Diplomarbeit

Systematic setup of compartmentalised systems on the seL4 microkernel

Lukas Hänel

February 26, 2010

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dr. Kevin Elphinstone

Technische Universität Dresden
Institute for Systems Architecture

TOPIC FOR DIPLOMARBEIT

Student Name: **Hänel Lukas**

Study programme: Computer Science

Student number: 3061420

Topic

Systematic setup of compartmentalised systems on the seL4 microkernel

Description

The seL4 microkernel was designed to fit high assurance goals: it exports services in a secure way using capabilities, the implementation has been formally shown to correctly implement its specification, and it employs new approaches to kernel resource management. The goal of this thesis is to leverage the properties of the microkernel to systematically support compartmentalised applications on the kernel, including compartmentalising access to hardware devices.

The thesis is expected to examine several issues. It will examine and develop a method for the specification of static, compartmentalised systems with specific focus on what needs to be specified on seL4, and the specification of the platform and its relationship to the system it is expected to support. The thesis will identify and implement the components that are required to support the instantiation of a specified system.

The proof of concept demonstrator will focus on the specification and instantiation of paravirtualised instances of Linux with direct (but secure) access to I/O devices. Thus the specification of devices is within the scope of this thesis, and a method for locating and securely accessing the specified devices will be supported in Linux.

The overall system design should take into consideration the minimisation of trusted code required to instantiate the system, and any space and time overheads introduced where relevant.

Supervisor:	Dr. Kevin Elphinstone
Chair:	Operating Systems
Start:	10.9.2009
Finish by:	10.3.2009
Supervising Professor:	Prof. Dr. Hermann Härtig

Dresden, 01. 09. 2009

Signature

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den February 26, 2010

Lukas Hänel

Acknowledgement

I would like to thank everybody that made my overseas trip such a rich experience. My special thanks goes to Hermann Härtig for affording me the opportunity to study abroad and for his inspiring lectures that first awakened my interest in operating systems. I want to thank Gernot Heiser for the ERTOS group at NICTA. My thanks goes to Kevin Elphinstone and Peter Chubb for welcoming me in Australia and guiding my work. I would like to thank the many people who worked with me in the project, namely Ben Kalman for developing the build tools, Michael von Tessin for support on the seL4 microkernel, and Ihor Kuz for guidance on component architectures and proof-reading my thesis. Thanks to Robert Sison for asking for more features and Adam Walker for maturing my code. I would like to thank the people responsible for my decision to go overseas, namely Christian Helmuth for his supervision of my study project, the guys at ST Microelectronics for the opportunity of my internship, and my dear parents for supporting me in many ways.

Contents

List of Figures

Listings

Chapter 1

Introduction

Developing high assurance systems is a hard problem. It requires intimate knowledge of system hardware and careful design and verification of the controlling software. When following a security architecture like MILS [AFOTH06] a microkernel controls the computer's central processing unit. The microkernel abstracts hardware protection mechanisms and provides compartments that are perfectly isolated except for explicitly-allowed compartment communications. The microkernel does not control buses and I/O devices of the system. They are controlled by bus managers and user level device drivers. The system architecture grants those compartments access to their hardware and connects them to client applications that require services of these devices. This thesis is about instantiation of such architectures on modern x86 computers using the seL4 microkernel [EDE08].

Several developments in computer technology call for more complexity in trustworthy systems. First, ever capable mobile devices create the desire to have a do-it-all device that handles entertainment, communication, personal data and financial transactions. Banks will only accept such devices when they are convinced that they are secure. Second, increasing performance capabilities of servers make it intolerable to have only one customer per machine. While untrusted server virtualisation is not a problem for many industries, military and governments demand solutions that provide isolation equivalent to physical separation. By default, embedded systems in aircrafts and automobiles require high assurance of their safety. With the advent of smaller sizes and lower costs, these systems control more and more parts of their environments and hence increase in complexity.

While microkernels improve the system architecture, they do not necessarily make a system secure. The next operating system issue that a secure system has to address is device and resource management. Common solutions to these tasks fail to be secure. I therefore propose an approach that leverages security architectures to do secure resource and device management.

Common operating systems are designed with extensibility and universality in mind. Device and resource management is geared to support new devices and components with unknown resource requirements. In turn, the kernel does not preallocate resources but follows a demand-allocation scheme where modules and applications continuously request resources. The kernel serves these requests with a best effort to achieve a fair resource distribution. In reality, however, faulty or malicious applications can request all resources of the OS and thereby deny service to other applications. Additionally,

availability of resources can be used as a covert channel. Two applications that can each exhaust and determine exhaustion of a resource can communicate.

In contrast, applications in seL4 cannot exhaust kernel resources. Isolated compartments only access seL4 to manage their kernel objects or to communicate with other compartments. Standard resource management in microkernel systems is traditionally performed by user level resource managers [GJP+00, APJ+01]. However, these resource managers also implemented a best-effort strategy and were therefore also prone to provide covert channels. More generally, global services that connect untrusted applications can compromise application isolation. Because of this threat, the developer must either analyse them rigorously or avoid them entirely.

A high assurance system cannot give control to untrusted applications. Hence this thesis proposes an architecture where applications do not request resources at runtime. By not allowing applications to control availability of resources, denial-of-service attacks and resource-exhaustion channels are prevented. To replace on-demand requests, resource allocations are determined at development time. An architecture specification is used to describe the resource requirements of components and the assignment of devices to components.

1.1 Outline

The remaining part of this thesis is structured as follows:

Chapter 2 Chapter 2 introduces the background and basis for this thesis. I will define information flow architectures and compartments and show their relation to computer security. I will then survey software architectures for secure and embedded systems. Those are based on hardware protection mechanisms that I will explain afterwards. Specialisation in computer hardware emerged the requirement for an I/O device discovery mechanism that plays an important role in computer compartmentalisation as well. I will therefore explain it. Then I will give an introduction to virtualisation as background for the compartmentalisation of PCI device discovery. Also I will review related work.

Chapter 3 Following this, I will introduce the seL4 project because the component developed in this thesis forms a basic component of every new seL4 system. I will summarise the goals and security guarantees, explain the microkernel API, and show current applications.

Chapter 4 In chapter 4, I will design a secure approach to device discovery for the system and for compartments. Then I will design the models for I/O devices, compartments, and connections and show the seL4 objects they consist of.

Chapter 5 In this chapter, I will describe my implementation and substantiate implementation decisions. I will handle the loading process, device access in Linux, device discovery mechanisms and compartmentalisation of the timer.

Chapter 6 In chapter 6, I will evaluate the achievement of my goals, show limitations of the approach and outline possibilities for future work.

Chapter 7 Finally, in chapter 7, I will draw conclusions about my work.

Chapter 2

Background and Related Work

This chapter will give the necessary background for this thesis. I will define the term *information flow architecture* that is the basis for my work. I then survey existing system security research and highlight its relation to information flow architectures. I introduce hardware protection mechanisms of modern computers because I employ them to implement an information flow architecture. I will then explain PCI device discovery and the issues and opportunities of using it. The chapter continues with a survey of virtualisation solutions and their approach to virtualisation of device discovery. I will conclude with related work.

2.1 Information flow architectures

2.1.1 Definition

Access control policies [San93] define how information can flow between subjects and objects. Subjects and objects get assigned security classifications, or labels and the policy defines how information can or may not flow from one label to another. For example, information may not flow from high to low in the Bell-La Padula model. However, special secure applications may be authorised to allow such an information flow when they ensure the secrecy goal by other means. For example, a declassifier could remove the high label from outdated information, an encryption program could maintain secrecy by outputting scrambled data only, and a filtering component could remove classified bits from documents.

In a simple example (see Figure 2.1), a filtering system would accept data with secret classification on one I/O device, remove compromising parts in a filter component and then transmit the data as unclassified on another I/O device.

Subjects and objects are implemented by a secure operating system. This system is *secure*, when it always enforces the restrictions set by the access control policy. That is, the OS has to make sure that information can flow from one subject to another, only if this flow is allowed in the policy. To do so, the OS splits the computer system into partitions or *compartments* that are isolated from each other. Between these compartments, the OS may enable monitored or unmonitored communication. A compartment hosts a subject and the OS ensures that two compartments are only connected if information can flow between respective subjects.

A compartment as part of a computer system consists of several parts. Because a compartment is an active software component it consists of one or more threads of activity

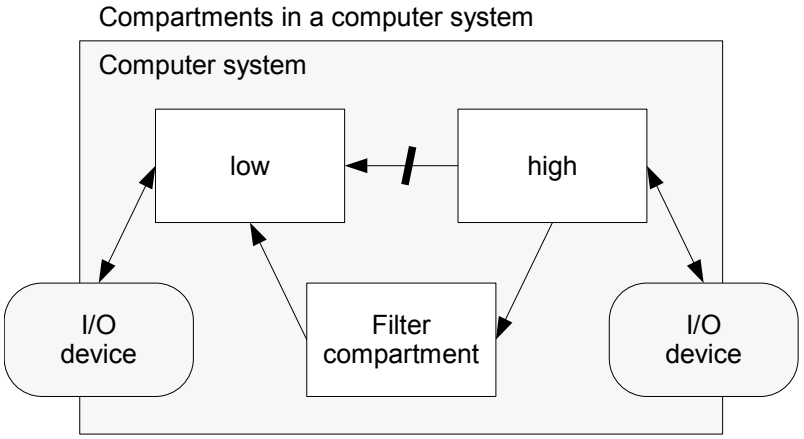

Figure 2.1: Example information flow architecture

running on one or more processors. Consequently, a compartment also comprises a partition of the computer's memory to store the program code and data. A compartment can also contain *input–output* (I/O) channels and hence include I/O devices. If a compartment is connected to another compartment or to the OS, one side of the connection is also part of the compartment.

Compartments and inter-compartment connections form a graph. This graph specifies how information can flow in a computer system. I call this graph an *information flow architecture* to highlight that it represents both a software architecture and an information-flow restricted system.

2.1.2 Problems

Systems implementing access control policies face two main problems: Privilege escalation attacks and the presence of covert channels. I will briefly outline the two.

Privilege escalation

In a privilege escalation attack a program breaks out of its compartment. It can then control a bigger part of the computer and eventually access classified data.

Privilege escalation attacks require that a malicious program can communicate with programs with higher privileges or with the OS. During this communication, the malicious program tries to drive the attacked component into a state the developer did not foresee. In a successful attack, the attacker then communicates prepared data that overwrites the program in the other compartment with his own program.

Common operating systems and applications suffer from these attacks because their specification does not disallow this behaviour or because they do not implement their specifications correctly. *High-assurance secure systems* have to show convincingly that they are correct and that their specification does not allow privilege escalation attacks. However, they may still have covert channels that allow unauthorised access to classified data.

Covert channels

A system contains covert channels [Lam73] if unconnected compartments can communicate. Because this connection conflicts with the access control policy it allows for two types of attacks. First, a malicious program can observe classified data from another program. Second, a confined but untrusted program handling classified data can communicate this data to a less classified program.

A system contains covert channels when compartments can induce and assess change in global resources. Depending on the type of resource, channels are classified as storage channels or timing channels. Furthermore, resources can be provided by the OS or by the hardware.

Typical OS provided resources for covert channels are namespaces and file contents. Also, the exhaustion of disk or main memory can be used as a covert channel. An example for a covert timing channel in hardware is the utilisation of a cache. Two compartments can communicate, if one compartment can deliberately cause cache misses in the other compartment and the other compartment can monitor the resulting changes in access time.

To mitigate covert channels, the OS has to be designed in a way that does not allow a compartment to induce or assess changes in global resources. This requirement in turn demands a thorough examination of every shared hardware and software component.

In this section I defined the requirements for a high assurance secure system. I showed how privilege escalation attacks and covert channels can overcome the system compartmentalisation set up by the OS. The next section will show how to mitigate these problems when building high-assurance secure systems.

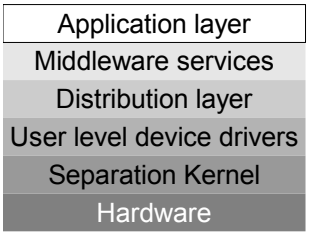

Figure 2.2: MILS layer

2.2 Security Architectures

Developing complex high-assurance systems seems to be impossible. Formal verification as developed in the L4.verified project [KEH+09] shows the feasibility of writing correct software. However, the technology does not scale to complex operating systems. Complexity is often addressed with modularisation and componentisation techniques because complex problems often consist of various sub problems of lower complexity. Individual components can then be developed and examined independently.

Usually however, systems are decomposed into functional units. While this approach aids in developing and managing the system, it does not necessarily make the system secure. When component boundaries are not enforced at runtime, such systems cannot use untrusted components and rely on the correctness of all components. Systems based on secure languages (e.g., Singularity/Sing# [FAH+06]), enforce boundaries but do not support legacy components and require expensive reengineering of all parts. Microkernel-based security architectures [HHF+05, FH06] enforce boundaries and support legacy components.

To increase trustworthiness, a security architecture splits the system into trusted parts and untrusted parts. The trusted parts are called the *trusted computing base* (TCB) and it is a goal of security architectures to minimise the trusted computing base. The TCB includes the OS kernel and core components that all need verification. Classical systems based on a trusted computing base became too complex to verify. The MILS architecture tries to remedy this situation by a new approach that I will introduce next.

Multiple Independent Levels of Security and Safety

The ERTOS group tries to develop and verify a high-assurance secure system following the Multiple Independent Levels of Security and Safety (MILS) approach [AFOTH06]. I will introduce this approach to give an overview of the architectures my work shall support.

The main idea of the MILS approach is the system decomposition rule. MILS decomposes the security goal of a complex system into several security properties that are established by components at different security layers (see Figure 2.2). Each component is build upon the guarantees of its lower layers and is only responsible for its own security property. Security analysis focuses on individual security components to achieve the overall security goal.

At its lowest layer the MILS architecture requires modern processors that support efficient context switches to allow isolated OS components. Also, a privileged mode to control the security features and I/O protection is required. (see Section 2.4 for an explanation of the x86 architecture.)

The second layer is a separation kernel that controls the processor and abstracts the processor's features. It has to provide isolated partitions (compartments) of the computer that can only communicate via confined inter-partition communication primitives. It must be impossible to circumvent the kernel separation, that is, to communicate to other partitions by other means than the communication primitives.

The next layer in the MILS architecture is formed by userlevel device drivers that control I/O devices. Because these devices are not used to partition the computer's computing resources they need not be controlled from inside the separation kernel. Instead they are controlled by device driver partitions. Access control to devices is implemented using the separation kernel's communication primitives.

The MILS architecture also contains a layer for distribution and parallelisation that I disregard in this thesis.

The next layer is the middleware services layer. This layer is built on top of the separation kernel's communication primitives and provides more traditional OS services and information flow control. Example services include resource allocation, higher-level communication services and real-time data distribution. Information flow control in the middleware layer ensures that partitions and messages are labelled and inappropriate messages are filtered. Also it should enable one-way communication between partitions. To do so, the middleware service layer sets up an information flow architecture and controls key nodes in the information flow graph.

The preceding layers sum up to what a secure operating system would usually provide. Based on these guarantees, the final application layer is responsible only for security functionality. Typical applications are cryptographic services that take data at a high classification level, encrypt it, and sent it out at a low classification level.

The MILS approach provides guidelines and requirements for the design of secure systems. However, it does not address software development issues like software reuse and hardware specifics like bus mastering by I/O devices and device discovery that are popular on modern computers. These issues are addressed by other means that I will present and use during this thesis. Also the MILS approach does not present a separation kernel. Research into microkernels promises to achieve the separation kernel's requirements (see Chapter 3). The next section will introduce a software development technique for microkernel-based systems.

2.3 Component Architecture for microkernel-based Embedded Systems

CAmkES, the Component Architecture for microkernel-based Embedded Systems [KLGH07, NIC] is a framework for the specification and instantiation of software architectures. Because its architecture is based on a microkernel, CAmkES was a natural basis for this thesis. However, it was not used.

CAmkES consists of three parts: The architecture description language specifies the components and how they are connected. A code generator transforms this description into stub code and an initialisation routine. A startup program and template code bind the architecture to the underlying microkernel system. In this system, compartments are connected via synchronous and asynchronous inter process communication (IPC) interfaces, and via shared memory. The component model takes this into account by providing components and connectors that map to these concepts. Additionally, CAmkES is based on typed interfaces between components. These interfaces are specified in an *interface description language* (IDL) and components can provide or use (require) such an interface.

The architecture description specifies components with their component interfaces and connections that connect components (see Figure 2.1 for an example). The specification starts with a list of *import* statements that include specifications for IDL interfaces and connectors. It then contains a list of component descriptions. Each component can *provide* or *use* an IDL interface. Components can *emit* and *consume* typed events. To specify usage of shared memory, components can have a typed *dataport*. Components with the *control* keyword, have a thread of control and thus will run from the beginning. Components can be composed of other, internally connected components. After the list of component descriptions, an *assembly* and *composition* section specifies, which components are instantiated and how the instances are connected. It starts with a list of instantiated components. Each entry specifies the type of the component and the name in this composition. The composition then contains a list of connections. Each connection specifies the connector, a name for the connection, and the partners. For each partner the connections also specifies the used stub.

