicated by the shared memory round-trip time (RTT) value in Figure 6.2. From these measurements we can conclude that the shared memory accesses contribute at least 51% of the total overhead of VM/EPT gates. Note that this is only a lower bound for the time spent on shared memory accesses because over the course of an VM/EPT gate call there are more accesses to shared memory. For example, the state part of the CCB is set multiple times during the execution of a single VM/EPT cross-compartment call (see Sections 5.1.2, 5.2.2 and 5.3).

```
*state = CALLED;
while (*state != RETURNED) {
    asm volatile("pause" ::: "memory");
}
```

Listing 6.3: The code executed and measured in the first compartment to determine the minimum overhead due to accesses to shared memory locations. The `pause` [46, Vol. 2B] instruction is a hint to the processor that this is a spin-wait loop and allows for more efficient execution.

```
while (1) {
    while (*state != CALLED) {
        asm volatile("pause" ::: "memory");
    }
    *state = RETURNED;
}
```

Listing 6.4: The code executed in the second compartment. This simulates the access pattern to the state part of the CCB during the execution of a VM/EPT gate call.

6.2.3 Possibility for Optimisation of VM/EPT Gates

Since we identified the frequent writes to the state part of a CCB as a source of significant overhead, we should consider optimising this aspect of the implementation of VM/EPT gates. Therefore, we first take a closer look at the MESI cache coherence protocol to understand why frequent writes from different compartments to the same memory location degrade performance. Then we see how security considerations hamper the optimisation of this particular memory access pattern.

The MESI Cache Coherence Protocol

Processors implementing the Intel64 architecture use the MESI cache coherence protocol to maintain consistency between the caches that are exclusive to individual cores [46, Vol. 3A]. The MESI protocol associates each cache line with one of four states: modified (M), exclusive (E), shared (S), or invalid (I) [31, p. 299-301]. The invalid state indicates that the cache line does not hold consistent data, resulting in a cache miss on access. A cache line in the exclusive state is only present in the cache of one core. The shared state indicates that the same cache line is replicated across multiple caches and all copies are clean, guaranteeing that they hold the same value. Finally, the modified state indicates that a cache line is valid but dirty, meaning that copies in other caches must be invalidated on transition to this state. The state transitions according to the MESI protocol are depicted in Figure 6.3. Transitions in response to processor read (PrRd) or write (PrWr) accesses are shown in blue. The transitions shown in red happen in response to observed bus activity (accesses of other processors to their caches), BusRd indicates a read access, and BusRdX a write access.

Cache Behaviour of VM/EPT Gates

Now we can trace what happens to the cache line of the state part of the CCB over the course of a VM/EPT gate call. For the sake of simplicity we assume that the two interacting compartments execute on different processor cores, subsequently called P1 and P2. We also assume that initially the cache line in P1's cache is present in the modified state and invalidated in the cache of P2. When the VM/EPT gate call is issued from compartment one, it sets the state twice, first to IDLE and then to CALLED after the CCB is set up correctly, thus obtaining exclusive ownership of the cache line which is now in the modified state. After the message to initiate the call is posted to the message queue of compartment two, the handling thread first checks the state in the CCB and then sets it to IDLE. Therefore, the cache line is invalidated in P1's cache and now is present in the modified state in the cache of P2. When the function requested by the gate call returns, the state is updated to RETURNED by P2 and a message is posted to the message queue of compartment one. When the handling thread in compartment one resumes execution, it first checks the state (to determine whether a return or a nested call is at hand) and then sets the state back to IDLE. Thus, the cache line is invalidated in P2's cache (after briefly transitioning to the shared state) and present in the modified state in P1's cache, which is the initial state of the caches we assumed earlier. Therefore, we observe that the

