An I/O Architecture for Microkernel-Based Operating Systems

Hermann Härtig Jork Löser Frank Mehnert Lars Reuther
Martin Pohlack Alexander Warg

Dresden University of Technology
Department of Computer Science
D-01062 Dresden, Germany
contact@os.inf.tu-dresden.de

Abstract

In a *workhorse* operating-system architecture, a slightly modified or unmodified off-the-shelf operating system runs as workhorse on a core operating system, next to application processes which have requirements not supported by the original workhorse alone. Examples include off-the-shelf operating systems running on real-time executives or trusted kernels. For such architectures, several challenges and opportunities must be addressed with respect to I/O drivers. If an I/O device is needed by the workhorse and by the other core applications, the driver may have to be separated from the workhorse. A welcome side effect of separating drivers is the opportunity to make the workhorse more robust against faults in the driver. On the other hand, if drivers remain in the workhorse and thus the workhorse needs I/O privileges, the workhorse needs to be effectively encapsulated even under the assumption of full penetration of the workhorse by an adversary. This paper describes how these challenges are met by the design and implementation of our microkernel-based operating system DROPS — the Dresden Real-Time Operating System [15] — using L^4Linux [16] as workhorse.

1 Introduction

In recent years, a novel approach towards building operating systems has been investigated and used to support applications with special trustworthiness or real-time requirements. It becomes widely accepted, that off-the-shelf operating systems are way too complex for guaranteeing them to be correct. On the other hand, these operating systems come with a large base of legacy applications and *drivers* that for economical reasons should be reused whenever possible. Consequently, rather than building completely new systems from scratch or modifying off-the-shelf operating systems significantly, off-the-shelf operating systems are just slightly modified to run atop a core operating systems that support these required properties. Successful examples for that approach are real-time operating systems, where modified Linux or Windows kernels run on top of real-time cores, for instance, RTLinux [32], Oncore, and Radisys Windows [27]. In other, more recent proposals, slightly modified off-the-shelf operating systems run on trusted cores, such as Perseus [26] and — as far as we understand from announcements on the WWW [8] — Microsoft's Palladium (alias NGSCI).

In such systems, applications either use the off-the-shelf system with its large functionality and driver base, or they use the core directly if the core's specific properties are required. For example, a robot application may run collision-avoidance processes on the real-time core while long-term planning processes run on the off-the-shelf non-real-time operating system.

For the scope of this paper, we refer to the off-the-shelf operating system as a *workhorse*, to the underlying operating system as the *core*, and to the applications that run on the core besides the workhorse as *core applications* or more specifically real-time processes or trusted applications. We investigate an aggressive approach towards a workhorse operating-system architecture: We support the reuse of untrusted off-the-shelf operating systems including their unmodified drivers, and we reuse the workhorse's untrusted or non-real-time functionality for trusted and real-time applications. That approach is based on the observation that many real-time applications have relatively simple parts with real-time (small periodic processes) and more complex parts without real-time requirements. We will substantiate a similar observation for trusted applications in Section 4.

We expect that even completely unmodified off-the-shelf operating systems can be used as workhorses if adequate virtualization support is present in hardware; though we know of no such example.

In this paper, we investigate opportunities for and challenges from I/O drivers in a microkernel-based workhorse approach. These include:

- I/O drivers may have to be separated from the workhorse and run as separate components. For exam-

ple, a keyboard driver may belong to the trusted base for a banking application and thus needs to be removed from the untrusted workhorse.

- A related but different motivation for separating drivers is to protect the workhorse from faulty drivers, since I/O drivers are known to belong to the most error-prone parts of operating systems [9]. Successful removal of drivers from the workhorse then leads to an operating system which is more robust than the original off-the-shelf operating system.

- A real-time system is one application of the workhorse approach. In such a system, we want to make sure that a crash of the workhorse does not induce the crash of the potentially critical real-time processes. Thus, we must effectively encapsulate the workhorse so that it can crash and be rebooted without affecting real-time processes. Drivers remaining in the workhorse require the encapsulation to be done even if the workhorse has I/O privileges. In a trusted-core scenario, our working assumption is that the workhorse can be fully penetrated and thus can use or induce malicious driver behavior to attack the trusted applications or the core.

- I/O drivers that remain in the workhorse can be used from applications that otherwise run on a trusted core. For example, a secure file system can use a disk driver of the workhorse under certain assumptions.

The remainder of the paper is structured as follows. In Section 2, we describe the application models of the workhorse-OS approach in more detail, concentrating on a trusted core scenario, and we discuss the software architecture. In Section 3, we investigate the separation of drivers from the workhorse. Section 4 discusses the use of services of an untrusted workhorse, such as hardware drivers, by trusted applications. In Section 5, we discuss how the workhorse can be encapsulated such that even full penetration cannot harm confidentiality and integrity of secured applications. Section 6 discusses some results of measurement experiments using DROPS with L^4Linux as workhorse. Section 7 discusses related work. Finally some conclusions are drawn and open issues are pointed out.

2 Application Scenario and Software Architecture

In the following sections we firstly discuss an application scenario, and then explain the software architecture.

2.1 Workhorse Architecture Applications in a Trusted-Core Scenario

The application scenario we use and the overall system architecture is depicted in Figure 1. A small secure core of the system runs both, an untrusted workhorse and some applications with higher trustworthiness requirements. For brevity reasons, we will refer to such core applications as *trusted applications*. The workhorse is used to provide functionality needed by trusted applications. However, the workhorse needs to be separated such that even full penetration of the workhorse does not harm the trusted applications, with respect to their information security.

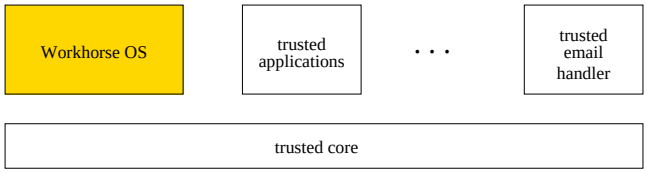


Figure 1: *Application scenario.* Trusted applications and the core are protected from bugs and malicious behavior of the workhorse.