From an architecture description, the build system will generate stub code and infer a directory layout to find and build the components. The stub code consists of headers and library functions that translate the invocation of an interface into the appropriate microkernel communication primitive. Also a startup program is generated that instantiates the architecture.

To port CAmkES to seL4, we have to integrate the CAmkES and seL4 build systems. This would ease development of seL4 component systems. Instead of this step, this thesis will provide a basis for seL4 stub code and the startup program.

CAmkES is a powerful tool for the specification and instantiation of software architectures, however, it lacks in two main areas: While being designed to specify operating systems, it does not take into account the specification and integration of I/O devices. Also, the current connectors do not restrict information flow. While CAmkES can specify information flow architectures, two connected components can communicate in both directions. To overcome the first problem area in my thesis, I will present the x86 I/O architecture and hardware protection mechanisms in the next section.

```
import "Hello.idl4";
import "std_connector.camkes";

component HelloComponent {
  provides Hello h;
  dataport Data buf;
  emits EHello hevent;
}

component HelloClient {
  control;
  uses Hello h;
  dataport Data buf;
  consumes EHello hevent;
}

assembly {
  composition {
    component HelloComponent hcomp;
    component HelloClient hclient;

    connection IguanaRPC hello(from hclient.h, to hcomp.h);
    connection IguanaAsynchEvent hello_e(from hclient.hevent,
                to hcomp.hevent);
    connection IguanaSharedData hello_d(from hclient.buf,
                to hcomp.buf);
  }
}
```

Listing 2.1: Example of a CAmkES architecture specification

2.4 Hardware compartmentalisation support

A modern Intel x86 computer contains processors [Int08a] and PCI devices [SA99] as active components, the DRAM as passive memory and the North Bridge and buses as connection devices (see Figure 2.3). Whereas PCI devices provide specialised operations for specific I/O protocols, the CPU performs general purpose processing. The CPU is also in charge of controlling the system compartmentalisation. As such, the CPU executes instructions that modify CPU and machine state. CPU compartmentalisation is achieved via the controlling *supervisor mode* and the restricted *user mode*. I/O compartmentalisation is achieved via the CPU, the PCI bus, and the IOMMU. In this section I will first present CPU compartmentalisation concepts and then explain how I/O channels are controlled. Afterwards I introduce the IOMMU and PCI device discovery.

2.4.1 CPU compartmentalisation

The CPU performs general purpose processing and system control operations. Complete system control is only possible in supervisor mode. In user mode, only a subset of the CPU state and machine state can be accessed. This subset is defined by data structures that are ultimately set up in supervisor mode. To support independent, isolated execution environments the CPU supports controlled switching between the modes. The program that runs in the supervisor mode is called the *operating system* (OS) or the *kernel*.

Exceptions

To isolate a single program the CPU has exceptions. When the processor is in user mode and the user program issues a privileged instruction or addresses unmapped memory, the processor switches to supervisor mode. The operating system can then decide how to handle that exception. That is, to stop the program, to handle the exception and restart the program, or to ignore the exception and continue the user program.

System Calls

To make functionality that requires the processor's privileged mode to be available to user mode programs, operating systems use the system call functionality of the processor. A system call instruction is used by a user mode program to voluntarily switch to the OS and to pass parameters to it. The OS then decodes the parameters, checks the validity of the request, and performs the requested operation.

Timer

To limit the time a user mode program can use the processor, the operating system uses a timer device. When the time of the currently running program has run out, the timer sends an I/O interrupt request to the processor. The processor will then perform a switch of the execution environment to supervisor mode. The OS can then decide

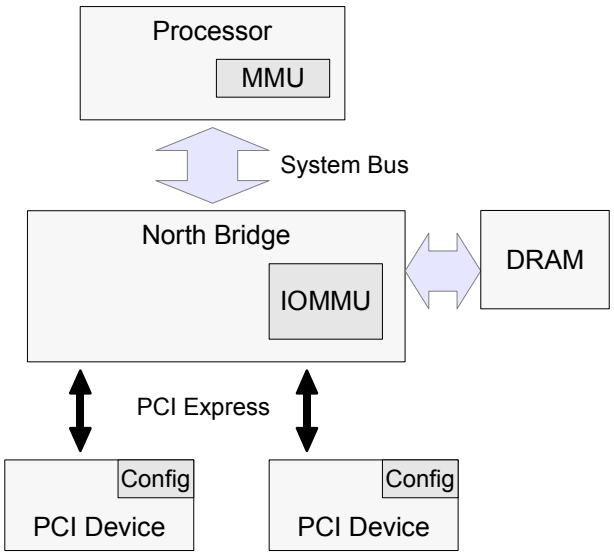

Figure 2.3: Computer architecture

whether to schedule another task, continue the current task, or to terminate the current task.

2.4.2 I/O Channels

The CPU accesses memory and I/O devices via memory requests on the system bus. The North Bridge decodes these requests and forwards them to either DRAM or to I/O devices (see Figure 2.3). The so addressed memory is called the physical memory, the set of addresses the physical address space. The CPU supports memory partitioning and protection via the memory management unit (MMU).

MMU

The MMU divides the physical address space into 4 kByte pages. It confines isolated programs into virtual address spaces that hide the physical addresses. Virtual address spaces are implemented by a translation from virtual addresses to physical addresses that traverses page directories and page tables. For 32-bit processors, the 10 most significant bits of a virtual address are used to index into the page directory. A page directory entry contains the physical address of a page table. The next 10 bit are used to index into the page table and to get the physical address of the memory frame. The resulting memory access uses the remaining 12 bit as offset into the frame. The currently used page directory can only be changed in supervisor mode. When the supervisor program refrains from mapping page directories and page tables into the virtual address space, the corresponding user program cannot access any other parts of the memory directly.

As devices are memory-mapped, they can, giving a mapping into a virtual address space, be accessed in user mode.

I/O Ports

Prior to the introduction of 32-bit memory addresses, x86 CPUs accessed I/O devices via I/O ports in an I/O space. The I/O space is a separate, 16-bit address space that is still used in modern computers to control legacy hardware, compatibility interfaces of modern devices, and the PCI bus. The I/O space can only be accessed by special I/O load/store instructions. It has a separate protection concept but is also routed via the system bus. Using the I/O permission bitmap, a supervisor mode program can give user mode tasks access to individual I/O ports.

I/O Interrupts

While I/O devices are active components, they usually perform tasks for the processor. When they have finished a task, they notify the processor via an I/O interrupt. The interrupt causes the processor to switch to supervisor mode. The OS then identifies and handles the interrupt. To continue normal operation, the processor must also acknowledge the interrupt.

2.4.3 IOMMU

Modern I/O devices are active components and can access memory independently of the CPU. Using bus mastering [SA99] they can perform *direct memory access* (DMA) and bypass the CPU's MMU protection. To set up a device for DMA, a user mode program requires the physical addresses of its virtual address space. More importantly, DMA is not subject to MMU protection. A user program can instruct a device to use DMA to access any part of memory. The introduction of an *input–output memory-management unit* (IOMMU) [Int08b] between PCI devices and the DRAM solves that problem. It introduces device virtual address spaces for I/O devices. Several devices can share one address space, and IOMMU address spaces can share pages with MMU address spaces. IOMMU address spaces are implemented using a similar structure, but with several 9-bit page tables to support 64-bit addresses. The CPU programs the IOMMU via memory-mapped registers. Those link to context entry tables that contain entries for each PCI device. Context entries link PCI devices to IOMMU address spaces. Using proper setup of these data structures, a PCI device can use the same virtual addresses as its controlling program and can only access the same memory.

2.4.4 PCI device discovery

The operating system has to perform device discovery to identify the I/O devices present in the system. To do so, it accesses common device addresses and evaluates the reaction (probing). For legacy x86 devices, the OS assumes their presence and uses them. To find ACPI [Hew06] reported devices, the OS scans the BIOS part of the memory to find,

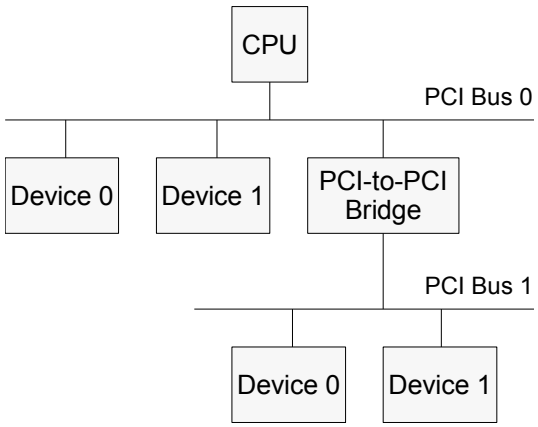

Figure 2.4: PCI bus structure

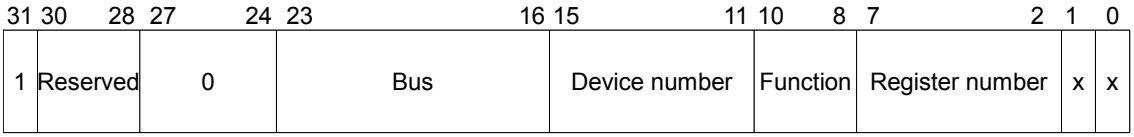

Figure 2.5: PCI configuration address format

by means of a magic number, the pointer to the ACPI tables that describe devices. To find PCI devices, the OS scans the PCI bus [SA99].

A system can have 1 to 256 PCI buses, each bus can have 1 to 32 devices and each device can have 1 to 8 functions. Functions correspond to different functional units a chip might have. For means of identification and configuration every PCI function must implement a PCI configuration space. A PCI configuration space consists of 64, 32-bit words of which the first 16 are defined by the standard (see Figure 2.6). However, Intel x86 processors cannot access the PCI configuration space directly. Instead, they use the north bridge as proxy. They access the north bridge via the following two 32-bit I/O ports:

- The Configuration Address Port, at 0CF8h.
- The Configuration Data Port, at 0CFCh.

To access configuration data, the OS writes the address to the address port and then reads or writes data from the data port. The address is decoded as follows: Bits 23–16 denote the Bus number, Bits 15–11 denote the Device number, and Bits 10–8 denote the Function number. Bits 7–2 address the word in the function's configuration space. The remaining bits are either reserved or have fixed values (see Figure 2.5).

To scan for PCI devices, the OS starts reading identity information of all devices of bus 0. If a device slot does not contain a device, the PCI bridge reports values of all 1's. If one of the devices was a PCI bridge, the OS starts probing the bus behind that bridge (see Figure 2.4). To identify devices, the OS reads the Vendor ID and Device ID from the first word of the header. Via the Status register, the OS can find out about the

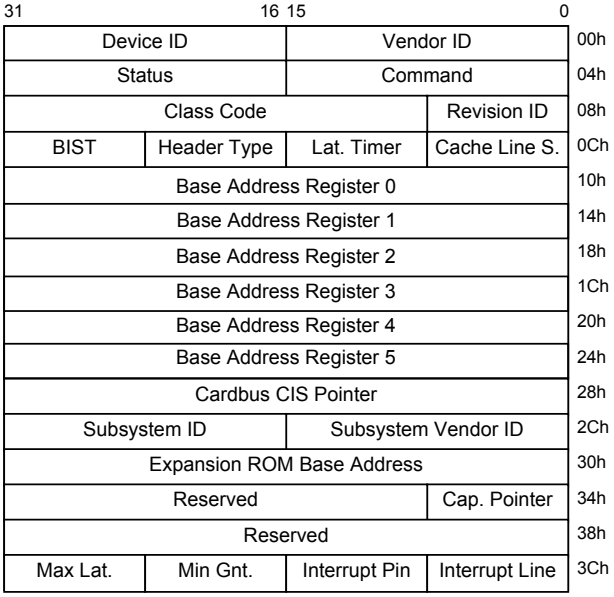

Figure 2.6: PCI configuration header

capabilities of the device. Base Address Register (BAR) 0 to 5 configure I/O regions of the device. Usually the BIOS configures these regions already such that they do not overlap. The OS still has to find out where each device's memory-mapped I/O regions are mapped in the physical address space. The same applies for I/O regions accessed by I/O ports.

While access to individual devices can be compartmentalised via memory-mapped I/O regions and I/O port protection, PCI device configuration cannot be compartmentalised in hardware. Access to the configuration I/O ports allows a program to change the hardware configuration. This leads to two problems: a) Two programs can use writeable configuration space memory as covert channel. b) Specific configurations of the hardware may break isolation. For example, the Base Address Registers of a PCI device use physical addresses. When a program changes those to overlay a portion of the physical memory, another program will read and write to the device's memory instead of to DRAM. When the original program changes the configuration back, it can read the data of the other program (see Figure 2.7).

In summary, PCI configuration space is an important means to communicate device information to the OS. The device discovery protocol is mainly based on reading information. However, writing data back is necessary as well. There is no compartmentalisation support for PCI configuration in hardware. When compartments have to use the PCI configuration space, an OS component must dispatch accesses securely.

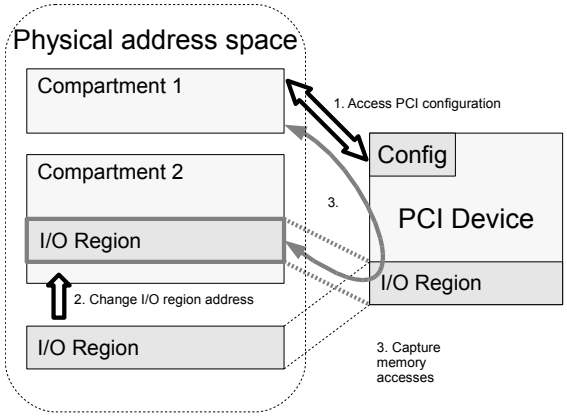

Figure 2.7: Capturing memory accesses using a memory-mapped I/O region

2.5 Virtualisation

Part of this thesis is about reusing existing commodity software using virtualisation. I will introduce techniques and terminology and outline existing solutions for virtualisation of PCI device discovery.

Various flavours of virtualisation exist for different use cases. In all kinds, a virtual machine is supposed to be an efficient, isolated duplicate of a real machine [PG74]. However, often, virtual machines do not exist as real machines. The computer the virtualisation system runs on is called the host system, the software running inside the virtual machine is called the guest system, the OS the guest OS.

A virtual machine consists of virtual CPU's, virtual memory and virtual devices. They are implemented by a combination of hardware and software mechanisms. A virtual machine monitor (VMM) software controls the physical machine's hardware that might have more or less support for virtualisation.

In modern computers, virtual memory can be used to virtualise the memory.

Processor virtualisation requires virtual instructions that modify virtual processor state. Depending on the physical processor and the virtual processor, virtual instructions must be implemented in a software interpreter (Emulation) or can run directly on the physical processor (Virtualisation) [NSL+06].

To maintain proper isolation, virtualised programs cannot be allowed to execute privileged instructions. Virtualisation systems handle privileged instructions in several ways [NSL+06]:

- they emulate them in hardware (Hardware Virtualisation),
- they emulate them with the help of the OS (Trap and Emulate), or
- they avoid them all together (Binary Rewriting, Paravirtualisation).

For microkernel-based systems, paravirtualisation is used to leverage the code base of legacy operating systems. To paravirtualise an operating system is to modify its source

code to run on the microkernel instead of physical hardware. Code portions that would use privileged instructions are replaced by calls to the microkernel system call API.

A virtual machine can use the following virtual devices [AJM+06]:

- devices simulated by software (Emulation),
- abstract software interfaces (Paravirtualisation),
- physical devices passed-through (Direct Assignment).

To virtualise an I/O device, we have to consider four types of interactions of the virtual machine and the physical device: device discovery and configuration, device control (MMIO), data transfers (DMA), and I/O interrupts. Modern systems support secure efficient virtualisation of the last three operations in hardware.

2.5.1 Virtualisation of PCI device discovery

PCI device discovery on x86 processors uses I/O port access. Respective instructions are privileged and have to be handled by the virtualisation solution. The virtual machine monitor then presents the guest OS the devices the user has configured for this virtual machine. Respective devices can be physical devices passed through to the virtual machine or virtual devices emulated by the VMM.

Hardware support for virtualisation of device discovery is currently not available. Virtualisation solutions therefore have to use the software-based mechanisms they use for privileged instructions. Two such examples are the QEMU open source processor emulator [Bel05] that uses emulation and the Xen hypervisor [FHN+04] that uses paravirtualisation. Respective uses of I/O port instructions or PCI configuration space operations are translated into accesses to a PCI device model. The VMM implements this model and emulates a reaction as specified by the PCI discovery protocol and the virtual devices the VMM wants to present. For a more detailed example see also [Zeh06].

Because modern computers are modular regarding their device configuration, device discovery mechanisms are a requirement. When compartmentalising the computer, device discovery must be compartmentalised as well. Virtualisation is a well-known compartmentalisation tool. I have surveyed existing techniques to extend an existing virtualisation solution with support for I/O device discovery (see Section 4.1). The next section will show related work.

