Improving System Security Through TCB Reduction

Bernhard Kauer

March 31, 2015

Dissertation

vorgelegt an der

Technischen Universität Dresden

Fakultät Informatik

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

Erstgutachter	Prof. Dr. rer. nat. Hermann Härtig Technische Universität Dresden
Zweitgutachter	Prof. Dr. Paulo Esteves Veríssimo University of Luxembourg
Verteidigung	15.12.2014

Abstract

The OS (operating system) is the primary target of todays attacks. A single exploitable defect can be sufficient to break the security of the system and give fully control over all the software on the machine. Because current operating systems are too large to be defect free, the best approach to improve the system security is to reduce their code to more manageable levels. This work shows how the security-critical part of the OS, the so called TCB (Trusted Computing Base), can be reduced from millions to less than hundred thousand lines of code to achieve these security goals.

Shrinking the software stack by more than an order of magnitude is an open challenge since no single technique can currently achieve this. We therefore followed a holistic approach and improved the design as well as implementation of several system layers starting with a new OS called NOVA. NOVA provides a small TCB for both newly written applications but also for legacy code running inside virtual machines. Virtualization is thereby the key technique to ensure that compatibility requirements will not increase the minimal TCB of our system.

The main contribution of this work is to show how the virtual machine monitor for NOVA was implemented with significantly less lines of code without affecting the performance of its guest OS. To reduce the overall TCB of our system, other parts had to be improved as well. Additional contributions are the simplification of the OS debugging interface, the reduction of the boot stack and a new programming language called B1 that can be more easily compiled.

Acknowledgements

First, I would like to thank my supervisor Hermann Härtig for his support as well as the freedom in choosing my research directions. Thanks are also due to the members of the OS group for uncountable discussions during my time at TU Dresden. I am particular indebted to Udo Steinberg, which I had the pleasure to share a room with. His NOVA Hypervisor proved to be an excellent base for my virtualization research. I am also grateful to Alexander Böttcher and Julian Stecklina for extending the software I have developed.

During my thesis I had the privilege to get to know industrial and academic research in the US and Portugal. Thanks to Marcus Peinado and Sebastian Schoenberg for the summers at Microsoft and Intel as well as to Paulo Veríssimo and the navigators for the rewarding time and excellent coffee in Lisbon.

I am grateful to all who have read and commented on drafts of this thesis. Monika Scholz, Anne Kauer, and Norman Feske deserve particular thanks for spending hours of their live to improve what I had written. The errors still remaining in this work are mine alone.

Finally, I want to thank my family for always believing in me and supporting this endeavor for so a long time. Without Anne and my parents I would not have made it.

Thank you all!

Contents

List of Figures

Chapter 1

Introduction

> The plague of software explosion is not a "law of nature." It is avoidable, and it is the software engineer's task to curtail it.
>
> N. Wirth in *A Plea for Lean Software* [Wir95]

1.1 Motivation

Security-sensitive data is more and more entrusted to world-wide accessible computers, may they be personally owned or reside inside the so called *cloud*. This includes financial and even medical records, but also private messages and pictures that are better kept secure. However, current systems are plagued by malware, which aims to break the confidentiality (*identity theft*), availability (*denial of service*), and integrity (*ransom-ware*) of the data.

The OS (operating system) of these machines is thereby the primary target because successfully exploiting one defect in it gives an attacker full control over the platform. Given the hundreds of vulnerabilities found every year in operating systems, there are always defects to exploit [AAD+09]. For example during the year 2013,

- Debian warned of 771 vulnerabilities affecting 154 packages [DSA],
- Microsoft published 334 vulnerabilities in 106 security bulletins [MSB], and
- the Common Vulnerabilities and Exposures (CVE) list got 170 new entries for the Linux kernel [CVE].

Thus, "we have a long way to go in making existing OS kernels secure" [CMW+11]. It is the aim of this work to contribute towards a *secure operating system*.

Current systems are too large to be bug free. Debian Wheezy, for instance, is built from 419 MSLOC[1] [Bro12]. A small subset of this codebase is usually sufficient for a certain scenario. For example, using Debian in a cloud-computing setting requires only 14.5 MSLOC (§*A*). This can further be reduced to 1.2 MSLOC by removing unnecessary features and choosing smaller programs whenever possible. However, when assuming "good quality software" with "1 defect per every 1,000 lines of code" [Cov14], even this smaller number translates to more than a thousand defects. Moreover, the steady code

[1] Million Source Lines of Code. See Appendix A or the Glossary (§*B*) for a detailed description.

inflation, which can be observed on nearly all layers of the software stack[2], will lead to additional defects in the code.

We deduce that the best approach towards a secure operating system is to reduce its size to more manageable levels. Shrinking the OS has several positive effects on the security:

- It reduces the number of exploitable defects and the attack surface of the system.
- Minimizing the "self-inflicted complexity" [Wir95] improves the system security because "security's worst enemy is complexity" [FS03]. In fact, smaller and therefore less complex systems have a lower defect rate. For example, [Cov14] reports that codebases below 100 KSLOC are only half as defective as those above 500 KSLOC.
- Advanced testing methods such as formal verification [KEH+09, Ler09, LS09] or automatic bug finding [CDE10, ZC10, BUZC11] do not scale to the million lines of code of current systems yet. By reducing the size of the OS, these methods can be applied to further minimize the defect rate.

As we like to strengthen the security of the OS, the parts of the system, which cannot influence the availability, confidentiality, and integrity of private data, do not need to be considered. Instead, an OS reduction can focus on the security relevant part of the system, commonly called the TCB (Trusted Computing Base)[3]. The aim of this work is therefore to *improve the OS security by significantly reducing its TCB, ideally below 100 KSLOC.*

1.2 Approach

Shrinking the software stack by more than an order of magnitude is a veritable challenge, as no single technique is known that could perform this reduction [Bro87]. Disaggregating an existing system does not reduce it far enough [MMH08, CNZ+11]. Instead, one has to consider the whole system. Even a rather innocent requirement, for instance retrieving an interrupt number from an ACPI table, may need already half of the 100 KSLOC (§5.4). However, advancing the design and implementation of each system layer is only feasible if we *build a new system from ground up*, an approach we share, for instance, with the Oberon and STEPS projects [Wir95, AFK+11].

1.2.1 Minimizing the OS

Minimizing the OS kernel and its services is a well-studied area. We can therefore rely on a large foundation of previous work on this layer of the system.

Most importantly are *microkernel-based systems* that "minimize the functionality that is provided by the kernel" [EH13] and thereby reduce the amount of code running in the highest-privilege CPU mode.

Nevertheless, just moving OS functionality like the filesystem or the network stack to the user level is not sufficient if the code keeps full control over the platform. One also needs to decompose this user-level environment into smaller deprivileged services [GJP+00] to get a minimal application-specific TCB. Additionally, one has to ensure that

[2] A basic Debian installation, for instance, grows around 10% every year. The Linux kernel increases even faster (§A.3.3).

[3] In contrast to the classical definition of the Orange Book [Dep85], I deliberately ignore the hardware here because hardware security is beyond the scope of this work.

the services have only a small impact on the TCB of the applications that are dependent on them[4].

User-level device drivers improve the OS security as well [LCFD+05]. Running device drivers in dedicated address spaces increases the robustness of the system as potentially faulty code is isolated and can be easily restarted [THB06]. More importantly, it ensures that the driver code will not be part of the TCB of any unrelated application. This holds especially true if DMA (direct memory access) and IRQ (interrupt request) attacks from a malicious device driver can be mitigated with an IOMMU (I/O Memory Management Unit) [BYMX+06].

1.2.2 Ensuring Compatibility

Unfortunately, traditional microkernel-based systems are hindered by compatibility issues. Porting legacy applications, services, and device drivers to a new OS interface is a tremendous task that may require 90-95% of the OS development effort [Pik00]. Moreover, the functionality needed by the legacy interfaces of applications [BdPSR96, HHL+97] and device drivers [FBB+97, Fri06] will increase the TCB for both ported and newly written software.

The broad success of hypervisors has shown that compatibility can be elegantly ensured through *virtualization*: "By virtualizing a commodity OS over a lowlevel kernel, we gain support for legacy applications and devices we don't want to write drivers for" [REH07]. Virtualizing a hardware platform like a x86 PC once is much simpler than implementing all the interfaces existing software might need. This holds especially true since CPUs natively support VMs (virtual machines).

Furthermore, virtualization can be employed to counteract the code explosion at the operating system level: Deprecated OS interfaces can be removed without breaking compatibility, if previous versions of the very same OS can run concurrently on a single machine.

However, contemporary systems with virtualization support typically rely on a large hypervisor, a complex VMM (virtual machine monitor), and a single VM for device drivers, which leads to millions of lines of code in their TCB (§*A*.2). Or in other words, "a VMM that does not inflate the system's minimal TCB with a large emulation framework has not been demonstrated yet" [HHF+05].

Our new OS should bridge the gap between microkernel-based systems and hypervisors to get the benefits from both worlds. It should provide a small TCB for both unmodified guest operating systems running inside virtual machines as well as applications not depending on virtualization at all.

1.2.3 Additional Layers

Just minimizing the kernel, user-level environment, and VMM is not sufficient to reduce the TCB by more than an order of magnitude. One has to consider other system layers as well:

Debugger Operating systems usually include sophisticated debugging capabilities to ease the investigation of unexpected system behavior, caused by configuration errors, hardware and software failures, or even remote attacks. However, a feature-rich debugging infrastructure can easily double the size of a small OS (§*A*.4.3). To get

[4] Reducing the size of the application or the OS services, as done for instance in [SPHH06, SKB+07, CGL+08, Fes09, Wei14], is outside the scope of this work.

to a usable system, we need to resolve the conflict between rich debugging features and a small TCB.

Compiler Operating systems are typically written in high-level languages like C or C++. A toolchain including compiler and assembler has to translate this source code into machine code the CPU is able to execute. Because a subverted compiler can easily omit any security check in the OS [Tho84], this toolchain is security-critical and therefore part of the TCB. Unfortunately, a widely used C compiler like GCC consists of millions of lines of code (§*A*.4.4). To reach a TCB below 100 KSLOC, we have to drastically reduce the size of the tools that translate source code to machine code.

Boot Stack Booting adds the firmware and the bootloader to the TCB because they have full control over the platform and can manipulate the OS easily. Current boot stacks additionally increase the size of the OS by providing complex interfaces like ACPI (Advanced Configuration and Power Interface) (§*A*.4.2). To run our OS without loosing any previously gained TCB advantage, we have to limit the TCB impact of the boot stack.

In summary, we aim for a new OS that offers a significantly smaller TCB for both newly written and legacy applications. We try to improve the architecture, design, and implementation of the operating system at several layers. Additionally, we will minimize the TCB impact of debugging, compiling, and booting such a small system.

1.3 Contributions

Most of the contribution to this work were achieved within the scope of the NOVA project. With NOVA, a joint work with Udo Steinberg, we demonstrate that an operating system can offer a small TCB for both virtual machines and applications [SK10a]. While Udo Steinberg designed and implemented the NOVA microhypervisor [Ste11, Ste15], I developed the VMM and a user-level environment for NOVA to run multiple virtual machines in parallel.

The main contribution of this work is to show how the VMM for NOVA was made significantly smaller than comparable implementations without affecting the performance of its guests (§2). The most important techniques to achieve this are:

- A software component design of the VMM and the user-level environment enabling an application-specific TCB,
- Moving functionality out of the device models,
- Generating code for the instruction emulator, and
- Virtualizing the BIOS inside the VMM instead of running it in the virtual machine.

These improvements have led to a virtualization stack more than an order of magnitude smaller than state-of-the-art implementations. However, to reduce the overall TCB below 100 KSLOC, other parts of the system had to be improved as well. Additional contributions of this work are the simplification of the debugger, compiler, and boot stack:

Debugger I show in Chapter 3 that the tension between rich debugging features and a small TCB can be solved by removing most of the debug interfaces from the

operating system. Only memory access and some form of signaling is needed to efficiently debug any physical and virtual machine. Furthermore, I show in this chapter that no runtime code will be necessary if the debugging interface is provided through Firewire.

Compiler I argue in Chapter 4 that both the semantics of the programming language as well as the implementation have to be improved to achieve a smaller compiler. I therefore designed a new programming language called B1, which is simpler than C but still powerful enough for systems development. By reusing the syntax of Python for B1 and also implementing a corresponding toolchain in this high-level language, the TCB impact of compiling B1 programs is at least four times smaller than compiling C code.

Boot Stack I investigate in Chapter 5 how the TCB impact of the x86 boot stack can be minimized. I show that a decomposed design and several implementation improvements can decrease the size of the bootloader by more than an order of magnitude. I also analyze whether trusted computing techniques will lead to a smaller and more secure boot stack. Finally, I show that replacing the ACPI interpreter with a simple heuristic can reduce the TCB impact by more than two orders of magnitude.

Incoperating all these improvements into a single unified system is left as future work. Instead, the various techniques presented in this thesis should help to build smaller and more secure operating systems in the future.

The structure of this work can be summarized as follows. Each topic, namely virtualizing, debugging, compiling, and booting the system, is handled in a dedicated chapter, which are called *A Smaller Virtualization Stack* (§2), *TCB-aware Debugging* (§3), *Simplifying the Compiler* (§4), and *Shrinking the Boot Stack* (§5). These chapters are followed by the conclusions (§6). Finally, there are three appendices. The first appendix shows how the TCB of existing software was measured (§A). The later ones are Glossary (§B) and Bibliography (§C).

Chapter 2

A Smaller Virtualization Stack

> Keeping a virtual machine monitor (VMM), or hypervisor, small is difficult.
>
> P. Karger and D. Safford in [KS08]

Virtualization of the hardware platform is a widely used technique that saves resources in cloud data centers by consolidating multiple servers onto a single physical machine [Vog08]. Furthermore, virtualization is employed as a second layer of defense in intrusion detection, Byzantine Fault Tolerance, and malware analysis systems [GR03, DPSP+11, JWX07]. Virtualization improves the security of operating systems, device drivers, and applications [SLQP07, SET+09, CGL+08].

Adding virtualization to a host OS (operating system) can also reduce the TCB (Trusted Computing Base) in two ways:

1. By running legacy applications together with their original OS inside a VM (virtual machine), the host OS does not need to provide legacy interfaces anymore. For instance implementing a POSIX compatibility layer or porting existing libraries to the host OS becomes unnecessary.

2. By running different versions of an OS concurrently on a single machine, deprecated OS features can be removed much earlier than today without loosing backward compatibility. This can counteract the size inflation observable on many contemporary operating systems.

"However, using virtualization is not without risk", as we argued in [SK10a] because any defect in the virtualization layer jeopardizes the security of all applications on top of it. Unfortunately, the security of virtualization software has not kept up with its enormous success. Current implementations are considered "bloat" that "comes with its own security problems" [REH07]. Virtualization stacks are indeed huge today. Faithfully virtualizing the x86 architecture seems to require more than one million lines of code as shown in Appendix A. Furthermore, the defect rate per code line is likely higher in virtualization software than in any major OS, due to the shorter development time. Consequently, a large number of defects in virtualization stacks can be observed [Orm07], which have already led to escapes from the virtual machine environment [Woj08, Kor09].

With NOVA, we tackle the root of these flaws, namely the huge size of existing implementations. We improve the security of the overall system by minimizing the TCB of virtual machines and applications not using VMs at all. By starting from scratch, we were able to redesign the architecture as well as improve the implementation of the

virtualization software stack on the x86 platform. This has reduced the TCB of a virtual machine by more than an order of magnitude compared to existing implementations.

NOVA is joint work with Udo Steinberg who designed and implemented the hypervisor [Ste15, Ste11]. In this chapter, I describe my part of the work on NOVA, namely the development of a new VMM (virtual machine monitor) called Vancouver. Vancouver aims to be significantly smaller than existing VMMs while achieving performance on par with commercial implementations. I also shortly introduce the NOVA user-level environment (NUL) that multiplexes the platform resources between multiple Vancouver instances.

In the next section, I describe the necessary background and related work. The NOVA architecture, the design of the VMM and of NUL is discussed in Section 2.2. This is followed by three sections detailing the improvements in the device model (§2.3), the instruction emulator (§2.4), and the virtual BIOS (§2.5). In Section 2.6, I evaluate the performance of VMs on NOVA. I conclude the chapter with Section 2.7.

2.1 Background and Related Work

In a virtualized system, a hypervisor multiplexes a single host machine so that multiple operating systems and their applications will run concurrently on it. The hypervisor is thereby the highest privileged software in the system. It protects the operating systems from each other and ensures that a single OS cannot fully take control over the physical machine.

A virtualized OS runs in an execution environment called a VM (virtual machine) that is similar but not necessarily the same as the physical machine. This environment is provided by a VMM (virtual machine monitor) and consists of memory, one or more virtual CPUs, and all the peripheral devices that an OS expects on a platform. Examples for these are timer, PCI (Peripheral Component Interconnect) bus, disk, and NIC (network interface controller).

It is important to know that hypervisor and VMM are often implemented as a single entity. Consequently, many authors use both terms synonymously. However, I explicitly distinguish them throughout this work as they are separate programs in NOVA.

In the next section, I detail the history of virtualization (§2.1.1) and describe current implementations (§2.1.2). This is followed by a summary of related work to secure the virtualization layer (§2.1.3) and advanced use cases for virtualization (§2.1.4).

2.1.1 A short History of Virtualization

The Roots: Emulation

Virtualization has its roots in machine emulation. A machine emulator is a program that reproduces the behavior of a guest machine on top of a *host* machine as closely as possible. This allows to run applications and operating systems written for one machine architecture on another. Usually, one emulates an older machine on top of a newer one to execute legacy code [Law96, Bel05]. The opposite is also possible where a yet-to-be-built machine is emulated to estimate its performance or to port software to it [CCD+10, AMD09].

Whereas emulation is a powerful technique, it is often quite slow because all CPU instructions need to be interpreted or translated from the guest to the host instruction set (binary translation). Virtualization can simplify this task by assuming that the guest and host machine share the CPU architecture. Thus, the guest CPU need not to be emulated anymore. The hypervisor can simply timeshare the physical CPU between guest and host programs. Most of the guest instructions can thereby execute natively (*direct execution*).

However, not all resources can be shared this easily. Especially hardware devices such as the platform timers have still to be emulated by the VMM.

Starting on the Mainframe

The initial virtual-machine research was done for the IBM 360 mainframe [Gol74]. Running only a single OS on these machines was considered to be too expensive. Thus, virtual machines were invented in the mid '60s to run more than one operating-system kernel at the same time. Previously developing, testing, and debugging of a new OS release required reboots to switch between older and newer versions. With virtualization two or more operating systems could be run concurrently on a single machine.

The foundations of virtualization, as we know it today, were laid by the end of the '70s:

- Virtual machines were used to debug new operating systems, test network software, and develop multi-processor applications [GG74, Gol74].
- Device drivers were adopted to the hypervisor to reduce the I/O virtualization overhead (paravirtualization) [Gol74].
- The hardware virtualizer design minimized the size and overhead of a VMM by implementing most traps in hardware. It already supported recursive virtual machines [Gol73, KVB11].
- IBM improved the performance of virtual machines by handling time-critical virtualization events in hard- and firmware *(VM-assist)* [Mac79].
- Nested paging and IOMMU (I/O Memory Management Unit) functionality could be implemented in hardware [Gol74].

The 80s

After this promising start, virtualization got out of focus of the research community for nearly two decades. The new personal computers (PCs) with their single user and limited computing power would not benefit much from virtualization. The PC was cheap enough to buy a second machine if required. As a result, computer architectures such as VAX and x86 were not designed with virtualization in mind [BDR97, RI00]. Consequently, they would not satisfy the virtualization requirements described by Popek and Goldberg [PG74]. Running virtual machines is not possible on these CPUs without changing their architecture or without falling back to emulation techniques.

An exception to the missing virtualization research during the 80s are the UCLA and VAX security kernels [Wal75, KZB+91]. Both extended existing CPU architectures with virtualization support to strictly separate multiple operating systems so that different levels of confidential information could be handled securely within a single system.

Paravirtualization

Another line of virtualization research started at the end of the 80s with the microkernel movement. In these systems, only a small microkernel like MACH [ABB+86] runs in the most privileged processor mode, instead of the large and monolithic UNIX kernel. The remaining OS functionality such as filesystem or network stack is provided by user-level servers. Unfortunately, the first MACH implementations were much slower than the

monolithic architectures they aimed to replace. Liedtke could show later with L4 that a careful design can make microkernel-based systems nearly as fast as monolithic operating systems [Lie95].

Running legacy applications and having enough device drivers is a major problem of all microkernel-based systems. One approach to solve this issue is to reimplement the legacy interfaces. The SawMill project showed that the POSIX interface can be implemented within multiple servers [GJP+00]. However, starting from scratch turned out to be too laborious. Better approaches to reuse legacy code had to be found instead.