An application example is a trusted email handler on a mobile system, for example on a PDA. The trusted email-handler application needs to fetch an email from a server, decrypt it, present it to the user and possibly send a reply. Fetching the email from the server and storing it in the PDA's file system can be done using the untrusted workhorse; however, decrypting and presenting on the PDA requires services from the small trusted core, namely secure storage of keys and a user interface, for example a window manager. In addition and most importantly, the small trusted core must effectively encapsulate the workhorse.

From this email-handler example it becomes clear to what extent the trusted applications can be protected from a potentially penetrated workhorse. If the encapsulation of the workhorse is successful, confidentiality and integrity of trusted applications can be ensured, but obviously not their availability. If the workhorse's file system is used to store encrypted emails, a successful penetrator can erase or modify the file, thus successfully break availability. However, as long as the encryption keys are securely stored in the core's secure storage, neither confidentiality nor integrity, which means unnoticed modification of content, can be compromised as long as the workhorse is encapsulated effectively. This is acceptable, if — as with IMAP email servers — emails are held on the server as well.

If the secure storage is large enough to be able to hold all changes since the last connection with a server, better availability can be ensured against software attacks that rely on the penetration of the workhorse. In that scenario, secure storage just holds the changes. The complete document can be restored during the next connection to a server. However, availability cannot be ensured in case of a hardware

attack, for instance, imagine a thief throws the PDA into a deep near-by lake.

The protection of confidentiality and integrity against software attacks are valuable for a large class of applications. This especially applies to mobile devices where only a limited availability can be ensured anyway. An important requirement for such an architecture is the availability of secure booting techniques, as to establish the trusted core[17].

2.2 Software Architecture

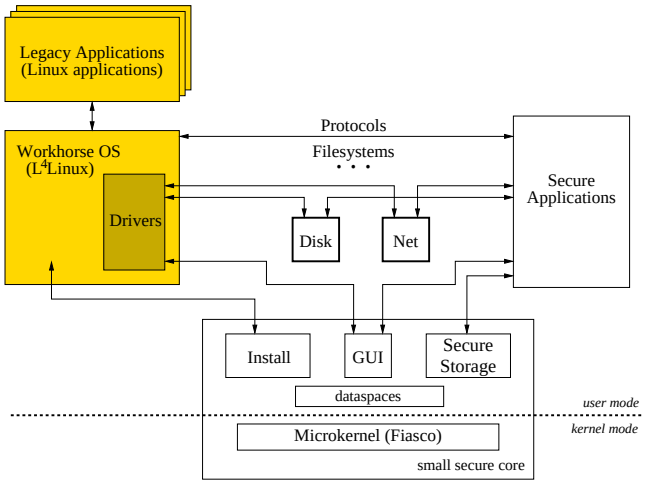


Figure 2: *The software architecture.*

Our architecture is based on a small trusted core that runs trusted applications. The trusted core is built atop a microkernel (Fiasco [19]) and consists of a few basic servers needed as part of a trusted computing base. The workhorse is L4Linux [16], a modified variation of the Linux kernel, which runs as user-level server in its own address space.

Linux applications call the L4Linux server using a binary-compatible system-call interface that relies on a transparent library and fast inter-process communication (IPC). In addition and most important, all drivers in L4Linux run unmodified, while the kernel had to be changed slightly[1]. However, we see no reason why an unchanged Linux kernel could not be used, in particular if there is better hardware support for virtualization, such as an additional execution mode separating virtual and bare-metal execution.

The Fiasco microkernel provides threads and address spaces as basic means of separation, and inter-process communication for well-defined interaction. Threads run at user level (e.g., ring 3 on IA-32), but can have I/O privileges (e.g., using IA-32's I/O bitmaps). On IA-32, accesses to I/O registers can be controlled on a per-register basis. The microkernel runs in kernel mode, while L4Linux (workhorse), the trusted applications, and all components of the trusted core run in user mode and have their own address spaces.

Access to any form of memory resources is provided via *dataspaces* which are unstructured containers for any type of data [6]. All address spaces are built using one or more dataspaces.

Dataspaces are provided by dataspace managers. Clients call a dataspace manager to create a data space and obtain a dataspace descriptor. A descriptor can be used to map dataspaces from one address space to another using IPC. Access rights can be restricted in such mapping operations. Dataspace managers construct dataspaces from other dataspaces of which some may represent physical memory.

Communication among address spaces can be based on messages that copy strings of data from one address space to another or by establishing shared regions of memory among address spaces using dataspace mappings. These mappings may be temporary, thus established on demand and revoked immediately after communication, or they can be permanent.

Drivers for bus-master DMA capable devices need to know the physical address of parts of dataspaces and may have to pin them in memory. The following simple scheme is used for that purpose: A dataspace manager keeps track of the physical pages that are given to an address space. Then, a client of a dataspace manager can request to pin a location referenced by a dataspace/offset pair and determine the physical address corresponding to the dataspace/offset pair [21]. If dataspace/offset is currently not mapped to a physical address space, the dataspace manager either tries to establish such a mapping or faults.

Apart from a dataspace manager, the trusted core contains a window manager, secure storage, and a component to install new trusted applications. Secure storage is a small storage, which is under complete control of the small core. It is based on secure booting techniques, this means access is only possible if the platform is under control of a trusted core system. The trusted applications use the secure storage to maintain keys and other credentials. Therefore, methods as proposed in [12] can be used. The window manager can have the workhorse and for example X-Windows, as well as trusted applications as clients. It is the window manager's responsibility to clearly express which stack of software is currently controlling the active window of the display. The window manager is described in [13].

Trusted applications can use both, the workhorse and the components of the trusted core. For example, files are securely stored by first encrypting them using some key from the secure storage and then delivering them to the file system of the workhorse.