2.6 Related Work

Device management

In my study thesis [Hän07], I analysed solutions for device management and I/O compartmentalisation. In summary, current server and desktop OSes do not encapsulate drivers. Drivers run in the supervisor mode of the processor and can access any memory

and I/O channel. Nevertheless, these OSes have mechanisms for controlled access to I/O resources [CRKH05]. However the OSes do not enforce usage of these mechanisms and trust all drivers.

Microkernel-based systems mitigate this problem and encapsulate drivers in address spaces [HLM+03, LCFD+05]. To manage I/O access these system reused the aforementioned mechanisms [Hel01, Hei05]. While they can guarantee that a driver compartment only accesses the I/O resources it requested, a driver can request any unclaimed resource.

Memory management

Similarly to I/O resources, common OSes handle memory and kernel object requests without restrictions. Hence a process can request arbitrary amounts of memory and can request creation of all kinds of kernel objects, that is, file handles, network connections, new processes, page tables, and so on. Again, microkernel-based systems faced similar problems [Elp04].

Virtual machine architectures

The recent move to system architectures based on virtual machines changes the resource management problem. Applications are no longer handled as processes that request resources at runtime, but as virtual machines with a predetermined resource allocation. This changes the approach to resource management, because a (guest) operating system can usually not send requests to the (virtualised) hardware to provide more resources. Therefore, I/O device allocations and physical memory allocations are defined by the user and not by the application.

However, virtualisation solutions often introduce load-balancing techniques that result in another form of resource sharing. Common memory load-balancing mechanisms like swapping and the more recent ballooning allow a virtual machine monitor to distribute physical memory between virtual machines depending on the load. This can lead to a covert timing channel because the physical memory becomes a cache of disk memory (see Section 2.1.2). While this attack against resource management is viable, it is rather theoretical. Because physical memory is big in comparison to disk transfer rates this attack would only provide a low-capacity covert channel. Additionally, virtualisation solutions are usually based on common operating systems and hence have a huge trusted computing base that should have bigger holes in other areas.

In conclusion, high assurance secure systems require isolated components. This can be achieved with microkernel-based system architectures. However, microkernels alone provide no security. The next step is secure I/O and resource management. Traditional solutions are not secure, however, virtual machine architectures have more secure I/O and resource management.

In this chapter I laid the basis for this thesis. I started with the theoretical background, access control policies and computer compartmentalisation. Then I introduced a security architecture that promises to enable high assurance secure systems. For such

architectures, we require appropriate development methodologies. For this purpose I outlined CAmkES, the Component Architecture for microkernel-based Embedded Systems. However, my work is not based on embedded systems. Instead I use the x86 and PCI hardware architecture. I introduced the relevant hardware compartmentalisation mechanisms and detailed the I/O memory management unit and PCI device discovery. Also I showed how virtualisation is used to compartmentalise systems and how related work addresses I/O and resource management. The next chapter will introduce the seL4 microkernel and show how it abstracts the hardware compartmentalisation mechanisms.

Chapter 3

seL4

Because the software developed in this thesis is a major component of a seL4 system, I reserve a complete chapter for seL4, the secure embedded L4 microkernel [EDE08]. I first restate the reasons to use seL4. Then I define the kernel, highlight associated security guarantees and explain the programming interface. I will conclude with the state of current user-level applications.

3.1 Motivation

Common operating systems were designed with performance and functionality in mind. They run most of the OS in the processor's supervisor mode. Hence a flaw in one part of the OS kernel affects the security and safety of the entire system. Software verification is used to prove software bug-free. However, software required to operate today's systems has reached a complexity that is beyond conventional analysis and verification methods. Microkernel-based systems remedy that situation by splitting the system into parts that can be verified independently. The seL4 microkernel has in turn undergone functional specification and formal verification [KEH+09].

To give guarantees about subsystem isolation, the microkernel may not enable information flow except through defined interfaces. However, OS kernels and initial L4 microkernels enabled communication over global name spaces and resource availability [Elp04]. Restricting access to resource availability implies an explicit resource policy. The main resources the microkernel cares about are microkernel objects and memory to back these objects. The seL4 microkernel therefore exports kernel memory management to user level. To address kernel objects, seL4 defines capabilities. Capabilities form local name spaces and hence are kernel objects as well. In turn processes also need to manage their capability spaces.

3.2 What it is

The seL4 microkernel operates in the privileged mode of the processor. As a microkernel it only contains functionality required to export hardware features securely and efficiently. The basic security guarantee states that a properly isolated compartment cannot extend its authority.

The following abstractions are defined by seL4:

Threads as units of activity that time-share the processor and use system calls. Threads are handled via *thread control blocks* (TCB) Thread control blocks have a thread state, a register set, a scheduling priority, an address space, a capability space, an IPC buffer, and a fault endpoint.

Address spaces called VSpaces, as memory abstractions that partition the main memory. VSpaces are build up from page directory objects, page table objects and frame objects.

Endpoints as means to communicate between threads. Endpoints can transfer data and capabilities. IRQs and exceptions are received on Endpoints.

Capability spaces called CSpaces, as containers for references to kernel objects. CSpaces are implemented as guarded page tables and built up from page-table-like CNodes.

Physical memory called untyped memory, to control kernel and application memory usage. Untyped objects have different power-of-2 sizes and can be retyped into kernel objects.

Hardware control objects like IRQ control, address space ID control, IOMMU address space ID control, I/O port access. Hardware control objects have specific interfaces.

Via the untyped memory abstraction seL4 externalises kernel memory management and prevents kernel memory exhaustion. Any memory the kernel consumes is abstracted in a kernel object. Applications can only create kernel objects to the extent of their access to untyped memory. Furthermore, once retyped, untyped memory cannot be typed into something else before it is revoked. Hence, the seL4 kernel guarantees type safety of its objects. Together with the local name spaces provided by the CSpaces, this property enables the seL4 security guarantee that "a capability cannot ever cross an isolation domain".

For the isolation guarantee to hold, isolated domains must meet the following restrictions [EDE08]:

- No sharing of writeable page tables or writeable CNodes.
- No shared endpoint that can transfer capabilities.
- No access to thread control blocks across domains.

In addition to the seL4 isolation guarantee, a secure system often requires spatial and temporal isolation between components. While temporal isolation is beyond the scope of this thesis, spatial isolation is not.

For spatial isolation, additional restrictions must be met:

- No sharing of frame objects.

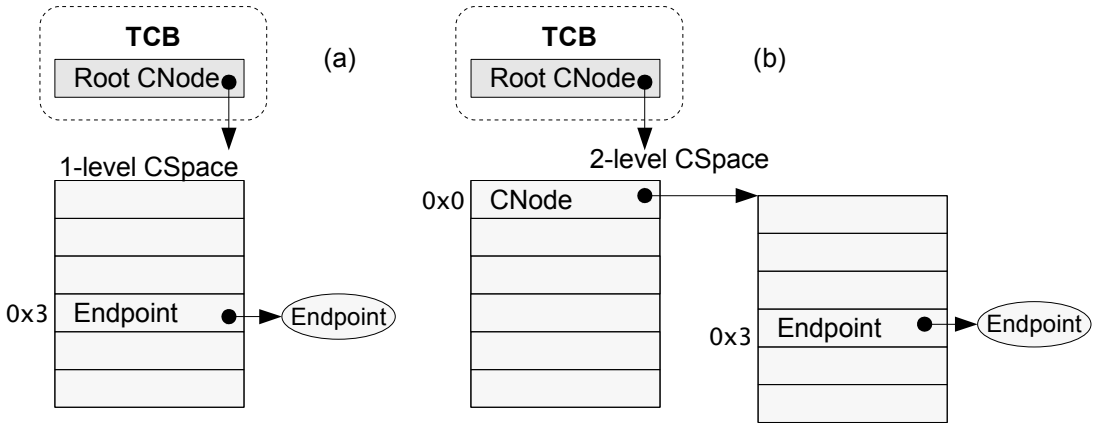

Figure 3.1: Capability references: (a) CSpace with one level: capability reference 0x3 is used as an index into the one CNode. (b) CSpace with two levels: capability reference 0x03 is translated, first 4 bits as index into the level-1 CNode, bits 4-7 as index into the level-2 CNode.

That is, no shared memory. A frame may also not be shared via IOMMU address spaces.

- No sharing of endpoints, untyped objects, or hardware control objects.

  That is, no communication via seL4 objects.

- No sharing of I/O devices.

  That is, one domain per device. seL4 represents a device by several capabilities. All those should be allocated to one compartment.

3.3 How to

Because the software developed in this thesis closely interacts with the seL4 kernel, I will give a more detailed explanation of the kernel API [NIC06]. I will define the different seL4 abstractions more precisely and explain their interactions.

3.3.1 Capabilities and CSpaces

A capability gives a user the right to use a kernel object. Using the capability is called invoking the capability. To invoke a capability, a user specifies the *capability reference* during the system call. The kernel uses the capability reference to index into the user's 32-bit wide CSpace that contains the capability (see Figure 3.1). A capability also states the seL4 permissions, like read, write and grant, a user has to the kernel object.

In an analogy, capabilities are like pointers to kernel objects. However, these pointers cannot be on the stack, but come from an array on the heap. To access an object, an index into the pointer array must be used. It is now the job of the user program to manage empty slots in this array and to maintain the semantics and relationships of

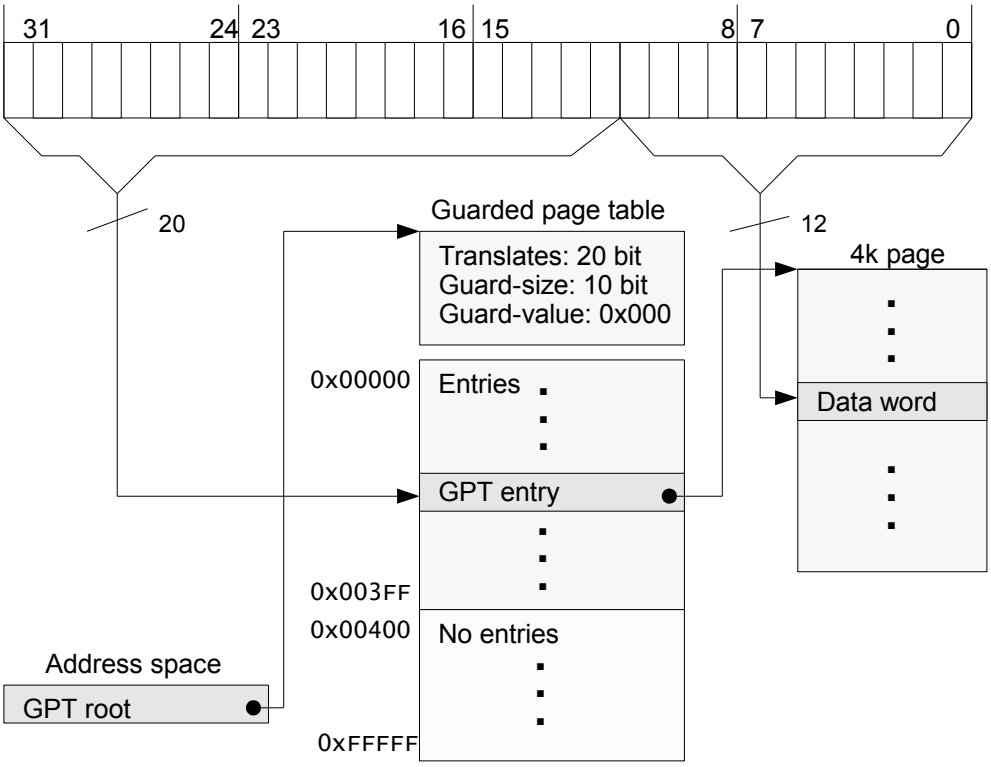

Figure 3.2: Guarded page table example. One GPT can cover a complete 32-bit address space.

filled slots. In another analogy, capability references resemble Unix file handles, they are pure numbers to the application but array indexes to the kernel. However, CSpaces are not simple arrays.

CSpaces are implemented like page tables because they require fast, bounded lookup times, and big maximum sizes, but low average sizes. To allow for variable-sized CSpaces, seL4 uses *guarded page tables* (GPT) [Lie94].

In a GPT, every table can have a different, power-of-2 size. Additionally, a table can cover a bigger address range than it actually translates (see Figure 3.2). Usually, when a page table translates x bits, it has $n = 2^x$ entries. A GPT however can have less entries. Therefore, every GPT has a number t of bits it covers, a number n of entries, and a guard with size s and value v such that $n = 2^{t-s}$. The value v determines the value that the first s address bits must have to be translated by the GPT. Addresses that fall into the address range of the GPT, but do not *pass* the guard, cause page faults.

The guarded page table equivalent in seL4 is the CNode that has a guard and a number of entries. Capabilities are entries (slots) in CNodes. A CSpace consists of an hierarchy of CNodes (see Figure 3.1).

Similarly to a virtual address space, not every capability reference in a CSpace translates to a final object. Empty slots can be used to accommodate new capabilities. Generic capability invocations allow a user to move or copy capabilities from one place to another. Also this can be used to reduce permissions and to set further attributes.

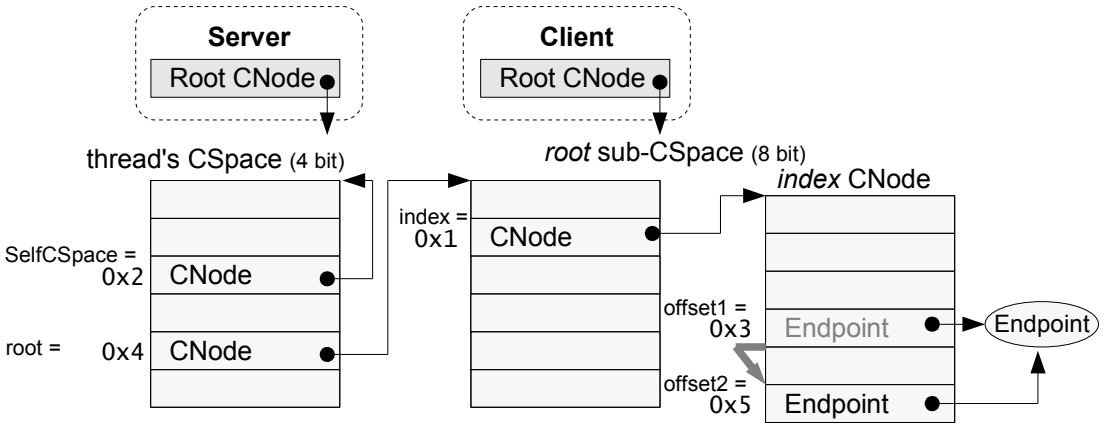

Figure 3.3: Moving an endpoint capability. To specify the endpoint capability-slot as the server, set root to 4, index to 1, offset to 3 and depth to 8.

When doing so, the user specifies for source and destination respectively, four parameters as addresses into the guarded CSpace.

Moving capabilities in a CSpace:

root is a capability reference in the thread's CSpace and has to point to a CNode that starts the *root* sub-CSpace. To modify its own CSpace, a thread needs a capability in its CSpace, to the top-level CNode of its CSpace. This CNode capability is usually placed at constant capability reference `seL4_SelfCSpace := 2`. Most capability invocations will set root to `seL4_SelfCSpace`. However, a server with capabilities to client CSpaces, that is, CNode capabilities, can modify client CSpaces.

index is a capability reference in the root sub-CSpace and points to the CNode that the user wants to modify.

offset notifies the slot or entry in the index-CNode that the user wants to modify.

depth allows to stop the translation before the lowest level of the CSpace and enables a user to place capabilities in earlier CNodes. It denotes the number of bits translated during the lookup.

For an example, see Figure 3.3.

3.3.2 Kernel memory management - Untyped_Retype

Untyped objects can be cast into other kernel objects. To do so, the user invokes *retype* on an untyped object, specifying the type and size for the new objects, the number of new objects to create, and a range in one CNode to place these objects. For an example, see Figure 3.4. Space not used by the retype operation is lost to internal fragmentation. However, the untyped object can be revoked, resulting in the deletion of all derived objects. Deletion of individual derived objects increases internal

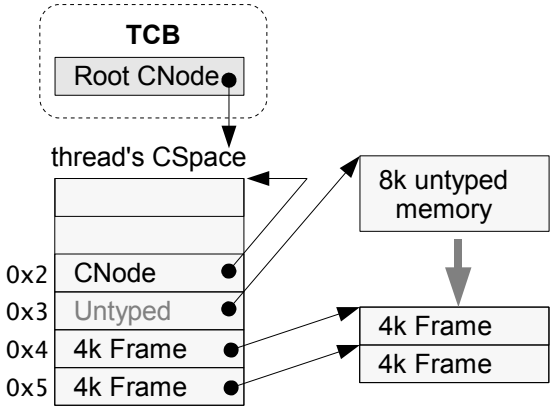

Figure 3.4: Retyping untyped memory. Transform 8k untyped-memory at capability reference 3 into 2 4k frames, create capabilities in the CNode with capability reference 2, in the range 4 to 5.

fragmentation. Untyped objects can be retyped into other, smaller untyped objects to prevent fragmentation.