In many cases a Unix kernel such as Linux or FreeBSD was ported to run as application on the microkernel. Privileged kernel instructions usually faulted and could therefore be emulated from the outside. The small set of *sensitive* instructions that would not fault on non-virtualizable architectures like x86 had to be replaced in the source code or on the assembler level. Denali later coined the term paravirtualization for this approach [WSG02]. Many projects reused Linux together with its device drivers in this way. Examples are MkLinux, L4Linux, User-Mode Linux, and Xen [BdPSR96, HHL+97, Dik01, BDF+03].

Paravirtualizing a system requires a certain manual effort because the guest OS and sometimes even its libraries have to be explicitly ported to a new architecture. While some steps can be automated [LUC+05], the sources or at least special annotations to the binaries have to be available. This effectively limits this approach to open-source operating systems.

Even if recent hardware extensions have diminished the need for paravirtualization on x86 CPUs, the technique survives on non-virtualizable architectures. Moreover it is still used to improve the performance of virtual I/O by letting guest drivers directly talk to the VMM [Rus08].

The Revival

The revival of virtualization as an active research area dates back to the Disco project [BDR97]. At the end of the '90s server and workstation operating systems did not scale well to upcoming multi-processor machines. Instead of undertaking the elaborate task of fixing all scalability bottlenecks found in a general-purpose OS, Stanford researchers around Rosenblum searched for a more elegant way to solve this problem. They came up with the idea of running multiple OS instances on a single machine. Whereas each instance would use only a single processor, the physical resources of the machine could still be fully utilized if multiple VMs were executed in parallel. In this way, they reinvented virtual machines on modern platforms.

Disco relied on paravirtualization to run multiple operating systems on the non-virtualizable MIPS architecture. Unfortunately, this technique significantly limits the number of virtualizable guest operating systems. VMware, a company co-founded by Rosenblum in 1998 to commercialize virtualization, chose binary translation to overcome the non-virtualizable nature of the x86 processors [AGSS10]. This technique allows to run closed-source operating systems in virtual machines by replacing the sensitive instructions found in their binaries with faulting ones. Such a binary translator is a complex piece of software [Bel05] because it not only needs to understand all x86 instructions and to hide itself from the guest, it also needs to do this as fast as possible.

VMware proved soon that virtualization can be commercially successful if it is used to consolidate multiple physical servers on a single machine (*server consolidation*). This unexpected result boosted academic and commercial efforts. Virtualization became a hot research topic again.

Hardware Support for Virtualization

The significant overhead of the first x86 virtualization products [AA06] could only be reduced by changing the CPU architecture. In 2005 Intel and AMD announced two different processor extensions, called VT-x and AMD-V respectively [UNR+05, SVM05]. Both added a new processor mode (root and guest mode) and introduced a couple of new instructions to configure and start virtual machines. Some processor resources such as the interrupt state are completely multiplexed in hardware whereas other resources like pagetables, MSRs (Model-Specific Registers), or `cpuid` leafs, have to be virtualized in software. To ease this task, accessing sensitive resources will interrupt the virtual machine and trap to the hypervisor (*VM exit*). The hypervisor may then emulate, in cooperation with the VMM, the side effects expected from the guest before resuming the virtual machine.

The virtualization extensions reduced the complexity of virtualization software dramatically. Techniques like binary translation were not required anymore because operating systems will run directly inside hardware supported VMs. Furthermore, the overhead of virtualization could be significantly reduced by implementing common virtualization tasks in hardware (hardware-assisted virtualization). Especially the support for nested paging [BSSM08], introduced in 2007, gives virtual machines near native performance, as I will show in Section 2.6.

2.1.2 Virtualization Today

While the previous subsection gave a short overview on the history of virtualization, I will now describe current virtualization projects related to our work. I focus on open source and research projects because many details of commercial products are undisclosed and therefore unavailable for evaluation.

x86 Emulators

Even though x86 emulators are usually slower than hardware-assisted virtualization, they are still widely used due to several reasons. First, they are more portable. An emulator written in plain C is likely to run on any platform, for which a C compiler exists. Second, emulators can support multiple guest architectures. Finally, developers like to reuse the device models and instruction emulation code in their virtualization stacks.

The grandfather of x86 system emulators is considered to be **Bochs** [Law96]. The development started in 1994 as a commercial product to emulate i386-based machines. Bochs was bought by Mandrake and made open-source software in 2000. It is still under active development today and supports many x86 CPUs between the i386 up to the latest 64-bit CPUs including many CPU extensions such as SSE4 (Streaming SIMD Extensions) and VT-x. Bochs is written in C++ in a verbose coding style. For instance, its device models implement the same functionality with nearly twice the lines of code as Qemu's (§2.3.5). Bochs' instruction emulation is complete and can be considered nearly bug free [MPRB09]. The BIOS (Basic Input/Output System) of Bochs and its VGA Option ROM are still used in many other virtualization projects.

In 2000, Bellard started to build a new system emulator called **Qemu** [Bel05], probably due to Bochs low performance at that time. Qemu is based on a fast, but still portable, binary translator. Consequently, it supports many guest and host architectures besides x86. Examples are ARM, ALPHA and even s390. Since Qemu is written in C and not C++, Bellard had to write its own device models. Even if they are not directly

derived from Bochs, both share common bugs[1]. Qemu's device models are reused by many virtualization projects such as Xen, KVM, and VirtualBox. Today, Qemu is actively developed by a large open-source community.

There are many commercial emulators available. Most notably are **Simics** [MCE+02], which supports a wide range of CPU architectures, as well as AMD's **SimNow** emulator [AMD09] that was used to port software to the AMD64 architecture before real CPUs were available.

Moreover, there are cycle-accurate emulators such as **PTLsim** [You07], which can model a CPU on a more detailed level by also emulating cache behavior and bus messages. Cycle accuracy can be useful to prototype and evaluate new CPU features [CCD+10]. Unfortunately, such emulators tend to be orders of magnitude slower than the physical CPU they emulate.

Finally, projects such as **Pin** [LCM+05] or **Valgrind** [NS07] rely on a binary translator for instruction-level instrumentation. These systems cannot be used as stand-alone x86 system emulators because they lack the necessary virtual devices to run an unmodified operating system.

Commercial Virtualization Stacks

Many commercial products provide virtual machines on the x86 architecture. They usually have a larger feature set than the open-source implementations. Most notably, they come with many redundant device models and include a complex software stack for managing and even migrating virtual machines in the data center [CFH+05]. However, commercial vendors seem to prefer performance and manageability over TCB size and security.

In the following, I will shortly describe those virtualization stacks we were able to evaluate for performance and size as described in Section 2.6 and Appendix A. Whereas many more commercial virtualization environments exist, for instance from Parallels, WindRiver, GreenHills, Trango, or VirtualLogic, we could neither obtain source code nor binaries for either of them. Furthermore, the lack of research papers detailing these projects makes it impossible to compare NOVA to them.

VMware was the first to successfully commercialize virtual machines on the x86 platform. VMware uses two different architectures for their virtualization products: First, there are the original hosted VMMs that run on another OS such as Windows or Linux [SVL01]. They rely on binary translation to run on x86 CPUs without hardware extensions. Nowadays they can also benefit from hardware support for virtualization [AGSS10]. Second, there are the stand-alone hypervisors from the ESX server product line [Wal02, ESXb]. ESX has a similar architecture as NOVA with a dedicated hypervisor and one VMM instance per VM. However, the little information available indicates that the device drivers are part of the monolithic hypervisor. Furthermore, the ESX hypervisor seems to offer a POSIX-like interface to its applications, which is much larger than NOVA's small microkernel interface.

Xen started as a research project at the University of Cambridge, UK. It was commercialized via XenSource in 2004. Originally Xen aimed at a hundred paravirtualized Linux VMs on a hypervisor [BDF+03] to better utilize the available hardware resources. Support for hardware-assisted virtualization was added as soon as the x86 virtualization extensions became available. The first version of Xen took its device drivers from Linux and linked them directly to the hypervisor. A different architecture was introduced with Xen 2.0: the first virtual machine, called domain zero (dom0), provides drivers for the

[1] Example: Both set the interrupt-pending flag in the RTC (real-time clock) model only if interrupts are unmasked.

rest of the system [FHN+04]. Furthermore, dom0 hosts the management services and the Qemu instances that act as VMMs for the faithfully virtualized domains. While this architectural change solved the problem of the limited driver support, it also increased the TCB of a VM by the size of the operating system in dom0.

The **KVM** [KKL+07] development was started in 2006 by Qumranet to add virtualization support to the Linux kernel. KVM is not a stand-alone hypervisor, but an extension to the Linux kernel. Thus it can rely on Linux for booting, device drivers, and resource management, which makes it significantly smaller than Xen. KVM uses Qemu as its user-level VMM for device and CPU emulation. In contrast to NOVA, KVM emulates a second set of interrupt controller and platform timer inside the hypervisor. Even though this reduces the virtualization overhead, it also increases KVMs code size.

VirtualBox [Wat08] is a hosted virtualization environment for many operating systems such as Windows, Linux, MacOS, and even Solaris. Initially released in 2007 by Innotek, it is now maintained by Oracle. VirtualBox runs on CPUs with and even without hardware-assisted virtualization. In the latter case ring-compression techniques [KZB+91] are leveraged. This makes VirtualBox quite similar to early VMware workstation products, except that most parts of it are open source. Many device models of VirtualBox are derived from Qemu. An additional software layer makes these models rather heavyweight, but unifies device management and adds new features such as SATA and debugging support.

Microsoft entered the virtualization market by acquiring Virtual PC and Virtual Server from Connectix in 2003. These two products were hosted VMMs running on Windows as well as MacOS and quite similar to early VMware products. Virtual PC seems to still be the base for the so called *Windows XP mode*, which allows to run legacy applications on Windows 7. In 2008, Microsoft released *Hyper-V* [LS09], a stand-alone hypervisor for the server market. It requires 64-bit Intel or AMD CPUs with virtualization extensions. Hyper-V shares its architecture with Xen. It also has a privileged virtual machine, called the parent partition. Guests run in their own child partition. However, the OS inside the parent partition is not a paravirtualized Linux as in Xen anymore, but instead a Windows Server 2008. The VMM of Hyper-V resides inside the parent partition and emulates all device models with the exception of the Local APIC (Local Advanced Programmable Interrupt Controller), which is already implemented in the hypervisor for performance reasons. This is similar to KVM, which has interrupt and timer models inside the kernel. The hypervisor interface is much wider than in NOVA [Mic12]. Besides communication and resource management tasks, it also provides *virtualization assists* that allow a paravirtualized or *enlightened* VM to call the hypervisor to execute batched virtualization events instead of trapping multiple times.

Virtualization on Microkernels

Various people have leveraged hardware-assisted virtualization on existing microkernel-based systems.

Biemueller [Bie06] studied how the L4-API can be extended to support faithful virtualization. He modified the L4/Pistachio kernel to run virtual machines on Intel VT-x CPUs. His code seems to never have left the prototype state, even though the architecture is similar to NOVA. Furthermore, the evaluation section of the thesis is still unpublished, which excludes any performance comparison.

Schild et al. [PSLW09] ported KVM to the L4/Fiasco microkernel. Most of the KVM code could run unmodified inside the L4Linux [HHL+97] kernel, whereas Qemu provided the VMM functionality. In contrast to NOVA, the **KVM/L4** project did not try to mini-

mize the virtualization layer itself. Instead it aimed for a minimal impact of virtualization to other applications both for security and maintenance reasons.

With **Karma**, Liebergeld et al. [LPL10] aimed at a minimal VMM. By relying on hardware support for CPU as well as memory virtualization and omitting legacy devices as well as CPU mode support, neither device nor instruction emulation is necessary in their VMM anymore. Instead, the guest solely relies on paravirtualized I/O. Whereas this approach reduced the VMM to approximately 4 KSLOC, unmodified guest operating systems are not supported anymore. Operating systems need to be explicitly ported to the Karma environment.

Sandrini implemented a virtualization library for the Barrelfish OS called **VMkit** [San09]. It supports a minimal set of device models to run virtual machines on AMD processors with nested paging. Similar to NOVA the virtualization functionality is split between a kernel part and a user-level monitor. Unfortunately, the user-level monitor has direct access to the security-critical virtual machine control block (VMCB). It is therefore part of the TCB of all other components in the system.

Other Open-Source Virtualization Stacks

There are other open-source virtualization stacks not aiming at a microkernel-based system.

FAUMachine [SPS09] uses virtual machines for fault injection scenarios. While the CPU virtualization is provided by Qemu and KVM, it comes with its own set of device models. This includes modems and serial terminals seldom found in other virtualization software. Device emulation is done at a finer-grained level than required in a general purpose VMM. One can for instance remove a single RAM module during run time. This leads to a larger codebase, but it also means many parts of the device emulation tend to be incomplete.

Palacios is an open-source virtualization library developed in the V3VEE project [LPH+10]. It is released under BSD license and comes with its own device models. Even though it supports more virtual devices than Vancouver, their implementation is often incomplete. The virtual i8259 PIC (Programmable Interrupt Controller) for instance neither supports special fully-nested mode nor rotating priorities. Similarly the i8254 PIT (programmable interval timer) supports only five out of six timer modes.

2.1.3 Securing the Virtualization Layer

Many projects tried to strengthen the security of the virtualization layer. Most of them targeted existing implementations like Xen [MMH08, CNZ+11] or KVM [WWGJ12, WWJ13] whereas others propose to protect the hypervisor [WJ10, WSG10, ANW+10] or aim to remove it from the TCB of a VM [SKLR11]. In the following, I detail these approaches and compare them to NOVA.

Decomposing Xen's Domain Zero

The security of the virtualization stack is especially important on a Xen-like architecture where a full-fledged OS is used in domain zero (dom0) to manage and support the virtual machines. Two projects have aimed to reduce the TCB of Xen.

In 2008, **Murray** et al. [MMH08] introduced another privileged domain, called the domain builder (domB), to create new domains instead of performing this operation from dom0. Full memory access privileges can then be removed from dom0 user space, which

significantly shrinks the TCB. Interestingly, they borrow techniques from microkernel-based systems such as an IPC (Inter-Process Communication) abstraction, the use of an IDL (Interface Description Language) compiler to automatically generate IPC stub code [Aig11], and the forwarding of system calls to untrusted code [WH08].

However, in our opinion, this pioneering work missed a couple of opportunities. Foremost, the dom0 kernel is still part of the TCB because IOMMUs were not (widely) available in 2008, so that device drivers may compromise the whole system via DMA (direct memory access). Second, they did not aim for the minimal functionality in domB, but they argued instead that disaggregating Xen at a higher level makes it simpler and therefore more secure. Furthermore, they reused a surprisingly large *MiniOS* as base for their domain builder. Consequently, domB consists of nearly 10 KSLOC. This is as much code as the complete hypervisor in NOVA. Finally, hardware-assisted domains may not work anymore because the Qemu processes can no longer emulate DMA-able devices when memory access from dom0 user space is forbidden.

In 2011 Colp et al. [CNZ+11] continued this line of research with **Xoar**, a modified version of Xen. They partitioned the management and device-driver functionality found in the monolithic dom0 into nine different service VMs. A small TCB is achieved by giving these VMs only the privileges absolutely necessary for their operation. Furthermore the service VMs live only as long as they are needed. This reduces the surface an attack can target. The initial boot code, for instance, is removed from the system before any client VM is started. Finally, service VMs are rolled back to the initial state after each request, which should improve the security as well. Nevertheless, driver VMs are only restarted after several seconds due to performance reasons. This leaves a large window of opportunity for any automated attack.

Even though both projects share our goal of a smaller TCB for a virtual machine, they started from existing implementations. Consequently they were not able to reduce the TCB as far as we did because a large amount of legacy code had to be kept in the system. Especially the Xen hypervisor and the Qemu VMM make the TCB of a decomposed Xen VM more than an order of magnitude larger than in NOVA.

Improving KVM

Improving the security of a KVM-based system was the focus of the following two research projects.

HyperLock [WWGJ12] moved the KVM code into an address space distinct from the Linux kernel. The code still runs with full privileges, due to performance reasons. Additional protection is ensured by recompiling and verifying the KVM binary. Thus `vmread` and `vmwrite` instructions can be executed directly but Linux kernel data structures cannot be touched anymore.

DeHYPE [WWJ13] continues this work by moving the KVM code out of the kernel. The result is similar to KVM/L4 [PSLW09], except that the KVM code had to be explicitly ported to user level. Only 2.3 KSLOC remain inside the kernel to execute privileged instructions on behalf of KVM. Caching guest state limits the performance impact of this design.

Both projects do not significantly reduce the TCB of the overall system because the large Linux kernel remains. However, they limit the impact of bugs in the KVM codebase. Both add a certain overhead to each guest-to-host transition without having the fine grained transfer of VM state as in NOVA.

Removing the Hypervisor

NoHype [SKLR11] shows that a large hypervisor can be removed from the virtualization stack if the hardware is leveraged to partition a multi-core CPU. The hypervisor just boots the virtual machine and shuts it down if a virtualization event occurs. It does not need to virtualize any device nor provide any runtime service. Consequently, there is no support for memory overcommitment or for fine-grained scheduling of virtual machines in the NoHype system.

While this restriction might be acceptable in a cloud-computing setting where one OS runs on a set of dedicated CPUs for a long time, it severely limits the server-consolidation scenario. Furthermore, the hypervisor code is minimized by not handling any virtualization exit. However, at the same time, generality is lost. Legacy operating systems will not run on a NoHype system anymore. Instead, any OS kernel and application might need changes to not execute certain instructions like `cpuid`. Finally, the system seems to be vulnerable to denial-of-service attacks. A malicious VM may send any type of interrupt to other unrelated virtual machines and a malicious application can kill the VM without OS kernel involvement.

Protecting the Hypervisor

Several projects do not reduce the TCB but instead aim to protect existing hypervisors.

HyperSafe [WJ10] ensures control-flow integrity in the hypervisor by making code as well as pagetables read only and making jumps only through the edges of the control flow graph. However, the simplicity of their approach makes it easy to skip important security checks by returning to the second invocation of the very same function.

HyperCheck [WSG10] and **HyperSentry** [ANW+10] periodically measure the integrity of the hypervisor from SMM (System Management Mode). Thus they can only provide coarse grained protection against known classes of attacks.

Hypervisor protection can be applied to our system as well. Nevertheless the security benefits the small NOVA hypervisor gains might be less than the complexity it adds to the overall system.

2.1.4 Advanced Use Cases for Virtualization

Many papers describe advanced use cases for virtualization. Virtual machines are used to migrate operating systems from one physical host to another [CFH+05], for time-travel debugging [KDC05], intrusion detection [DKC+02], and to improve software rejuvenation [SAT09]. Most of this research has relied on an existing virtualization layer and can also be applied to our work.

Different projects try to harden the operating system inside a virtual machine or protect an application from a compromised OS kernel. For example, **SecVisor** [SLQP07] intercepts all kernel entry and exit paths at the hypervisor level to ensure that only kernel code runs in kernel mode. **BitVisor** [SET+09] on the other hand intercepts drivers accessing devices to ensure that DMA requests do not overwrite security-critical memory. This is similar to Mehnert's PhD thesis [Meh05], except that they also intercept the data path to add transparent encryption.

Overshadow "offers a last lines of defense for application data" [CGL+08], by giving an untrusted OS only encrypted access to it. The OS can still perform its resource management task but can neither modify nor leak application secrets. **CloudVisor** [ZCCZ11]

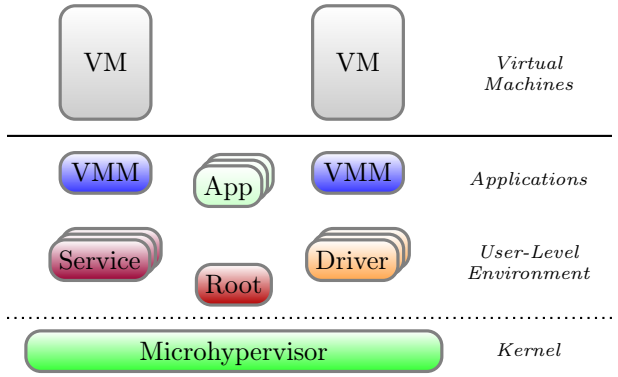

Figure 2.1: The NOVA architecture reduces the TCB of applications and VMs by splitting the virtualization layer into microhypervisor, user-level environment, and one VMM per virtual machine.

protects a guest operating system in the same way by adding another hypervisor underneath an untrusted virtualization stack with the help of nested virtualization [BYDD+10].

TrustVisor [MQL+10] uses a hypervisor to shield security relevant parts of legacy applications from the untrusted operating system. To achieve protection across system reboots, Trusted Computing techniques are employed.

Even though all these approaches are orthogonal to our work, most of them would benefit from the small and general-purpose virtualization layer NOVA provides.

2.2 Design

With NOVA, we aim to minimize the TCB impact of the x86 virtualization stack for both unmodified guest operating systems running in virtual machines and applications that do not depend on virtualization at all. We redesigned the architecture as well as the different software layers from ground up to achieve this goal.