[1] about 7000 LOC almost exclusively in the architecture dependent kernel subtree

3 Separating Drivers from the Workhorse

I/O drivers are known to belong to the most error-prone parts of operating systems [9]. Nevertheless, in virtually all operating systems drivers still run at the highest privilege level, this means in the kernels of operating systems. Thus drivers have full access to all vital OS data structures and hardware components; a faulty driver can easily crash a system. In our workhorse approach, some drivers can be separated from the workhorse and run in their own address spaces. This makes the workhorse more robust against faults in those drivers than the original off-the-shelf operating system. In addition, if a workhorse architecture is used in a trusted scenario, drivers have to be separated from the workhorse if they belong to the trusted base of trusted applications. This reduces the number of drivers to be evaluated drastically in contrast to using a workhorse with all its drivers as the trusted base.

This section discusses more detailed the interaction of separated drivers with the workhorse, beginning with a general description and concluding with an example.

Running a driver in its own address space instead of in the workhorse increases the complexity of an I/O request.

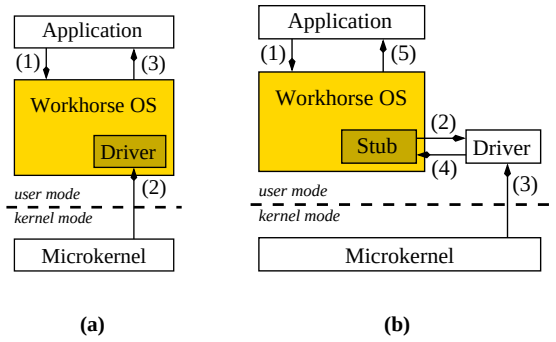


Figure 3: I/O requests with internal and separated drivers.

Figure 3(a) shows the original situation of an I/O request. An application invokes the request (e.g., a *read()* system call) calling the workhorse. The driver inside the workhorse then issues the request directly to the device, and upon completion the device responds with an interrupt, which the microkernel maps to an IPC call to the workhorse. The driver completes the request and wakes up the application by sending a reply to the original system call. With a separated driver, the request is not issued by the workhorse itself, instead a stub, which replaces the driver in the workhorse, forwards the request to the driver, which now runs in its own address space. This architecture raises several problems:

- The driver can no longer access components of the workhorse directly. In particular, it cannot access the network or file-system buffers, data must be transfered by other means between the driver and the workhorse.
- The communication between the stub and the driver introduces additional costs. These costs do not only include the costs of the two IPC operations, but also overhead introduced by the additional address-space switch between the workhorse and the driver. It is one of the aims of this paper to analyze those costs.
- Unique resources, such as the interrupt controller or PCI buses, must be shared between separated drivers and the workhorse (i.e., with the remaining drivers in the workhorse). The core must provide appropriate mechanisms to reliably share such resources.

3.1 Data Transfer

With separated drivers, I/O buffers (e.g., network packets to send) must be transfered between the address spaces of the workhorse and the driver. More generally, data must be transfered between drivers and their clients. As already mentioned in Section 2.2, data can be either copied between the address spaces or a memory mapping can be established by sharing dataspaces between the two address spaces.

To grant access to the dataspace to the driver, a client must ask the dataspace manager to share the appropriate access rights. Drivers of bus-master DMA capable devices pin the corresponding memory pages and request their physical addresses to initiate the data transfer between the device and main memory. Drivers using programmed I/O (PIO) have to map the pages of the dataspace to their own address space to be able to copy the data to the device registers.

3.2 Examples

To further examine the workhorse architecture, we separated two drivers from the workhorse.

3.2.1 Network Drivers

When we talk about separating network drivers, we really mean the drivers of the network interface cards (NICs), not a network stack, such as TCP/IP. Implementing network stacks is the responsibility of the workhorse, or other components of the workhorse architecture.

For separating the network drivers, we use the original driver code and put it into an environment that emulates the original workhorse. As described before, in the workhorse the original driver code is replaced by a stub that communicates with the separated driver. We refer to that separated driver as *network server*. Applications using the network server are called *network clients*, with the workhorse being one of them. The emulation environment in the network server takes care of communication with the network clients,

but also ensures correct memory management, thread management, and interrupt handling.

For transmitting data to the network, each network client uses a separate dataspace, which the network server uses for performing I/O. The network client puts its data into the dataspace and enables that part of the dataspace for device I/O. Then it passes the offset and length of the data to the network server, which checks these parameters and translates them into a physical address range. This range is given to the original driver code, which initiates the actual transfer.

For receiving, the situation changes a little bit. It is the nature of received network traffic that the final recipient of a data frame is not known at the time it enters the computer. In monolithic operating systems this led to the socket abstraction, where data is analyzed and dispatched into queues within the kernel. The application interface is entirely based on copying this data to user space. While early demultiplexing, which can be implemented in hardware [14, 25, 10], is a well-known solution to this problem, most network cards have no support for it.

In a workhorse architecture multiple network stacks can coexist using the same network device. Thus, the dispatching of the data must happen at the device driver, which is the network server. Therefore, the network server receives the data into its own address space. It determines the correct recipient for each frame by inspecting parts of the packet header. Then, it uses the underlying microkernel to copy the data into buffers provided by the recipient. This copying feature of the microkernel allows for a `read`-like semantics, where the client can specify where the data has to be put within its address space without the need of sharing parts of its address space with the network server. Note, that we cannot save this copy, unless data is early demultiplexed in hardware.

3.2.2 SCSI Block-Device Driver

As a second example for separating drivers from the workhorse we moved the driver of a SCSI host adapter to its own address space. Similar to the network server, we embedded the unmodified source code of the driver into an emulation environment, which provides a block-device interface to the workhorse and the other clients. In addition, the environment provides the appropriate memory and interrupt management functions the block driver uses within the original workhorse.