3.3.3 VSpaces

seL4 does not have memory mappings via flexpages as L4 [HWL96] has. Instead, system applications have to build and manage their address spaces with hardware objects. However, they do not need to take care to not overwrite them accidentally.

To create a new VSpace, an application associates a page directory object with an *address space id* (ASID). It then maps page tables and pages into the VSpace at an address and with associated rights. Existing mappings will be overwritten, and one page capability can only be mapped into one VSpace. To share a frame, the frame capability must be copied, and the copy mapped into the other VSpace.

For an example, see Figure 3.5.

3.3.4 Threads, Endpoints and IRQs

seL4 user-space threads have similar semantics as L4 threads [Lie96]. A thread is identified by a capability to its *Thread Control Block* (TCB). A TCB allows setting of the address space, the capability space, the IPC Buffer and the scheduling priority (*ThreadControl*). Furthermore, the register set can be read and written (*ExchangeRegisters*), and the thread can be stopped (*Suspend*) and be started (*Resume*). The *IPCBuffer* is a shared memory page between the kernel and the user-mode thread to exchange parameters during system calls. A thread has to map the IPCBuffer into its VSpace to access it.

Endpoints are connector objects that, when copied to several CSpaces, can be used by respective threads to exchange messages and capabilities via IPC.

Capabilities to endpoints, like all other capabilities, have permissions associated with them. Endpoint capabilities without *seL4_Grant* cannot be used to transfer capabilities,

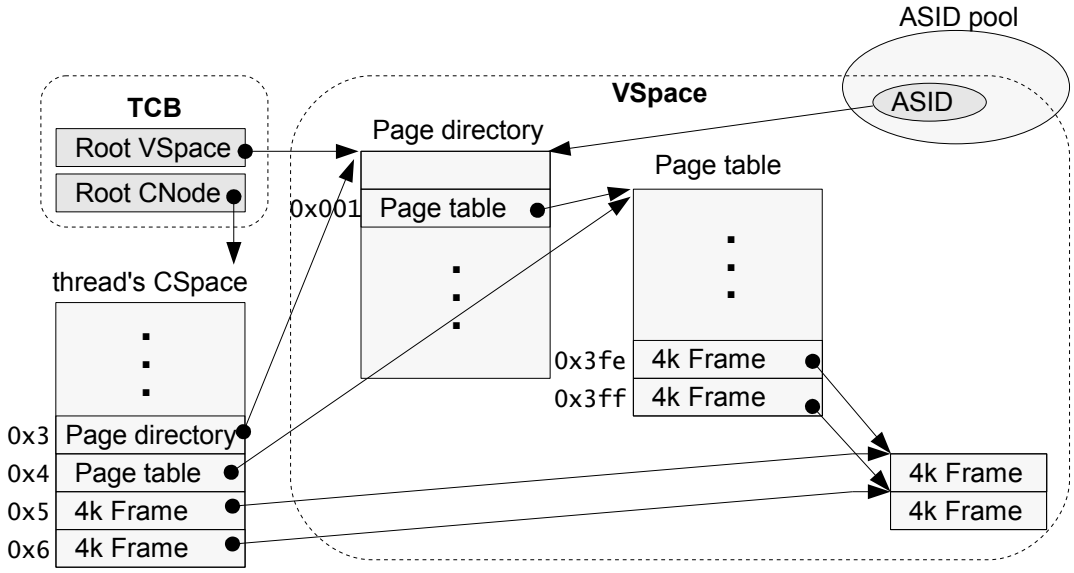

Figure 3.5: Example with seL4 address space objects

endpoint capabilities without *seL4_Read* cannot be used to receive from the endpoint, and endpoint capabilities without *seL4_Write* permission cannot be used to send to the endpoint.

In addition to permissions, endpoint capabilities have 32-bit *badges* that allow a program to distinguish between different senders on one endpoint. To use a badge, a server creates a copy with badge of an endpoint, and then moves the copy to the client. When the client sends to this endpoint capability with badge, the server receives the badge back.

seL4 supports synchronous and asynchronous endpoints. Synchronous endpoints are used for conventional, blocking or non-blocking L4 IPC with message transfer. Asynchronous endpoints are used for non-blocking send and only modify one buffer word of the endpoint by XOR-ing the bits of the sender's badge and the buffer word. This allows a receiver to distinguish between 32 senders. When reading from an asynchronous endpoint, the buffer word contains all bits set since the previous read and is reset.

Each thread also has a synchronous fault endpoint. When the thread pagefaults, traps or executes an undefined instruction, seL4 sends an IPC to the fault endpoint containing the registers associated with the exception. To continue the faulting thread, the fault handler thread has to reply with an IPC with new registers or restart the thread.

To use a service, clients call an endpoint associated with a server. Calling translates to synchronous sending and immediate waiting.

With the IRQ control object, individual IRQ objects can be created. Individual IRQ objects can link to an endpoint and send interrupts as asynchronous IPCs to this endpoint. To make it possible to receive an interrupt again, the user has to invoke the individual IRQ's acknowledge operation.

3.3.5 Start up

At startup, seL4 loads the initial program's ELF-file and starts an initial thread. This thread has an initial CSpace that contains capabilities to all the thread's objects, to all hardware control objects, and to all untyped memory objects (see Figure 3.6).

The initial CSpace is a 2-level CSpace with a big level-1 CNode and several level-2 CNodes with 256 entries, aka 8-bit translation, each. However, for historic reasons, level-2 CNodes translate 12 bit, to look like pages in a 2-level page table. Hence, level-2 CNodes have a 4-bit guard with value 0h and are 4096 capability slots distant from each other. (That leaves gaps of 3840 (=4096-256) slots).

To inform the thread about the semantics of its capabilities, seL4 provides a static *bootinfo* data structure at a fixed address. Many drawbacks of the bootinfo's coding make using it a nightmare. Entries in this array generally describe ranges of capabilities with the same semantics.

The following semantics are defined by the seL4 bootinfo for regions of the initial CSpace:

seL4_Region_Empty a region not covered by L1 or L2 CNodes

seL4_Region_RootTask frame objects for bootinfo, IPCBuffer and code

seL4_Region_L1Node number of entries and guard of the L1 CNode

seL4_Region_L2Node number of entries and guards of the L2 CNodes

seL4_Region_FreeSlots free slots in L2 CNodes

seL4_Region_InitCaps the initial thread's TCB, CSpace, VSpace and hardware control objects

seL4_Region_SmallBlocks 4-kB untyped memory objects

seL4_Region_LargeBlocks sets of different power-of-2 sized untyped memory objects

seL4_Region_DeviceCaps frame objects to I/O device memory

3.3.6 x86 and IO protection

seL4 does not use hardware protection for I/O ports (see Section 2.4.2). Instead, I/O ports are mapped to special system calls. To invoke these system calls, a thread needs the `seL4_IA32_InitIOPort := 7` capability.

seL4 uses IOMMUs (see Section 2.4.3) to give devices secure, direct memory access [Pal09]. seL4 only allows one PCI device per IOMMU address space. The seL4 kernel already sets up root tables for all IOMMUs and context tables for all PCI buses. To create an IOMMU address space object, the user needs the IOMMU control object and invokes the create operation with the address (bus, device, function) of the PCI device. The user can then map IOMMU page tables and frames into the address space.

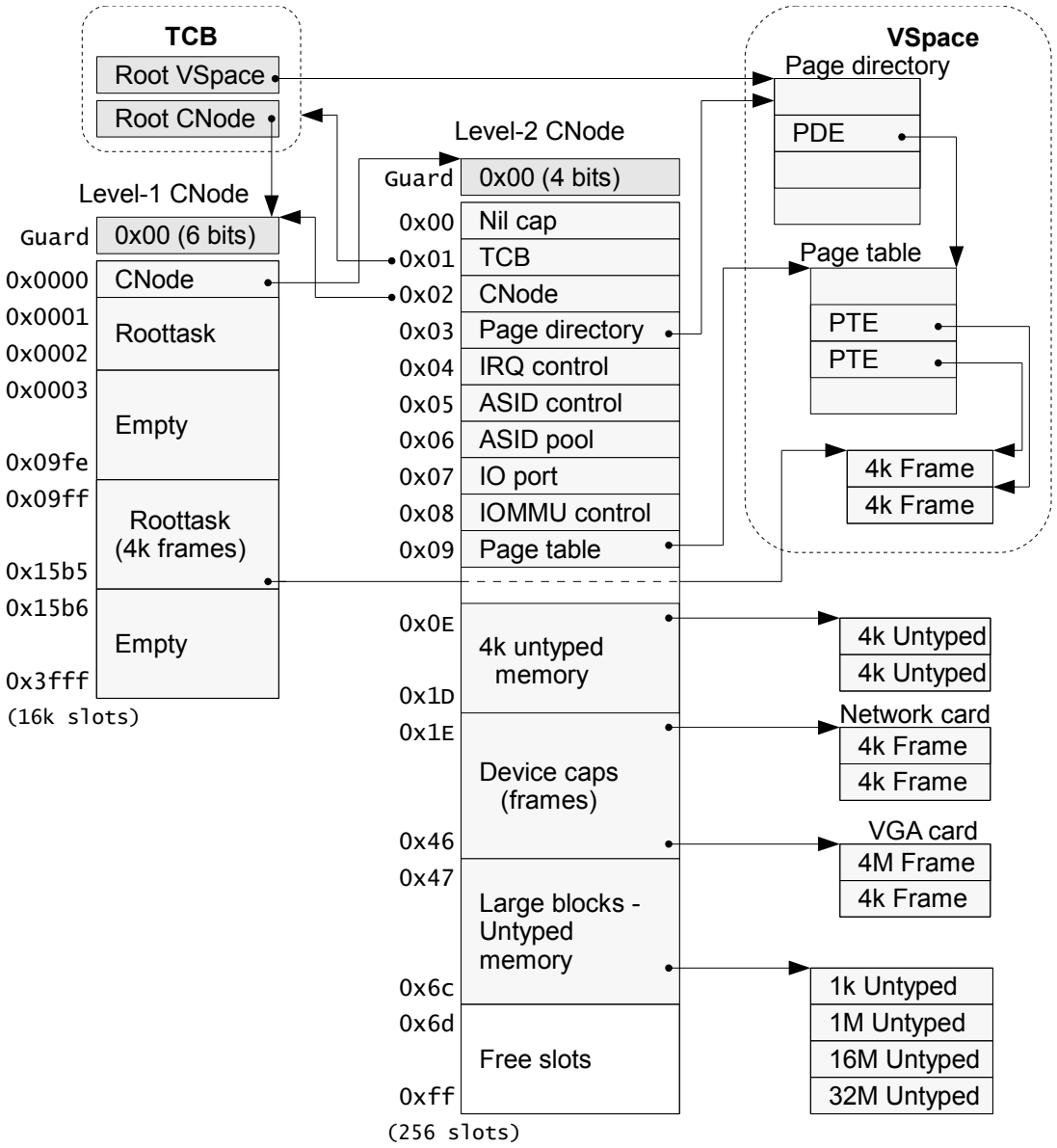

Figure 3.6: seL4 initial CSpace on QEMU

3.3.7 Devices

In the current IA32 version of seL4, the kernel does PCI device discovery (see Section 2.4.4) at boot time. However, it only reads out the BARs and creates frame capabilities for every I/O page. These capabilities end up in the CSpace of the initial thread and are reported to it as DeviceCaps in the bootinfo (see for example Figure 3.6) Regions with contiguous physical addresses are merged to one capability region in the bootinfo. To identify them, bootinfo entries contain the first physical address of the I/O memory region. As device regions often span several, for example 64, pages, several devices fill up the 256-entry L2 CNodes. As it happened that caused regions of one device to span two L2 CNodes. Finally, in the initial CSpace (see Section 3.3.5), L2 CNodes have gaps between them, so contiguous physical ranges are represented by non-contiguous capability ranges.

A user program is supposed to do PCI device discovery as well and parse the bootinfo to match the BAR-reported I/O region with frame capabilities. To do PCI device discovery, a user program uses the seL4_IA32_InitIOPort capability.

3.4 Current implementation status

At the time this thesis was started, the major challenges introduced by the new seL4 concepts already had detached solutions. There was a paravirtualised version of Linux that used a SLAB-like memory allocator [Bon94] (Iwana). Also small examples demonstrated the use of PCI device discovery and direct memory access via the IOMMU [Pal09]. In this section, I will explain the object allocator and paravirtualisation in more detail.

3.4.1 Iwana - the kernel object allocator

The Iwana memory library [Elk09] is closely interwoven with the startup program. This program handles the initial CSpace and transforms it into a much easier CSpace. The Iwana allocator is initialised on that CSpace and used to load Linux. Linux in turn uses Iwana to create processes and to manage their address spaces.

In contrast to the initial CSpace (see Figure 3.6), Iwana requires a CSpace where the level-1 CNode and the level-2 CNodes both translate 10 bits. It only support 4 kB and 1 MB untyped memory objects. Capabilities for each memory type must be in one contiguous capability region. The startup program creates that Iwana-CSpace by using ad hoc object allocation. Using seL4 compile time options, the developer ensures that the initial program has enough objects for the ad hoc allocation.

After setting up the Iwana-CSpace, the initial program stores information about that CSpace into a static *bootinfo* data structure at a fixed address. It then starts a new thread in the new CSpace. The thread then initialises the Iwana allocator with the range of 4 kB untyped memory objects, the range of 1 MB untyped memory objects, and the location of free slots in the L1 CNode and the current L2 CNode.

After initialisation, programs can use Iwana to allocate any seL4 objects: thread control blocks, endpoints, page directories, page tables, frames, CNodes, and slots. Iwana internally allocates CNodes to grow the CSpace as necessary and retypes untyped

memory in larger chunks of 256 objects at a time. It stores the affected capability references in pools per object type and hands them out when requested. Although objects can be freed into these pools, they are never revoked into untyped memory.

3.4.2 Paravirtualisation - seL4::Wombat

The ERTOS group has a paravirtualised version of Linux [Elk09] that follows the L4Linux approach [HHL+97] and is based on Wombat [LvSH05]. This version uses the Iwana allocator and was the only application loadable by the startup program. After initialising Iwana, the startup program loads the Linux ELF-image into a VSpace that it creates on-the-fly. For simplicity reasons, Linux uses the same CSpace as the startup program. After starting Linux, the startup program assumes the role of the timer-server. The timer-server, too, is in the same CSpace. Linux will initialise the Iwana library in a free area of this CSpace.

When Linux is started, it allocates a heap with a known mapping from physical address to capability reference, allocates and starts the system call thread that initialises the architecture independent part.

Processes are implemented as individual seL4 threads when they are in user mode and execute on the L4Linux system call thread when they are in the Linux kernel. Kernel entry and exit are mapped to fault IPC and reply. As such, Linux maintains endpoints for all its processes and waits on them whenever it has scheduled the corresponding process.

When this thesis was started, paravirtualisation of processor and memory was working, however Linux did not have access to I/O devices. The only device access was an interface to an I/O port based timer server and the seL4 kernel's debug putchar system call.

3.5 Application to this thesis

In this chapter I presented seL4. The seL4 microkernel employs new design approaches that require proper handling at user level. Object capabilities allow fine-grain control over resources and can prevent covert channels. However, they require handling of local names and local name spaces. Also, I/O devices must be identified and capabilities to access device resources must be created. To create kernel objects, the user must manage physical memory. To create compartments, an entity has to create all required kernel objects and assemble them. When instantiating an information flow architecture, compartments must get access to their program, to device resources, and to other compartments via defined interfaces. The next chapter will present my solutions to these tasks.

Chapter 4

Design

The task for this thesis is to compartmentalise an x86 system using the seL4 microkernel. The goal is to achieve the strongest compartmentalisation possible by design. Therefore, standard OS services and features are cut back and replaced by a build time specification mechanism. The resulting system is static and cannot change at run-time. However, a specification mechanism shall allow to describe different hardware-software compositions.

I extended an existing seL4 startup program with support for I/O devices and multiple compartments. This program can now be parameterised by data structures that describe I/O devices and compartments.

In this chapter I will develop a secure approach to device discovery and show how I built seL4 compartments and how these compartments are parameterised. I will first provide my solution for PCI device discovery by the OS. Then I show how a similar mechanism can be used in untrusted, deprivileged, paravirtualised guest operating systems. I will then give an approach to secure PCI configuration. In the second part of this chapter I will derive the specifications that are required to instantiate secure architectures.

4.1 Handling I/O devices

In an information flow architecture a computer consists of I/O devices that require special consideration. Four kinds of interaction between devices and compartments are important: device discovery, device control, data transfers, and I/O interrupts. seL4 already provides the last three operations, the device communication channels, to compartments. However, it lacks understanding of what a device is (as appropriate for a microkernel). To give a compartment access to a device, the loading process has to discover devices on the machine and collect their associated seL4 resources. It then has to give these resources to the compartments.