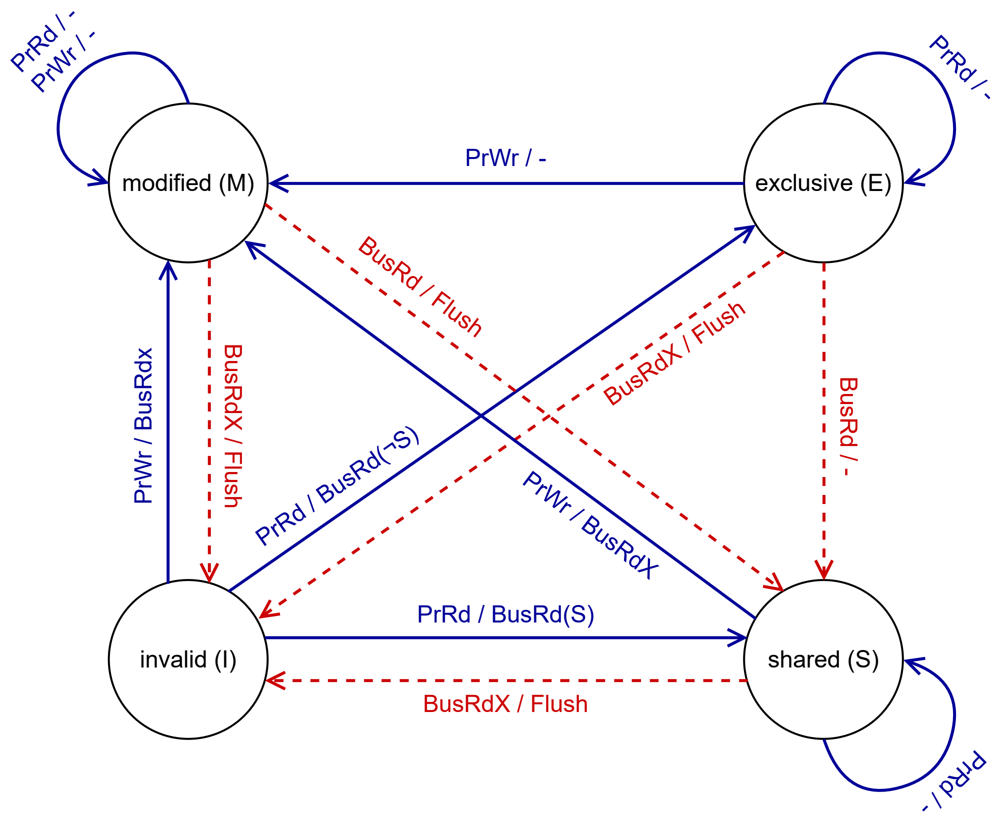


Figure 6.3: State transitions according to the MESI cache coherence protocol.

cache line containing the state part of the CCB is invalidated and refilled frequently, which degrades performance and explains the overhead incurred by shared memory accesses.

Performance vs. Security Considerations

The observation that the memory access pattern of VM/EPT gates has adverse effects on performance leads to the question of whether this can be optimised. Unfortunately, security considerations make obvious optimisations, such as including the state in the messages sent to message queues, impossible. This is because we have to ensure that VM/EPT gate calls can only be processed when the communication partner has fully set up the CCB. Otherwise, since the message queues are shared between all compartments, a malicious third compartment could post spoofed messages in order to cause a cross-compartment call with the CCB in an inconsistent state (see

Section 5.1.2). Note that the CCBs are only shared between each pair of compartments thus forming private communication channels. By carefully setting the state part of the CCB we can prevent this kind of attack, although at the cost of higher overhead as discussed above.

6.3 Network Throughput (iPerf)

In order to test the network throughput, we build FlexOS in a configuration where the application receiving data over the network is isolated from the rest of the system, including the network stack. This configuration is used both for the VM/EPT and MPK backend. For comparison we built the same application with Unikraft without any isolation. All images are compiled with optimisation enabled (-O2). The LwIP network stack is configured with a TCP window size of 64KB and window scaling is disabled. We use the iPerf [3] network performance measurement tool in client mode to connect to the server application and measure the TCP throughput.

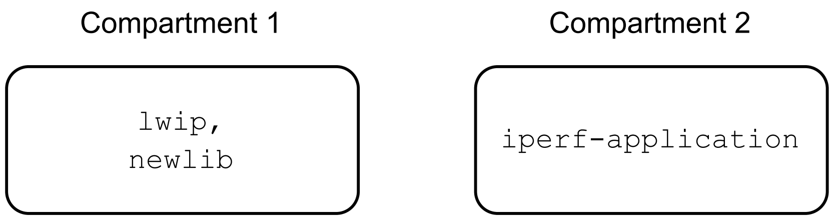


Figure 6.4: Compartmentalisation configuration for the iPerf benchmark. The iperf-application which receives data over the network is isolated from the rest of the system, including the network stack (lwip).