For both, reading and writing blocks from/to the disk, clients pass a reference to a dataspace, the block number, and the number of blocks to the driver. This dataspace contains the target/source buffer of the request. In L^{4}Linux, a stub is used to access the disks of the separated driver. This stub uses the interface of the driver to provide a block device to the workhorse on his part.

The environment used to emulate the original workhorse roughly consists of 7500 lines of code. While this seems to be quite expensive at first glance, it is still affordable because with this environment we can reuse a wide range of network, block device, and other drivers of the original workhorse.

4 Reusing an Untrusted Workhorse in Trusted Applications

I/O drivers that remain in a workhorse can be reused from trusted applications that otherwise run on the trusted core directly, or from the trusted core via upcalls. For example, a secure file system can reuse a disk driver of the workhorse under certain assumptions.

This section describes the general architecture of reusing untrusted workhorse drivers for trusted applications and details the reuse of a disk driver as example.

4.1 Architecture and Principles

Reusing drivers of an untrusted workhorse in trusted applications is an important technique to keep trusted bases small. Cryptographic techniques can be used to ensure the confidentiality and integrity, this means to notice unauthorized modification of data. For example, an untrusted disk driver can be used for the implementation of a secure file system. However, availability cannot be ensured using that technique.

Integrity of data to be submitted to the workhorse or received from the workhorse cannot be trusted. Thus, this data is encrypted before its submission to the workhorse respectively decrypted after reception. The keys used for that purpose remain in secure storage, which can be implemented using techniques as proposed in Microsoft's *Sealed Memory* [12].

The principle architecture is rather simple: a server is started either as an application of the workhorse or as an in-kernel process[2]. The server accepts requests from trusted applications and forwards them to the driver (or to another part) of the workhorse. Data are passed between trusted applications and the server either by copying messages or using temporarily shared mappings.

4.2 Example Implementation

To estimate the costs of using the drivers of the workhorse from core applications, we implemented a small application running atop the workhorse. This application accepts IPC messages from core applications, which contain requests to read and write files. Because of the limitations we described above, data is copied between the two applications.

[2]Kernel here refers to the part of the workhorse that runs in kernel mode if run as an off-the-shelf system, but runs in user mode and in its own address space if used as a workhorse.

5 Encapsulating Untrusted Components with I/O Privileges

In a trusted-core context, we must assume full penetration of the complex workhorse, and even in this case still be able to protect applications that run on the trusted core directly. This requires that the workhorse including its remaining I/O drivers must be effectively encapsulated. Even though malicious drivers have not been reported yet, we have to deal with the worst possible malicious driver if a workhorse contains drivers and is fully penetrated by an adversary.

In this section, we discuss the adaption of virtual-machine-monitor techniques and discuss a detailed example: the encapsulation of a network-interface driver, either as part of the workhorse or as separated driver.

A driver running with I/O privileges can attack the system by:

- accessing device registers of other devices (PIO)
- monopolizing or disabling interrupts
- initiating bus-master DMA from devices

Monopolizing other resources, such as CPU and memory, is prevented with the standard operating-system techniques scheduling and memory management.

Accessing device registers is typically done using memory-mapped I/O. Standard memory protection techniques prevent attacks here. A specialty of the IA-32 is a second I/O address space, the I/O port space. Access to it can be restricted to specific I/O ports with per-task permission bitmaps [20].

Interrupts are managed at a central instance in a workhorse architecture, which alone has access to the interrupt controller. At least with the IA-32-based PC architecture, there are enough interrupt lines to assign a different interrupt line to each device. Interrupt lines must not be shared between devices to prevent denial-of-service attacks between devices.[3] Disabling interrupts at the host processor is often used for synchronization, but is known to be fully replaceable by appropriate locking mechanisms [18, 32].

What remains to be solved is bus-master DMA, which allows devices to access arbitrary physical memory locations. To prevent this, the access rights of devices to memory must be restricted.

5.1 Restricting Devices

We apply the principle of address spaces to the I/O bus: We map each address that is used by a device for accessing memory to an address in main memory. Once we have an instance doing this mapping reliably, this instance can also deny accesses to addresses the device should not have access to. We refer to such an instance as an *I/O-Memory-Management Unit* (I/O-MMU).

Figure 4(a) shows the scheme of a peripheral bus with such an I/O-MMU. The devices are not connected to the bus directly, instead each bus must pass the I/O-MMU. This way, device-to-device accesses can also be controlled. For purposes of intended device-to-device transfer, a window in the physical address space of the main memory must be reserved, but this does not need to contain any real memory. Although this I/O-MMU allows a full isolation, we are not aware of such a hardware being implemented today. A possible implementation could be a riser card that is placed between the bus and the proper device. This riser card must contain the logic to intercept and validate memory transfers.

Within the architecture depicted in Figure 4(b) one I/O-MMU for the whole peripheral bus protects the host memory from illegal accesses by devices. Nevertheless, inter-device accesses cannot be prevented. Systems of this type are already available, for example the UPA-to-PCI bridge for the UltraSPARC architecture [29], HP ZX1 chipset for Itanium2 [1], and the Alpha host bridge [11] provide some means to translate bus-master DMA addresses into physical host memory addresses.

Figure 4(c) shows an architecture that contains no I/O-MMU at all. This is what most current PC architectures provide. As we will show in Section 5.3, a reliable protection for a large group of devices can be achieved with this architecture as well.

For now, let us assume a system having per-device I/O-MMUs.