Device discovery is in conflict with a completely static system. I therefore propose to treat x86 systems as embedded systems for security purposes. Instead of dynamic discovery and usage of devices, the system developer has to specify all I/O devices he wants to use. The loading process then assumes these devices are present and fails otherwise.

The OS learns about PCI devices by scanning the PCI bus (see Section 2.4.4). To dispatch PCI devices to compartments, the loading process has to know the communication channels each device uses. For each device, the PCI configuration space contains

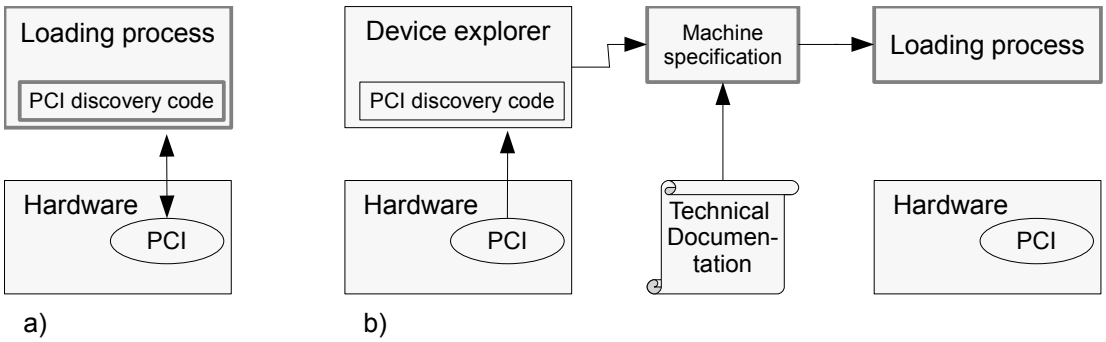

Figure 4.1: Offlining PCI device discovery: a) Device discovery code is in the trusted computing base, b) only the machine specification has to be trusted.

this information. The allocation of I/O resources to PCI devices is performed by the BIOS on system startup.

4.1.1 PCI device discovery by the system

The loading process is responsible for splitting the resources between compartments. As such it is in the trusted computing base and must be correct. To show correct device discovery, one must have a formal model of PCI and show that the loading process uses it correctly (see Figure 4.1, a). Under the assumption that PCI discovery always returns the same result for the same machine, this result can be saved and reused. That way, device discovery can be removed from the loading process. By offlining access to the actual hardware, verification of the loading process is simplified. The offline device discovery yields a copy of the PCI configuration space. This representation is then analysed, confirmed, and linked into the binary of the loading process (see Figure 4.1, b).

The prototype implementation showed that this design is feasible and that PCI devices can be operated without prior access to their configuration spaces. To ease machine specification, I developed the *deviceexplorer* tool to read out the PCI configuration space of a machine. It runs on Linux and accesses the /proc/bus/pci virtual directory to read device information. It then formats this information according to a device specification and writes it into a C-header file. I will present this specification in Section 4.2.1 of this chapter. The developer then assigns identifiers to the devices he wants to use. In the architecture description the developer uses these identifiers to grant a compartment access to a device.

4.1.2 PCI device discovery by compartments

Compartments, too, must have a way to discover the I/O devices they consist of. Because the PCI configuration space can also be used to modify the I/O resources and has no protection concept, hardware access to it cannot be given to isolated compartments. Instead, we require another way to transfer device descriptions into a com-

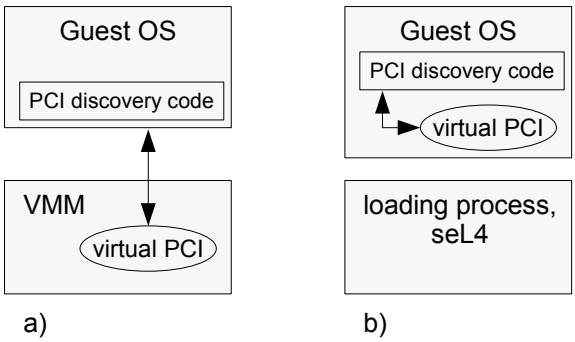

Figure 4.2: PCI virtualisation. a) classical: emulation by VMM, b) emulation inside the VM compartment

partment. For custom made components, device information can be encoded in a data structure or the initialisation code.

To make a paravirtualised OS use a set of devices, the developer can configure the OS to only have drivers for these devices. For PCI devices, one additionally needs to make the OS activate the drivers. In a non virtualised OS, this activation is done by the PCI discovery code that starts drivers for every device it finds. This same functionality is required in the virtual environment as well. Instead of scanning the hardware, the OS has to scan a software generated list of devices. It then has to activate the PCI drivers. However, the interface between the PCI code and PCI drivers is quite complex, is a moving target and generally belongs to the OS internals. A less intrusive approach is the paravirtualisation of the PCI device discovery code itself. Instead of replacing that code, you can intercept its access to hardware. By emulating reads and writes to the PCI configuration space, the existing code can be reused without changes. To do so, the paravirtualised OS must be provided with a copy of the configuration space of all devices it is supposed to drive (see Figure 4.2). Reads can then just return values of the copy. However, the discovery protocol also contains writes. On closer analysis, these writes are only required to access another static value, the I/O region sizes. In this case, the guest OS writes all 1's to each base address registers (BAR) of every PCI device. This action has to make the next read to the BAR report the encoded size of the I/O region. Implementing these two mechanisms, read from copy, and emulate write to BARs, proved to be enough to paravirtualise PCI device discovery.

4.1.3 PCI configuration

PCI is not only used for device discovery, but also to configure individual devices. For devices that require configuration via PCI, a secure way to access the PCI configuration space must be devised. I will present three designs.

First, the system developer can try to avoid using devices that require PCI configuration access. Under the assumptions that the configuration set up by the BIOS is secure, this approach reduces the complexity of the necessary trusted software. However, it might be impossible to avoid these devices because certain features cannot be config-

ured by the BIOS. Examples include message signalled interrupts [SA99] and DMA modes of popular disk controllers [Int07].

Second, the developer can offline PCI device configuration when his system does not change device configurations at run time. Many device drivers only configure devices when they are started. A developer could therefore extract the configuration part from the driver and move that code to a trusted instance like the loading process. The loading process then configures the device before it starts the untrusted OS. The OS then only accesses a local copy of the PCI configuration space. To make the OS think it is controlling the device, the PCI virtualisation layer might be required to emulate effects of configuration space writes.

Third, if configuration access is necessary during the runtime of the system, the developer has to include a secure server that controls writes to the PCI configuration space. That server needs access to the SEL4_IA32_InitIOPort capability (see Section 3.3.6) and needs to have an interface that allows confining access to individual devices (see also [Hän07]). Similar to the seL4 kernel interface, it could provide a control capability that allows creating capabilities for single device objects. A secure instance then creates these capabilities and hands them over to untrusted components. Such capabilities can be implemented with endpoints (see also [LW09]) and it would be the servers responsibility to ensure that requests on one endpoint only access the associated device. Additionally, the developer has to decide, whether to export byte access to the configuration space or just a set of functions like `enable_PCI_COMMAND_MEMORY()` and `set_PCI_PM_CTRL(int value)`. However, most settings determine the consumption of PCI resources by the device. As these resources are also global, they cannot be controlled by an untrusted compartment. For example, access to the PCI BARs determines the mapping of device memory into the physical address space. If an untrusted application can change this mapping it can capture accesses to memory (see Section 2.4.4).

For the prototype I chose the first design, because the devices I used did not require PCI configuration.

In this section I handled device discovery and configuration. I showed that it is useful to separate PCI device discovery from PCI device configuration. I also moved virtualisation of device discovery from the virtual machine monitor into the virtual machine compartment. Thereby I reduced the complexity of the VMM and made a global PCI discovery service unnecessary. The next section will handle further device operations and the modelling of compartments.

4.2 Architecture specification

This section describes the manifestation of an information flow architecture in an x86-seL4 system. I define the relevant partitions of a computer (compartments) and derive a method to specify them. This specification is then processed by a loading process. The main tasks of the loading process is to split resources and to instantiate an architecture on top of the seL4 microkernel. This thesis is about the second part. To instantiate all the compartments I needed to find out their minimal resource requirements. The remainder of this section will show what seL4 resources are required to control a device,

to create different types of compartments, and to connect compartments. Furthermore, I will give the developed specifications as well as the interface by which compartments get to know about their resources.

4.2.1 Device specification

To assemble the seL4 resources of a device, the loading process must know the identity and communication channels of the device. It can acquire them via PCI device discovery or via a static configuration.

To assemble the seL4 resources of a device, the following information is required:

- the physical addresses of the I/O memory regions to control the device,
- whether the device is controlled via I/O ports,
- the IRQ number to receive I/O interrupts from the device,
- the PCI device address to create an IOMMU address space.

When this information is acquired, the loading process has to create or find the corresponding seL4 capabilities.

The following seL4 capabilities are required to use an I/O device:

- a range of frame capabilities to I/O memory of the device, mapped into a VSpace
- the seL4_IA32_InitIOPOrt capability for I/O port access
- an IRQ capability and an endpoint capability associated with the IRQ
- a device-specific IOMMU address space id, IOMMU page tables and frame capabilities mapped into this IOMMU space and a VSpace.

To reduce the trusted computing base I chose a static configuration. The developer uses another OS and my deviceexplorer tool to dump the PCI configuration space into a list of I/O devices of the target machine. Using this list, the loading process acquires the relevant capabilities and copies or maps them to the compartments driving the devices. To maintain the meaning of the newly created capabilities I derived a data structure that stores device capability references. This data structure contains sections for each device operation. Each section describes both the physical property of the device as well as the corresponding seL4 capabilities.

The first section is about device identification. Currently only PCI devices are handled. A PCI identifier contains the device address (source id) and a copy of the PCI configuration space (see Listing 4.1).

Memory mapped I/O regions are characterised by a range of physical addresses and a range of capabilities. For later purposes, flags and seL4 permissions were also required in this specification (see Listing 4.2).

The complete device specification (see Listing 4.3) contains up to 6 MMIO regions, the PCI structure and capability references for the IRQ and the IOMMU space. The

```
typedef struct {
    unsigned int pci_source_id;
    unsigned int config_space[64];
    unsigned int bar_size[6];
} wombat_pci_device_t;
```

Listing 4.1: PCI device identification data structure

```
struct seL4_resource {
    uint32_t            start;
    uint32_t            end;
    unsigned long       flags;
    seL4_CapRights      rights;
    seL4_CPtr           cap_start;
    seL4_CPtr           cap_end;
};
```

Listing 4.2: I/O region data structure

IO port capability is not mentioned because it is always at the same capability reference. However, the device specification states whether a device requires IO port access, whether it uses DMA, and whether it can send IRQs. The IRQ number is also included.

Using the device specification, the loading process assembles the seL4 resources and assigns them to a compartment. To report the device and its seL4 resources to the compartment, the loading process copies the device specification to the compartment's bootinfo.

4.2.2 Compartment specification

To create a compartment in seL4 is to assemble all the necessary kernel objects. Different compartment types require different seL4 resources. Resource requirements are based on four classes: minimal isolated domains, device driver domains, virtual machine domains, and compartment connections.

Minimal isolated domains

For every isolated seL4 application, the following capabilities must be assembled:

- a Thread Control Block (TCB), to execute the application,
- a VSpace, filled with enough page tables and page frames, to
  - hold the program,

```
typedef struct seL4_device {
    /* A numeric identifying name for this device. */
    enum device_name name;

#define SEL4_DEVICE_USES_DMA     0x00000002
#define SEL4_DEVICE_HAS_IOPORTS 0x00000004
#define SEL4_DEVICE_HAS_IRQ      0x00000008
#define SEL4_DEVICE_VIRTUAL_IRQ 0x00000010
    unsigned int     flags;

    unsigned int     irq;
    unsigned int     irq_cap;

    unsigned int     iommu_space_cap;

#define SEL4_DEVICE_IS_PCI       0x00000000
    unsigned int     type;
    union {
        wombat_pci_device_t pci;
    } id;

    struct seL4_resource resource[6];

} seL4_device_t;
```

Listing 4.3: Device description data structure

  - hold the IPCBuffer, and to
  - hold the stack, and

- an empty CSpace to confine authority of the application.

Most of these properties are determined by the application binary. The existing loader has a function to build a VSpace from an application binary. Binaries are identified in a combined ELF-file by a string. To parameterise loading of a program, I refer to the binary's file name. For a sample specification of a simple compartment, see Listing 4.4.

Device driver domains

Device driver domains have all the requirements of a simple application. In addition, they need access to devices. As discussed previously, the following is required to drive a device:

- a description of the device and its seL4 capabilities,

```
component_t app = {
    .name = "Simple App"
    .binary_name = "app.elf"
    .CSpace_depth = 4,
};
```

Listing 4.4: Example minimal compartment specification

```
component_t networkserver = {
    .name = "Network server"
    .binary_name = "network-server.elf"
    .CSpace_depth = 4,

    .devices = {HARDWARE_NIC_0, },
};
```

Listing 4.5: Example device server specification

- the I/O memory of the device mapped into the VSpace, possibly involving page tables,
- the I/O port access capability,
- the IRQ capability and the associated endpoint capability,
- an IOMMU address space that shares memory with the driver's VSpace.

The device-specific capabilities to create all these resources are already referenced by the device specification. Device specifications in turn are referenced by an identifier. To notify assignment of devices to a compartment, I gave the compartment specification a list of device identifiers. For a sample specification of a device server, see Listing 4.5.

Virtual machine domains

Virtual machine domains also have the requirements of seL4 applications and of driver domains, when they drive devices. In addition, they must be able to manage and control compartments on their own. Therefore, they need access to many seL4 objects. To keep the complexity of the loading process small, the virtual machine should create all these objects itself. Therefore, a virtual machine domain requires the following:

- write-access to its CSpace,
- untyped memory to create seL4 objects,

```
component_t networkrouter = {
    .name = "Wombat 1"
    .binary_name = "vmlinux.elf"
    .CSpace_depth = 20,

    .devices = {HARDWARE_NIC_0, HARDWARE_NIC_1},

    .smallUT = 9, //(i.e 9 * 4k)
    .largeUT = 21, //(i.e. 21 * 1MB)
    .cmdline = "init=/etc/init.d/dropbear0",
};
```

Listing 4.6: Example virtual machine specification

- a big, or a grow-able CSpace,
- the ASID control capability,
- a capability to its own VSpace, and
- a description of its initial capabilities and initial CSpace and VSpace.

To mark a compartment as virtual machine domain, I use the size of the CSpace in the compartment specification. A `CSpace_depth` of 20 signifies to the loading process to create an Iwana-compatible 2-level CSpace. The loading process will also give the compartment the required, above mentioned, capabilities. Of these requirements I parameterised the amount of untyped memory. I also added a command-line string to the compartment description. This parameter can be used to start virtual machines with different kernel command-lines. For a sample specification of a virtual machine, see Listing 4.6.

For the entire compartment description data structure, see Listing 4.7.

4.2.3 Connection specification

To connect seL4 domains, the following options exist:

- connect via shared seL4 objects, like endpoints, TCBs, CNodes, page tables, and so on;
- connect via shared hardware objects, like memory frames, devices.

To maintain isolation, only shared memory and non-grant endpoints are acceptable (see Section 3.1). Because connections do not belong to one of the connected parties, I treat them as separate objects in the specification.

A connection object can be shared between multiple compartments. To model this, the object can be modelled as a passive compartment and links between the object and

```
typedef struct {
    char *name;
    int id;
    loader_DeviceIDs_t devices[5];
    char *cmdline;
    char *binary_name;
    char *ramdisk_name;
    unsigned int CSpace_depth;
    unsigned int smallUT;
    unsigned int largeUT;
    unsigned int deviceOffset;
    //internal
    struct app_info *app;
} component_t;
```

Listing 4.7: Compartment specification data structure

a compartment denote mapping of the object into the compartment (see Figure 4.3, a). To ensure information flow control, mappings can be restricted to read-only or write-only access to the shared object. For simple two-way connections, this model adds overhead, that is, specifying the object and two mappings, instead of one specification object. Therefore I used a simpler version. In my model, connections are specified for two compartments only (see Figure 4.3, b).

In the specification (see Listing 4.8), a connection item specifies the two compartments it connects. Furthermore, I developed two connection types, one based on endpoints, the other based on shared memory. A type parameter specifies this type. The next parameter encompasses type-specific information and the final two parameters specify the seL4 permissions each compartment has to the objects of this connection.

In addition to setting up the connection, the loading process also has to inform both domains about their communication capabilities. Later argument will show that I model connections close to I/O devices. I therefore overloaded the list of I/O devices in a compartment's bootinfo to also describe compartment connections.

The problem with select

When a compartment has several endpoint connections, it might want to wait on all connections at the same time (`select()`). Especially in a client-server scenario, the server wants to wait for all clients. This can be achieved with a multithreaded server that has one thread per endpoint, or with an endpoint shared between the server and all clients.