The results of the measurements, averaged over five measurement periods of ten seconds for each receive buffer size, are shown in Figure 6.5. We can see that the overhead for cross-compartment calls degrades the performance in both cases where isolation is employed. Since VM/EPT gates introduce significantly more overhead than MPK gates, the throughput for small buffer sizes is correspondingly lower. For buffer sizes between 16 and 64 bytes, the TCP throughput with the VM/EPT isolation backend is about 3.6 times lower than with the MPK backend and about 4.7 times lower compared to Unikraft. Also, the peak throughput of 2.95Gb/s is reached later with the VM/EPT isolation backend because of the higher gate latency, but is close to the peak throughput of 3.1Gb/s achieved by the MPK backend and Unikraft.

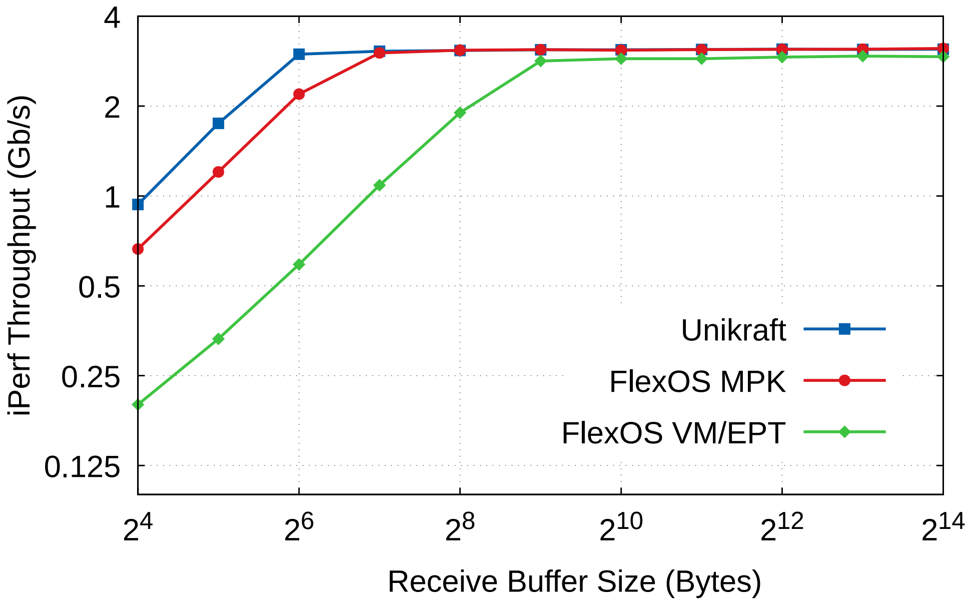


Figure 6.5: Network throughput measurements for Unikraft, FlexOS with the MPK backend, and FlexOS with the VM/EPT backend with different sizes of the receive buffer. For buffer sizes between 16 and 64 bytes throughput with the VM/EPT backend is about 3.6 times lower compared to the MPK backend and 4.7 times lower compared to Unikraft without isolation. For larger buffer sizes the throughput with the VM/EPT backend converges to a similar maximum.

6.4 File System Intensive Workloads (SQLite)

In order to evaluate the performance of the VM/EPT isolation backend on file system intensive workloads, we isolate the file system from an application that performs operations on an SQLite [12] database. All images are built with compiler optimisation enabled (`-O2`). All systems use a RAM-based file system and the SQLite version 3.30.1. We measure the time it takes to complete 5000 `INSERT` operations.

Figure 6.7 shows the results of the measurement, averaged over ten measurement runs each. We can see that Unikraft, providing no isolation, performs by far the best with approximately half the runtime of FlexOS with the MPK isolation backend. FlexOS with the VM/EPT backend requires almost three times the runtime compared to Unikraft and roughly 1.5 times the runtime measured with the MPK backend.