5.2 Workhorse Support

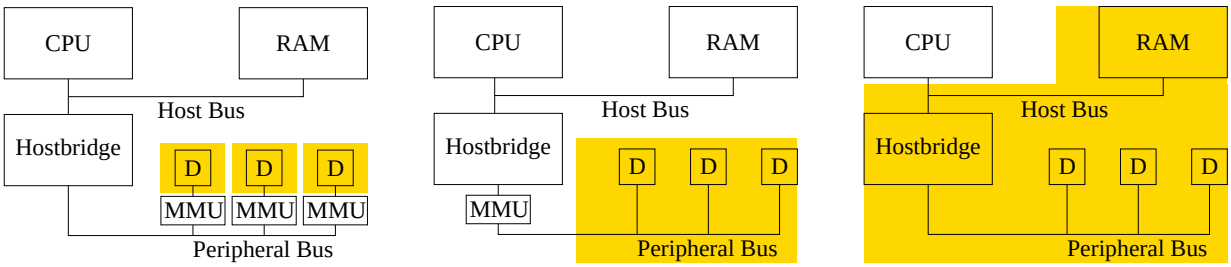
Establishing and revoking mappings at these I/O-MMUs must only be done by a trusted-core application. We call this application the *mediator*. The purpose of the mediator is to map dataspaces, or parts of them, into the address space of a device.

A client has to perform the following steps to enable a dataspace, which must be accessible by that client, for I/O requests:

1. The client asks the mediator to make a reservation in the address space of the desired device. Thereupon the mediator asks the dataspace manager to pin that dataspace on behalf of the client. The dataspace manager checks the access rights of the client and performs the pinning. Then, the mediator reserves the memory region determined by the dataspace in the address space of the device, and binds the device to the dataspace.

2. To execute a request that includes bus-master DMA between the device and a part of the dataspace, the client asks the mediator to enable transfer at the I/O-MMU for exactly this part in the dataspace. Then, it sends the

[3] As it is unknown to which device a specific interrupt on a shared interrupt line belongs all interrupt handlers connected to the interrupt have to be invoked.



(a) Peripheral bus structure with a per-device I/O-MMU.

(b) Peripheral bus structure with a per-bus I/O-MMU.

(c) Peripheral bus structure without an I/O-MMU.

Figure 4: Different types of peripheral bus structures. The shaded boxes mark the protection domains.

dataspace ID together with the offset of the part to the driver.

3. The driver performs the I/O operation and informs the client upon completion.

4. The client asks the mediator to disable device access to the part of the dataspace.

The first step allows for an optimization of the second and the last steps: A dataspace-specific permission bitmap could be used by the I/O-MMU, with each bit representing a certain fraction within the dataspace. Enabling and disabling I/O for such a fraction is than as simple as enabling and disabling this bit.

Note that for un-pinning a dataspace at its dataspace manager, the dataspace manager must explicitly revoke all mappings of that dataspace in any device address spaces.

5.3 Architectures without I/O-MMU

Without hardware support in form of I/O-MMUs we use virtual-machine-monitor techniques to emulate in software what the I/O-MMU is supposed to do in hardware: Each address that is given to a device for doing bus-master DMA is translated in software to the correct physical main-memory address. This way, we can prevent malicious drivers from telling their devices to access arbitrary memory locations without prohibiting bus-master DMA at all.

We encapsulate drivers by intercepting accesses to I/O data structures. Intercepting is done using three building blocks: We use hardware protection mechanisms to detect I/O accesses. On IA-32 these are either page-tables for memory-mapped I/O or I/O-permission bitmaps for port I/O. An *emulator* is executed in the same address space as the driver and interprets the intercepted operations. The interpretation is sent to the *mediator* that checks whether or not the access is allowed and performs the I/O if it is allowed. For this, the mediator must have an understanding of what the device is doing in response to that specific I/O operation. This requires the mediator to emulate some of the logic of the device. After the mediator performed the I/O operation, the emulator completes the instructions by setting the program counter to the next instruction.

Both the emulator and the mediator are device-specific implementations and have to be extended for new devices or classes of devices. As a result of our work, we claim that emulation is ways easier to implement/verify than the whole driver, since only a small subset of the device specification (how to access the DMA unit of the device) has to be considered.

Note, only the detection mechanism and the mediator must be trustworthy. Detection and emulation do not need to rely on hardware protection mechanisms. Instead they can be accelerated using software patches, either extending macros in an open-source system or patching at binary level. The mediator must be trustworthy, and hence runs in a separate address space. Malfunctions in the emulator can in the worst case degrade availability of the provided service.

On architectures without I/O-MMUs, un-pinning dataspaces must be done in accordance with the mediator. As device accesses to main memory cannot be stopped by simply revoking mappings of device addresses, pinned dataspaces must stay pinned as long as they may be referenced by outstanding or ongoing bus-master DMA operations. The mediator is the only one that knows for sure if addresses passed to the device for doing bus-master DMA are still in use or not.

Examples

ATA devices can be intercepted quite easily as long as they do not use command queueing. To perform bus-master DMA requests, ATA devices use a table (DMA table) holding an array of pointers to DMA blocks. The table is created at the beginning of a request and its address is written to a device register. The device signals the end of the request by raising a device interrupt. From then, the DMA table is no longer used.

To control the bus-master DMA transfers of an ATA device, it is only necessary to virtualize the DMA table. Therefore, write accesses of the driver to the device register, which holds the address of the DMA table, have to be intercepted. The mediator checks if bus-master DMA access to all entries of the DMA table is allowed for the driver and finally passes a *copy* of that table to the device. Thus, the driver cannot manipulate the table after the request has been started.

The terminating interrupt of the request can be uniquely assigned to the request whose DMA table is currently active.

Command queueing complicates the virtualization of the DMA table, because several tables are active concurrently and it is not clear to the mediator which of these tables corresponds to a raised interrupt without having explicit knowledge of the driver.[4]

Network devices usually use descriptor lists to communicate with the driver. Simplified, a descriptor consists of the address and size describing the physical memory region the device should receive network frames to or send network frames from and a flag stating the owner of the descriptor. If the aforementioned flag is set, the device may access the frame, but the device does never access a frame with a cleared flag.

To receive packets from the network device, a driver provides a receive descriptor list. Once the driver has written the address of the list to a device register, the device starts using the buffers the descriptor entries point to. The device writes received frames until it encounters the first descriptor with a cleared flag. After receiving a network frame, the device raises an interrupt. To send packets, the driver provides a send list with all flags set.