A shared endpoint poses a problem to the compartmentalisation (see Figure 4.4, a). When an endpoint is shared between a server and its clients, the clients can communicate via this endpoint. To prevent this unintended information flow, the shared endpoint

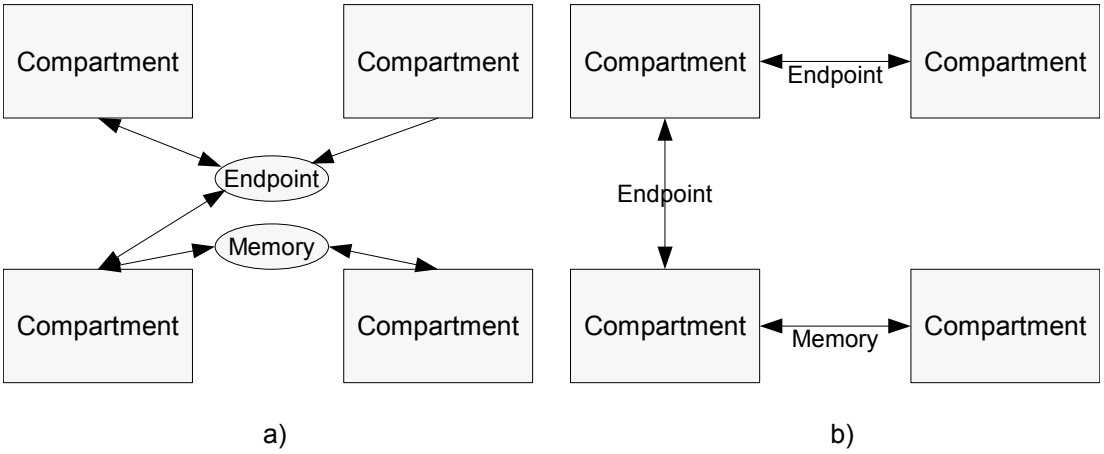

Figure 4.3: Connection options. a) Objects shared between multiple compartments, b) Connection between two compartments

```
typedef struct {
    component_id_t      from;
    component_id_t      to;
    connection_types_t  type;
    union {
        unsigned int        protocol;
        unsigned int        irq;
        void *              dev;
    } data;
    seL4_CapRights      from_rights;
    seL4_CapRights      to_rights;
} connection_t;
```

Listing 4.8: Connection specification data structure

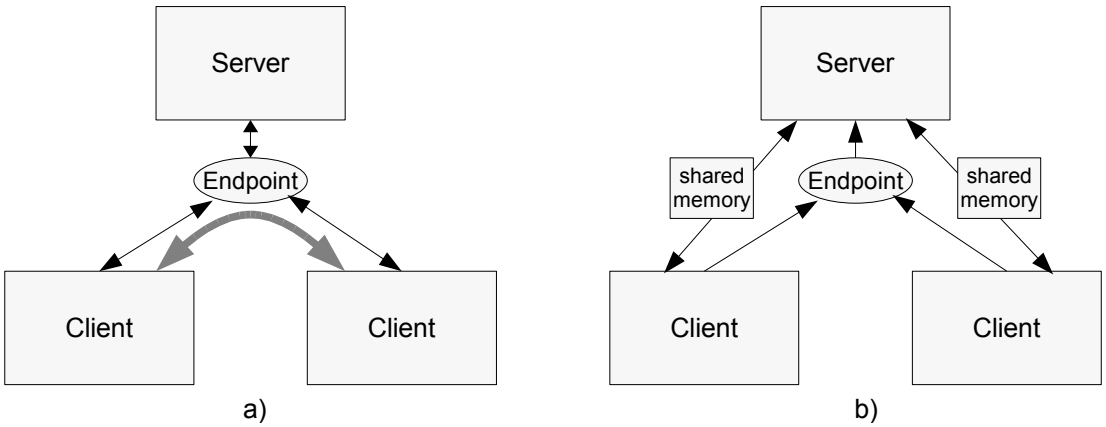

Figure 4.4: a) Conflict of select and compartmentalisation. b) Solution

must be read-only or write-only for all clients. This restriction prevents the classical call-reply server protocol. My conclusion from this observation is that the current seL4 API does not allow call-reply with a shared endpoint without enabling communication between all partners.

However, the unidirectional notification mechanism, that is, write-only clients, is still useful, if a client and a server are connected via multiple links (see Figure 4.4, b). Compare the situation to an I/O device that has I/O regions shared with a compartment and can send an interrupt to the compartment. The compartment can receive interrupts from this or other devices, however devices cannot send interrupts to each other.

I disregarded the multithreaded server as being too complex a solution for this problem. Instead, I use shared notification endpoints for client-server scenarios.

Asynchronous endpoint connection

Following the above discussion, I decided for a model where each compartment can be a server waiting for several clients at once. Also, a compartment can send notifications to several servers. Therefore, each compartment has one endpoint to receive from several clients and multiple endpoints to send to different servers. For one compartment I call these endpoints the *incoming endpoint* and the *outgoing endpoints* (for an example, see Figure 4.5).

Note also that a write-only endpoint has a covert back-channel, if it is a synchronous endpoint. Although a sender may only write to an endpoint, it can determine whether the receiver is waiting or not. To enforce unidirectional connections, I have to use asynchronous endpoints.

An asynchronous endpoint does not allow data transfer. Instead, non-blocking IPC only modifies one buffer word of the endpoint by XOR-ing it with the bits of the sender's badge. When reading from this endpoint, the buffer word contains all bits set since the previous read. The server uses these bits to distinguish between different clients. In addition, the incoming endpoint is used to receive I/O interrupts. In my model, clients

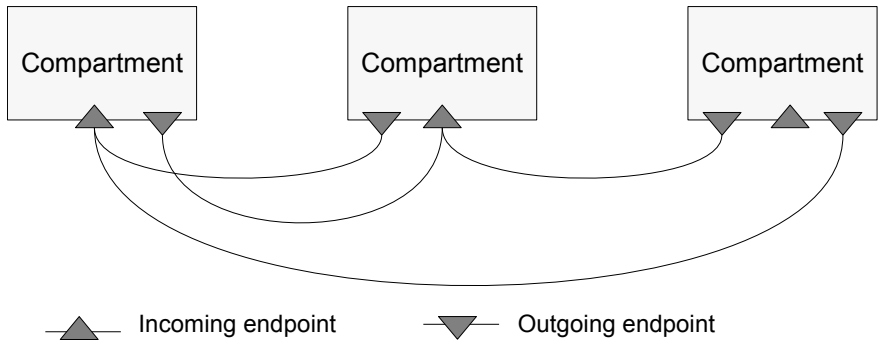

Figure 4.5: Example of endpoint connection model

```
struct seL4_outgoingEP {
    seL4_CPtr       cap;
    unsigned int    protocol;
};
```

Listing 4.9: Outgoing endpoint reporting data structure

are represented to the server as connections. A connection specifies the "IRQ number" as the bit the server will see when somebody uses this connection.

When specifying an endpoint connection, the 'from' compartment is the client id, and the 'to' compartment is the server id. The server receives client requests as virtual interrupts, whereas the client has an endpoint to send notifications. The data parameter of an endpoint connection specifies the IRQ number the server associates with this client.

An extension to the current endpoint connection could allow several incoming endpoints to allow for the call-reply scheme between two compartments.

When a compartment is the target (`to`) of an endpoint connection, the loader will create a new device entry in the compartment's bootinfo. The loader sets the device's IRQ number and sets the `flags` field to `HAS_IRQ` and `VIRTUAL_IRQ`. This tells the interrupt code to correctly dispatch this notification but to not acknowledge an interrupt.

For the `from` compartment, the loader creates a new entry (see Listing 4.9) in the outgoing endpoints list in the compartment's bootinfo.

Shared memory connections

To connect compartments via shared memory, I have to devise virtual addresses in the VSpaces to which to map the memory. To allocate virtual addresses I have two options: Reserve a portion of the virtual address space and allocate ranges from that area. Or, ask the developer to provide virtual addresses. I decided for the second option to refrain from putting restrictions on the usage of virtual addresses.

The second task is to report the shared memory region to the compartment. When the developer devises the virtual addresses, he can also tell its program where each shared memory region is. That is, use the virtual addresses as identifiers for the regions. However, when he reuses his program in another context he has to adapt his program to the new memory layout. To increase modularity I propose to use identifiers for shared memory connections and to report shared region descriptors via the compartment's bootinfo. Also I want to allow a developer to specify multiple shared memory regions between two compartments in one connection object. I decided to represent shared memory as a virtual device, both in the connection specification and when reporting to compartments.

To allow a developer to specify virtual addresses for a shared memory object, I reused the device specification data structure (see Figure 4.3). The `data`-part of the connection data structure is therefore considered a pointer to a device data structure that contains an identifier and memory regions. The regions' memory is then allocated and mapped into the compartments at the specified addresses. Consequently, these device specifications are also used to report the shared memory connection to the compartment. Because of this specification method, I call shared memory connections *virtual device* connections.

In this section I showed how to model connection of compartments using endpoints and shared memory. I analysed the problem with shared endpoints and came up with a solution. I have devised a way to specify shared memory and a way to report connections to compartments. The next section will summarise the reporting facility.

4.2.4 Compartment bootinfo

When giving a compartment capabilities, the compartment also has to know their meanings. For that purpose, the loading process generates a static bootinfo that contains the necessary information. This bootinfo is a data structure (see Listing 4.10) that the loading process fills when creating the compartment. The loading process then copies the data to a fixed address in the compartment's VSpace.

The bootinfo first describe the capability reference ranges for untyped memory of 4kB and 1MB sizes. Then, it describes the CSpace layout, assuming a two-level CSpace, that is filled from 0 onwards, the first entry states the first free slot. The second entry states the first free slot in the level 1 CNode. Of the next entries, the command line reports the string the developer wants this virtual machine to use as kernel command line. The next entries are self-describing, the incoming endpoint and array and count of outgoing endpoints describe corresponding capabilities. The device count and array describe I/O devices and connections of the compartment. Note that the device descriptions contain copies of the devices' PCI configuration spaces and form part of the PCI virtualisation support.

This data structure concludes the design chapter. I have developed a secure solution for PCI device discovery that was not handled by seL4. I have designed models of seL4 compartments including I/O devices and connections. I explained what seL4 resources are required to set up these models and how to parameterise them. The next chapter will show how these compartments are instantiated.

```
struct bootinfo {
    /* Start / end of small untyped objects. */
    unsigned long small_ut_start;
    unsigned long small_ut_end;

    /* Start / end of large untyped objects. */
    unsigned long large_ut_start;
    unsigned long large_ut_end;

    /* CNode layout. */
    unsigned long cspace_free_slot;
    unsigned long cspace_free_L1_slot;

    /* Kernel command line. */
    char command_line[128];

    /* Endpoint to receive IRQ events. */
    seL4_CPtr incomingEP;

    /* Endpoints to e.g. timer-clients, network server */
    unsigned long outgoing_count;
    struct seL4_outgoingEP  outgoingEPs[8];

    /* Device information. */
    unsigned long device_count;
    struct seL4_device devices[8];
};
```

Listing 4.10: Compartment bootinfo data structure

Chapter 5

Implementation

This chapter illustrates the implementation part of this thesis. I will show individual steps of loading an information flow architecture and substantiate implementation decisions. I start with an in-depth discussion of the loading process. Then I discuss the other half, device access in seL4::Wombat. Afterwards, I resolve implementation issues of PCI virtualisation and conclude with the details of the timer server.

5.1 System initialisation

In the existing system, a bootloader loads the microkernel and the initial program into memory. The microkernel then starts the initial program. An ELF-combining tool concatenates multiple ELF files to the initial program. The first program then reads the combined ELF-file's header and loads additional ELF files. I extended this mechanism to make it instantiate a specified software architecture.

To transform the architecture specification into a running system, I first allocate and load the compartments. Then I give each compartment the device capabilities it needs. Afterwards, I allocate and connect communication channels. Finally, I start the compartments. Because the loading process has the highest scheduling priority, the compartments start when the loading process finishes.

5.1.1 The loading process

I based the loading process on an existing startup program (see Section 3.4.1). This program was used to load one instance of the paravirtualised Linux. The startup program uses two stages. It starts off with the initial CSpace in the setup stage. There it creates a new CSpace that is usable by the Iwana object allocator. The startup program then starts the loader-thread in this CSpace, initiating the second stage, the loading stage. The loader first initialises Iwana and then loads the Linux ELF image. Note that the first and the second stage use the same VSpace and therefore share code and data structures.

Considering device support, the existing startup program already transferred the generic hardware control capabilities into the second CSpace. However, it did not transfer the page frames of the I/O regions. When adding this functionality, the mapping of frame-object to physical address must be preserved. First however, the loader has to calculate this mapping from the initial CSpace layout (see Section 3.3.7).

In the design section I defined the list of devices of the target machine (see Section 4.2.1). The loading process uses this list to find all the I/O regions whose frame capabilities need transferring to the second CSpace. For each device in the list, each I/O region is processed. I use the existing implementation that, given a physical address range, finds the list of non-contiguous capabilities ranges. I then move the frame capabilities into a contiguous region in the second CSpace. Then I update the device's I/O region description with the new capability range. After this procedure, the second CSpace has all capabilities required to control the requested I/O devices.

5.1.2 Loading compartments

The loading process loads compartments by parsing the compartment-specification data structures and allocating and assembling corresponding seL4 objects. Another data structure is used to track the allocated objects during different phases of the loading process.

Creating the minimal compartment

First, the *Thread Control Block* (TCB), IPCBuffer and incoming endpoint are allocated. Then, the compartment's ELF file is located in the combined ELF file and loaded into a VSpace. The loader therefore allocates page tables and frames as necessary. Frames are first mapped into the loader's VSpace, filled with content and then mapped into the compartment's VSpace.

Building CSpaces

Afterwards, the loader creates a CSpace for the compartment. Two types of CSpaces are currently supported. The *simple CSpace* consists of one CNode with 16 entries and can be used for compartments that only have a maximum of 16 capabilities, that is, drivers that only have endpoints to clients and to hardware control objects. This CSpace does not contain an entry to its own CNode and hence cannot be modified. The *VM CSpace* is a two-level CSpace with the same layout as the Iwana CSpace to support the Iwana allocator. Initially, this CSpace is constructed with two 1024-entry CNodes, one for the first level, one for the second level. The second CNode's capability is copied into the first CNode at slot 0 to establish valid capabilities from 0 to 1024. To give the compartment authority over its CSpace, the first CNode's capability is copied into the second CNode at `seL4_SelfCSpace := 2` with all permissions and the 20-bit depth information. Also, the VSpace and TCB capability are copied into the second CNode at the standardised slots (`seL4_SelfTCB := 1`, `seL4_SelfVSpace := 3`).

Allocating capabilities

When the CSpace is initialised, the loader can fill the CSpace with the requested capabilities. To do so, the loader has to devise capability references in the new CSpace. Many of them were already defined by the existing program, for example slots 20 to 99 for 4k untyped memory, slots from 100 for 1MB untyped memory. While these ranges

work for standardised anonymous objects, endpoints and hardware control objects have individual semantics. I already defined data structures to track these semantics, however, I also had to allocate capability references for them. Because these references can be chosen freely in a free range, I implemented a simple allocator with a moving free-pointer. For each CSpace type, the allocator is initialised with the range of free slots. On every allocation, the free-pointer is moved by the number of requested capabilities. This allocator is also used in other places, for example to track the loader's untyped memory objects that are allocated to compartments.

Adding optional objects

In the next step the loader copies untyped memory objects to the compartment. The required amounts for 4k and 1MB-sized objects are specified in the compartment description. Because a compartment has to know the number of objects, it is here also that the loader updates the compartment's bootinfo with the ranges and amounts of copied objects.

Then, the loader connects devices to the compartment. Therefore it loops through the compartment's array of assigned device identifiers and uses them to index the global device array. It then calls a function to connect this device to this compartment. I will describe this function in the next subsection.

Finally, the loader copies the incoming endpoint into a free address in the compartment's CSpace. It also updates the compartment's bootinfo. In the next step connections are handled, after device access in the next subsection.

5.1.3 Mapping devices to compartments

To allow a compartment access to a device, the four device operations must be accessible to the compartment.

Device discovery and configuration

For identification, I copy the device specification into the compartment's bootinfo. I use the compartment's device counter to index into the compartment's device array. Also I keep a reference to this object to update it with compartment-local capability references.

The prototype implementation does not contain support for secure access to PCI configuration. For uncontrolled access, the compartment can use I/O ports to configure PCI devices.

I/O Interrupts

If the device has an interrupt, I have to give the compartment the IRQ's capability and an endpoint associated with this IRQ. I allocate a free slot in the compartment's CSpace, update the bootinfo with this slot, and copy the IRQ capability to the compartment's CSpace. Note that the IRQ capability is derived from the IRQ control capability using the IRQ number. As IRQ endpoint I use the compartment's incoming endpoint. To

allow the compartment to distinguish different IRQs, I associate this IRQ with a badge. Therefore I have to create a copy with badge of the endpoint capability. The badge has only the IRQ number's bit set. Then I associate this badged endpoint capability with the IRQ capability. Thereby, the IRQ is linked to the incoming endpoint.