The performance of FlexOS with the VM/EPT backend is similar to

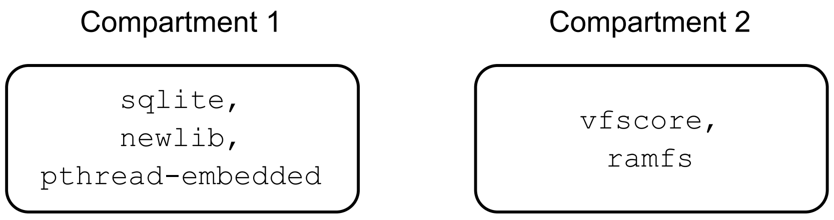


Figure 6.6: Compartmentalisation configuration for the SQLite benchmark. The sqlite application is isolated from the file system.

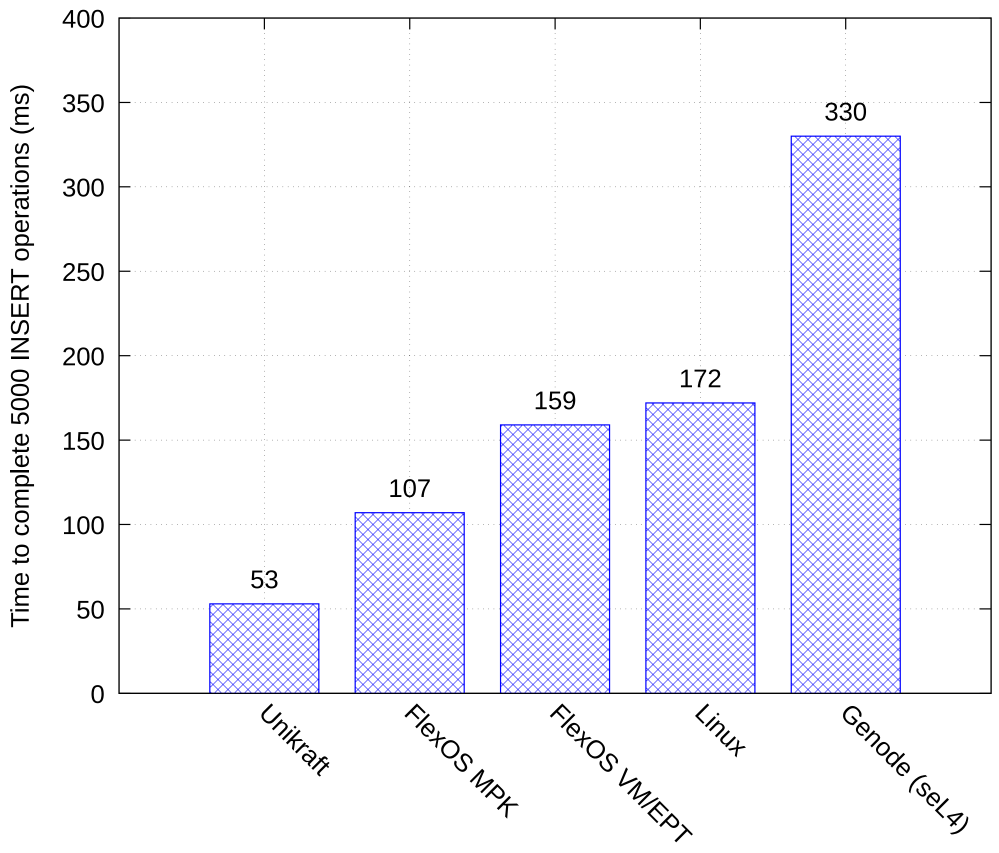


Figure 6.7: Time to complete 5000 INSERT operations into an SQLite database on Unikraft, FlexOS with the MPK backend, FlexOS with the VM/EPT backend, Linux, and Genode with the seL4 microkernel. FlexOS with the VM/EPT backend is about 3 times slower than Unikraft and about 50% slower than FlexOS with the MPK backend. The performance achieved by the VM/EPT backend is comparable to that of native Linux and about twice that of Genode with the seL4 microkernel.

that of a native Linux system, even comparing slightly favourable in our measurements. Compared to Genode [36] with the seL4 [54] microkernel, the runtime required with VM/EPT isolation is less than 50%. Therefore, we can conclude that in this measurement the VM/EPT isolation mechanism provides similar or better performance as systems relying on page table based isolation, i.e. Linux and Genode with the seL4 microkernel.