In this section, I give an overview of the NOVA architecture (§2.2.1), briefly describe the hypervisor interface (§2.2.2) and the user-level environment (§2.2.3). At the end, I discuss the design of the Vancouver VMM (§2.2.4).

2.2.1 NOVA Architecture

The NOVA OS Virtualization Architecture [SK10a], as depicted in Figure 2.1, is based on a small kernel called the **microhypervisor**. The microhypervisor, in the following just called hypervisor, is the only part of the system, which is executed in the highest privileged CPU mode[2]. The user-level environment runs above the hypervisor. It consists of the initial program called the **root** partition manager that bootstraps the system. Additionally it provides **services** like a GUI, a network stack, or a file system, and the necessary device **drivers** for graphics, disk controller, and other hardware. This software layer provides OS functionality to native applications (**APP**) and to multiple **VMMs**, one for each **virtual machine**.

[2]This ignores system-management mode and CPU microcode, which are typically not under the control of the OS developer.

The NOVA architecture bridges the gap between microkernel-based systems and hypervisors [HWF+05, HUL06]. It inherits the small kernel and the implementation of OS functionality as well as user-level device drivers from microkernel-based systems [EH13]. It shares the support for running unmodified operating systems inside virtual machines with hypervisor-based systems [Gol73]. Finally, NOVA borrows many features from existing systems such as capabilities, portals, the use of IOMMUs, and the decomposition of the OS into smaller services [SSF99, GSB+99, LUSG04, GJP+00].

The hypervisor relies on hardware support for virtualization as provided by Intel VT [UNR+05] or AMD-V [SVM05] to run unmodified legacy operating systems in *VMX non-root mode* or in *SVM guest mode* respectively. The virtual machine monitor completes the platform virtualization by emulating devices and CPU features not virtualized in hardware yet.

Splitting the virtualization layer into hypervisor and VMM has several advantages:

1. It minimizes the TCB impact of virtualization. Native applications not depending on virtualization do not have the VMMs in their TCB.

2. It removes a potentially vulnerable VMM from the TCB of unrelated virtual machines.

3. It allows to tailor the VMM to a specific guest OS. If a VMM feature is not needed, it can be easily removed from the TCB.

4. It improves performance and simplifies the implementation, as the VMM needs to interact with a single guest only.

Virtualization support in NOVA is orthogonal to other hypervisor concepts. Any application can run a virtual machine and each VM could drive physical devices. Additionally, a VMM may offer operating system services. Legacy OS code can thus easily moved into a virtual machine while keeping its external interfaces.

2.2.2 The NOVA Microhypervisor

The NOVA microhypervisor was designed and implemented by Udo Steinberg in parallel to my work on the user-level environment and the Vancouver VMM. The hypervisor consists of approximately 10 KSLOC, which is significantly smaller than other implementations. This reduction in size is achieved by adopting a design principle of L4 [Lie95] and including only those features inside the hypervisor that could not be securely and efficiently implemented in user mode[3].

In the following, I briefly describe how the hypervisor supports native applications, OS services, user-level device drivers, and virtual machines. More details on the hypervisor design and implementation will be available in Udo Steinberg's PhD thesis [Ste15] and the NOVA Hypervisor Interface Specification [Ste11].

Native Applications

The hypervisor executes native applications inside PDs (*protection domains*). A PD is a resource container similar to a process in Unix. It is composed of three spaces: the virtual memory address space backed by physical memory or MMIO (Memory Mapped

[3]With NOVA we do not aim at the minimal kernel size. Instead the hypervisor includes several performance optimizations such as providing both IPC and shared memory for communication or having an in-kernel virtual TLB (Translation Lookaside Buffer).

I/O) regions, the I/O space holding the rights to PIO (Port I/O) regions and the object space consisting of *capabilities* to kernel objects. Capabilities are PD-local names to global kernel objects, similar to file descriptors in Unix. They are protected by the hypervisor and provide a unified way to manage access rights. If a PD possesses a capability, it can use the corresponding kernel object[4].

Multiple threads called *execution contexts* (EC) are scheduled by the hypervisor according to their *scheduling context* (SC). The SC defines the priority and time-slice length for a single EC on a particular CPU. All ECs within a PD run in the same address space. Concurrent access to shared variables can be synchronized with counting *semaphores* [Dij68].

OS Services

OS services are supported in NOVA through IPC, worker threads, and portals.

An application requests OS services, like a file or disk operation, via synchronous IPC. An IPC call transfers parameters between two ECs and *maps* access rights to hypervisor-managed resources like physical memory and kernel objects. Mappings are used to establish shared memory between PDs and to distribute capabilities in the system. The rights can be revoked at any time via recursive *unmap* [Lie95].

Worker threads in a service wait for incoming requests. They do not need an SC of their own because calling clients donate their CPU time. ECs with a higher priority help a worker to finish the current request. This limits the time a high-priority EC waits on a lower priority one (*priority inversion*) [SBK10].

An IPC message does not directly target a service EC. Instead a *portal* is used as a layer of indirection to hide the internal thread structure of a service. Clients can be identified through the portal they have used or by *translating* a client capability into the object space of the server.

If an EC causes a page fault or another CPU exception, the hypervisor forces it into an IPC through an exception portal. The exception handler can define, which part of the EC state will be transferred during such a call. This feature limits the overhead of frequent exceptions and user-level paging.

Device Drivers

The hypervisor manages all CPUs including their MMU (Memory Management Unit) and FPU (Floating Point Unit). Furthermore, it controls the IOMMUs (I/O Memory Management Units) of the platform if present. Finally, it drives the interrupt controller and at least one timer for preemptive multitasking. All other devices have to be programmed in user mode.

The hypervisor includes several features to support user-level device drivers similar to L4 [EH13]. First, it manages MMIO and PIO regions in the very same way as physical memory. Any PD can get direct access to device registers. This enables device programming through otherwise unprivileged code.

Second, the hypervisor converts taken interrupts into a semaphore wakeup. Any EC, which has access to such a semaphore, can block on it until the corresponding device triggers the interrupt. This lightweight mechanism is not only fast but also secure. As the hypervisor controls the interrupt path, it can prevent event storms by masking all interrupts nobody waits on at the interrupt controller or the IOMMUs.

[4]Several right bits allow fine-grained access control over the operations allowed with the object.

Third, the hypervisor ensures through the IOMMU that each driver can perform device DMA (direct memory access) only into its own protection domain. This protects the hypervisor and other PDs from malicious DMA. Furthermore it enables DMA transfers to user-level memory without requiring an additional memory copy operation.

Finally, the device driver features of the hypervisor are orthogonal to the virtualization support. Physical devices may be driven from a VMM or can even directly attached to the virtual machine.

Virtual Machines

The hypervisor multiplexes CPU support for virtual machines between multiple unprivileged VMMs. Each PD can act as virtual machine monitor and create one or more virtual CPUs (VCPU). VCPUs are special ECs that are not executed in an address space like threads. Instead they run in a VM environment provided by Intel VT-x or AMD-V.

If a VCPU accesses a virtual I/O device or uses a CPU feature not yet virtualized by the hardware, the CPU exits virtual-machine mode. The hypervisor either handles this condition on its own[5] or it suspends the VCPU and sends an IPC with the current register state to the VMM. The VMM emulates the necessary side effect of the faulting operation and resumes the VCPU by sending the updated registers back. A VMM may also *recall* a currently running VCPU to deliver interrupts timely.

The hypervisor allows the VMM to choose what part of the VCPU state is transferred for a particular exit reason. This feature can significantly reduce the serialization costs of VM exits [SK10a]. A `cpuid` exit for instance requires only five general purpose registers, whereas exits handled by the instruction emulator need the whole VCPU state.

The hypervisor hides many differences between Intel VT-x and AMD-V from the VMM. For instance, the VCPU state is unified, whereas the exit reasons are still vendor specific. Furthermore, the hypervisor already implements a virtual TLB to improve the performance on older CPUs that do not implement nested paging in hardware [BSSM08].

Virtualization is orthogonal to other hypervisor features. The VMM can use OS services as backends for its device models. A filesystem, for instance, can provide storage for a virtualized hard disk. Additionally a VMM may offer services on its own. Finally, applications may use virtualization to run existing library code within their original environment.

2.2.3 NUL: The NOVA User-Level Environment

In a NOVA system, multiple applications compete for the few physical devices of the platform. Not every VMM can get exclusive access to the devices its virtual machine requires, like a timer, a disk, or a network controller. Instead, the user-level environment needs to drive the devices and time or space multiplex them between different clients.

Furthermore, the hypervisor implements only basic resource management. Initially, it delegates all low-level resources like memory or CPU time to the root partition manager and provides mechanisms to securely forward access to other domains. However, it does not implement any particular policy to further distribute the resources in the system. Instead, this functionality has to be provided by a user-level environment. If it is properly implemented, one can run untrusted virtual machines and secure applications concurrently on the same host.

Porting older user-level environments written for L4 to NOVAs capability interface seemed to be too complicated. Moreover, their huge size would have significantly increased

[5] For instance virtual TLB related exits are handled directly in the hypervisor.

the TCB of nearly all applications. I therefore developed a new user-level environment for NOVA.

The NOVA User-Level environment (NUL) is an experimental OS that allows to run virtual machines and applications on the NOVA hypervisor. NUL is similar to Bastei/Genode [FH06] and L4.re [LW09]. The main difference is that it implements just the necessary functionality to run multiple virtual machines with a user interface, disc, and basic network access. Additional features like a file provider or the remote management of PDs were later added by Alexander Böttcher and Julian Stecklina. The source code of NUL can be found at [NUL].

NUL extends previous work on user-level environments in three areas:

1. It shares the component design with the VMM,
2. It uses a new communication protocol for a multi-server environment, and
3. It aims at scalability on multi-core platforms.

In the following, I explain these features in detail and mention areas for future work.

A User-Level Environment from Components

The Vancouver VMM is built from flexible software components, as described in Section 2.2. There are several reasons to use the very same approach for NUL as well.

Foremost, it enables the flexible placement of the device drivers throughout the software stack. Device-driver components may run in the root PD or in another stand-alone PD. They may also be linked to a VMM or an application which has exclusive access to a particular device.

Second, the component design allows to adapt the user level to the particular platform. A driver not required on a certain hardware, or a service not needed for a usage scenario, can easily be removed from the user-level environment and thus from the TCB.

Finally, it allows to share components between Vancouver and NUL. One example is the PS/2 keyboard driver that is also used by the virtual BIOS (`vbios`) to retrieve keystrokes from the emulated keyboard controller. Another example is the instruction emulator and the i8254 PIT emulation that is reused by the `vesa` driver to execute BIOS code to switch the graphics mode.

The Multi-Server Protocol

Initially, the services and device drivers were all running inside the root PD. However, this monolithic approach did not scale well with the increasing complexity of a growing implementation. Additionally, it conflicts with the goal of NOVA to build a system out of unprivileged parts. Consequently, the user-level environment had to be separated into dedicated PDs. To enable the evolution of NUL towards a multi-server environment, I developed the following multi-server protocol for NOVA.

The protocol involves three **subjects**: the server offering a service, a client using it and the parent connecting both parties.

Four different **capabilities** are used: i) The identity of the server and the identity of the client at the parent. ii) A pseudonym that hides the clients identity. iii) The portal where a server EC waits for client requests. iv) A session is later added to the protocol to reduce the overhead by letting the server cache data between subsequent requests.

Finally, the following **data structures** are referenced: Services are called by name, a null-terminated string without any defined semantics. A service can enforce a policy,

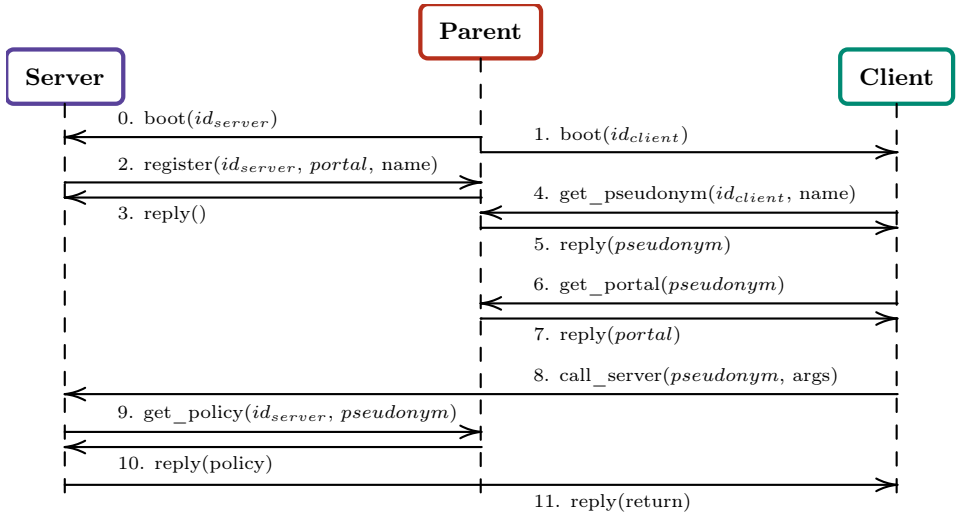

Figure 2.2: The session-less multi-server protocol in NUL.

which might be as simple as an `uid` for a UNIX-like filesystem, but can also be a complex ACL (Access Control List) or a quota for a certain resource. The called service may support any number of arguments, including an operation code to distinguish different suboperations. It can return any value.

Session-less The session-less version of the protocol is depicted in Figure 2.2. The **parent** is responsible for booting its subsystem. It thereby gives all of its children their identity capability to distinguish requests coming from them (id_{server} and id_{client} in the figure). The parents in a NOVA system form a tree rooted in the initial PD. This leads to a recursive OS structure similar to Fluke and Bastei/Genode [FHL+96, FH06]. Whereas client and server do not need to trust each other beyond the service requirements, there will be one common (grand-)parent they both have to trust for keeping them running and for letting them communicate with each other.

The **server** creates worker ECs and portals, one for each CPU. It then registers the service under a certain name by delegating the portal capabilities to the parent. The parent can deny such requests, for instance if the name is already taken or if the policy of the parent states that the child is not allowed to offer a service under this name. The parent may also forward the request to the grandparent to make the service more widely available to the system. It can either register the retrieved portals directly or can wrap the service by using its own set of portals instead.

A **client** that likes to use a service needs to request a pseudonym for his identity from the parent first. The pseudonym is an additional layer of indirection that allows the client to use multiple server connections with different rights. For instance, a library might need read-only access to a filesystem whereas the main application logic likes to have read-write access to another part of it. The pseudonym also enables the consolidation of multiple applications into a single PD. By splitting the right to communicate (the portal capability) from the right to a server object (the pseudonym or session capability), the number of capabilities in the server does not scale with the number of CPUs in the system anymore, but just with the number of connections each client opens. With the pseudonym, the client can request a portal from the parent and call the server through it. The server will

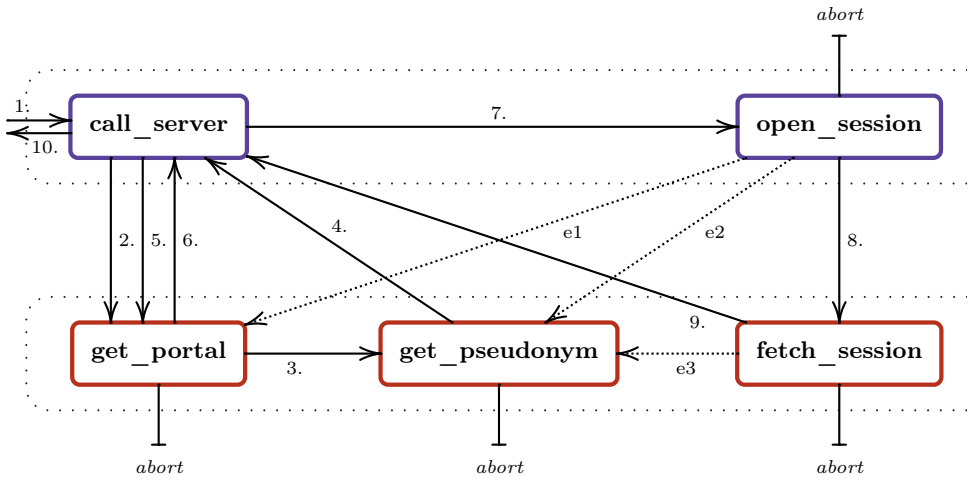

Figure 2.3: The session-based multi-server protocol in NUL from the client perspective.

use the client pseudonym to retrieve the policy from the parent it should enforce. It then handles the client request accordingly and replies with the return values.

Session-based Providing the policy to the server gives the parent fine-grained control over the client rights. Moreover, requesting the policy on every request makes changes immediately visible[6]. Unfortunately, it doubles the number of IPC calls, which slows down any server invocation. Adding sessions to the protocol will reduce this overhead through caching.

The simplest way to implement sessions on NOVA is to map a capability from the server directly to the client. This session capability can identify the client in further calls and allows the server to retrieve cached data including a previously requested policy. However, when implementing sessions this way, policy changes are not immediately visible anymore. Note that the parent cannot force the client to request a policy change. Even revoking the portals is not sufficient here because the two parties may have already established shared memory for faster communication. Besides calling the server from the parent can weaken the security as the parent not only depends on the policy enforcement of the server, but also on the timely completion of any request to it.

A more powerful, but also more complex mechanism, has to be employed: Instead of a direct mapping, the server will deposit the session capability at the parent under the pseudonym of the client. The client can then fetch the capability from the parent. Because the parent is involved in the capability distribution, it can revoke the session at any time. The parent can change the policy of a single session, but also ban a misbehaving client completely. Additionally, this mechanism does not only work for session capabilities, but also for shared memory and semaphores. Finally, it can be used in the opposite direction, when the client wants to give memory or other capabilities to the server.

Figure 2.3 shows how the session-based protocol is implemented by the client. For the first call to the server, the following transitions are made: 1. The client calls the server. 2. This fails because no portal is available yet. 3. Requesting the portal fails as well

[6]The NOVA hypervisor does not abort existing calls if the access to the portal is removed. Thus a policy change will be delayed until the currently running request is finished.

because no pseudonym was requested yet. 4. With the pseudonym the client can retry the call. 5. The call will fail again because no portal is mapped yet. 6. The portal is now available and the call can be retried a third time. 7. This fails because no session was opened yet. 8. The session capability needs to be fetched from the parent. 9. The client will retry the call with the portal and the session capability mapped. 10. The fourth call to the server succeeds.

In total, ten transitions are needed to securely establish a session between server and client[7]. As long as the portal and the session capability remain available, further calls to the server will immediately succeed. Thus, most server invocations will require only a single IPC including one translation of the session capability, which can be efficiently implemented as a single memory read.

Developing the parent protocol has led to the integration of the capability translation mechanism in the NOVA hypervisor. The hypervisor interface is now powerful enough for a multiserver environment where a hierarchy of parents starts services, which can be accessed from untrusted clients running on different CPUs.

Comparing the parent protocol to the corresponding protocols in Genode and L4.re [FH06, LW09] reveals three differences. First, the parent never trusts and therefore never calls one of its children. This saves resources as a thread per client is unnecessary. Second, the protocol fetches missing capabilities automatically. Thus policy changes become visible immediately. Reconfiguring or restarting a service is also easily possible. Third, the server and the client have little information about each other. The client neither knows the thread structure in the server, nor can the server correlate the pseudonyms of the same client.

Towards A Scalable User-Level Environment

The multi-server protocol defines the basic mechanism how a client can communicate with a server. It does not specify how a service should be implemented, which is especially difficult if the service has to run on multiple CPUs. In the following, I will briefly present the ideas we proposed in [SK10b] that should lead to a user-level environment scalable to many CPUs. They still have to be fully implemented in NUL.

The server should provide non-blocking operations, such that an application can submit multiple requests without having to wait on the hardware for completion. Additionally, the clients should provide the CPU time for all expensive operations. A secure filesystem like VPFS [WH08], for instance, will spend most of its CPU cycles on data encryption for write operations and decryption for read operations. Finally, the chosen server interface should not depend on the number of workers, the synchronization method employed or special hardware features.

A server invocation in [SK10b] consists of three phases: a synchronous request, an asynchronous completion signal and a synchronous retrieval of the reply. Whereas complex operations need all three phases, simpler ones may work with only a subset of them. The server interface need to explicitly expose these phases, such that IPC can be used for the synchronous parts.

Because a single threaded implementation can be a performance bottleneck, a service should instantiate at least one worker EC on each CPU. A worker will be used by all the client threads on the same CPU. Helping, as described in [SBK10], ensures that a preempted scheduling context and an aborted request will not block a high-priority requester longer than the WCET (worst-case execution time) of a single request. Relying

[7] The error transitions (e1, e2, e3) may further lengthen this path. However they are taken only if the parent revokes capabilities in between.

on helping reduces resources compared to a thread per client but leads to more jitter in the execution time. Splitting the interface into simpler functions and instantiating dedicated workers will improve both WCET and jitter.

Device drivers and asynchronous signaling requires IRQ threads to wait on semaphores and forward completion events.