I/O Ports

For devices that are controlled using I/O ports, the seL4_IA32_InitIOPort capability is required in the compartment. When copying it into the CSpace at the standardised slot, I have to make sure it is copied only once. Note also that the current seL4 implementation does not allow to compartmentalise access to individual I/O ports. Therefore, a compartment with this capability can control many devices and the PCI bus. However, for testing and development, it is vital to have I/O port access in compartments.

Memory-mapped I/O

When giving a compartment access to memory-mapped I/O regions, there are two options. Either I can copy the frame capabilities into the compartment's CSpace, or I can map the frame capabilities into the compartment's VSpace. The first option removes code from the loader and gives the compartment more control over these frames. However, a compartment now needs to access its CSpace and VSpace to map the frames into its VSpace. Also, the compartment might need page tables to establish the mapping and untyped memory to create the page tables. While all these requirements are fulfilled by seL4::Wombat, this option adds requirements and complexity to simple driver servers. The second option allows to use driver servers that contain only minimal seL4 code. However, with this option, a VM cannot map I/O memory regions to compartments it creates on its own. Despite this drawback I chose the second option.

To map I/O regions into the VSpace, I now have to devise virtual addresses for the I/O regions. Because I virtualise both PCI discovery and device addresses, I can almost freely choose an address for the I/O region. However, I decided to not change the PCI configuration space information and to keep the original physical addresses as virtual addresses. Note that on the testing machine, those were in the range `0xf0000000 - 0xffffffff`. This same range is also occupied by the seL4 kernel that reserves this part of each compartment's VSpace as kernel address space (256MB). I therefore devised the range `0xe0000000 - 0xefffffff` for memory mapped I/O device regions and implemented a simple offset of `-0x10000000` from physical to virtual addresses. Compartments have to respect this offset to access I/O regions. For Linux, a central place to install this offset exists.

For mapping I/O regions in the VSpace, an existing helper function tests whether mapping of page tables is necessary, and if so, allocates and maps them, prior to mapping the page. The bootinfo is not updated, because the addresses are kept. However, it is used to determine the addresses where to map the pages to.

Data transfer - DMA

To enable a compartment to have DMA data transfers with an I/O device, the compartment's VSpace must share frames with the device's IOMMU space. To create the IOMMU space, a program needs the device's IOMMU-address-space capability, IOMMU page tables and useful frame capabilities, that is, frames that are or can be in the compartment's VSpace. Similar to the argument for memory-mapped I/O regions, either the loader can set up this IOMMU address space or it can be made a task of the compartment. In this case I decided for the compartment option, because this reduces complexity of the loader more than is the case for MMIO regions and because of the assumption that compartments using DMA-capabilities of devices are implemented using VMs. Hence, giving a compartment DMA control over a device, consists in creating the IOMMU ASID from the device's PCI address and copying this capability into the compartment's CSpace. Also, the compartment's bootinfo is updated with the newly allocated IOMMU ASID capability. For more information on IOMMU address spaces, see the next section.

After these operations, compartments have all capabilities to drive I/O devices. However, they still need glue code to use them. Also, the loader has to connect compartments to establish an information flow architecture.

5.1.4 Establishing connections

After the loader has assembled compartments, it connects them. Therefore it loops through the list of connections and establishes each connection. It uses the connection's `from` and `to` field to find the compartment's data structures and seL4 objects. Each connection type is handled differently.

For the endpoint connection, the loader finds the already allocated incoming endpoint of the `to` compartment. Then it copies the incoming endpoint to a newly allocated slot in the `from` compartment's CSpace. Thereby it assigns a badge to the endpoint using the IRQ number of the connection. It reports this connection as a new device to the `from` compartment and as new outgoing endpoint to the `from` compartment.

For the shared-memory virtual-device connection, the loader parses the device description and allocates the required amount of pages for each I/O region of the device. Simultaneously, it updates the capability ranges in the I/O region. Then it connects this device to one compartment, using the method described above. Afterwards it creates copies of the frame capabilities and connects the device to the other compartment.

5.1.5 Starting compartments

When all generic seL4 objects are allocated and assembled, the loader will start the applications. Prior to this, for virtual machines, it copies the command line into the compartment's bootinfo and copies the `seL4_InitASIDPool` capability. Note that sharing of the initial ASID pool also provides a covert channel. Future work should allocate separate ASID pools for all compartment.

To start the application, the loader maps into the compartment's VSpace, the stack, the IPCBuffer, and the bootinfo. It assigns the VSpace and CSpace, IPCBuffer and

priority to the TCB. Then it sets the TCBs registers to the ELF-file's entry point and the stack position, and starts the compartment with this operation.

5.2 Device access in seL4::Wombat

The existing seL4::Wombat version already had paravirtualisation code that interacted with Iguana [Hei05]. My work was to find these places and replace them with seL4 equivalents. Note that the Wombat/Iguana system was based on a request system, where I/O-region requests from Linux drivers were forwarded and served by Iguana. In my thesis, I/O regions are premapped. The I/O region request functions [CRKH05] therefore only calculate the offset. Furthermore, I mapped I/O port functions to the seL4 capability invocations.

I implemented an initial version for I/O interrupt dispatching and acknowledging. The interrupt thread waits on the incoming endpoint and reads the badge on receipt of an IPC. It then calls the Linux interrupt handler. Also it has to acknowledge the interrupt by invoking the acknowledge operation on the IRQ capability. Initially, it finds this capability in the bootinfo, later it uses an array that maps IRQ numbers to IRQ capabilities.

Linux initiates DMA for objects on its heap. During an I/O operation, Linux will ask the paravirtualisation code to give an I/O device access to a part of the heap. While a lazy approach of mapping only the requested portions of the heap into the IOMMU address space has its merits, I decided to premap the Linux heap into each IOMMU address space Linux controls. One of the first actions of the paravirtualisation code is the allocation of the heap, with a fixed mapping from virtual address to capability reference. Later in the initialisation code, I added functionality to create IOMMU spaces. Therefore I loop through all devices in the bootinfo, get the IOMMU ASID capability, calculate from the heap size the required number of I/O page tables, and allocate and map them into the IOMMU space. Then I create copies of the heap's frame capabilities and map them into the IOMMU space at the same virtual addresses. Therefore I also had to extend the Iwana object allocator with support for allocation of I/O page tables.

When first activating I/O device drivers in the existing seL4::Wombat version, a strange Linux kernel page fault had to be solved. Further analysis revealed that arbitrarily data structures were overwritten and indicated a memory corruption. Digging through the driver code while trying to determine the first occurrence of this fault revealed that unrelated data structures had the same addresses. Eventually I could pinpoint this to the allocation of arrays in the driver's code that were allocated using vmalloc() [CRKH05]. In the seL4::Wombat code, the vmalloc() area was set to the heap region. I resolved that problem by reducing the heap region and reserving a part as vmalloc() area. Note that the seL4::Wombat kernel cannot resolve its own page faults.

With these provisions in place, seL4::Wombat can access I/O devices. To initialise PCI drivers, I gave Linux access to the PCI bus at first and prevented it from using

other devices by a hack. Later I removed that requirement by implementing virtual PCI discovery.

5.3 Virtual PCI configuration space

When paravirtualising PCI device discovery, a few implementation problems have to be solved. Following the design (see Section 4.1), the paravirtualised OS is intercepted at the `pci_write_config` and `pci_read_config` interface. Accesses specify the bus, the device, the function, and the address in the function's configuration space; also the size of the access (8-bit, 16-bit, 32-bit) and a value to set or to return is specified. On the other hand, a list of device specifications contains the copies of the devices' configuration spaces. A translation has to map the guest OS's access to the device list. This translation should be efficient, because it has to be invoked on every configuration access during device discovery.

A simplistic translation uses only the device specification list as data structure. On every access the glue code walks through this list and checks whether the current device entry has the requested PCI address. If so, it performs the requested operation on the entry in the list. If no device is present, the emulation returns a value of all 1's, corresponding to the discovery protocol. This approach does not scale with bigger numbers of devices but has a low memory overhead.

A simple fast translation uses lookup tables to translate accesses. It would need one table with 256 entries for each bus, one table for each bus with 32 entries for each device and one entry for each device with 8 entries for each function. The function entries then point to the device specifications in the device list. To reduce the space consumption of this device tree, paths that do not lead to devices can be pruned. The lookup process then has to check whether an entry points to a table before it dereferences the pointer.

I used a translation with lookup tables and a pruned device tree. This tree is built in the startup code of the paravirtualised OS. Later the paravirtualisation code redirects the configuration space accesses through the lookup tables to the device list entries.

This way, I made the performance overhead of PCI-device-discovery paravirtualisation constant. While this feature is not performance critical, it reduces the loading time of seL4::Wombat. I estimate that the loading time is already reduced because of using memory instead of I/O devices as store for device information While the constant number of accesses is similar to the simplistic translation, it should scale better with more devices. However, I did not evaluate this prediction because my focus was implementability of the high-assurance design and not performance.

5.4 Timer server

seL4::Wombat relies on a hardware time source for scheduling and keeping time. The existing system already had a timer server similar to the one in Iguana [Hei05]. However, the server was not compartmentalised and relied on residing in the same CSpace as seL4::Wombat.

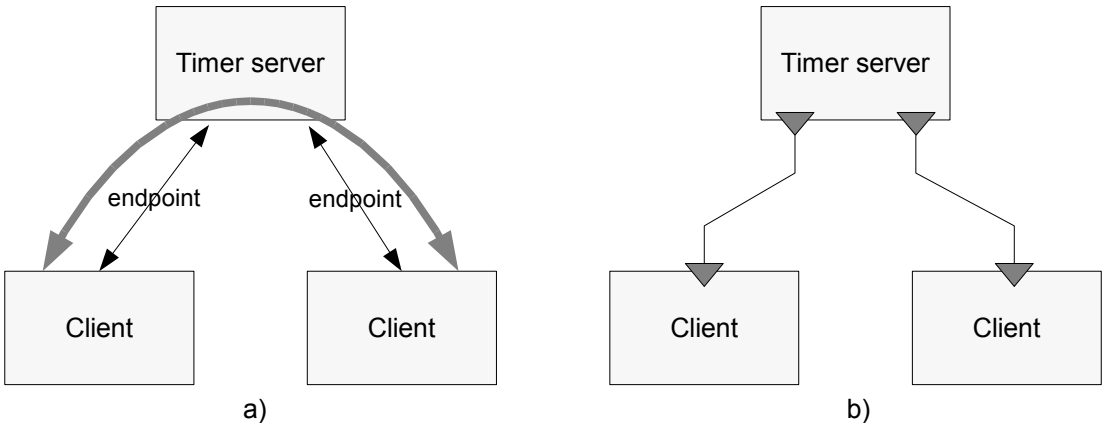

Figure 5.1: Information flow of the timer server. a) The timer server must be trusted to not transfer information, b) Isolation is independent from the timer server

To run multiple instances of seL4::Wombat, each instance requires access to a time source. To maintain isolation between these instances, they cannot share direct access to the timer hardware. Instead, access to the timer hardware must be compartmentalised. That is, arbitrary numbers of individual instances must be able to program timeouts and receive notifications independently and isolated from each other. Such a functionality is not provided by current x86 hardware. Instead, the original timer server uses the programmable interval timer of the chipset [Int07] that receives timeouts on IRQ 0. The timer server then sends IPCs to endpoints that it had previously created for clients registering a timer.

Looking at the information flow of the existing timer server, clients that program a timeout send data to the server and clients that receive return-values or timeout-IPC's receive data from the server. Hence, the timer server must be trusted to not transfer information from one client to another (see Figure 5.1, a). To alleviate this situation, I propose a simpler timer server that does not allow programming. Instead, it is configured to send timeout IPC's at a preconfigured rate to a programmed set of clients. This way, information only flows from the timer server to its clients. No information can flow via the timer server from one client to another (see Figure 5.1, b).

To drive the timer hardware, the timer server requires several seL4 capabilities. It needs the SEL4_IA32_InitIOPort capability to access the timer's registers, the IRQ capability of IRQ 0 to acknowledge the interrupt, and an endpoint capability associated with the IRQ capability to receive interrupt IPCs. To notify its clients, the timer server requires one endpoint for each client. These endpoints have to be asynchronous and write-only for the timer server. Otherwise, clients can send information to the server by selectively waiting or not waiting on the synchronous endpoint.

The required seL4 capabilities are available if the timer server is specified as seen in Listing 5.1 in the architecture specification.

When started, the timer server reads out the capability references and the number of clients. It then programs the timer hardware and waits for timer interrupts. On every interrupt it sends IPC's to each of its client endpoints.

```
SYSTEM_COMPONENTS[COMPONENT_TIMER] = (component_t){
      .name = "timer-server",
      .devices = {HARDWARE_TIMER, },
      .binary_name  = "timer-server.elf",
      .CSpace_depth = CLIENT_CSPACE_BITS,
};

SYSTEM_CONNECTIONS[0] = (connection_t){
      .from = COMPONENT_TIMER,
      .to = COMPONENT_WOMBAT_1,
      .type = INCOMING_EP,
      {.irq=0},
      .from_rights = seL4_CanWrite,
      .to_rights = seL4_AllRights};
```

Listing 5.1: Specification of timer compartment

With the timer server I showed how to compartmentalise access to a simple platform device. This approach does not require verification of the timer server and relies solely on the correctness of the compartmentalisation mechanisms. However, the functionality of the timer server is reduced. Future work has to determine whether this type of time source is sufficient.

This chapter showed individual steps of loading an information flow architecture. I covered details of the loading process, device access in seL4::Wombat, implementation issues of PCI virtualisation, and the timer server. The following chapter will evaluate the achievement of my goals.

Chapter 6

Evaluation and Future Work

In this chapter I will revisit my goals and evaluate how I achieved them. Then I show limitations of my approach and give an outlook to future work.

6.1 Evaluation

Security

The task for this thesis was to compartmentalise an x86 system using the seL4 microkernel. The goal was to achieve the strongest compartmentalisation possible by design. To evaluate this goal I will summarise my approach here. I started by defining how a compartmentalised system should look like. Then I showed how the x86 architecture supports compartmentalisation and how an operating system can use these mechanisms. Afterwards I introduced the seL4 microkernel and showed how it uses the x86 mechanisms. I also highlighted the parts that still needed compartmentalisation. For one of them, device discovery, I developed an approach that has a low runtime-code complexity. Then I showed how to achieve a compartmentalised system by assembling compartments only of the required seL4 resources and I/O resources. Compartments as developed in the design chapter are isolated except for explicitly specified communication channels. In contrast to other systems, compartments cannot use resource requests, I/O devices, the timer server, or the PCI bus to break the compartmentalisation. While I am confident about these claims, I cannot prove them. Future work to develop a high assurance system has to analyse the correctness of my work.

In addition to the assumption that my code is correct, I make assumption about the BIOS. For the correctness of device discovery, I assume the BIOS creates a compartmentalised PCI device configuration. I also assume this configuration does not change across reboots.

Complexity

The trusted computing base of my compartmentalisation solution encompasses the seL4 microkernel and the loading program. To give an impression of the complexity of these components I will use the *source lines of code* (SLOC) count.

The seL4 microkernel has about 8700 SLOC. The original loading program had a 1916 SLOC count. It is dependent on the Iwana library that adds about 1500 SLOC. From the original loading program I removed the timer server that has now 364 SLOC.

This leaves 1552 SLOC for loading VSpaces and building the Iwana CSpace. My loading program now has 2260 SLOC. So I added 708 SLOC to support I/O devices and compartmentalisation.

Because of my decision to use static device discovery I did not require PCI device discovery code in the trusted computing base. This decision removed otherwise necessary 700 SLOC of similar projects. However, I only worked on a small variety of devices. Other devices and features will make this code necessary.

When adding virtualisation support for PCI discovery I chose to emulate PCI configuration space inside the virtual machine. The resulting code only required 65 new SLOC in the paravirtualised Linux. Hence this extension is small and easy to maintain. Because I am intercepting hardware accesses, I do not even touch Linux software infrastructure.

Performance

I did not develop a new component for a well evaluated system. Instead, my thesis adds a fundamental component for the new field of compartmentalised systems on seL4. Performance of these systems is expected to be in the range of previous microkernel-based systems. I did not conduct measurements to validate this assumption, because I expect the performance to be dominated by the seL4 microkernel's design decisions.

During testing however, the paravirtualised Linux on a Core 2 Duo 2.33 GHz with an Intel PCIe Gigabit Ethernet card was reactive via a serial console and via an ssh connection.

Functionality

A goal of this thesis was to leverage the properties of the seL4 microkernel that employs new approaches to kernel resource management. This feature removes the necessity for a memory management server. To leverage this property, the initial seL4 program has to split and distribute kernel memory between compartments. Different approaches to solve this memory allocation problem in seL4 have not resulted in a usable system. Only the Iwana memory allocation library was sufficiently sophisticated and developed to provide a basis for this thesis. Choosing Iwana allowed me to concentrate on instantiating compartmentalised systems.