6.5 Discussion of Security

After evaluating the performance of the VM/EPT isolation backend, we now want to discuss various security aspects, address shortcomings of the implementation, and show that they are not fundamental in nature.

6.5.1 Private Communication between Compartments

As described in Section 4.2, the information required for cross-compartment calls is stored in a section of memory that is shared between each pair of compartments. Therefore, a corrupted compartment cannot manipulate this information for calls in which it is not involved. However, to enforce this pairwise sharing, our current prototype implementation sets up the memory mapping in each compartment accordingly via the uk_page_map function provided by Unikraft. This means that a corrupted compartment can easily revert this protection by changing the mapping to gain access to the memory regions shared between other compartments. However, this is not a fundamental limitation since techniques similar to the sealing of unikernels [67] can be employed. After initialisation, a hypercall is issued that instructs the VMM to prevent further page table modifications. This approach of dropping the privilege to modify the memory mapping after initialisation follows the same idea as the seccomp[2] system call in Linux, applied to the page table instead of the system call interface.

While this can be implemented when shadow page tables are used for translation of guest-virtual to host-physical addresses, a different solution is needed when the extended page tables (EPT) mechanism is used for address translation. This is because with EPT the guest has full control over the mapping from guest-virtual to guest-physical addresses. However, the mapping from guest-physical to host-physical addresses is still controlled by the VMM. Therefore, we only have to ensure that the extended page table does

[2]When executed, the seccomp [50] system call prevents the process from executing any further system calls with the exception of read or write on open file descriptors, exit, or sigreturn (in the most restrictive configuration).

not provide a valid mapping to pages that must not be accessible to the guest after system initialisation.

6.5.2 Shared Memory Allocation

The VM/EPT isolation backend uses one shared allocator per compartment, each of which allocates memory from a different part of a memory section that is shared between all compartments. Currently, the ukallocbuddy buddy allocator of Unikraft is used for the shared allocators. With this allocator metadata is stored in the same memory region from which allocations are made which means that this metadata is also accessible to all compartments. Therefore, a malicious compartment could corrupt the shared allocator of another compartment.

A solution to this problem is to use a special allocator implementation that segregates metadata from the memory region from which allocations are made. The metadata must then be kept on memory pages that are private to the respective compartment.

6.5.3 Addressing Race Conditions on Shared Memory

In Section 4.5.2, we outlined a general solution to the problem of race conditions on shared memory. Our prototype implementation of the VM/EPT isolation mechanism did not address this problem since we first wanted to demonstrate the feasibility of VM-based isolation before committing to the significant engineering effort associated with the proposed solution. We expect that implementing the solution we proposed comes at a significant runtime overhead for shared data involved in cross-compartment function calls. Unfortunately, we could not investigate the suitability of the proposed solution to the problem of race conditions on shared memory in terms of additional runtime overhead and therefore have to leave this to future work.

6.6 Summary

In this chapter we saw that, although the VM/EPT isolation mechanism introduces high overhead on cross-compartment calls compared to more lightweight MPK isolation, it still performs similar or better to page table based isolation for file system intensive workloads. The network throughput benchmark confirmed that the higher overhead for cross-compartment calls leads to a degradation of performance roughly proportional to the frequency of compartment crossings. We also discussed practical limitations of the current prototype implementation of the VM/EPT isolation backend and how these can be overcome.

Chapter 7

Conclusion

In this thesis, we explored the use of high-guarantee isolation mechanisms for compartmentalisation at the granularity of a library. For this purpose, we performed a systematic analysis of existing isolation mechanisms, motivated for VM-based isolation, before designing, implementing, and evaluating a prototype in the FlexOS framework.

The evaluation of our prototype implementation showed that, while the radical approach of VM-based isolation comes at the cost of high latency for cross-compartment function calls, adequate performance is still obtained in scenarios where the frequency of cross-compartment function calls is moderate. We can therefore conclude that the cost of VM-based isolation is high but not prohibitive in scenarios where security is the primary concern.