Threads on different CPUs can access shared resources. However, a general and fast approach to thread synchronization does not exist. Recent research suggests that simpler synchronization primitives tend to be faster [DGT13]. Nevertheless, synchronization should be an implementation detail and not exposed at the interface level.

Current State and Future Work

NUL offers the following services to NOVA applications:

- It helps booting them through ELF (Executable and Linkable Format) decoding and the support for multiboot modules including a command line.
- It implements libc-like functionality like `printf`, `malloc`, and string functions.
- It manages the memory, PIO, MMIO, PCI config spaces, and interrupts.
- It provides network as well as disk access and allows applications to request a notification at a certain point in time.
- It forwards human input to applications and displays their textual as well as their graphical output.

Additional functionality, like a file service and remote management of PDs, were later implemented by Alexander Böttcher and Julian Stecklina.

NUL comes with the following device drivers[8]:

- PCI config space access (`pcicfg, pcimcfg`) and IRQ routing (`acpi, atare`),
- Timeouts via `pit` and `hpet`; wall-clock time via `rtc`,
- Disk access through `ide` and `ahci`; in-memory disk via `memdisk`,
- Simple network driver `ne2k`,
- Human input from PS/2 `keyboard` and mouse as well as from `serial` ports,
- Text output via the `vga` console and graphics through `vesa` mode switching.

My work on NUL has shown that the NOVA hypervisor interface is powerful enough to implement a multi-server environment. However, three performance optimizations still wait to be implemented. First, managing the UTCB (User Thread Control Block) as a stack would avoid most callers the effort to save and restore its contents during nested IPC calls. Second, if the hypervisor would export a CPU-local timer to user level, the cross-CPU synchronization overhead in the timer service would be unnecessary. Finally, the timer service could be completely removed if the hypervisor provided fine-grained timeouts.

In the future, one should complete the transition to the multi-server design and ensure that the servers scale to many-core platforms. Furthermore, one could add new

[8]The filenames are denoted in teletype font without their `host` prefix.

functionality. Foremost the simple FIFO approach to resource sharing does not fully decouple clients. A malicious client can significantly influence the throughput of other VMs. Adopting time-multiplexing approaches as in [PKB+08] to all dynamic resources (disk, network, graphics) could solve this issue.

Moreover, device drivers for USB (Universal Serial Bus) devices and for 3D graphic acceleration would extend the usage scenarios of NUL. Adding power-management support would enable mobile platforms. NUL could be used for daily work with a persistent filesystem, a TCP stack, and a shell. Porting development tools like compiler, editor, and a version control system to it would lead to a self-hosting system.

2.2.4 Vancouver: A Small VMM for NOVA

The design of Vancouver was driven by the goal of a small TCB and the limited amount of time, which favored any reduction of the development effort. In the following, I will describe the goals of the VMM design and explain how I used software components to achieve them. At the end of this subsection, I will compare this approach to related work.

Design Goals

Virtual machine monitors for the x86 platform are between 220 KSLOC and 2 MSLOC large as shown in Section A.2.2. The main reason for this huge size can be attributed to the support of multiple guest operating systems and different use cases at the very same time. However, most of the features are not required simultaneously. Qemu for instance comes with 24 NIC models even though a single model is sufficient to give a particular guest network access[9]. Similarly VMMs support sound output even in cloud computing environments where nobody could ever listen to it. The VMM should therefore be easily *specializable* to the particular use case to reduce the TCB impact of unneeded features.

Removing features should not require any programming skills. A system administrator or even an ordinary user should be able to configure the virtual machine accordingly. Similarly, existing VMMs often only emulate a standard PC platform, which excludes use cases beyond the original developers imagination, like using the VMM for chipset or BIOS development. A *flexible configuration* mechanism supports such cases.

The VMM code needs to be *reusable* on other platforms, without inflating the TCB through multiple layers of indirection.

The complexity of the x86 platform, together with the huge size of legacy operating systems, makes debugging of a VMM a time-consuming task. The VMM should therefore not be a monolithic implementation, but built from smaller parts that can be *independently tested* for correctness.

In summary, a design for Vancouver should have the following properties: specializable to the guest operating system, highly configurable, and build from reusable code blocks, which can be validated independently.

Software Components

The VMM design is based on a small number of abstractions: component, message, bus, and motherboard. In the following, I will explain them in detail.

Vancouver is built from independent software **components**. Most important are the device models (§2.3), the instruction emulator (§2.4), and the virtual BIOS (§2.5). Even

[9]Multiple models of the same device class are seldom required at the same time. For instance, if a scenario needs driver diversity to increase the reliability of the guest OS.

most of the platform-specific glue code holding all the components together is implemented as component as well. The granularity of the components varies. A device model for instance can resemble an entire chip on the mainboard or just one functional block of it.

Components do not depend on each other, which means they will not directly call other components or modify their internal variables. Components can therefore be written in different programming languages and multiple components of the same type can be instantiated. Finally, components can run in their own address space to isolate untrusted (third party) implementations[10].

Components interact by sending and receiving structured **messages** of various types. These messages resemble the communication that happens between real hardware devices. There is, for instance, a message type to signal interrupts and a message type that resembles the communication of the PS/2 bus. Furthermore, there are messages for memory, PIO, and PCI config space access. Finally, there are artificial message types not modeled after real hardware. Examples are the messages for communicating with device backends such as the timer, disk, and network drivers.

Messages do not directly target a destination component but they are broadcasted over a virtual **bus**. Thus, multiple components can receive the same message. This allows to easily snoop messages. It also ensures the correct semantics for many corner cases in virtualization. One example are two devices configured with overlapping address ranges.

The address decoding logic is not part of the bus itself. Instead it is implemented inside each component. Any message will be delivered to all receivers on the bus. A receiver decides internally how to handle a message. The selection of valid receivers is thereby marginally slower compared to a tree-based lookup. However, local decisions simplify the code, as there is no need to register and invalidate address ranges. Furthermore, the number of components on a bus is typically small, which limits the performance impact of a distributed implementation.

The C++ type system ensures that each bus supports only a single message type. Thus interrupt messages cannot be sent over a memory bus. This reduces the number of receivers on a bus and avoids type checks during runtime.

Components are created from the VMM command line where the user has fine-grained control over their parameters[11]. They add themself at creation time to one or more busses, which implements late binding of independent components. Thus one does not need to have a priori knowledge of other components in the system.

A virtual **motherboard** combines all known bus types into a single object. Multiple motherboards can coexist in the system where each one acts as its own namespace. This feature can be used to filter the messages going in and out an isolated component. The virtual BIOS, for instance, uses a motherboard to encapsulate the keyboard driver from the host in this way.

Example

Figure 2.4 shows how a keystroke from the console that was received by the VMM glue code, is forwarded to the guest. i) The key is broadcasted over the *keycode bus* where the keyboard model is listening. ii) The keyboard tells the PS/2 controller that its input buffer is non-empty by sending a message over the *ps2 bus*. iii) The controller fetches the keystroke and raises its interrupt line by sending a message to the *irq bus*. iv) The *PIC* snoops this messages and notifies the *vCPU* component that an interrupt is pending. v)

[10] Device models that support DMA need a virtual IOMMU for this purpose.
[11] Aliases for the most common configurations exists.

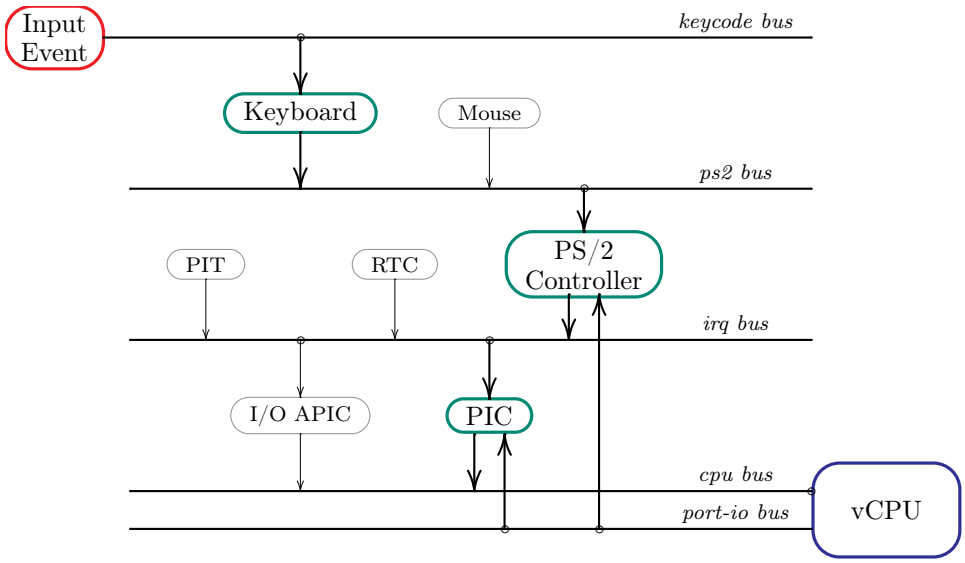

Figure 2.4: Multiple components have to interact with each other to forward input events to the VM. The PIC, for instance, listens on the *irq* and *port-io* busses and sends messages to the *cpu* bus.

The virtual CPU is recalled and the corresponding interrupt vector is injected into the guest. The interrupt handler calls the keyboard driver inside the guest. vi) The driver then reads the keystroke from the *PS/2 controller* via PIO instructions. This leads to VM exits that are converted by the VMM glue code to messages on the *port-io* bus. vii) The guest acknowledges the interrupt at the virtual *PIC* by using PIO instructions again.

Discussion

The concept of using software components was proposed multiple times to cope with the complexity of operating system construction [FBB+97, GSB+99, FSLM02, HF98]. To our knowledge, Vancouver is the first componentized VMM.

Vancouver uses fine-grained components like Think or MMLite [FSLM02, HF98], instead of following the more coarse grained approaches taken by the OSKit and Pebble [FBB+97, GSB+99].

Furthermore, components do not export a function pointer based interface as in COM [COM95], but instead receive broadcast messages as in the Information Bus Architecture [OPSS93]. This message-based approach makes Vancouver also similar to event- and flow-based programming paradigms [Fai06, Mor10].

Finally, the design is related to multi-server environments [GJP+00, FH06] where clients use IPC to call servers to obtain a certain service because both rely on message based communication to interact with each other. Vancouver therefore inherits many of the properties of a multi-server design such as:

Flexibility The flexibility to easily specialize the VMM to the guest OS has the most impact on the VMM size. The disk models, for example, can be easily removed if no storage emulation is required.

Additionally, one can create multiple instances of the same component. This allows

to add another network card or interrupt controller to a virtual machine without changing the source code.

Easier Development Splitting the codebase into many small components allows parallel development. Furthermore, as device models are basically stand-alone state machines with all external interaction exposed through the busses, they can be validated independently from each other.

Communication Overhead Communicating with messages is certainly slower than normal function calls in a program because arguments have to be marshalled at the sender and extracted at the receiver. Furthermore, the compiler cannot inline the called code. However, this overhead is much smaller than the cost of IPC in a multi-server environment.

Portability The messages provide a layer of indirection and enforce clear interfaces. This can be used to increase the portability of the code by splitting generic and OS-specific code into two components. The device models for instance can be independent from the underlying platform, if all backend accesses are forwarded via messages to OS-specific glue code.

The achieved portability eased porting Vancouver from F2[12] to the NOVA hypervisor. Currently, Vancouver and its forks run on NUL, Genode, NRE, and Linux in 32-bit as well as 64-bit configurations.

Structure

In this section, I presented the NOVA architecture and described the hypervisor interface. I introduced the user-level environment and discussed the design of the Vancouver VMM.

In the following three sections, I will focus on the VMM implementation. I start with the device models (§2.3), which are responsible for the majority of the VMM code base. This is followed by the single largest component the instruction emulator (§2.4). Finally, I describe how Vancouver provides BIOS functionality to its guest operating system (§2.5).

[12] F2 is an experimental microkernel that I have built for the AMD64 platform to extend the kernel-memory management research from my Master's thesis [Kau05].

2.3 Device Models

A device model emulates the behavior of a physical device in software to provide the illusion to the guest OS of running on real hardware. A model does not need to emulate the device in its entirety. Instead, it is sufficient to mimic the externally visible interface that is used by the device drivers. Similarly, it is not required to emulate the very same timing behavior of physical devices because depending on the exact timing is typically considered a driver bug[13].

In this section, I describe the device models of Vancouver. First, I explore different implementation approaches that may lead to smaller device models (§2.3.1). Second, I show that the standard technique of wrapping existing code is not a viable solution for Vancouver (§2.3.2). Third, I describe the implementation in general (§2.3.3) and with more detail (§2.3.4). Finally, I evaluate the code size of the device models (§2.3.5) and propose future work (§2.3.6).

2.3.1 Approaches to Reduce the TCB

In this subsection, I evaluate three approaches to device-model implementation that promise a smaller codebase: containerizing existing implementations, synthesizing the device models, and using paravirtualized interfaces.

Containerizing Untrusted Device Models

To reduce the TCB, one may take NOVAs fine-grained decomposition of the virtualization stack a step further and put untrusted device models into their own address space. One would use a virtual IOMMU to restrict DMA operations into guest memory. Finally, one would limit the interactions with the core of the VMM such as the number of interrupt injections or the number of backend requests. Together, these changes would ensure that a buggy model cannot fully take over the whole VMM and thereby remove the model from the TCB.

While this idea seems promising at the first glance, it has a couple of drawbacks. Foremost, it requires additional code to reuse and containerize existing device models. This code will come with its own defects. Moreover, the strict isolation between device models is only fruitful if the guest OS utilizes a virtual IOMMU. An attacker could otherwise easily circumvent the isolation through a DMA attack against guest memory. Furthermore, the guest OS needs to protect its device drivers as well for instance by using user-level device drivers. However, mandating an IOMMU and user-level device drivers reduces the number of operating systems that will benefit from the smaller TCB drastically. Most operating systems will suffer from the additional complexity. Finally, a further decomposition of the virtualization stack would increase the overhead as more context switches are required to handle a single event. This will be significant on devices with a high interrupt rate such as a 10 Gbit NIC or a high-end SSD (Solid State Disk).

In summary, containerizing untrusted device models reduces the TCB only for limited number of guests operating systems whereas adding non-negligible overhead to all of them.

[13]The *ne* driver in Linux cannot handle completion interrupts directly after the instruction that issued a request. The IDE driver of Windows XP has a similar bug.

Source-Code Synthesis

Another idea to reduce the TCB of the device models, would be to employ a code synthesis approach similar how Termite does it for device drivers [RCK+09]. Such a synthesis would be based on a formal device specification that can be more abstract than the corresponding low-level source code. It would therefore be smaller and easier to verify for correctness.

However, such an endeavor is limited by the unavailability of device specifications. Vendors, which are in the ideal position to release them, are hindered by the additional costs and liability issues. Most importantly, a complete device description makes it easy for a competitor to clone the device while having less costs for device verification and device-driver development.

Reverse engineering the device behavior from the driver as suggested by RevNIC [CC10] is possible as well. This task is more complicated for device models as one needs to analyze a large number of device drivers to cover all operating modes a chip provides.

Finally, the tools required to analyze and to generate the source code of the devices have to be counted to the TCB as well. They will have a significant TCB impact because they are not used very often. This is a minor issue on device-driver synthesis where hundreds of them exist in each device class whereas a VMM only needs a few device models per class.

In summary, synthesizing device models seems to be impractical.

Paravirtualizing the Guest

Instead of modeling existing legacy devices, one may also invent new interfaces [Dik01, Ste02, LPL10]. Following this paravirtualization approach has several advantages:

- It reduces the VMM size, as only one device model per device class needs to be provided. A single NIC model for instance is sufficient for all operating systems.
- A paravirtualized device is not bound to any legacy interface. One can easily design a simpler interface that minimizes the TCB because both sides, namely the device model in the VMM and the device driver in the guest OS, are under control of the developer.
- A better interface will reduce the overhead of device emulation.

Using paravirtualization has also disadvantages. Most importantly, one looses the ability to run unmodified operating systems. Instead, one needs to develop and inject a set of paravirtualized device drivers. This excludes closed-source operating systems that do not come with a driver development kit. It also results in a significantly larger implementation effort, which now has to scale with the number of supported operating systems instead of the smaller number of required device models.

Due to these reasons, I excluded the paravirtualization approach as well.

2.3.2 Reusing an Existing VMM

In the previous subsection, I explored three approaches for smaller device models. Whereas none of them was entirely satisfactory, I will now focus on reusing existing code to reduce the development effort but still gain a small TCB.

Many projects implemented their own hypervisor, but very few have built a VMM as well [Law96, Bel05, LPH+10, SPS09], mainly due to the significant effort this requires. Porting an existing codebase to another environment is an established technique to reduce

the development effort compared to a reimplementation of the very same functionality. Especially wrapping existing code on the source-code level, as done for instance by Flux [FBB+97] and DDE [Fri06], combines little effort with a certain forward compatibility. Furthermore, reusing an existing codebase allows to benefit from the development and bugfixes of a whole community. Consequently, several projects have derived their VMM from Qemu [Bel05]: Xen [BDF+03] has forked it, VirtualBox [Wat08] wrapped the device models, and KVM [KKL+07] seems to have overtaken the Qemu project.

Whether the device models from Qemu should be used in the VMM was controversially discussed at the beginning of the NOVA project. In the following, I will describe four reasons why wrapping an existing codebase will not lead to a small and secure implementation.

1. Wrapping Adds Code Software wrapping is required to benefit from further updates and bugfixes. However the wrapping layer adds new lines to the original code and thereby increases the TCB.

To evaluate the impact of wrapping, I ported two network models, namely `ne2k` and `e1000`, from Qemu to Vancouver. The models were originally 2400 SLOC (source lines of code) large. The reused Qemu infrastructure accounted for 1150 SLOC and 250 new lines of code had to be written for a wrapping layer. Altogether this was as much code as the twenty device models Vancouver supported at that time. Furthermore, a reimplementation of the two models should require no more than 1000 SLOC. Even though one saves approximately 750 SLOC through code reuse, the overall code size is increased significantly. The very same effect can be observed on VirtualBox where wrapping the device models from Qemu more than doubled their size.

2. Incompatible Goal Foreign code is seldom written with a small TCB in mind. Instead, the code might be optimized for performance or written in a hurry to achieve a short time-to-market. The developers might also aim for a minimal maintenance effort. One example are Bochs device models that are written in a verbose coding style. Consequently, they are twice as large as our own as shown in Section 2.3.5. Reusing such an inflated codebase will not result in a small TCB.

3. Inheriting and Introducing Defects When reusing legacy components one inherits all the defects in the code. Furthermore, by wrapping legacy code with another software layer new lines are added that may itself contain bugs. Additionally, the component may be used in a way it was never tested before. It is therefore likely that more bugs emerge.

Defects in existing virtualization code are not an abstract problem. Figure 2.5 provides an (incomplete) list of bugs that I found in open-source virtualization software while implementing Vancouver. These defects were only detected after getting familiar with the documentation and the experience of implementing similar code. Even if most of these issues are not security critical, they can lead to subtle misbehavior of a guest that is hard to diagnose and repair.

4. Limited Benefit of the Community Peter et al. [PSLW09] have argued that reusing an existing codebase should be preferred over a new development because it allows to benefit from the development of a whole community. To argue to the contrary I describe three cases where the open-source virtualization community could not help:

Device	Codebase	Description
CPU	kvm xen	decode only 8 prefixes instead of 14
CPU	qemu	eventinj fails with #PF
CPU	qemu	invlpga does not generate a #GP on user access
CPU	qemu	ioio intercept fails on unaligned access
CPU	qemu	ioio intercept just checks single bit
CPU	qemu	rdpmc intercept missing
CPU	qemu	cpuid 0x8000000A unimplemented
CPU	qemu	SVM rdtsc intercept has wrong priority
CPU	qemu	LTR allows to load 0 selector
CPU	qemu	vmptrld does not generate a #GP
CPU	qemu kvm	more than 15 byte opcode length - no #GP
CPU	xen	locked operation in emulator
PIC	bochs	does not check for iolen
PIC	qemu	poll mode does not set the ISR
PIC	qemu	special mask mode missing
PIT	qemu	periodic modes starts with new counter immediately
PPI	qemu xen	dummy refresh clock instead of PIT_C1 output
RTC	all	irq flags not set when irq disabled
RTC	qemu	12hour format not in range 1..12
RTC	qemu	alarm test broken
RTC	qemu bochs	seconds bit7 not readonly
RTC	qemu bochs	regs visible during update-cycle
APIC	all	change of DCR causes CCR jumps
APIC	qemu	off-by-one when calculating periodic value
HPET	qemu	IRQs are re-enabled in legacy mode when reprogramming the PIT
HPET	qemu	does not support level triggered IRQs
KEYB	bochs	panic when OUTB and cmd-response needed and on resend
KEYB	bochs	panic when controller buffer overflows or written in the wrong order
KEYB	qemu bochs	immediately reset
KEYB	qemu	SCS1+SCS3 does not work
KEYB	qemu	output port IRQ1+IRQ12 are set both
KEYB	qemu	overflow could result in corrupted data
KEYB	qemu	read-mode response is put into aux buffer
KEYB	qemu	cmd response buffer missing
KEYB	qemu	does not indicate a scan-code buffer overflow
KEYB	qemu	keyboard buffer too large
MOUS	qemu	does not apply scaling
UART	bochs	iir id bits wrong
UART	qemu	does not implement irq disable on OUT2 and character loopback

Figure 2.5: Various defects were found in open-source virtualization software while implementing Vancouver during 2008/09. Most of them are fixed now.