The mediator intercepts write accesses to the device registers that which contain the addresses of the descriptor lists. A *copy* of the descriptor list is passed to the device. The entries of that lists point to addresses in the drivers address space. The driver cannot manipulate the list after it has been passed to the device. Once the device has completed a request and sent an interrupt, the list of descriptors has to be copied back to the driver, so that the driver can inspect the descriptors for the results of the I/O operations.

This method works nicely for devices which are based on descriptor lists that are separated from the I/O buffers (e.g., Tulip 2114x). Everything is much harder for devices whose descriptors are adjacent to the I/O buffers. In that case not only copies of the descriptors but also copies of the adjacent data buffers have to be created and passed to the device (e.g., Intel EEPRO100).

Measurements presented in Section 6 are based on experiments using the Tulip 2114x chip.

[4]Command queueing for ATA devices is currently not in wide-spread usage.

6 Performance Impact of the Workhorse Architecture

In this section we explore the impact of separating and encapsulating drivers in our workhorse architecture, which is based on L^{4}Linux. We study both, the effects on the performance of the device drivers itself using micro-benchmarks, as well as the effects on the overall system using an application benchmark.

When doing micro-benchmarks not only the resulting performance but also the CPU utilization during the measurement is of interest, as it shows the additional overhead introduced by our architecture. Therefore, we used low-priority looper which consumed all idle CPU resources with a cache-footprint of one cache line.

The basic test environment is a 1.6 GHz Pentium 4 running on a Intel 845 chipset with 256 MByte SDRAM.

6.1 Block-Device Performance

The hardware used for the block device measurements consists of an IBM Ultrastar 36Z15 disk (18.4 GByte, 15,000 rpm, 4 MByte disk cache) attached to a Tekram DC-390U2W disk controller (U2W SCSI, 80 MByte/s).

To explore the performance impact of our architecture on the SCSI driver, we used *tiobench* [4] to measure the low-level performance of the disk system. Tiobench reads and writes blocks of files in sequential and random order.

Table 1 shows the results of tiobench. As it can be seen, we still reach the bandwidth of Linux running on the plain hardware. For random writes we even exceed the bandwidth of Linux, which we believe is due to changes in the timing in the stub in L^{4}Linux. This result could be expected, as the SCSI disk handles several requests at a time using command queueing, which hides additional overhead. However, the overhead can be seen with a slightly higher CPU utilization.

To show the impact on overall system performance we set up an application benchmark, which consists of the following five parts:

1. Extraction of a Linux kernel source archive (`tar -xjf linux-2.4.20.tar.bz2`)
2. Configuration of the Linux kernel with the supplied predefined configuration (`make oldconfig` and `make dep`)
3. Build of the kernel (`make bzImage`)
4. Duplication of the build tree and three searches through them (`grep -r 'random search pattern' .`)
5. Clean up of the build tree (`make clean`) and removal of the scratch directory

	Read				Write			
	sequential		random		sequential		random	
	bandwidth	CPU	bandwidth	CPU	bandwidth	CPU	bandwidth	CPU
Linux	46.5 MB/s	14%	14.1 MB/s	4%	43.7 MB/s	24%	18.4 MB/s	8%
L^{4}Linux (int. driver)	46.5 MB/s	19%	13.8 MB/s	6%	43.4 MB/s	26%	18.3 MB/s	9%
L^{4}Linux (ext. driver)	46.5 MB/s	20%	13.3 MB/s	7%	43.7 MB/s	27%	20.4 MB/s	12%

Table 1: *tiobench* results for IBM Ultrastar 36Z15 disk. Average of 10 runs, command line `'tiotest -f 1024 -b 65536 -r 16384 -d /mnt/disk'` (file size 1GB, block size 64KB, 16384 requests for random read/write).

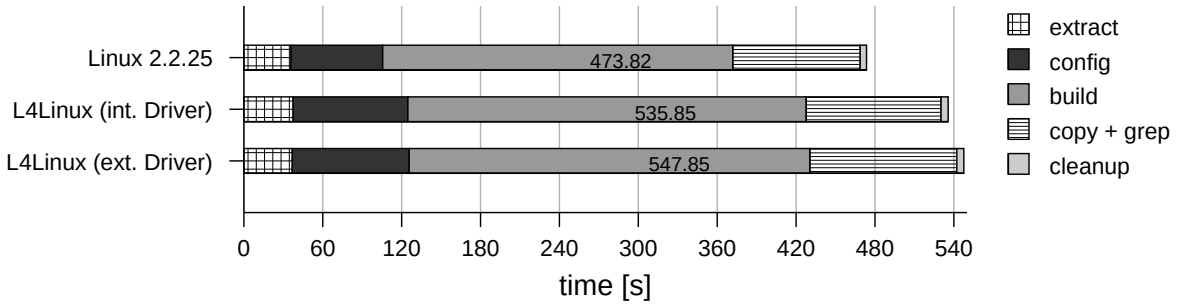


Figure 5: Application benchmark running on Linux, L^{4}Linux with internal driver and L^{4}Linux with external driver.

We measured the time for each single part and the overall time consumed.

Figure 5 shows the results of that test. The overhead introduced by separating the driver from L^{4}Linux is about 2%, which is less than the overhead caused by moving Linux to user level, which is about 13%.

6.2 Overhead of Using Drivers of L^{4}Linux from Core Applications

To measure the overhead caused by using drivers in L^{4}Linux by core applications, we used the application described in Section 4.2 to perform a test similar to tiobench used in the previous section.

	Bandwidth	CPU
Sequential Read	46.6 MB/s	28%
Sequential Write	43.6 MB/s	32%

Table 2: Using the disk driver in L^{4}Linux from core applications using the same hardware as in Table 1.

The application running on the core sequentially reads and writes an 1 GB file issuing 64 KB requests to the application running on L^{4}Linux. Table 2 shows the results of this test. We can still achieve the same bandwidth as tiobench, however, because of the additional copy operation the CPU utilization increases.