7.1 Future Work

Our prototype implementation of the VM/EPT isolation suffers from several practical limitations. The RPC servers that are at the heart of the RPC mechanism responsible for the implementation of cross-compartment function calls rely on polling for receiving messages from their respective message queues. This choice was made purely for ease of implementation and is not practical in real-world scenarios. Not only is this inefficient from the perspective of power consumption but it also precludes resource consolidation as the virtual CPUs of all VMs are kept busy at any time.

Furthermore, there are several security concerns that are not addressed by our prototype implementation. This includes the fact that the private communication channels that are an integral part of our RPC mechanism are only secured by the memory mapping set up by each individual compartment. There is no mechanism in place to prevent malicious compartments from changing this mapping to get access to the private communication channels shared between other compartments. Fixing this problem requires an extension of the VMM to enable a form of sealing the memory mapping as outlined in Section 6.5.1. Also the shared allocators used by the VM/EPT compartments presents an attack surface. This is because they rely on the ukallocbuddy allocator implementation which allocates the metadata in the same memory region from which allocations are made, in this case the memory region shared between all compartments. Therefore, this metadata is accessible by any malicious compartments and presents an obvious attack surface. The solution to this problem is, as already outlined in Section 6.5.2, to use shared allocators that segregate metadata from the memory pool from which allocation requests are satisfied. All the problems mentioned above are not fundamental limitations and should be solvable with moderate engineering effort.

The part of our design of the VM/EPT isolation mechanism that addresses the problem of race conditions on shared memory was not implemented. This is because we first wanted to demonstrate the the feasibility of VM-based isolation before committing to the considerable effort we estimate it takes to implement the solution proposed in Section 6.5.2. The time-frame of this work did unfortunately not allow us to implement this part of the design after establishing the fundamental feasibility of VM-based isolation and we have to leave the implementation of a practical solution to this problem to future work.

An interesting topic for potential future research is the hybrid use of isolation mechanisms, particularly the use of intra-address space isolation within individual VM/EPT compartments. This would allow enforcing strong isolation via the VM/EPT mechanism at a coarser level while enforcing isolation between several sub-compartments via a lightweight intra-address space mechanism, further increasing the flexibility of the user to define a compartmentalisation configuration that strikes the optimal trade-off between performance and security.

Appendix A

Reproducibility

The experiments conducted in Chapter 6, specifically the microbenchmarks (Section 6.2), iPerf benchmarks (Section 6.3), and SQLite benchmarks (Section 6.4) are documented in this GitHub repository:
https://github.com/SebastianRauch/evaluation.

Docker images are provided which can be used to reproduce the results of the experiments.

Bibliography

[1] Coccinelle. https://coccinelle.gitlabpages.inria.fr/website.

[2] Cscope. http://cscope.sourceforge.net.

[3] iperf2. https://sourceforge.net/projects/iperf2.

[4] kraft. https://github.com/unikraft/kraft.

[5] The linux kernel. The kernel's command-line parameters. https://www.kernel.org/doc/html/v4.19/admin-guide/kernel-parameters.html. accessed 14 February 2022.

[6] Microsoft Hyper-V. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about.

[7] MirageOS. https://mirage.io/.

[8] nginx. http://nginx.org.

[9] Oracle VM VirtualBox. https://www.virtualbox.org.

[10] QEMU. https://www.qemu.org.

[11] Redis. https://redis.io.

[12] Sqlite. https://sqlite.org/index.html.

[13] Unikraft 0.5.0 Tethys documentation. Application development and porting. http://docs.unikraft.org/developers-app.html. accessed 10 January 2022.

[14] Unikraft 0.5.0 Tethys documentation. Unikraft libraries. http://docs.unikraft.org/intro.html#unikraft-libraries. accessed 10 January 2022.

[15] Using the gnu compiler collection (GCC). Extended asm. https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm. accessed 22 February 2022.

[16] Using the gnu compiler collection (GCC). Program instrumentation options. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html. accessed 10 Januarary 2022.

[17] VMWare ESXi. https://www.vmware.com/products/esxi-and-esx.html.