Extensibility

One main goal of this thesis was to systematically support compartmentalised applications on the seL4 kernel, including compartmentalising access to hardware devices. To achieve this goal, I developed the loading process that can be parameterised with the specification of a static compartmentalised application. It also includes compartmentalisation support for hardware devices that are connected via I/O ports, memory-mapped I/O regions, interrupts, and DMA. That is, legacy x86 devices, PCI devices, and ACPI-reported devices.

To demonstrate the generality of this approach I will present two subsequent works that were based on my work. In his honours project, Robert Sison [Sis09] developed a

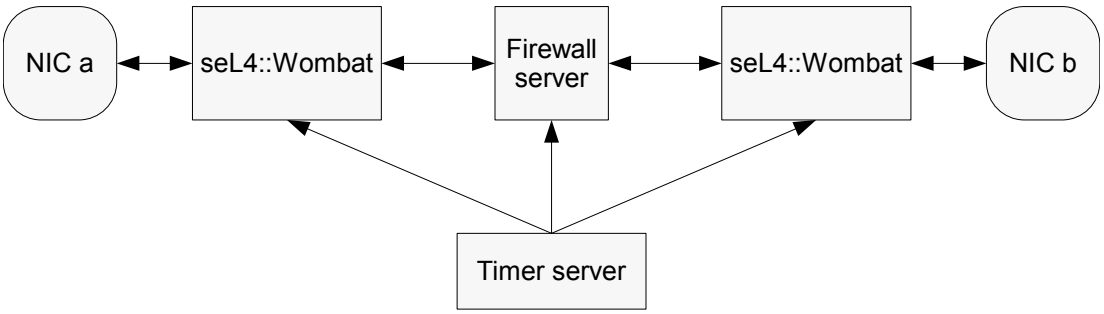

Figure 6.1: Architecture of secure firewall

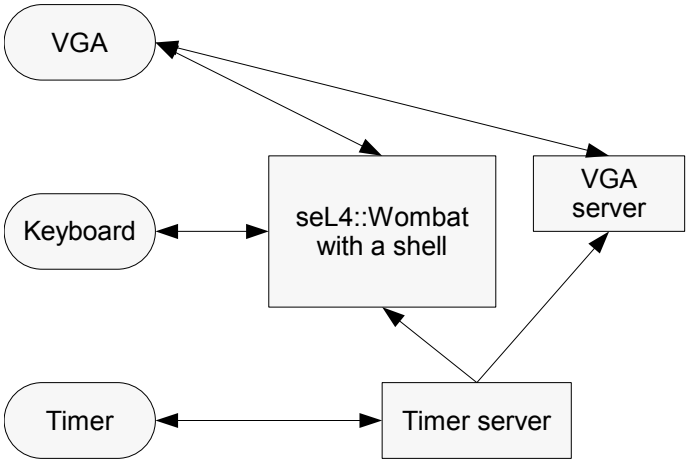

Figure 6.2: Architecture of SOSP demo

secure firewall that contains two Linux instances each connected to one network card and to each other via a trusted firewall compartment (see Figure 6.1). His work made use of the hardware compartmentalisation functionality and the endpoint and shared memory connections. Also the timer server was used to implement the timeout of network connections in the firewall. To instantiate his architecture he only had to specify it using the data structures developed in my thesis.

The ERTOS group presented the seL4 work at SOSP and included a demo. This demo ran a Linux instance next to a VGA server on a notebook (see Figure 6.2). Therefore it was required to specify the x86 keyboard and VGA devices. This was possible with minor additions and compartments were then able to use these devices. Note that Linux and the VGA server were sharing the VGA device, however, the VGA server was unaffected by restarts of Linux. Additionally, the VGA server also used the timer server to show time elapsed since bootup. Instead of producing an implementation only valid for this specific demo, the demo resulted in a reusable toy VGA server and evaluated the requirements for a secure terminal (VGA and keyboard access).

6.2 Limitations

Connections

In the design chapter I derived ways to connect seL4 compartments and a method to specify these connections. When analysing the design spectrum I only found three useful and secure connection types. However, I only implemented two of them. Synchronous endpoints cannot be used in the current version.

Considering the abstraction of connections my solution has two drawbacks. seL4 can connect compartments in a finite number of ways, using a shared object and the capabilities that connect the object to the CSpaces of the compartments. I modelled connections as links between two compartments. A more generic model would model connections as connections, thus include as separate objects, shared objects and mappings of shared objects to compartments. This model allows to specify connections between multiple compartments and multiple endpoints per compartment. In the design chapter I showed that the resulting connections conflict with compartmentalisation or do not lend themselves to a single-threaded server scheme. However, for mutually trusting compartments a call-reply connection or globally shared memory reduces complexity. Also, a compartment does not have to wait on all its endpoints at the same time or can use multiple threads to do so.

To specify shared memory connections I use the virtual device abstraction and require the developer to provide virtual addresses. In hindsight I think these two decisions were wrong. Component developers often need not care about virtual memory layout but have to with my solution. Also, virtual addresses in the architecture specification introduce a dependency between the architecture and the component that reduces modularity. I also think that developers do not need multiple shared memory regions between two compartments. A simpler specification could therefore only state the size and an identifier for a shared memory region.

Limitations of the approach

Static system

The current design trades adaptability and online reconfiguration for simplicity in an attempt to increase trustworthiness. The analysed system is entirely static in what it expects from the hardware and in what it creates in terms of software components. Once started, changing the architecture is not possible.

Static device discovery

The current system uses offline device discovery. When that information is assembled into the loader binary, the loader can only run on machines closely resembling the specified machine. For faster development, it is useful to have only one binary execute on different machines. That can be achieved by adding device discovery code to the loading process. That code adds to the trusted computing base and the overall size of

the trusted code. However, its functionality is fairly isolated and does not interact with other components.

Information flow architectures

This thesis is limited to static information flow architectures. Only information can flow between components, not resources. Therefore it is only useful for single purpose machines that do not change their functionality at runtime.

6.3 Future Work

In this section I show problem areas that involve my work and have to be solved to deliver a secure and versatile high-assurance system.

Verification

Several steps are necessary to analyse the correctness of my work. The goal of this analysis would be, to show that my code indeed transforms a specified information flow architecture into a compartmentalised x86 system. Therefore, a formal model has to describe the I/O channels and I/O devices of the x86 architecture and the PCI bus. This model also has to contain the definition of compartment and it has to be decidable whether a given hardware configuration is compartmentalised, that is, divided into isolated partitions. A next model has to show how seL4 interacts with the I/O hardware model and how the seL4 API controls this interaction. My program instantiates an architecture by performing seL4 system calls. Given an architecture specification and a model of my program, these system calls can be calculated. Applying the sequence of systems calls to the seL4 API model will create a sequence of transitions in the seL4 kernel model, which will in turn create a sequence of transitions in the hardware model. Starting from an initial state in the hardware model, this sequence will create a new hardware configuration. Using the compartmentalisation test, we could now show that we created a compartmentalised hardware state.

Dynamic information flow architectures

This thesis is based on static information flow architectures. This allows to build systems that only contain the kernel and untrusted run-time components. Future work should also investigate using trusted run-time components to enable dynamic information flow architectures.

In such systems, untrusted compartments would still only have explicit data-transfer connections to other compartments. Thus, the same class of secure information flow graphs can be instantiated. However, that loading process does not have to run at startup time only. Instead, the system specification should be modifiable at runtime and incur a change of the current compartment configuration. That way an authorised user of the system can set up a new use case.

Because a dynamic system will deviate from its initial state, a full specification at development time is impossible. Instead, a new component, the compartment manager, has to maintain the system configuration. It also has to accept configuration changes and organise transitions between configurations. The compartment manager has to bookkeep resource information for compartments to destroy them and to reuse the resources. To set up compartments, it also requires compartment descriptions similar to the ones defined in this thesis. Otherwise it could not create isolated compartments that do not request resources.

I/O compartmentalisation

A further challenge is the compartmentalisation of more I/O devices. In general, this will incur device servers that compartmentalise I/O devices for compartments.

For the simple timer device, I showed a compartmentalisation solution that does not require verification. This solution was based on unidirectional information flow. Future work should investigate whether a similar mechanism is possible for other I/O devices that, conceptually, are input-only or output-only. That is, keyboard, mouse, sensors for input-only devices, sound, graphics for output-only devices.

Network and storage devices are conceptually bidirectional and require a more advanced compartmentalisation solution. One compartment would directly control the device and provide virtual devices or a similar service to client-compartments. The device server has to ensure that no information can flow between its clients. Therefore I propose to not allow untrusted compartments to create, modify, delete, or list virtual devices. Instead, these operations should be reserved for trusted components.

My thesis analysed I/O compartmentalisation for the PCI bus. The current design only supports I/O devices that are statically connected via I/O ports, MMIO, DMA, and interrupts. Modern device classes exist that use more abstract, software-like interfaces. Examples are USB devices, Intel network cards implementing VT-c technology and graphic cards supporting Microsoft DirectX 10. These devices provide compartmentalisation for I/O channels similar to the device servers I just described and can result in faster and less complex I/O compartmentalisation.

In this chapter I evaluated the achievement of my goals, showed limitations of my approach and presented my ideas for future work. The next chapter will conclude.

Chapter 7

Conclusions

In my thesis I developed a key component of future seL4 systems. This component is responsible to instantiate a compartmentalised software architecture on the seL4 microkernel. I showed what architectures secure high assurance systems require and how microkernel-based architectures can be described in component systems.

One requirement of security architectures is the compartmentalisation of the underlying hardware. I showed how both the x86 architecture and the seL4 microkernel support compartmentalisation of the system. I further showed how to handle the resources both mechanisms provide. I therefore developed a description of seL4 compartments including I/O devices and explicit compartment connections. I implemented a loader component that interprets these descriptions and instantiates the specified compartments.

Of the four operations of I/O devices, x86 and seL4 already compartmentalise device control, data transfer and I/O interrupts. In this thesis I proposed to make device discovery static to reduce the interdependability of system components. I demonstrated that this idea can be implemented for the operating system and for compartments. For compartmentalised operating systems I showed a mapping from the OS's native device discovery to the static device list. For static device discovery I developed specifications for I/O devices that comprise the four device operations and describe the x86 I/O channels as well as the corresponding seL4 capabilities.

I showed that seL4 compartments have five classes of requirements. I showed how minimal compartments look like and argued that they are dominated by their address space. Then I showed that for a compartment to access a device the compartment must have the capabilities referenced in the device specification. Virtual machine compartments manage subcompartments. Therefore they need access to their capability space, address space, and to seL4 kernel memory. I analysed compartment connections in seL4 and devised secure connections based on message passing and shared memory. Finally, I developed a reporting facility to tell a compartment its capabilities.

I evaluated my goals and concluded that my work can be used as a central component in a seL4 system. I also argued that my design decisions removed many attack vectors of common operating systems. However, I can only hope that I did not miss other vulnerabilities. Therefore I outlined how future work could formalise the x86 architecture and proof correctness of my results.

Glossary

ACPI advanced configuration and power management interface

API application programming interface

ASID address space identifier

BAR base address register (PCI)

BIOS Basic Input Output System

CNode capability table (seL4)

CPU central processing unit

CSpace capability space (seL4)

DMA direct memory access

DRAM dynamic random access memory

ELF executable and linking format

IDL interface description language

IO input–output

IOMMU input–output memory management unit

IPC inter process communication

IRQ interrupt request

MILS multiple independent levels of security and safety

MMIO memory-mapped I/O

MMU memory management unit

OS operating system

PCI peripheral component interconnect

seL4 secure embedded L4 microkernel

TCB thread control block

VSpace (virtual) address space (seL4)

Bibliography

[AFOTH06] Jim Alves-Foss, Paul W. Oman, Carol Taylor, and Scott Harrison. The MILS architecture for high-assurance embedded systems. *International Journal on Embedded Systems*, 2:239–247, 2006.

[AJM+06] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3):179–192, 2006.

[APJ+01] Mohit Aron, Yoonho Park, Trent Jaeger, Jochen Liedtke, Kevin Elphinstone, and Luke Deller. The SawMill framework for VM diversity. In *Proceedings of the 6th Australasian Computer Systems Architecture Conference*, Gold Coast, Australia, January 2001. IEEE CS Press.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, Anaheim, CA, April 2005.

[Bon94] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Technical Conference*, Boston, MA, USA, Winter 1994.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly & Associates, Inc, 3rd edition, 2005.

[EDE08] Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. Kernel design for isolation and assurance of physical memory. In *1st Workshop on Isolation and Integration in Embedded Systems*, pages 35–40, Glasgow, UK, April 2008. ACM SIGOPS.

[Elk09] Dhammika Elkaduwe. *A Principled Approach To Kernel Memory Management*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, February 2009.

[Elp04] Kevin Elphinstone. Future directions in the evolution of the L4 microkernel. In Gerwin Klein, editor, *Proceedings of the NICTA workshop on OS verification 2004, Technical Report 0401005T-1*, Sydney, Australia, October 2004. National ICT Australia.

[FAH+06] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and

reliable message-based communication in Singularity OS. In *Proceedings of the 1st EuroSys Conference*, pages 177–190, Leuven, Belgium, April 2006.

[FH06] Norman Feske and Christian Helmuth. Design of the Bastei OS architecture, subsequently called Genode OS Framework. Technical report, TUD-FI06-07-Dezember-2006, TU Dresden, 2006. 6, 74, 2006.

[FHN+04] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.

[GJP+00] 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 SIGOPS European Workshop*, pages 109–114, Kolding, Denmark, 2000. ACM Press.

[Hei05] G. Heiser. *Iguana User Manual*. National ICT Australia, 2005.

[Hel01] Christian Helmuth. Generische Portierung von Linux-Gerätetreibern auf die DROPS Architektur. Master's thesis, TU Dresden, Germany, July 2001. URL `http://os.inf.tu-dresden.de/papers_ps/helmuth-diplom.pdf`.

[Hew06] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., Toshiba Corporation. *Advanced Configuration and Power Interface Specification*, 3.0b edition, October 2006. URL `http://acpi.info`.

[HHF+05] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza secure-system architecture. In *colcom05*, San Jose, CA, USA, December 2005.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 66–77, St. Malo, France, October 1997.

[HLM+03] Hermann Härtig, Jork Löser, Frank Mehnert, Lars Reuther, Martin Pohlack, and Alexander Warg. An I/O architecture for microkernel-based operating systems. Technical Report TUD-FI03-08, TU Dresden, July 2003.

[HWL96] Hermann Härtig, Jean Wolter, and Jochen Liedtke. Flexible-sized page-objects. In *Proceedings of the 5th IEEE International Workshop on Object Orientation in Operating Systems*, page 102, Seattle, WA, USA, October 1996.

[Hän07] Lukas Hänel. ACPI for L4Env. Student thesis, TU Dresden, Germany, June 2007. URL `http://os.inf.tu-dresden.de/papers_ps/haenel-beleg.pdf`.

[Int07] Intel Corp. *Intel I/O Controller Hub 8 (ICH8) Family*, 2007. URL `http://www.intel.com/Assets/PDF/datasheet/313056.pdf`.

[Int08a] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manuals*, 2008. URL `http://www.intel.com/products/processor/manuals/index.htm`.

[Int08b] Intel Corp. *Intel Virtualization Technology for Directed I/O*, 2008. URL `http://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf`.

[KEH+09] 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 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[KLGH07] Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems*, 80(5):687–699, May 2007.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, 1973.

[LCFD+05] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting (Rita) Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, September 2005.

[Lie94] Jochen Liedtke. Page table structures for fine-grain virtual memory. *IEEE Technical Committee on Computer Architecture Newsletter*, 1994.

[Lie96] Jochen Liedtke. *L4 Reference Manual — 486/Pentium/PentiumPro, Version 2.0.* GMD, Schloß Birlighofen, Germany, September 1996. Working Paper 1021.

[LvSH05] Ben Leslie, Carl van Schaik, and Gernot Heiser. Wombat: A portable user-mode Linux for embedded systems. In *Proceedings of the 6th Linux.Conf.Au*, Canberra, April 2005.

[LW09] Adam Lackorzynski and Alexander Warg. Taming subsystems: capabilities as universal resource access control in L4. In *2nd Workshop on Isolation and Integration in Embedded Systems*, Nuremburg, Germany, 2009. ACM.

[NIC] NICTA. CAmkES website. `http://ertos.nicta.com.au/software/camkes`.

[NIC06] National ICT Australia. *seL4 Reference Manual*, 2006. URL `http://www.ertos.nicta.com.au/research/sel4/sel4-refman.pdf`.

[NSL+06] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), August 2006.

[Pal09] Ameya Palande. Capability-based secure DMA in seL4. Masters thesis, Vrije Universiteit, Amsterdam, January 2009.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):413–421, 1974.

[SA99] Tom Shanley and Don Anderson. *PCI System Architecture*. Mindshare, Inc., 1999.

[San93] Ravi S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[Sis09] Robert Sison. Secure Web Server on seL4. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, October 2009.

[Zeh06] Myrto Zehnder. Virtualisation of PCI devices in LinuxOnLinux. Masters thesis, Department of Computer Science, Swiss Federal Institute of Technology, Zurich, Switzerland, August 2006.