6.3 Network-Device Performance

To investigate the influence of our architecture on the network performance, we measured the maximum throughput and CPU utilization on an unmodified Linux kernel, on L^{4}Linux with an integrated driver, on L^{4}Linux with a separated driver, and on encapsulated L^{4}Linux as described in Section 5.3.

In all of our experiments, we connected two nodes with a Fast Ethernet switch D-Link DES-1008D. To generate a similar network load, we used *netperf 2.2pl4* [2]. One node comprised an AMD Duron 800 MHz on a VIA KT133 chipset, 128 MByte SDRAM and a DECchip 21142/43 Ethernet controller. The other node was a Pentium III Coppermine 800 MHz with 256 MByte RAM and different network cards. On the latter machine, we changed the various operating-system setups and measured the CPU utilization.

For determining the performance impact of separated drivers, we used an Intel EEPro100 network card (100MBit/s) in the Pentium III. We used netperf to measure TCP and UDP send and receive throughput. Additionally, the UDP request/response rates were acquired. All measurements were done three times, first with the original Linux on the Pentium III, second with L^{4}Linux on the Pentium III, and third with the separated driver and L^{4}Linux on the Pentium III. Table 3 shows the corresponding results.

For determining the performance impact of encapsulating L^{4}Linux, we used a DECchip 21142/43 Ethernet controller network card (100MBit/s) in the Pentium III. The netperf setup was the same as in the previous experiments. The three

		TCP_STREAM		UDP_STREAM		UDP_RR	
		CPU	Rate	CPU	Rate	CPU	Req/s
Linux	send	18.0 %	94.1 MBit/s	7.0 %	95.7 MBit/s	36.2 %	10441
Linux	receive	30.0 %	93.9 MBit/s	21.2 %	95.7 MBit/s		
L^{4}Linux (int. driver)	send	27.0 %	86.8 MBit/s	10.5 %	95.4 MBit/s	49.0 %	7058
L^{4}Linux (int. driver)	receive	26.0 %	53.0 MBit/s	37.0 %	95.7 MBit/s		
L^{4}Linux (ext. driver)	send	44.8 %	94.0 MBit/s	20.0 %	95.6 MBit/s	73.0 %	7163
L^{4}Linux (ext. driver)	receive	66.0 %	93.9 MBit/s	52.0 %	95.7 MBit/s		

Table 3: Impact of separated network drivers. Throughput and request-response rates for different operating system setups.

operating-system setups are original Linux, L^{4}Linux with an integrated driver, and the encapsulated L^{4}Linux with an integrated driver, as described in Section 5.3. Table 4 shows the corresponding results.

7 Related Work

Using unmodified or slightly modified legacy operating systems on top of other systems in general is an old technique. IBM's VM [23] system supports other IBM operating system without modifications. MACH's single-server approach[5] used a modified variant of BSD Unix on their microkernel. However, changes were significantly more substantial than in our system. Drivers in general were part of the MACH microkernel, rather than encapsulated. RT-MACH [31] — probably the first attempt to run a legacy operating system besides applications with specific requirements — suffered from the same shortcomings.

A different approach often taken to supporting a legacy interface is to rebuilt large parts of a system call interface based on a new kernel. Notable examples include QNX [3] that provides a Unix-compatible interface based on a small real-time kernel, and EROS [28], a capability-based system built from scratch that plans to emulate a Linux interface.

Several systems use small real-time executives to run legacy operating systems besides real-time applications. RTLinux [32], a widely used system, includes the real-time executive and the real-time tasks into the address space of the Linux kernel. Thus neither can the Linux kernel be protected from the real-tasks nor the real-time tasks from Linux crashes. In earlier papers [24], this group presented results using real-time cores with address-space-based encapsulation of real-time tasks.

An attempt to protect an operating system from faulty drivers is the NOOKS architecture [30]. Drivers remain in the kernel's address space, but run at lower privilege level (ring 1 of IA-32). Calls from the kernel to drivers and vice versa are controlled via wrapper functions which can emulate accesses to device registers as well. However, no performance results are provided so far. In contrast to our approach, NOOKS considers the Linux kernel (still about 500K LOC) as trustworthy base. No attempt is made to protect trusted applications from a penetrated Linux kernel that can make use of all drivers and driver's I/O privileges. In addition NOOKS does not address threats via bus-master DMA, which is admissible since NOOKS explicitly does not consider malicious drivers.

Related work includes VMware [5] and Disco [7]. VMware is often used in a scenario where different sets of applications are run on different virtual machines to effectively separate them. The underlying assumption is that separation using a hardware abstraction is safer than separation using a software platform like an operating-system interface. This may indeed be true for today's most widely used operating systems. However, VMware does not encapsulate existing drivers. Rather it provides its guest operating systems with a fixed set of available I/O devices. These devices have been selected to optimize performance, this means drivers with as few I/O-port accesses as possible are used. In our approach, we try to support as many different devices as possible, with as little knowledge as possible. VMware enables the usage of legacy software on incompatible legacy systems. Our approach aims at running trusted applications besides legacy applications on a small trusted core, while safely reusing complex functionality of an untrustworthy workhorse.

The architecture of the Fluke device-driver framework [22] is quite similar to ours. They run Linux device drivers as user-level applications atop the Fluke microkernel, either co-located with an application or in separate address spaces. The latter mode corresponds to L^{4}Linux with internal drivers, compare with Figure 3(a). While their results using this mode are similar to ours with L^{4}Linux, they don't mediate bus-master-DMA operations and therefore are vulnerable to errors in the driver.

Palladium probably comes closest to the work described in this paper. An announcement by Microsoft Palladium [8] mentions a system Nexus to run underneath (as far as we understand) a version of the windows operating system. But although there is a huge number of WWW-sites available containing (hearsay) knowledge, no publications on details

[5] MACH's single server was more an intermediate step towards an intended multi-server architecture than an explicit attempt to exploit a legacy operating system for more than a development step.