[18] VMware Workstation Player. https://www.vmware.com/products/workstation-player/workstation-player-evaluation.html.

[19] Xen project. https://xenproject.org.

[20] Xen project wiki. Dom0. https://wiki.xenproject.org/wiki/Dom0. accessed 02 February 2022.

[21] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pages 340–353. Association for Computing Machinery, 2005.

[22] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13. Association for Computing Machinery, 2006.

[23] Bill Alexander, Andy Anderson, Barry Huntley, Gil Neiger, Dion Rodgers, and Larry Smith. Intel® architected for performance - virtualization support on nehalem and westmere processors. *Intel® Technology Journal*, Volume 14(3):84–102, January 2010.

[24] M. Anderson, R. D. Pose, and C. S. Wallace. A password-capability system. *The Computer Journal*, 29(1):1–8, January 1986.

[25] ARM Ltd. Building a Secure System using TrustZone Technology. https://developer.arm.com/documentation/PRD29-GENC-009492/c, 2009. Online; accessed 09 January 2022.

[26] ARM®. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*, 2021. https://developer.arm.com/documentation/ddi0406/cd, accessed 20 January 2022.

[27] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, and Neugebauer Neugebauer. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM Symposium on Operating systems principles (SOSP)*, pages 164–177. ACM Press, October 2003.

[28] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12(4):271–307, November 1994.

[29] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU pitfalls: Attacks on PKU-based memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1409–1426. USENIX Association, August 2020.

[30] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016.

[31] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[32] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 191–206. Association for Computing Machinery, 2015.

[33] D. M. England. Capability concept, mechanisms and structure in system 250. In *Proceedings of the Intrnational Workshop on Protection in Operating Systems*, pages 68–82, 1974.

[34] Dawson R. Engler, M. Frans Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266. Association for Computing Machinery, 1995.

[35] Robert S. Fabry. Capability-based addressing. *Commun. ACM*, 17(7):403–412, July 1974.

[36] Norman Feske. *Genode Foundations.* 2021.

[37] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 38–51. Association for Computing Machinery, 1997.

[38] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, pages 109–114. Association for Computing Machinery, 2000.

[39] Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, 1973.

[40] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean application compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CSS)*, pages 1016–1031. Association for Computing Machinery, 2015.

[41] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 66–77. Association for Computing Machinery, 1997.

[42] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process isolation for High-Throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504. USENIX Association, 2019.

[43] Germont Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Softw. Pract. Exper.*, 28(9):901–928, July 1998.

[44] Hewlett Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1994. Third Edition.

[45] IBM Corporation. *Power ISA^{TM}, Version 3.1*, 2020.

[46] Intel Corp. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, June 2021.

[47] Michael Kerrisk. cgroups(7). Linux man-pages project, Release 5.13 https://man7.org/linux/man-pages/man7/cgroups.7.html, 2021. online, accessed 08 Januarary 2022.

[48] Michael Kerrisk. namespaces(7). Linux man-pages project, Release 5.13 https://man7.org/linux/man-pages/man7/namespaces.7.html, 2021. online, accessed 08 Januarary 2022.

[49] Michael Kerrisk. readelf(1). Linux man-pages project, Release 5.13 https://man7.org/linux/man-pages/man1/readelf.1.html, 2021. online, accessed 23 February 2022.

[50] Michael Kerrisk. seccomp(2). Linux man-pages project, Release 5.13 https://man7.org/linux/man-pages/man2/seccomp.2.html, 2021. online, accessed 16 February 2022.

[51] Michael Kerrisk. syscall(2). Linux man-pages project, Release 5.13 https://man7.org/linux/man-pages/man2/syscall.2.html, 2021. online, accessed 13 February 2022.

[52] Michael Kerrisk. syscalls(2). Linux man-pages project, Release 5.13 https://man7.org/linux/man-pages/man2/syscalls.2.html, 2021. online, accessed 08 Januarary 2022.

[53] Michael Kerrisk. taskset(1). Linux man-pages project, Release 5.13 https://man7.org/linux/man-pages/man1/taskset.1.html, 2021. online, accessed 15 February 2022.

[54] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 207–220. Association for Computing Machinery, 2009.