- In mid 2009, I discovered an ancient bug in the RTC model of Qemu [Kau09b] while developing a driver for NOVA. The model does not set the update-in-progress interrupt flag, which will livelock a guest operating system that depends on this behavior. This defect can be tracked through all Qemu versions and up to the first public release of its predecessor Bochs. This bug also existed in Xen, VirtualBox, and Palacios. More than 10 years ago a developer had missed one sentence in the manual. Afterwards, everybody had copied the wrong behavior and nobody has made the effort to look into the documentation again. It took more than 2 years to get this bug fixed in Qemu and another year for Xen. Even after more than 4 years this bug is still present in Bochs, VirtualBox, and Palacios. A bugfix for one codebase can take years to spread to another.

- Xen, Qemu, and Palacios emulate the i8259 PIC each to a varying degree. Whereas Xen correctly supports special-mask mode and special-fully nested mode, Qemu does not implement the former, and Palacios does support neither of them. Because these are all open-source projects nobody is systematically filling feature gaps. Every new OS version might trigger defects or virtualization holes throughout the whole codebase. However, debugging such issues, especially if they are timing or interrupt related, is very hard. It may take days to find out why an interrupt is lost under certain load conditions.

- If nobody takes the trouble to understand the low-level details anymore and everybody just ports existing code to another environment, the numbers of experts and with it the quality of a codebase decreases over time. In the end more bugs will be introduced. This is especially visible in handling the plenty corner cases that rarely trigger.

  One example was a patch on the Qemu mailing list [Poe09]. For the maintainer the explanation was sound, however the reporter had misunderstood the documentation. If there are no experts that detect such things, new defects will be added to the codebase. I therefore see a clear risk of reusing existing low-level code without becoming an expert in the field, as this means to depend on the future of a community one does not actively participate in.

In summary, solely depending on an open-source community is limited by long-standing defects, incomplete features, and missing experts.

2.3.3 Implementation

In the previous two subsections, I showed that neither a different implementation approach nor wrapping of existing code will lead to small and secure device models. Instead, I choose to build the necessary device models from ground up. In this subsection, I describe general implementation issues that affect all device models. In the following, I will focus more on the specific details.

In the following, I show how the component design is applied to the device models. I then discuss two features to significantly reduce the code size, namely externalizing checkpointing and debugging functionality. I list the reasons for choosing device models to implement and describe my approach to increase the overall code quality.

Implementing the Component Design

The design of Vancouver, as outlined in Section 2.2.4, provides the framework for the implementation of the device models. However, it does not define on what granularity the software components, the busses, and the messages are employed. I used the following approach to implement the device models:

One Component per Instance One component is used for each device instance in the virtual platform. The device-internal state is thereby kept in private class variables and only modified through incoming messages. These messages originate in VMM glue code to forward VM exits or backend completion events. Messages can also be received from other device models.

Multiple instances Multiple instances of the same device are multiplexed through a bus. This centralizes the multiplexing code and allows starting a VM with any number of virtual device instances. It is, for instance, possible to have a VM with multiple PCI host bridges or with three instead of the usual two i8259 PICs. The actual configuration of the virtual platform is thereby specified at the (multiboot) command line.

Backend Communication Communication with performance-critical backends such as timer or disk are done asynchronously. Messages issued on a request bus include a tag that identifies the response later received on a completion bus. This design improves the virtualization performance by allowing multiple outstanding requests. This parallelism can even reach the physical device if its interface and the driver supports request queues like NCQ (Native Command Queuing).

Checkpointing

The ability to checkpoint a virtual machine is the base for several advanced techniques. Most importantly, it allows the (live) migration of virtual machines between different physical hosts for load balancing or software maintenance reasons [CFH+05, LSS04]. Furthermore, it can be used to improve the fault tolerance of a virtualized environment [GJGT10].

The ability to checkpoint requires support code in the VMM to save and restore the guest memory as well as the state of the virtual CPUs. Furthermore, the VMM needs to preserve the state of the device models. Whereas the first is easily accomplished as the guest state is directly visible to the VMM, the second point requires additional lines of code.

Traditional implementations add save and restore functions to every device model. Each member of a device model class[14] that needs to be preserved, leads to one line in both functions. Newer implementations, for example in Qemu, reduce this code by defining the offsets as well as the types of the class members only once. Furthermore, they heavily rely on macros to reduce the common code. However, the manual effort of keeping the save/restore functions in sync with the definition of the class remains.

I therefore investigated how the maintenance effort for the checkpointing code and with it the TCB of the VMM could be reduced. The basic problem in the traditional approach is that the data layout of the device model is defined twice: when specifying the object layout in a `class` statement and then again when adding checkpointing support.

[14] Whereas this subsection assumes a C++ implementation, the very same holds true for VMM implementations in other programming languages like C.

If the data layout could be extracted from the first definition, one could automatically generate a C++ function to save/restore the members.

I implement this functionality in small Python script by using GCC-XML, an extension to GCC that outputs the C++ parse tree in XML format. Class members that should not be saved had to be manually annotated in the class definition. To minimize the number of changes to existing code I used the following default rule: integral types are saved whereas pointers are not saved. This reduced the annotations to the cases where host specific values such as file descriptors or time offsets had to be excluded. Additionally, the script implements a simple, but powerful optimization. All members that are contiguous in the object layout are saved with a single `memcpy`. Manually shuffling the class members ensures that this optimization can be used in nearly all cases. Even though this approach reduced the manual written checkpointing code to 150 lines of Python, it required special compiler support and source code access. With the experience later made with VDB, as described in Chapter 3, I would nowadays extract the object layout from the DWARF debug information that are already produced by may compilers and present in debug builds even of closed-source programs.

In discussions with Jacek Galowicz during his work to add live migration to Vancouver [Gal13], we found a much simpler approach to VM checkpointing. If one treats the VMM as an ordinary process, special code is only needed to reconnect the backend connections and update the host specific state. This observation was also independently made by [GSJC13]. Note that this process migration approach is less powerful than the previous device migration approach as it excludes software updates of the virtualization layer and limits migration to similar host platforms.

In summary, checkpointing is a device model feature than can be externally provided. Device models do not need to have save/restore functionality. Instead a generic process-checkpointing approach is sufficient for migrating virtual machines between similar hosts. Finally, one can derive the checkpointing code from the debug information, if migration between different software versions should be supported as well. Externalizing checkpointing code reduces the size of the device models.

Debugging

Defects in the VMM are much more likely than in hypervisor or even application code due to the high complexity of badly documented hardware[15]. Especially interrupts and timing requirements make the device models more complex and thus error prone. However, each defect can crash or livelock a VM or, even worse, lead to silent data corruption. Furthermore, a device model needs to implement many corner cases that are rarely executed, but required for full device emulation[16]. Defects in such code paths may be hidden for a long time.

The guest can be faulty as well: It can make wrong assumptions on the functional[17] and the timing behavior of the device[18].

[15] Publicly available hardware documentation usually aims at the driver/OS developer but seldom includes all details a chip designer needs to rebuild the device.

[16] The RTC supports daylight saving mode that is only meaningful twice a year in the US. Similarly, a ne2k NIC has to signal an error if the OS cannot get received packets fast enough. This seldom triggers with fast CPUs but it happens in a VM that runs on an overloaded hypervisor.

[17] Linux 2.6.24 crashes if the AHCI (Advanced Host Controller Interface) CAP register reports too many enabled ports.

[18] The `ne` driver from Linux assumes that it can execute a couple of instructions before a request completes. These assumption breaks in a VM and on physical hardware if an SMI (System Management Interrupt) delays the driver until the interrupt fires.

As the publicly available documentation is often incomplete, a VMM developer is forced to either guess or reverse engineer the behavior of the physical device. However, the driver developer might have guessed differently or the OS just violates existing standards[19]. Subtle differences between physical and virtualized hardware may break a virtual machine.

However, a rich debugging infrastructure conflicts with a small TCB. I therefore used the approach presented in Chapter 3 and employ VDB (Vertical DeBugging) to remotely control the machine without increasing the VMM codebase. With VDB one gets full access to the hypervisor, VMM, and guest memory. Furthermore, it allows to inspect the state of all components even if the guest is running. Debug code in the device models is therefore rarely needed, which reduces their size.

However, manual inspection is not sufficient to reveal the dynamic behavior of the system. I therefore used a simple tracing approach based on `printf()` output to a remotely readable tracebuffer to record the interactions between different VMM components. Lightweight profiling counters, which measure event frequencies, complete the debugging features available in Vancouver.

Selecting Device Models

The developer of a VMM faces the choice what device of each class to model. Whereas Linux for instance supports hundreds of network cards, emulating just one of them is sufficient to get network access from a VM. I used the following criterions to decide what devices to model:

Mandatory Even though hundreds of different storage and network controllers exists, there are a couple of devices that are mandatory to implement because nearly all x86 operating systems depend on them. Especially platform devices such as interrupt controller or the real-time clock fall into this category.

Coverage Not all operating systems have drivers for all devices. Choosing a device that is covered by many systems reduces the number of models in a particular device class. It is therefore useful to implement standardized device interfaces like AHCI that are supported by many vendors instead of implementing proprietary ones.

Documentation The value of good documentation of the device behavior and the programmers interface cannot be underestimated. Any missing information complicates the implementation because one may need to reverse engineer device drivers, probe the physical device, or at last resort correctly guess it. Devices with simple and well-documented interfaces are therefore preferred over complex ones where little information is available. However, this criterion needs to be balanced against the performance impact of implementing a simpler interface.

Performance The virtualization overhead can be reduced by choosing a device model that leads to less VM exits during device programming. For example, issuing a disk request to an emulated IDE controller needs twelve PIO exits whereas the same on a virtual AHCI controller requires only six MMIO exits.

Some devices need no exit at all by resorting to polling-based emulation where a thread running on a second CPU periodically checks whether the guest has added

[19] Windows XP does not search the RSDP, which indirectly points to the ACPI tables, in the EBDA (Extended BIOS Data Area) as mandated by the ACPI (Advanced Configuration and Power Interface) specification.

new entries to a request queue. This minimizes virtualization overhead while increasing overall CPU utilization.

Similarly, emulating a device with interrupt coalescing support reduces the virtualization overhead by accepting more latency. The `e1000` driver in Linux, for instance, accepts an additional receive delay of up to $50\mu s$ by limiting the interrupt rate to 20,000 events per second.

If the device behavior is well documented, the actual implementation of the device model becomes straight forward. One has to basically emulate the state machine of the physical device.

Increasing the Code Quality

Finding defects in the VMM, while running a full-blown guest OS on it, is an elaborate task. One not only has to debug VMM code, but also the (closed-source) OS. Furthermore, one may not find race conditions or defects in one of the plenty corner cases using this approach. It is therefore reasonable to avoid such defects in the first place. I tried to increase the code quality as follows:

- I limited the complexity by finishing existing device models before implementing new ones. By implementing them in one step, one can better acquire a deep understanding of the device behavior instead of paying the additional effort to refresh and extend the knowledge every time a new feature has to be implemented.
- I ensured that a missing feature can be easily detected, by not letting the VMM silently fail if it is triggered. Instead the VMM will panic or at least output a warning.
- I classified the VMM code into three categories. Approximately one third of the code was throughly reviewed. It should require only few updates for feature extensions and hopefully no bugfixes. This code is called *stable*. Another third of the VMM codebase has not received the same amount of review. It is in the *testing* state. The remaining code is actively developed and too *unstable* for elaborate testing.
- I experimented with unit tests to verify that all state changes of a model are inline with the documentation. The component design makes it easy to test a single or even a group of device models in isolation.

  I implemented table driven tests for the `pit` and the `rtc` model. Each test was approximately twice as large as the device model itself. Writing one test took approximately one day. It will likely be harder for newer and more complex devices. Unit testing has increased the code quality significantly. Even after several years the two device models were involved in only two defects. Both were found outside of the tested core logic[20].

In the future one could use comparison based testing, as proposed in [MPRB09, MMP+12] for instruction emulators, to compare the behavior of different device model implementations. Additionally, test cases generated through symbolic execution of different device drivers could ensure OS compatibility with less effort compared to specifying the tests manually.

[20] The RTC miscalculated the current time during reset. A negative timeout from the PIT was misinterpreted by the backend.

2.3.4 Implementation Details

In the following, I will briefly describe the different device models I have developed and mention noteworthy implementation details.

Interrupt Virtualization

IRQs (interrupt requests) on the x86 platform can follow different paths until they reach the CPU. The original IBM PC relied on a single i8259 PIC (Programmable Interrupt Controller) to multiplex the interrupt pin of the CPU between eight devices [PIC88]. The IBM PC-AT introduced a second cascaded PIC to support up to 15 interrupt sources. In newer platforms I/O APICs (I/O Advanced Programmable Interrupt Controllers) [Int96] are used, which typically come with 24 input pins each. I/O APICs send interrupts to the Local APICs. A Local APIC is an interrupt controller in the CPU. It was added to the platform to support multiple processors. MSIs (Message Signaled Interrupts), available since the P4 and PCI v2.2, are replacing the interrupt pins on PCI devices with special messages on the data bus. An MSI bypasses the traditional interrupt controllers and directly targets one or more Local APICs.

Vancouver models all of this functionality as accurate as possible. This is especially important because a lost interrupt can stop the whole guest OS. Vancouver supports edge- and level-triggered as well as spurious and non-maskable interrupts. Edge-triggered interrupts are just interrupt numbers sent as messages from a device model to the virtual interrupt controller. Level-triggered mode allows to share an interrupt line between multiple devices. Furthermore they are used by timer models to dynamically adapt their frequency to the system load. A level-triggered interrupt message additionally requests a callback from the IRQ (interrupt request) controller that is triggered by the guest when it acknowledges the interrupt. This mechanism allows the device models to re-raise the interrupt if the reason to do so still exists.

Interrupts usually originate in a backend completion message received by a device model. This leads to a message from the device model through the interrupt controller hierarchy. At the end, the virtual CPU gets recalled. This ensures that the virtual CPU will fault to the VMM to inspect the `vcpu`[21] interrupt state before it executes any further guest instruction. If the interrupt state indicates a pending interrupt, a vector is requested either from the `lapic` or from the `pic8259`. This request races with a failing edge message and may therefore lead to a spurious interrupt. The interrupt handler in the guest will later acknowledge the interrupt and allow it to happen again.

The `pic8259` device model uses the PIC bus instead of a fixed master/slave design commonly found in other implementations. It therefore supports up to 8 slaves connected to the master PIC and additional PICs driven in poll mode. This resembles the original hardware more accurately than other implementations and correctly virtualizes many corner cases such as a misconfigured slave PIC.

The `ioapic` does not implement the original 3-wire APIC bus. Instead it models a P4-like I/O APIC that sends messages to the `lapic`. Similarly the `msi` component translates upstream memory accesses from device models to the MSI address range into messages targeting the corresponding `lapic`.

The `lapic` model prioritizes the different incoming interrupts and forwards them to the `vcpu` model. Lowest-priority interrupts are delivered round robin due to performance reasons, as otherwise the current priority of all CPUs need to be inspected. The `lapic` model can also be accessed through the x2APIC interface but without supporting the

[21] Teletype font indicates the prefix of the file the device model is implemented in.

recently added TSC deadline timer, yet. Another implementation challenge was to make the `lapic` model purely optional, if it is put into hardware disabled state. This feature is seldom found in other implementations.

Timer Device Models

An x86 operating system can use different devices as time stamp and timer interrupt source [VMw11]. First there are the i8254 PIT (programmable interval timer) and the mc146818 RTC (real-time clock) that date back to the IBM PC and the IBM PC-AT respectively. Newer CPUs include the TSC (Time-Stamp Counter) and a timer inside the Local APIC. ACPI compatible systems come with a fixed-frequency PM timer that is used by newer OSes for timer calibration. Finally there is the HPET (High Precision Event Timer) [HPE04]. Vancouver emulates all of these devices, with the exception of the HPET, which was not required by any tested guest yet[22].

The current value of a timer is calculated by shifting and scaling guest time, which is itself derived from the host TSC. The timer state is changed only if a guest driver accesses the timer or if a previously programmed timeout happens. This lazy update mechanism improves upon other implementations, which commonly update for instance the RTC memory twice every second, even if the guest never observes these changes.

Timer models in Vancouver request absolute timeouts at the backend for both periodic and one-shot timer interrupts. Note that a guest OS may program a periodic timer with a frequency higher than it can temporarily handle, for instance due to contention in the host. The VMM needs to cope with this overload situation because requesting to much timeouts at the backend may livelock the VMM without giving the guest a chance to stop the resulting interrupt storm. Periodic timer interrupts are therefore modeled level triggered. This means a timer model delays the programming of the next timeout until it receives the message that a guest has handled and acknowledged the previous interrupt. The model can skip full timer periods at this time and thereby scale down the timer to a frequency the whole system can handle. The resulting jumps in guest time can be eliminated with a sophisticated TSC virtualization strategy.

Human Input Devices

Human input to the guest OS is available through serial port and PS/2 emulation. Modeling the PS/2 hardware was straight forward after the PS/2 hardware documentation from IBM was discovered [IBM90].

The `serial16550` model emulates all features of a National Semiconductor 16550 UART on the character level. This means the baud rate, as well as the number of stop and parity bits, can be configured but they are ignored by the model. Characters are sent and received independently of these settings. Consequently, the model does not exhibit the same timing as real hardware. Bridges between keyboard and serial port exists, so that a host keyboard can drive a serial port of a guest and vice versa.

The `keyboardcontroller` model implements a PS/2 compatible controller. It provides two ports on the PS/2 bus where keyboard and mouse models can listen. It is also possible to plugin two keyboard models or use multiple PS/2 controllers in a system. The model supports seldom used features such as password protection, platform reset and A20 masking. The only missing feature is the upstream propagation of the keyboard LEDs state.

[22] Mac OS X seems to mandate an HPET.

The `ps2keyboard` receives *make* and *break* codes from the backend in scancode set #2. The other scancode sets are translated from this format. The keyboard model is fully implemented with the exception of typematic keys, which would need a timer to periodically resend scancodes for longer pressed keys.

The `ps2mouse` supports packet merging, poll mode and coordinate scaling. It does not support a scrolling wheel yet because the backend does not report the required z-coordinate.

Graphics

The effort to virtualize graphics is reduced in Vancouver by relying on as much host functionality as possible and preferring time multiplexing over full emulation.

The GUI supports all graphic modes available through the VESA BIOS of the host. It also offers VGA text mode #3 including the hardware-accelerated cursor. Thus, a guest can often write directly into a linear framebuffer without inducing a VM exit. Only rarely used VGA modes like mode #13 need to be fully emulated by the VMM.

The guest framebuffer is periodically displayed by the GUI. Because frontend and backend use the same video mode, the output does not need to be scaled or color converted. A `memcpy` from the guest to the host framebuffer is sufficient. Mapping parts of the host framebuffer directly to the guest is also feasible within this design, even though it was not implemented yet.

The BIOS fonts and the mode information are taken from the host. Reusing host functionality improves the performance and simplifies the code. However, it also limits the migration of virtual machines to hosts with a similar graphic card [Gal13].

The `vga` model is still incomplete but already implements VGA text as well as VESA graphic modes, most device registers and many VGA and VESA BIOS functions.

Storage

An IDE controller is supported by nearly all operating systems whereas the newer AHCI controller with SATA disks promises better performance due to its DMA and NCQ features. I therefore implemented both controllers. Emulating SCSI disks for performance reasons as done in VMware and VirtualBox is not necessary anymore.

The IDE controller is quite simple as it supports only PIO operations. A corresponding bus-master DMA engine was not implemented. If disk performance is crucial for a scenario, the newer AHCI model should be used instead.

SATA support is more complicated. It is therefore separated into AHCI controller and SATA disk emulation. The former is a PCI device that forwards requests taken from an input queue to one of the 32 disks. The latter interprets the SATA commands that are tunneled through AHCI and executes the DMA programs. The DMA descriptor can often be translated directly to the backend format. However, large DMA programs have to be split into multiple requests. Similarly, disk requests that are composed of many small memory snippets may require double buffering.

Both device models support only hard disks. Implementing ATAPI (ATA Packet Interface) commands for CD and DVD virtualization is left as future work.

Network

To enable basic networking between virtual machines I implemented a `rtl8029` model within less than 300 SLOC. This PCI NIC is ne2k compatible and supported by most

operating systems. Unfortunately, the PIO based interface can require hundreds of VM exits to receive or send even a single packet. It is therefore one of the slowest device models. Even achieving a throughput as low as 10 Mbit per second is challenging with this interface.