		TCP_STREAM		UDP_STREAM		UDP_RR	
		CPU	Rate	CPU	Rate	CPU	Req/s
Linux	send	22.0 %	87.6 MBit/s	14.5 %	95.7 MBit/s	43.2 %	11651
Linux	receive	19.0 %	65.7 MBit/s	22.7 %	95.7 MBit/s		
L^4Linux	send	20.5 %	69.7 MBit/s	13.3 %	94.1 MBit/s	65.0 %	8647
L^4Linux	receive	32.0 %	65.0 MBit/s	40.0 %	95.7 MBit/s		
L^4Linux (Mediator)	send	41.0 %	66.4 MBit/s	44.0 %	95.1 MBit/s	75.0 %	7970
L^4Linux (Mediator)	receive	39.0 %	65.3 MBit/s	47.8 %	95.7 MBit/s		

Table 4: Network-related impact of encapsulating L^4Linux.

of the architecture are known to us except a paper on *Authenticated Operation* [8, 12] that describes Sealed Storage and its use in an authentication chain.

8 Conclusion

This paper presented a microkernel-based design and partial implementation of a workhorse operating system architecture with a Linux derivative as workhorse. We presented how drivers can be separated from the workhorse to 1) increase the robustness of the workhorse or to 2) make the driver usable as a component of a trusted base for trusted applications. We demonstrated how to encapsulate a workhorse such that even full penetration of the workhorse by an adversary cannot be used successfully to harm trusted applications. Last, we presented a technique how to use an untrusted workhorse to implement trusted applications. For our partial implementation of the architecture, we did some case studies with a small set of drivers. The measurement results, although not outstanding, are still good enough to encourage the pursue of microkernel-based workhorse architectures.

Additionally, we propose the introduction of hardware-based memory protection for I/O devices (I/O-MMU) in legacy hardware.

References

[1] IDF 2002: hp zx1 chipset. Available from URL: `http://www.hp.com/products1/itanium/idf/chipset/index.html`.

[2] Netperf Homepage. URL: `http://www.netperf.org/`.

[3] QNX Homepage. URL: `http://www.qnx.com/`.

[4] Tiobench Homepage. URL: `http://tiobench.sourceforge.net/`.

[5] VMware Homepage. URL: `http://www.vmware.com/`.

[6] M. Aron, L. Deller, K. Elphinstone, T. Jaeger, J. Liedtke, and Y. Park. The SawMill Framework for Virtual Memory Diversity. In *Proceedings of the Sixth Australasian Computer Systems Architecture Conference (ACSAC2001)*, 2001.

[7] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[8] Amy Carroll, Mario Juarez, Julia Polk, and Tony Leininger. Microsoft "Palladium": A Business Overview, August 2002. Available from URL: `http://www.microsoft.com/PressPass/features/2002/jul02/0724palladiumwp.asp`.

[9] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 73–88. ACM Press, 2001.

[10] U. Dannowski and H. Härtig. Policing offloaded. In *Proceedings of the Sixth IEEE Real-Time Technology and Application Symposium*, Washington D.C., May 2000.

[11] DIGITAL Semiconductors. *21174 Core Logic Chip*, 1997.

[12] Paul England and Marcus Peinado. Authenticated operation of open computing devices. In *Proceedings of the Australian Conference on Information Security and Privacy*, pages 346–361. Springer Verlag, 2002.

[13] Norman Feske. DOpE – a graphical user interface for DROPS. Master's thesis, Dresden University of Technology, 2002.

[14] A. Gallatin, J. Chase, and K. Yocum. Trapeze/ip: Tcp/ip at near-gigabit speeds. In *1999 USENIX Technical Conference (Freenix Track)*, Berkeley, California, June 1999.

[15] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[16] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[17] Hermann Härtig. Security Architectures Revisited. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[18] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, Adelaide, Australia, September 1998.

[19] Michael Hohmuth. *Pragmatic nonblocking synchronization for real-time systems*. PhD thesis, TU Dresden, Fakultät Informatik, September 2002.

[20] Intel Corp. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999.

[21] Simon Kagström and Rickard Molin. A device driver framework for multiserver operating systems with untrusted memory servers. Master's thesis, Lund University, Sweden, 2002.

[22] Kevin Thomas Van Maren. The Fluke Device Driver Framework. Master's thesis, The University of Utah, December 1999.

[23] R. A. Mayer and L. H. Seawright. A virtual machine time sharing system. *IBM Systems Journal*, 9(3):199–218, 1970.

[24] F. Mehnert, M. Hohmuth, and H. Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *Proceedings of the 23th IEEE Real-Time Systems Symposium (RTSS)*, December 2002.

[25] Myricom, Inc. GM: A message-passing system for Myrinet networks. Available from URL: `http://www.myri.com/`.

[26] Birgit Pfitzmann, James Riordan, Christian Stüble, Michael Waidner, and Arnd Weber. The PERSEUS system architecture. Technical Report RZ 3335 (#93381), IBM Research Division, Zurich Laboratory, April 2001.

[27] Krithi Ramamritham, Chia Shen, Oscar González, Subhabrata Sen, and Shreedhar Shirgurkar. Using Windows NT for Real-Time Applications: Experimental Observations and Recommendations. In *Proceedings of the Fourth IEEE Real-Time Technology Applications Symposium (RTAS)*, 1998.

[28] J. S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, April 1999. Available from URL: `http://srl.cs.jhu.edu/~shap/EROS/thesis.ps`.

[29] Sun Microsystems. *UPA to PCI Interface User's Manual, STP2223 Data sheet*.

[30] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, September 2002.

[31] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-time Mach: Towards a predictable real-time system. In USENIX, editor, *Mach Workshop Conference Proceedings, October 4–5, 1990. Burlington, VT*, pages 73–82, Berkeley, CA, USA, October 1990. USENIX.

[32] Victor Yodaiken and Michael Barabanov. A Real-Time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, Anaheim, CA, January 1997. The USENIX Association.