[55] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*, pages 376–394. Association for Computing Machinery, 2021.

[56] Stefan Lankes, Simon Pickartz, and Jens Breitbart. HermitCore: A unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*. Association for Computing Machinery, 2016.

[57] Hugo Lefeuvre. Toward specialization of memory management in unikernels. Bachelor's thesis, Karlsruhe Institute of Technology, 2020.

[58] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: Towards flexible OS isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2022.

[59] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the sixth USENIX Symposium on Operating systems design and implementation (OSDI)*. USENIX Association, December 2004.

[60] Henry M. Levy. *Capability-Based Computer Systems.* Digital Press, 1984. https://homes.cs.washington.edu/~levy/capabook.

[61] Guanyu Li, Dong Du, and Yubin Xia. Iso-UniK: lightweight multi-process unikernel through memory protection keys. *Cybersecurity*, 3(11):1–14, 2020.

[62] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles Principles (SOSP)*, pages 237–250. Association for Computing Machinery, 1995.

[63] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 418–429. Association for Computing Machinery, 2018.

[64] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[65] H. J. Lu, Michael Matz, Milind Girkar, Jan Hubicka, Jaeger Andreas, and Mark Mitchell. *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, 2018.

[66] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-in-time summoning of unikernels. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, pages 559–573. USENIX Association, 2015.

[67] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 461–472. Association for Computing Machinery, 2013.

[68] Toshiyuki Maeda and Akinori Yonezawa. Kernel mode linux: Toward an operating system protected by a type theory. In *Proceedings of the 8th Asian Computing Science Conference (ASIAN)*, pages 3–17. Springer, 2003.

[69] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 218–233. Association for Computing Machinery, 2017.

[70] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. Association for Computing Machinery, 2013.

[71] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel® virtualization technology: Hardware support for efficient processor virtualization. *Intel® Technology Journal*, Volume 10(3):167–177, August 2006.

[72] Ruslan Nikolaev and Godmar Back. VirtuOS: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 116–132. Association for Computing Machinery, 2013.

[73] Ruslan Nikolaev, Mincheol Sung, and Binoy Ravindran. LibrettOS: A dynamically adaptable multiserver-library os. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 114–128. Association for Computing Machinery, 2020.

[74] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 59–73. Association for Computing Machinery, 2019.

[75] Gabriele Paoloni. How to benchmark code execution times on Intel® IA-32 and IA-64 instruction set architectures. September 2010.

[76] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[77] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[78] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. CubicleOS: A library OS with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 546–558. Association for Computing Machinery, 2021.

[79] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 170–185. Association for Computing Machinery, 1999.

[80] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*. Association for Computing Machinery, 2016.

[81] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts Essentials*. Wiley Publishing, 10th edition, 2018.

[82] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[83] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 33–46. USENIX Association, 2010.

[84] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238. USENIX Association, 2019.

[85] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting software with code-centric memory domains. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, pages 469–480. IEEE Press, 2014.

[86] Robert Watson. The Arm morello board. *University of Cambridge. Department of Computer Science and Technology.* https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-morello.html. accessed 24 January 2022.

[87] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Simon W. Moore, Steven J. Murdoch, and Michael Roe. Capability hardware enhanced RISC instructions: CHERI instruction-set architecture. Technical Report UCAM-CL-TR-864, University of Cambridge, Computer Laboratory, December 2014. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-864.pdf.

[88] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*, pages 20–37. IEEE Computer Society, 2015.

[89] Jinpeng Wei and Calton Pu. TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study. In *Proceedings of the 4th Confer-*

ence on USENIX Conference on File and Storage Technologies (FAST) - Volume 4. USENIX Association, 2005.

[90] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, December 2003.

[91] M. V Wilkes. *The Cambridge CAP Computer and Its Operating System (Operating and Programming Systems Series)*. North-Holland Publishing Co., New York, 1979.

[92] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 304–316. Association for Computing Machinery, 2002.

[93] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for Linux using mondrian memory protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP)*, pages 31–44. Association for Computing Machinery, 2005.

[94] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfei Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. Kylinx: A dynamic library operating system for simplified and efficient cloud virtualization. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, pages 173–185. USENIX Association, 2018.