Emulating a newer gigabit or even 10 Gbit NIC that supports DMA and IRQ coalescing results in much better performance. Furthermore, modeling just one virtual function of an SRIOV NIC such as the Intel 82567 is even simpler because this abstracts away all link-level operations. However, the implementation of a faster network layer including newer device models and a virtual network switch is beyond the scope of this thesis and pursued by Julian Stecklina instead.

Legacy

An x86 platform comes with many legacy features that have to be emulated by a VMM. The ability to externally mask the 20'th address line (A20) of the CPU is probably the most widely known one. This feature was introduced with the PC-AT to be backward compatible with the original IBM PC. Vancouver implements the keyboard, BIOS, and fastgate method to toggle the A20 pin. However, the `vcpu` model currently ignores the A20 state for complexity reasons. Correctly supporting it would require a different address space for I/O devices. Alternatively one could force a CPU with A20 masking enabled into the instruction emulator.

Another less known legacy feature are bits 4 and 5 at system control port B (PIO 0x61). These bits are used by low-level software like the VESA BIOS or boot code to wait for a small amount of wall-clock time (`udelay`) before programming timing sensitive devices. Many VMM implementations incorrectly toggle these bits at each read or just give them a certain frequency. However, the state of these bits is determined by the output of the first i8254 PIT. Bit 4 reflects the output pin of the second timer, originally used to refresh DRAM whereas bit 5 reflects the output pin of the third timer that drives the PC speaker[23]. Vancouver correctly virtualizes these pins and thereby allows a guest to freely choose their frequency.

2.3.5 Evaluation: Code Size and Density

Aiming for less code and externalizing checkpointing as well as debugging functionality should make Vancouver's device model smaller. In this subsection, I will compare selected device models with corresponding implementations in Qemu, Bochs, and VirtualBox (VBox) to quantify this effect. I will also measure the code density to exclude coding style effects.

For the size comparison I take only those device models into account that are present in all implementations. Furthermore, to not overestimate the size reduction achieved, I've chosen only those device models from Vancouver that implement equal or slightly more functionality. Figure 2.6 shows the result of the comparison[24].

The device models of Qemu are on average 40% larger than our implementation. If the models are well written such as in the PS/2 and the I/O APIC case the sizes are nearly on par. If superfluous functionality is implemented such as programming a timer to update the RTC memory twice a second, Qemu's models can have up to 80% more lines of code. Bochs device models are on average twice as large as our implementation. The main reason is code duplication. The logic for the two PICs, for instance, is implemented

[23] Vancouver currently misses PC speaker output as no sound backend is available.

[24] The measurement was done on repository snapshots taken in November 2011.

Model	Vancouver	Additions	Qemu	x	Bochs	x	VBox	x
PIC	310	multiple instances	436	1.4	793	2.6	767	2.5
PIT	348	cycle accurate	428	1.2	948	2.7	892	2.6
RTC	303	polling mode	535	1.8	632	2.1	755	2.5
Local APIC	478	x2APIC	849	1.8	1085	2.3	1694	3.5
I/O APIC	209	MSIs	206	1.0	307	1.7	491	2.3
PS/2	846	password support	921	1.1	1311	1.6	1390	1.6
Sum	2494		3375	1.4	5076	2.0	5989	2.4

Figure 2.6: Size comparison in SLOC of selected device models in different VMM implementations. Only the models were chosen where Vancouver implements additional functionality.

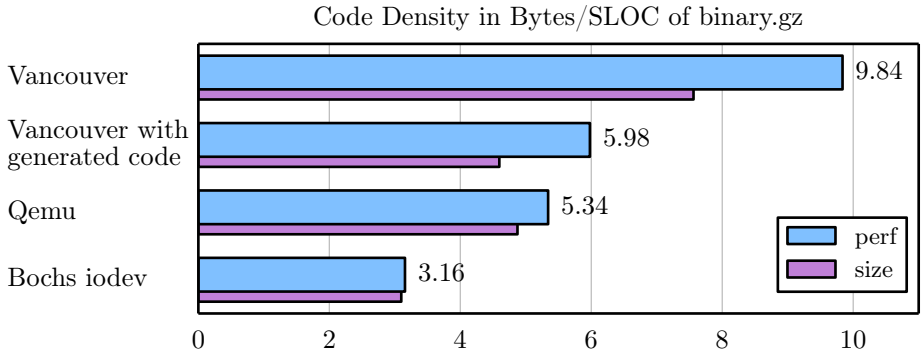

Figure 2.7: Code Density as Bytes per SLOC for gziped VMM binaries when the compiler optimizes for performance (perf) or code size (size).

twice, even though they are the very same chips in hardware. Furthermore, Bochs follows a verbose coding style. Bit fields for example are initialized one by one. Finally, Bochs contains a lot of debug code to warn on exceptional OS behavior. VirtualBox wraps the device models from Qemu and adapts them to its own interface. This approach increases the code by an additional 50%-75% and leads to the largest device models. The number of lines even doubles compared to Qemu, if debugging support is added, as in the Local APIC case.

Writing code more densely by pressing more statements in a single line, is one approach to reduce the number of SLOC in a program. The qualitative effect of such a programming style is disputable. On the one hand writing an algorithm on a single page might more clearly represent it, compared to spreading it over multiple pages. On the other hand this can lead if taken to the extreme to incomprehensible source code. To exclude coding style effects on the previous measurements, I additionally evaluated the code density by employing the approach described in Section A.1 to measure how much bytes a single SLOC adds to a compressed VMM binary. Figure 2.7 shows the results.

This measurement reveals that the device models in Bochs are written more verbosely. They need 50% more SLOC per gziped-byte than Qemu. Similarly, a Vancouver binary seems to be nearly twice as dense as the Qemu binary. However, when taking the code generated by scripts into account, this shrinks to barely 12% (5.98 vs. 5.34). Qemu is even denser, if the compiler optimizes for size instead of performance. In this case Vancouver needs 6% (4.6 vs. 4.87) more lines per byte than Qemu.

In summary, device models in Vancouver are on average 40% smaller than in Qemu due to a careful implementation and due to the externalization of checkpointing as well

as debugging code. Furthermore, the code density is approximately the same in both projects. Bochs and VirtualBox on the other hand need more code to implement the same functionality.

2.3.6 Future Work

Vancouver currently emulates more than twenty device models. This is enough to run most x86 operating systems inside VMs on NOVA. Adding more device models would support additional usage scenarios, for example:

- Emulating an USB controller including pass-through support allows to access the USB devices of the host.
- Sound emulation enables concurrent media playback from different VMs[25].
- Optical media support in the SATA emulation enables DVD and BD playback.
- GUIs, demos, and games greatly benefit from 3D graphics virtualization.
- Recursive virtualization and untrusted device models need an IOMMU.
- The TSC deadline mode and a HPET model would reduce the timer overhead.
- A TPM (Trusted Platform Module) model extends the trust chain into the VM.
- Floppy, DMA controller, and parallel port emulation are needed for very old operating systems.
- Various software stacks could be tested when implementing for instance WLAN, Bluetooth, Firewire, or Infiniband device models.

Furthermore one can improve existing code by:

- Validating more device models and closing virtualization holes.
- Adding power management including VCPU frequency scaling and hotplug support.
- Increasing the scalability on many core platforms with a better synchronization scheme [Par13].
- Reducing network, disk, and graphic virtualization overhead and thereby show that paravirtualization is unnecessary to achieve high performance.

Finally one should further research alternate approaches to device model creation like reverse engineering of existing code or code synthesis from formal specifications.

[25] Audio output from a single VM is already possible by directly assigning the sound card to it.

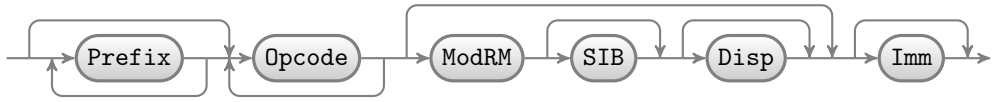

Figure 2.8: Layout of x86 instructions in 32-bit compatibility mode.

2.4 Instruction Emulator

A x86 VMM needs an instruction emulator to handle MMIO instructions targeting virtual devices. Moreover, the emulator is required to virtualize certain CPU transitions such as a hardware-task switch. Finally, it is needed to execute 16-bit realmode programs on most Intel processors.

Emulation of a x86 processor is a complex task, due to the huge number of instructions, the variable-length instruction format and many processor modes. Consequently, the instruction emulator ends up to be the largest and most complex component of the VMM. I therefore spent a significant effort on reducing its size.

I start this section with the technical background of x86 instruction emulation. I then introduce the first generation of the instruction emulator. Thereafter I detail two experiments using the developer documentation directly. Furthermore, I describe the current generation of the instruction emulator. I close the section with future work to further improve instruction emulation in a x86 VMM.

2.4.1 Background

The instruction emulator inside a x86 VMM can be simpler than a stand-alone implementation because many guest instructions can be directly emulated by executing the very same host instructions. Moreover, the emulator does not need to be very fast because it is only invoked rarely. Implementing a JIT (just-in-time) compilation system is therefore unnecessary. The traditional three phase approach to instruction emulation, which consists of instruction decoding, followed by execution, and a commit phase is sufficient.

Decoding x86 instructions is a laborious task due to the variable length instruction format. A single instruction is composed of six different parts as shown in Figure 2.8 and can be up to 15 bytes long[26]. Only the opcode part is mandatory for all instructions.

Various extensions to the instruction-set architecture have added new encodings to support additional instructions and to make more registers accessible. There are, for instance, 1-byte `REX` prefixes to use 16 GPRs in 64-bit mode and two- as well as three-byte `VEX` prefixes to implement the Advanced Vector Extension (AVX). Supporting these new instruction formats further complicates the decoding process.

Simple instructions can be executed natively. Only the memory operands need to be translated from the guest to the VMM address space. This requires a page-table walker to understand the virtual memory layout of the guest. MMIO operands cannot be handled this easily. Instead, they can be substituted with a pointer to a temporary *store buffer*. This buffer is flushed to the corresponding device model register during the commit phase.

Several dozen lines of high-level code may be required to emulate the side effects in complex cases such as exception delivery or hardware-task switching.

[26] Example: `lock addl $0x1234567,%fs:0x789abcde(%eax,%edx,4)` in a 16-bit code-segment.

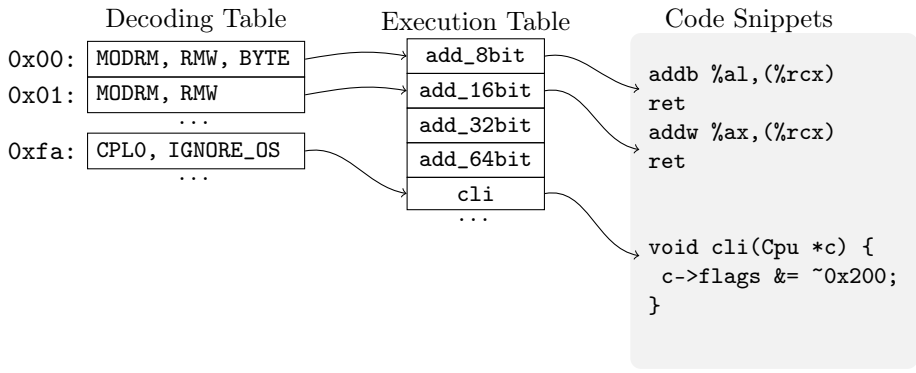

Figure 2.9: The first generation of the instruction emulator was based on a decoding and execution table as well as code snippets in assembly code and C++.

2.4.2 First Generation: Hand-written Tables

The first generation of the instruction emulator is based on two tables as shown in Figure 2.9. The handwritten decoding table holds the information for each byte in the instruction stream. A compact representation is used to minimize the lines of code. Each entry consists of a 32-bit value, which includes various flags to specify, for instance that another byte has to be fetched or that the encoding is invalid in 64-bit mode. An entry may contain an instruction number to lookup the function pointer in the execution table.

The execution table is combined by the preprocessor. It has one entry for any operand size of all instructions. An entry either points to a small assembler snippet that emulates simple arithmetic instructions such as `add` or it points to a function written in C++ to handle more complex instructions.

The first generation of the instruction emulator consists of more than 150 assembler snippets and around 100 helper functions. It emulates most arithmetic and operating system instructions but does not support floating point or vector instructions.

Please note that the table-based approach is very similar to the KVM instruction emulator introduced with Linux 2.6.37. However, the compact representation made it several hundred lines smaller than KVM's implementation. Nevertheless, the large number of table entries turned out to be hard to maintain in the long run. It was very easy to introduce a bug that could be only found after a long debugging session.

In summary, the first generation of the instruction emulator proved that a compact representation can significantly reduce the size of the instruction emulator. However, this did not increase the maintainability of the code. Instead, it revealed that collecting the instruction encodings manually does not scale to the enormous size of the x86 instruction set.

2.4.3 Automating the Knowledge Extraction

The first generation of the instruction emulator heavily relied on manual work. To add a new instruction to the decoding table, the developer would lookup the encoding in the published manuals and change the corresponding decoding table entries. Similarly, to emulate a new instruction, the developer would read the instruction documentation and the corresponding pseudocode before programming the helper functions.

Name	Description	Encoding
OR	Logical Inclusive OR register1 to register2	0000 100w 11 reg reg
OR	Logical Inclusive OR register2 to register1	0000 101w 11 reg reg
OR	Logical Inclusive OR memory to register	0000 101w mod reg r/m
OR	Logical Inclusive OR register to memory	0000 100w mod reg r/m
OR	Logical Inclusive OR immediate to register	1000 00sw 11 001 reg imm
OR	Logical Inclusive OR immediate to AL, AX, or EAX	0000 110w imm
OR	Logical Inclusive OR immediate to memory	1000 00sw mod 001 r/m imm

Figure 2.10: Encodings for the `OR` instruction from [SDM13] Table B.13.

A better approach is needed to reduce this manual labor and thereby the number of hand-written lines of code in the emulator. I therefore conducted two experiments to research whether certain information can be extracted from the documentation. The first experiment aimed at the generation of the decoding table. The second one tried to generate code snippets by translating the pseudocode.

Generating the Decoding Table

Intel's documentation for the x86 instruction set architecture [SDM13] includes a list of instruction encodings in several tables in Volume 2 Appendix B. See Figure 2.10 for an excerpt. The document is freely available in PDF format. It can be easily parsed after being converted to text. It should therefore be possible to write a small script that generates the decoding table from it.

The `pdftotext` tool from the *poppler* library can be used to convert the PDF file to plain text. The output of this tool without any parameters mimics the original page layout. Unfortunately this is hard to parse with a program because the tables do not follow a fixed format: newlines inserted in the wrong column break the table structure. Using `pdftotext -bbox` solves this issue by outputting not only the text from the PDF file but also the bounding box of each word. A simple Python script can split the output into pages, reconstructing the lines by grouping the words according to their y-coordinate and concatenating long entries spanning multiple columns. Altogether 150 lines of Python are enough to extract 3000 encodings for 900 instructions from the documentation.

However, it turned out that just knowing the encodings is not sufficient to reconstruct the decoding table. The flags, indicating for example the size of the operands or whether the instruction is valid in a certain processor mode, are required as well. Adding them manually would lead to nearly the same effort as before and would not improve the maintainability of the code. Additionally, a lot of syntax errors could be observed. This file seems to be maintained by hand without any automatic validation of its content. Fixing all of the errors turns out to be more costly than the time saved. In summary, the instruction tables in the documentation are too informal to directly derive the decoding table from them. Nevertheless, having a list of valid encodings is still useful for later testing the instruction emulator.

Translating the Pseudocode

Intel's [SDM13] and AMD's [APM13] documentation do not only describe the x86 instruction in plain English, they also come with pseudocode to further detail the semantics of certain instructions. It should therefore be possible to get code snippets for a large number of instructions with a small onetime effort by converting this pseudocode to C. Because both vendors use a different syntax, I implemented two different tools: one that

understands Intel's documentation and the other one to parse AMD's. See Figure 2.11 for example input and output of them.

The documentation from **Intel** contains twelve thousand lines of pseudocode for more than 450 instructions. The documents are freely available in PDF format, which can be easily parsed with a scripting language after a `pdftotext` conversion. In fact, by relying on regular-expression based string replacement, around 100 lines of Python are enough to extract the pseudocode from the text and convert it to C code[27]. The approach works well for simple instructions such as `cli` or `aad`, even if the pseudocode is sometimes suboptimal or in rare cases buggy. However, the tool fails in complex cases where not all information is expressed in pseudocode but only available in English words[28], or completely missing[29].

The pseudocode in the **AMD** documentation consists of approximately 1000 lines and covers 18 instructions. It can be parsed in the very same way as Intel's documentation. Fortunately it follows a more formal syntax. Thus only 60 lines of Python are needed to translate the pseudocode to C. The extraction works well for simple instructions, but the semantic gap in the complex cases is also present in AMD's documentation. Certain information is only available in English words[30] or completely missing[31].

Summary

Albeit the syntax problems can be fixed with some effort, the missing semantic information in the documentation makes any extraction method fruitless, except in the most simple cases. Putting more knowledge into the pseudocode would require tremendous work, which contrasts with the initial goal of reducing the developers effort. However, a complete formal and most importantly verified description of the instruction set, ideally provided by the manufacturer, would significantly improve the implementation of emulators, CPU testing tools and machine code verifiers.

2.4.4 Current Generation: Reuse the Assembler

While searching for a new approach for the current generation of the instruction emulator, I observed that the `GNU binutils`, which we use to assemble the NOVA binaries, already contain the encoding of x86 instructions in the assembler and disassemblers. They know the prefix bytes, after what opcodes a `ModRM` byte follows and what encodings include an immediate. Furthermore, these tools are used by a huge number of developers. The probability of defects is therefore much smaller compared to the hand-written decoding table of the previous generation of the instruction emulator. If this knowledge can be efficiently extracted, the encodings need not be specified manually anymore, which should result in a smaller emulator.

Knowledge Extraction

To recover the x86 instruction encodings from the assembler, I relied on a brute-force approach as depicted in Figure 2.12. Each combination of instruction name and operand string is assembled. The resulting ELF file is disassembled to retrieve the instruction bytes or a failure if an invalid combination was chosen. Finally, the `opcode` part of the

[27] The script could be simpler if many syntax errors would not complicate the extraction process. Intel's documentation seems not to be validated for syntactical correctness.

[28] `popf`: VIP is cleared, `ltr` locked operation to set busy flag, `pabsb` repeat operation for the next bytes

[29] `sti`: IRQ shadow, `wrmsr`: cpl0 check, `XCHG`: #LOCK assertion

[30] `pushf`: pushed with RF cleared, `int`: IF cleared if interrupt gate

[31] `sti`: IRQ shadow

```
IF 64-Bit Mode
  THEN
     #UD;
  ELSE
     tempAL <- AL;
     tempAH <- AH;
     AL <- (tempAL + (tempAH * imm8)) AND FFH;
     (* imm8 is set to 0AH for the AAD mnemonic.*)
     AH <- 0;
FI;
```

(a) Pseudocode for the `aad` instruction from [SDM13].

```
if (cs.l) {
    UD0;
} else {
    tempAL = AL;
    tempAH = AH;
    AL = (tempAL + (tempAH * imm8)) && 0xFF;
    /* imm8 is set to 0x0A for the AAD mnemonic.*/
    AH = 0;
};
```

(b) C code for the `aad` instruction converted from [SDM13].

```
IF (CPL <= IOPL)
  RFLAGS.IF = 1
ELSIF (((VIRTUAL_MODE) && (CR4.VME = 1))
    || ((PROTECTED_MODE) && (CR4.PVI = 1) && (CPL = 3)))
  {
    IF (RFLAGS.VIP = 1)
        EXCEPTION[#GP(0)]
    RFLAGS.VIF = 1
  }
ELSE
  EXCEPTION[#GP(0)]
```

(c) Pseudocode for the `sti` instruction from [APM13].

```
if (cpl() <= iopl())
  efl |= EFL_IF;
else if ((((cr0 & CR0_VM)) && (cr4 & CR4_VME))
  || (((cr0 & CR0_PE)) && (cr4 & CR4_PVI) && (cpl() == 3))) {
    if (efl & EFL_VIP)
        GP(0);
    efl |= EFL_VIF;
}
else
  GP(0);
```

(d) C code for the `sti` instruction converted from [APM13].

Figure 2.11: Converting the Pseudocode from Intel (above) and AMD documentation (below) to C Code works well for simple instructions.

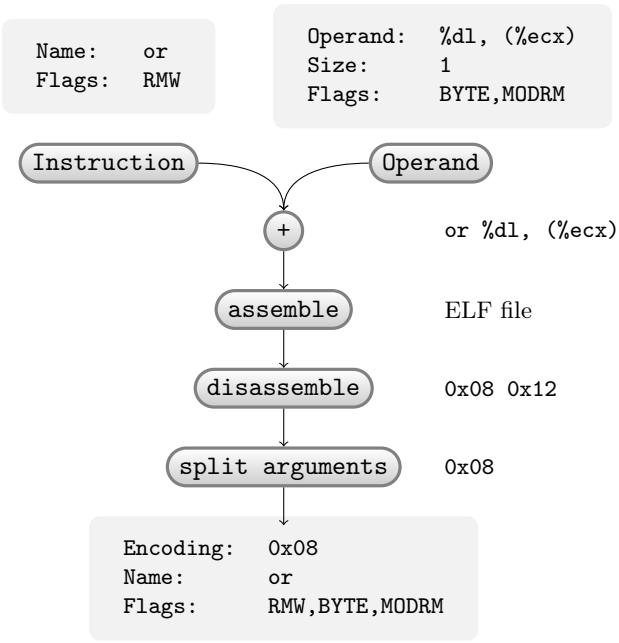

Figure 2.12: The encoding of an instruction can be automatically extracted from the assembler by trying to assemble and disassemble each combination of instruction name and operand.

instruction is isolated by stripping the number of bytes that are required to encode the chosen operand. This leads to the instruction name and a set of flags for every encoding. The flags indicate what parts of the instruction are present, where the operands can be found, and how the instruction behaves.

Please note that this approach is quite similar to [HEB01], where the authors used the assembler to learn the encoding of an unknown instruction set. However, I put more knowledge about the x86 instruction format in the extraction process, such that the search space will be smaller compared to their solver-based approach. While they directly analyzed the assembler output, I utilized the disassembler to split instructions. Finally, to speedup the process they use instruction batching and an heuristic to detect instruction boundaries. Instead, I cache intermediate results to achieve reasonable run times during development.

My approach reduces the effort to extract the x86 instruction encodings from the assembler by more than an order of magnitude to less than 200 SLOC of Python.

Code Generation

The script does not only extract the instruction decodings from the assembler. Another 220 SLOC of Python generate more than 8000 lines of C++ code for an instruction decoder and hundredths of handler functions. This code can be directly compiled with `g++` and linked to the emulator.

Instruction decoding is not based on tables as in the previous version of the instruction emulator. Instead the decoder uses nested switch statements as depicted in Figure 2.13. This code is automatically generated from the Python script and invoked for every byte of the instruction stream. It either returns an error or a pointer to an executor function.

```
switch (opcode_mode) {
 case 0x0:
  {
    switch (instruction_byte) {
  ...
      case 0x0f:
         opcode_mode = 0x1;
         break;
  ...
      case 0x8f:
         get_modrm ();
         switch (entry->data[entry->offset_opcode] & 0x38) {
           case 0x00 & 0x38:
               /* instruction 'pop' ['8f', '00'] ['OP1', 'MODRM', 'GRP'] */
               entry->flags = IC_MODRM;
               if (entry->operand_size == 1)
                   entry->execute = exec_8f00_pop_1;
               else
                   entry->execute = exec_8f00_pop_2;
               break;
```

Figure 2.13: The current generation of the instruction emulator uses nested switch statements for instruction decoding.

Furthermore, it forwards the flags such as `MODRM` or `RMW` to the caller, which are used for operand fetching. The decoder supports 375 different encodings of 175 instructions, which covers most of the integer and system instructions. However, it misses nearly all floating point and vector instructions because they are rarely used in an operating system kernel. Moreover, it does not detect the newest instruction formats such as AVX or the TSX prefixes. Finally, it will not detect alternate encodings such as a multi-byte `nop` or a 2-byte `inc`.

Whereas the generated code can be easier understood and debugged than the table-based one, it is also significantly slower because it causes more cache pressure and branch misses. Thus, the two implementation approaches for instruction decoding are instances of the typical tradeoff between performance and maintainability an OS developer faces.

The script also derives executor functions from small code snippets added to the instruction name list. These functions either emulate the side effects of the instruction via inline assembly or they call other C++ functions to achieve the same goal. Figure 2.14 shows two examples. As one can see, there is one function per operand size. Templates are used to avoid implementing helper functions twice. Furthermore, the `eflags` register, containing the arithmetic flags of a x86 instruction, need not to be saved in the assembler code. Instead, the caller has to make sure they are available, depending on the instruction flags returned by the instruction decoder. Overall more than 550 executor functions for 32, 16, and 8-bit operands are generated by the script.

Corner Cases

There are many corner cases in an instruction emulator where it is hard to implement the x86 semantics correctly. This is especially true when atomic operations are emulated. Exemplary, I detail how I solved two of these cases in the following.

Access and Dirty Bit Updates The instruction emulator contains a page-table walker that understands all x86 page-table formats. While filling the TLB, it also updates the

```
void exec_07_pop__es_2(Emulator *e) {
    unsigned sel;
    e->helper_POP<2>(&sel) || e->set_segment(&e->cpu->es, sel);
}

void exec_08_or_0() {
    asm volatile("movb (%edx), %al; lock orb %al,(%ecx)");
}
```

Figure 2.14: The instruction emulator calls automatically generated executor functions to emulate the side-effects of each instruction.

access and dirty bits in the page-table entries according to the x86 semantics. Using a plain `or` instruction to set these bits is dangerous because a second CPU might have disabled this entry concurrently by setting the present bit to zero. In this case, all other bits are reserved and must not be altered by the CPU or an emulator. If the emulator violates this condition, it will corrupt guest memory. Ideally, one could evaluate the page-table entry and write back the access as well as the dirty bits in one atomic operation. Unfortunately, this is not feasible without a recent CPU extension such as AMD ASF or Intel TSX [CCD+10, YHLR13].

Instead, one has to make sure that other CPUs cannot modify the page tables. Any ongoing DMA operation has to be stopped as well. Unmapping the page tables and mapping it into a CPU-local scratch area solves this problem. However, for complexity reasons I have not implemented this approach yet. Instead, I use the atomic instruction `lock cmpxchg` to check whether the page-table entry changed before modifying it. If an update failed, the operation is repeated by restarting the page-table walk. Please note that this loop may be unbounded and could consequently livelock the VMM.

Locked Instructions Whereas memory read and writes are itself atomic on a x86 CPU, most read-modify-write instructions, for instance `add` or `sub`, are not. Instead, they require a `lock` prefix to guarantee the atomicity of the memory updates. Locked instructions are important to synchronize memory updates between multiple processors.

On older CPUs, the atomicity was guaranteed by locking the front-side bus, such that no other CPU could change the memory while the instruction was executed. Nowadays the atomic update is implemented on top of the cache-coherency protocol. The relevant cache lines are pinned in the cache while the update is performed, thereby inhibiting any other CPU from interfering. Unfortunately, there is no x86 instruction that would allow an emulator to pin memory in the caches. Again AMD ASF and Intel TSX would help here.

One solution to the problem would be to copy in the data, execute the operation on the copy and then use an atomic `cmpxchg` to write back the results. If the `cmpxchg` instruction fails because another CPU has changed the value in the meantime, one would retry the whole instruction. Unfortunately, retrying this operation may lead to an unbound execution time, similar to the aforementioned page-table update. Malicious code running on one CPU can keep another CPU inside the VMM, which effectively livelocks the VM.

I have therefore chosen a more sophisticated solution. Because the VMM has mapped all guest physical memory, it is possible to directly execute the very same locked instruction on physical memory. In this way the semantics of the instruction is kept and the emulation time is bounded. Additionally, the VMM could map pages consecutively into virtual memory, if the destination operand crosses page boundaries and the two pages

are not already adjacent in guest physical memory. The mapping of pages is currently unimplemented because additional code is required for a case that never triggers when virtualizing all the major OSes. Instead, the implementation resorts to a non-atomic update, which is similar to the behavior of KVM in this case.

2.4.5 Summary and Future Work

The instruction emulator is the single largest and most complex component in the VMM. In this section, I explore several approaches to reduce its size. The first generation of the instruction emulator introduced a compact representation. It still required too much manual work. Better approaches had to be found.

Extracting the encodings and the pseudocode from the published documentation was unsuccessful due to the surprising informal nature of these files.

The current generation of the instruction emulator uses a semi-automatic approach, where the instruction encodings are automatically retrieved from the assembler. The executor functions are still based on compact code snippets. The instruction emulator remains the largest component of the VMM with more than 2000 lines of code. Adding missing features such as 64-bit or floating-point support will further increase its size. Nevertheless, by reusing a component already present in the TCB, the lines of code and thereby the development effort could be significantly reduced compared to the first generation. Please note that adding another dependency to a full-featured assembler is contrary to the goal of Chapter 4, which aims for a simpler toolchain.

In the following, I describe two areas that should be tackled in the future. The first suggest hardware improvements to speedup MMIO emulation. The second proposes a new approach to instruction emulator generation by extracting enough information from the physical CPU.

Hardware support to speedup MMIO emulation

On current hardware, any MMIO fault from a virtual machine has to go through the instruction emulator because the VMM does not have all required information available to directly perform the operation. This makes MMIO virtualization much slower than PIO emulation (See Section 2.6.4). Additional hints from the CPU would allow to handle the common MMIO instructions in a fast path.

The CPU could provide the instruction length, the source or destination register and the operand size. This, together with the already available instruction pointer, fault address, and error code would be enough to emulate all `mov` instructions, which are the canonical way to access MMIO regions, for instance, in Linux.

A more general solution would be to make the instruction bytes available to the VMM[32]. This would not only remove the effort to fetch them again, but it would also ensure that the reported fault address is valid as well. Thus, no page walk is necessary anymore for all instructions with only a single memory operand not crossing a page boundary. The CPU could also provide a second physical address to the VMM to support instructions that read from one memory location and write to another. This includes string `movs` and several stack operations.

Please note that these extensions would increase the speed of the common MMIO cases. They do not allow to remove the page walker completely because the VMM has to support special cases such as operands that cross page boundaries or page tables located in emulated device memory.

[32]Please note that AMD provides exactly this information starting with their Bulldozer CPUs.

Analyzing the CPU

The current generation of the instruction emulator has shown how the developer effort can be reduced. However the employed approach did not fully remove the dependency to the public documentation, but just outsourced its interpretation to the assembler writers. Note that using a formal description of the x86 instruction set such as [Deg12] does not solve this problem either because it is still derived from the informal documentation. The knowledge extraction from the published manuals remains a laborious and error-prone task.

To solve this problem, I propose to retrieve the required knowledge directly from the physical CPU. This refines an idea from Emufuzzer [MPRB09], which used the physical CPU as an oracle to decide whether an instruction emulator behaved correctly or not.

In the following, I will shortly describe several tests based on exceptions generated from a x86 CPU that reveal information required for instruction decoding and operand fetching:

Instruction Length The length of an instruction can be determined as follows: One places the first byte in the instruction stream just before a virtual memory hole. The code is then executed in single-step mode. If this byte is part of a longer instruction, an instruction-fetch pagefault will occur[33]. Any other exception means that a valid one-byte instruction was found. The test can be repeated with more bytes from the instruction stream to find larger instructions. This process is guaranteed to find an instruction boundary because x86 instructions are limited to 15 bytes.

Legacy and REX Prefixes An instruction-fetch pagefault happens if legacy or REX prefixes are repeated up to 14 times as redundant prefixes are ignored. Using the maximum of 15 prefixes in a row leads to an undefined opcode exception. This cannot be distinguished from any other unimplemented instruction.

Lock Prefix Whether an instruction supports a lock-prefix can be determined by executing the instruction with it.

System vs. User Instructions only valid in kernel-mode will fault if executed from user-level. Whether an instruction is invalid in a certain CPU mode such as 64-bit or realmode can be detected in the very same way.

Disabled Instructions Many floating point and some other instructions can be disabled via control or machine specific registers. Executing each instruction twice and toggling the corresponding bit in-between allows to distinguish such cases.

Memory Operands Pagefaults show whether an instruction accesses memory. Varying the GPRs and segment registers leads to the exact addressing mode. Executing the instruction with an address size or segment prefix should also result in a different behavior.

Operand Size Pointing the memory operand just before a hole in the virtual address space will reveal the operand size.

Fortunately many guest instructions can be directly executed without actually knowing their full semantics, as vx32 [FC08] has shown. It is often sufficient to just rewrite the memory operands. Direct execution is possible if the instruction does not harm the

[33] Alternatively one may use hardware breakpoints to trap the instruction fetch.

integrity of the emulator by returning the control to it and not causing unknown side effects. The few instructions that do not fall in this category such as `iret` or `sysenter` need to be special cased. Please note that any newly added as well as any back-door instruction will break this black-list approach.

In summary, this new implementation approach could lead to an emulator supporting all x86 instructions with an effort independent of the increasing number of arithmetic, floating point, and vector instructions. Only the general instruction format and the emulation of system instructions requires the informal documentation. Finally this approach enables the comparison of CPU behavior and the ability to adopt the emulator to the features supported by the host CPU it runs on.

2.5 Virtual BIOS

The BIOS is the first software running at boot time. Its main task is to initialize the platform and to start the bootloader. Moreover, it enumerates the system resources and initializes in-memory data structures such as ACPI tables to communicate the platform configuration to the OS. Finally, it includes device-driver functionality that allows to write bootloaders and early operating system code in a hardware-independent way. The BIOS drives for example keyboard and hard disk controllers. Option ROMs can be used to extend the BIOS. The VGA and VESA BIOS are such extensions found on graphics cards that add text as well as graphical output functions to the BIOS.

A VMM needs to provide BIOS functionality as well because bootloaders and operating systems that run inside the virtual machine will assume that a BIOS is readily available. They will therefore call through the standard BIOS entry points and search for BIOS data structures in memory.

BIOS functionality is typically implemented by injecting a BIOS binary into the virtual machine. The VM then executes the BIOS code in the very same way as it would happen on real hardware. Commercial vendors often licenced existing BIOS implementations and adapted them to their virtual environment. VMware for instance uses a BIOS from Phoenix and Virtual PC relies on an AMI BIOS. Open source projects opted for fresh implementations based on the publicly available documentation. Examples are Bochs BIOS and more recently SeaBIOS. However, injecting a traditional BIOS requires 25 to 300 thousand lines of code (§*A*.4.1). This is a significant increase of the TCB of any guest.

Injecting a BIOS not only adds to the TCB, it is also quite slow. The main reason is that the BIOS relies on its own device drivers to talk to the virtualized hardware. A single BIOS call, for example to read a block from disk, requires multiple traps to the VMM because the driver has to program multiple virtualized hardware registers. Furthermore, most BIOS code runs in the ancient 16-bit realmode. This processor mode cannot be directly executed in hardware, but needs to be emulated on all, except the latest, Intel CPUs. This further slows down injected BIOS code.

I therefore researched how BIOS functionality can be provided to a virtual machine in a faster way while adding fewer lines of code to the TCB.

2.5.1 Design

In the following, I will discuss three design alternatives. First, one could still inject the BIOS, but use a secure loader to remove it from the TCB. Second, one could paravirtualize the drivers to accelerate the I/O routines. Finally, one could virtualize the BIOS in the VMM. Figure 2.15 shows how a BIOS call will be handled in each of these cases.

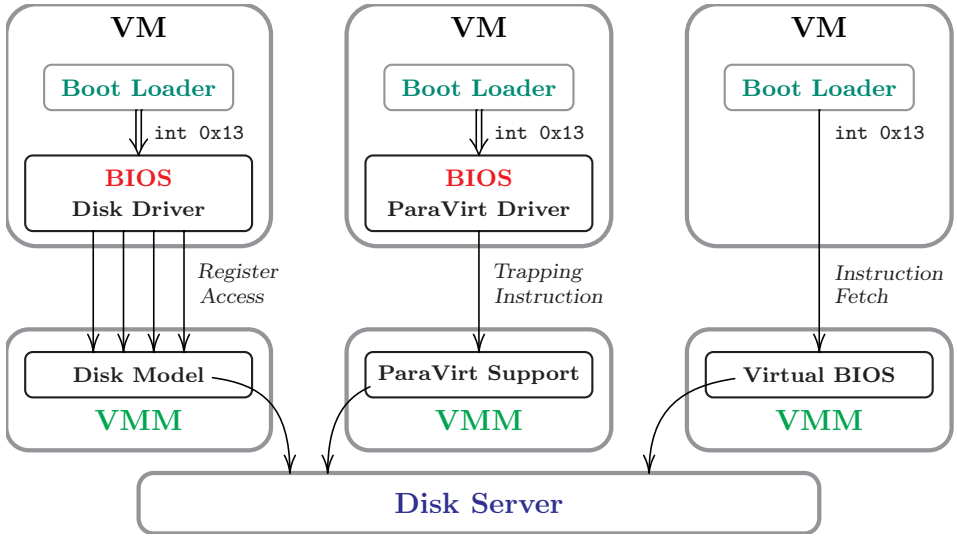

Figure 2.15: Handling a disk request from a bootloader in a traditional BIOS (left), in a paravirtualized BIOS (middle), and in a virtual BIOS implemented in the VMM (right).

Relying on a Secure Loader

A secure loader such as OSLO [Kau07b] can be used in a virtual machine in the very same way as on real hardware to remove the BIOS from the TCB (§5.3). However, the standardized interfaces for secure loaders, namely Intel's TXT (Trusted Execution Technology) or AMD's SVM (Secure Virtual Machine) require a TPM to store their secrets. Whereas emulating this device in the VMM is possible, it would require nearly 20 KSLOC [SS08].

One option to reduce this size would be to let the VMM emulate only a subset of the TPM functionality. Strictly required for a secure system are signed quotes of a single PCR (Platform Configuration Register), sealed memory, and monotonic counters. Other TPM functionality such as sophisticated key management, localities or multiple PCRs can be provided by the Trusted Software Stack instead. However, relying on a stripped down TPM requires software modifications inside each guest OS. Finally, it means that the missing functionality is not omitted from the TCB, but just moved into the VM.

Thus, using a secure loader will not make the TCB much smaller than relying on an injected BIOS in the first place. Moreover, it will not improve the performance of the BIOS at all.

Paravirtualizing the BIOS Drivers

Changing drivers inside the virtual machine to use a direct communication channel to the VMM, also known as paravirtualization, is an established technique to accelerate virtual device I/O [Rus08].

A paravirtualized driver in the BIOS would not program multiple hardware registers for a single request, but could rely on a special instruction[34] to communicate the request

[34]The `cpuid` instruction is usually chosen for this purpose because the instructions reserved for this task, namely `vmcall` and `vmmcall`, are not architectural defined but CPU vendor specific.

to the VMM. The VMM will then call the backend directly, without going through the device model emulation.

This approach will be faster then using unmodified drivers because it minimizes the number of VMM invocations. Furthermore, paravirtualization reduces the TCB by simplifying the drivers. However, it still requires to emulate the code before and after the trapping instruction on Intel CPUs.

Moving the BIOS into the VMM

Another approach would be to move the whole BIOS into the VMM. Instead of executing the instructions of the BIOS one by one, the VMM can already emulate the whole BIOS call on the first trap. Moreover, the VMM can already take control with the fetch of the first BIOS instruction, if the BIOS code is never mapped to the virtual machine. When handling this trap, the VMM can perform all the effects of the original BIOS call, for instance changing registers, accessing guest memory, and modifying device state. The virtual CPU can then directly return to the caller.

Executing the BIOS in the VMM is much faster than running a paravirtualized or unmodified BIOS inside the virtual machine because only a single guest instruction has to be emulated. Furthermore, not mapping the BIOS code makes sure that this approach works transparently on AMD and Intel CPUs. Finally, the BIOS is much smaller inside the VMM because it can directly access the device models as well as their backends and does not require device drivers as before.

2.5.2 Implementation

Due to the advantages of the third approach, I have virtualized the BIOS inside the VMM.

The `vbios` model thereby acts as special executor for the BIOS area. It handles single-stepping requests before the instruction emulator has a chance to do it. It checks that the CPU is in the right processor mode and that it faulted on one of the BIOS entry points, which are not mapped to the guest. If the checks are successful, the `vbios` model sends a message to the artificial BIOS bus.

Using a dedicated bus for this purpose allows to split the otherwise monolithic BIOS into multiple parts. Each part can be developed on its own. This also moves the BIOS code nearer to the device models. The VGA BIOS, for instance, is implemented directly in the VGA model where it can access the VGA registers without requiring a special interface for this purpose. This further simplifies the implementation.

BIOS Data Structures

A BIOS does not only provide entry points that can be called by an OS, it also initializes various data structures in memory. An example are ACPI tables that describe the platform configuration, such that bootloaders and operating systems can be written in a platform independent way.

A normal BIOS would execute code in the VM to enumerate the resources of the platform and discover how many CPUs and what devices are present. This knowledge is then used to fill the corresponding tables. Fortunately, a BIOS virtualized in the VMM does not need to run code inside the virtual machine to get the same information. Instead, all the required knowledge is already available in the VMM, even though it is distributed among the device models. Only the specific model knows how it is configured.

A straightforward implementation would collect this information and initialize the data structures in a central place. I have chosen another approach and generate the BIOS data structures in a distributed way. A read and write interface on named BIOS tables allows the device models to initialize the parts of the data structures they know of. This means the Local APICs can add itself to the `APIC` table and the VGA BIOS can easily register bitmap fonts in the realmode IDT (Interrupt Descriptor Table). By relying on a distributed implementation, the code and the interfaces to collect the platform configuration can be omitted. This makes the Virtual BIOS smaller than previous implementations.

2.5.3 Enabling Emulators

At a first glance, no BIOS code seems to be necessary in the VM in our design because the virtual BIOS can act directly on the trap caused by the first instruction fetch. However, this does not hold true anymore if the BIOS is emulated by the guest itself. If there is no BIOS code inside the virtual machine, the emulator cannot fetch it from the BIOS area. Even though the `vbios` observes corresponding memory reads, it cannot perform the effects of the BIOS call because it neither has access to the registers nor to the memory of the emulated machine. It just sees the CPU state of the emulator running inside the VM. The real register values are often deeply embedded in the emulator and cannot be easily accessed from the outside[35].

However, emulation is an established technique to run the BIOS outside its normal realmode environment. Especially the VESA part is often emulated by operating systems such as Linux, Fiasco, and even NOVA itself, to switch the screen resolution in a device independent manner. It is therefore necessary that the BIOS of a virtual machine can be emulated.

To make sure that an emulator can fetch valid instructions, I inject a very small BIOS stub into the virtual machine. This stub translates the original BIOS call to a paravirtualized BIOS interface that is based on a MMIO region. Using MMIO for this purpose makes sure that the emulator cannot substitute it with DRAM. The region therefore acts as shared memory between emulator and virtual BIOS. No additional trapping instructions are needed because all accesses to the MMIO region already trap to the VMM. The region can also be made private to the CPU, similar how any CPU sees its own Local APIC on the same physical address range. Thus, BIOS emulation can run concurrently on multiple CPUs.

Figure 2.16 shows the steps that are needed to emulate a single BIOS call. The BIOS stub copies the registers and all the needed memory from the emulated environment to the shared memory. It then traps to the VMM, which will perform all the effects on the copied-in values. Afterwards the stub copies the values back. The stub does not know what memory regions, besides the registers on the stack, are needed by the virtual BIOS in the first place because this heavily depends on the input parameters. The copy loop in the stub therefore traps to the VMM on every iteration, allowing the Virtual BIOS to request more data to be copied into shared memory.

The stub can be very small because it is unaware of the specific BIOS call. Moreover, the copy-in and copy-out parts differ only in the source and destination addresses. They can therefore use the same code. In fact, 30 assembly instructions are enough for a size-optimized BIOS stub. Additionally 60 lines of C++ code are required in the VMM to implement the MMIO region and to support the stub. Thus, a virtualized BIOS can be

[35] Virtual machine introspection could provide this functionality in some but not all cases [GR03].

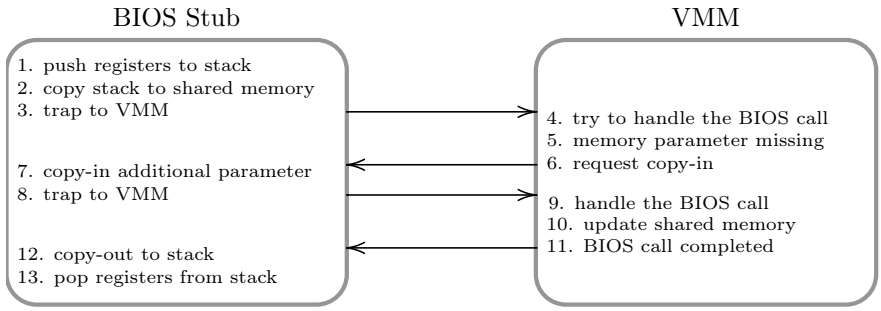

Figure 2.16: Control flow between BIOS stub and VMM while handling an emulated BIOS call.

emulated within less than one hundred SLOC.

2.5.4 Summary

The BIOS inside a virtual machine is a significant part of the Trusted Computing Base of a VM today. I did not follow the state-of-the art approach and injected a full BIOS into the virtual machine. Instead, I have virtualized the BIOS inside the VMM where it can be implemented more easily in a high-level language. The code inside the VMM is thereby reduced to 30 assembly instructions, which are only needed if the guest emulates BIOS instructions. The virtual BIOS has direct access to device models and their backends, such that device drivers in the BIOS are rarely needed. This makes the implementation not only smaller, but also faster.

The virtual BIOS implementation currently consists of less than 2 KSLOC. This number will probably increase in the future if missing features such as CD-ROM or floppy support will be implemented. Nevertheless, the virtual BIOS should remain an order of magnitude smaller than traditional BIOS implementations. In summary, the size and the complexity of the BIOS for a virtual machine can be significantly reduced by moving it into the VMM.

2.6 Performance Evaluation

Performance was an important but not the primary goal of the NOVA project. Nevertheless, a tiny but very slow system would be useless in practice. We therefore minimized the TCB of the system while still keeping an eye on the performance impact of our design decisions. In cases of conflict, we often preferred a smaller code base. For example, having no device models inside the hypervisor leads to less code. However, this also results in more overhead compared to KVM, which has timer and interrupt models inside the kernel. Furthermore, not having any paravirtualized I/O interface means that the I/O performance of a virtual machine might suffer. In some rare cases, we explicitly accepted additional lines of code to significantly reduce the virtualization overhead. Examples are the implementation of the virtual TLB inside the hypervisor and the optimization of the context-switch code between guest OS and VMM.

In this section, I present the performance of CPU as well as memory virtualization in NOVA and the performance of the virtual I/O path. I will not show any guest OS benchmarks such as `lmbench` or `SPECint` because their performance is largely independent of the virtualization stack if nested paging is employed. Furthermore, I will not

report any network performance numbers in this thesis because I have implemented only a model for an ancient ne2k card to get basic network access for a large number of legacy operating systems. However, the very simple interface of this card makes it several orders of magnitude slower than any modern NIC model. Instead, a high-speed network path will be implemented by Julian Stecklina.

This performance evaluation was performed in collaboration with Udo Steinberg. The results were partially presented in [SK10a]. In the following, I will give a more detailed analysis compared to the conference paper. Furthermore, I have reevaluated the raw numbers and corrected errors that have crept into the previous version.

2.6.1 Setting up the Benchmark

Measuring the time how long it takes to compile a Linux kernel is an easy way to test various parts of a system. This is also a popular macrobenchmark in the OS community because all ingredients are available as free software. Unfortunately, the benchmark was never formally defined and will therefore not produce an absolute number that can be used to directly compare different systems. Instead, the results can only be used to make relative comparisons between different software or hardware configurations.

In a nutshell, compilation is nothing more than number crunching within many processes. If the setup is done right, the I/O performance of the system will have only a very small impact. Instead, the results depend on the performance of system calls, process creation, demand paging and TLB handling. To evaluate the overhead of virtualizing CPU and TLB, we measured the time it takes to build a Linux kernel in 18 different environments.

Minimizing the Uncertainty

We developed an experimental setup that made the Linux compilation benchmark easily reproducible. It also ensured a minimal jitter of the results and a fair comparison of different configurations. In the following, I will describe the setup in detail because I have often observed that certain aspects of the benchmark setup were ignored, for instance in [PSLW09, LPL10, WWGJ12, WWJ13], which typically leads to non-reproducible results.

Task We compile the default i386 configuration of Linux 2.6.32[36]. A single benchmark run preprocesses, compiles, and links more than 1500 C files. It leads to a single compressed kernel image of approximately 4 MB after creating and destroying more than fifteen thousand Linux processes. Each benchmark run takes around 8 minutes. This is short enough to be repeated hundreds of times and long enough so that random events such as a spurious network packet or an additional rotation delay of the disk, have no measurable impact on the overall result.

Time Measurement Wall-clock time was measured by reading the TSC, the most accurate time source available on all machines. Because some virtualization layers are known for large clock drifts, we verified the results by also checking the real-time clock. Finally we asked an NTP (Network Time Protocol) server for a time stamp if networking was available inside the VM.

Hardware Most benchmarks were run on a pair of identical workstations[37]. Each was equipped with a single Intel Core i7 CPU clocked at 2.67 GHz, 3 GB DDR3 DRAM,

[36] The latest version at the time these measurements were done.

[37] We used two machines to reduce the benchmark time by running two configurations in parallel.

and a SATA hard disk from Hitachi (250 GB, 7200 RPM). Using the same machines for measurements ensured that hardware differences have no influence.

For workloads that run only on AMD CPUs, another machine equipped with a Phenom X3 8450 CPU clocked at 2.1 GHz, 4 GB DRAM, and a slightly slower 120 GB hard disk was used.

Single Core The Core i7 processor supports four cores with two hyperthreads each. Running the benchmark on all of these eight logical cores leads to a large jitter in the results. One reason might be the unpredictability of the mapping from Linux processes to cores. Another reason might be that the hyperthreads compete for shared resources. Disabling hyperthreading improved the situation significantly. However, the most consistent results were achieved by disabling all but one core in the BIOS.

With only a single physical core available to the host, it does not make much sense to give more then one virtual CPU to the guest. The overhead of inter-processor communication and memory synchronization would only negatively impact the measured performance. Finally, NOVA was not supporting VMs with more than one CPU at that time. We therefore run the benchmark only in single core configurations.

TurboBoost Disabling TurboBoost in the BIOS was necessary to get accurate and fair results of the benchmark for the following reason. Running the benchmark on a single, instead of two CPUs, reduced the number of CPU cycles by 10%. This high gain was especially surprising as kernel compilation is a workload with very little dependencies between two processes. Thus multi-processor scalability issues should not have such a large impact. Furthermore, this gain was not observed in some of the virtualized cases. It turned out that this reduction had to be attributed to Intel's TurboBoost technology, which allows one CPU to increase its core frequency if other CPUs of the same core are idle. This can be done as long as the produced heat fits into a processor specific thermal envelope. The Core i7 920 can increase its frequency in this way up to 2.93 GHz. This is exactly 10% more than the nominal frequency of 2.67 GHz and explains the observed CPU cycle reduction.

Guest OS To minimize the differences at the OS Level, the same guest OS binaries were used for all measurements. The same Linux Kernel (2.6.32) and Slackware installation was chosen for this purpose.

Device Models Linux has drivers for a number of devices of the same class. On a native x86 machine it can use for instance the Local APIC of the CPU, the PIT, the PM-Timer (Power Management Timer), or the HPET as timeout source. Unfortunately, many virtualization stacks support only a subset of these different devices. As the virtualization overhead depends on the device interface, we forced Linux to use the devices available in all environments. This means Linux got its timer interrupts from the i8254 PIT [PIT94] and acknowledged them at the i8259 PIC [PIC88].

Note that there was no common disk controller available everywhere. Thus, configurations using Qemu as their VMM had to be measured with a virtual IDE controller, while others were using an AHCI model[38].

[38]Qemu got AHCI support with version 0.14, which was announced only a couple of months later.

Physical Disk All configurations would use the physical disk controller in AHCI mode. Furthermore, NCQ had to be disabled on the Linux command line. Otherwise NOVA, which did not virtualize this feature, was actually running the benchmark faster than a non-virtualized system. One reason for this unexpected result seems to be that Linux delays disk requests longer if NCQ is enabled to be able to merge requests before they are submitted to the disk.

Fixed Timer Frequency Linux does not use a fixed periodic timer per default anymore, but dynamically adapts its frequency depending on the load of the system. This means the frequency of timer interrupts and therefore the overhead to virtualize them, will vary through the runtime of the benchmark. To exclude this variability and measure approximately the same number of timer interrupts in repeated runs, we forced a fixed timer frequency by adding the `nohz=off` parameter to the Linux command line.

Parallel Make The make should run in parallel to ensure that enough active jobs exist so that the CPU will not be idle if one job waits on the completion of disk I/O. This minimizes the influence of any disk delay. Using four parallel jobs led to the best results.

CPU idle times could not be completely avoided with this simple approach because kernel compilation cannot be fully executed in parallel. There are very few runnable processes at the beginning of the benchmark where dependencies are checked, and at the end where the kernel is linked and compressed.

Finally, having another process ready at the end of a thread's time slice, ensures an address space switch will happen. The kernel can otherwise avoid this costly switch by directly scheduling the previous thread again. Thus, a parallel build will simulate a loaded system more accurately.

Buffer Cache The Linux caches have a limited, but measurable influence on the duration of the benchmark. We therefore dropped the *buffer*, the *dentries*, and the *inode* caches before starting the benchmark. To avoid reading blocks twice from disk, we gave all guests 512 MB of DRAM. This should result in the same strategy for buffer cache prefetching during all measurements.

The virtualization layer was given all the memory of the host platform. Note that these gigabytes of memory could have been used for a second buffer cache in the host. A very intelligent virtualization layer could even trace and predict the access pattern of the benchmark. It could then fetch the disk blocks before our benchmark even runs. For NOVA, KVM, Xen, and the paravirtualized systems we could make sure that double buffering was not happening, either by design or by configuring the system accordingly. For the measured closed source systems we could not assure this. However, our results do not indicate that they gained much from it, if they have employed this technique.

2.6.2 Results

We compiled the Linux kernel on 18 different system configurations for more than 1500 times. This was only feasible by automating the measurements as much as possible. A single benchmark run booted the machine, compiled the kernel, uploaded the results to a server, and rebooted the system to start the next run. The results of these measurements are shown in Figure 2.17 as the median of the runs.

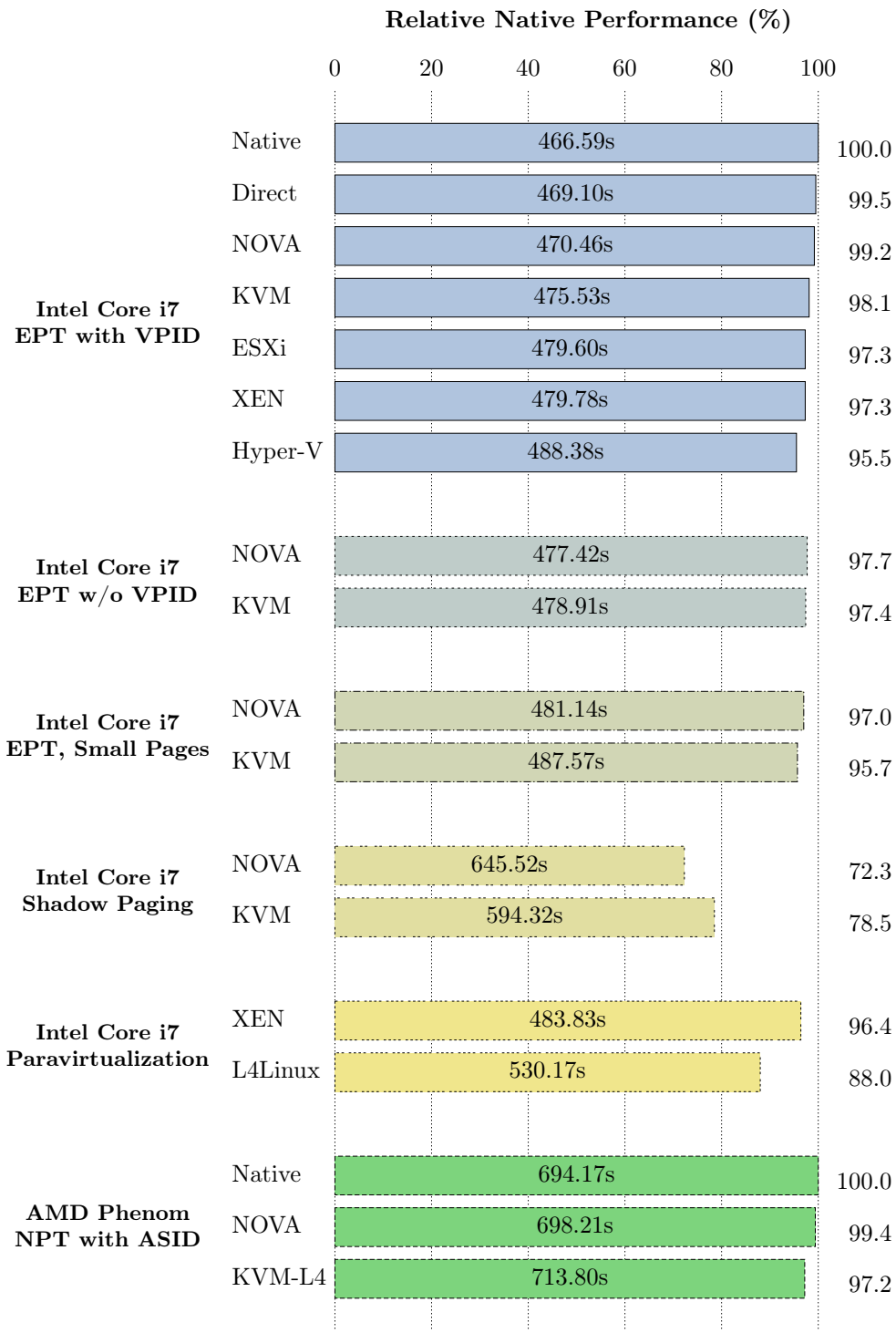

Figure 2.17: Performance of Linux kernel compilation within hardware-assisted virtualization and paravirtualized environments.

The two *Native* bars represents the time needed to run the benchmark on Core i7 and AMD Phenom machines respectively without any virtualization involved. They are the baseline for all other measurements. We normalized the results of the virtualized systems with respect to them. Thus, higher percentages in the Figure indicates better performance with less overhead.

To quantify the overhead caused by nested paging (EPT) alone, we run a virtual machine with all intercepts disabled[39]. Furthermore, the VM got direct access to all devices and received all interrupts without invoking the hypervisor. The second bar labeled *Direct* shows that even without virtualizing any devices, the benchmark needs 2.51 seconds more than on native hardware. This overhead can be attributed to the longer page walk that is needed to fill the TLB, which requires more memory accesses. Furthermore, additional TLB entries are needed to cache intermediate results. Note that this overhead depends not on the employed software, but only on the particular hardware. Every virtualization stack that relies on nested paging has to pay this overhead and cannot achieve more than 99.5% of the native performance on this particular machine.

NOVA

The third bar shows that compiling a Linux Kernel inside a virtual machine on NOVA v0.1 takes only 0.8% or 3.9 seconds longer than on bare hardware. I will later quantify how much we benefit from nested paging (EPT), large pages, and a TLB tagged with a virtual processor identifier (VPID). Furthermore, I will perform a detailed breakdown of this overhead in Section 2.6.4.

KVM, ESXi, Xen, and Hyper-V

The next four bars give the results for KVM 2.6.32, VMware ESXi 4.0 U1, Xen 3.4.2, and Microsoft Hyper-V Build 7100. KVM performed best in this group. ESXi and Xen are on the very same level whereas Hyper-V needs significantly longer. KVM surely benefits from the `hugetlbfs` Linux feature because this provides large pages for the guest memory and thereby reduces the TLB overhead. The bad results of Hyper-V are unclear. One reason could be that Hyper-V has a longer context switch path than NOVA and does not implement neither the interrupt nor the timer model inside the hypervisor as KVM and Xen do. Another reason might be that we have not found a way to disable NCQ in the parent partition. Finally, Hyper-V may not be using all of the optimizations to speed up virtualization such as tagged TLBs or large pages.

Tagged TLB

The second set of bars in Figure 2.17 shows the benefit of tagged TLBs. The benchmark is run with VPIDs disabled. The hardware TLB is therefore flushed on any VM resume and has to be refilled later. This costs NOVA 1.3% or more than 6 seconds. KVM is still slower than NOVA without VPIDs, but looses only half of that number. In other words KVM benefits less from tagged TLBs. KVM probably touches more memory pages than NOVA to handle a single virtualization event and therefore needs a larger portion of the TLB for itself. Consequently more entries of the guest will be aged out of the TLB before the VM is resumed.

[39]The *cpuid* exit cannot be disabled on Intel VT, but handling it during the benchmark took less than a millisecond.

Small vs. Large Pages

The third set of bars illustrates the influence of the page size to the benchmark results. By mapping the guest memory with small 4k instead of large 2M pages, up to 512 times more TLB entries are needed to access the same amount of memory. Kernel compilation on NOVA takes 10 seconds longer with small instead of large pages. This means its virtualization overhead is tripled. NOVA is now slightly slower than Xen or ESXi. Similarly KVM needs twelve seconds longer and more than doubled its overhead. The performance is now similar to Hyper-V. Thus, using small pages will lead to a significant performance drop in a TLB-hungry benchmark. These results are remarkable because certain techniques such as page sharing between virtual machines [GLV+10], VM migration [CFH+05], and VM forking [LCWS+09] assume small pages. Because Xen, ESXi, and Hyper-V support at least migration, they might not benefit from large pages.

Shadow Paging

The next two set of bars compare the performance with older systems, which do not support nested paging. We simulate a system without nested paging support by configuring NOVA and KVM to use shadow paging. This means the CPU will not flatten the nested page tables in hardware anymore. Instead, the hypervisor has to combine the guest and the host page tables into a single shadow page table on its own. Note that a parallel kernel build will be the worst case for such a software TLB because of many short living processes, frequent address space switches, and the huge number of page faults caused by memory-mapped files. In fact, kernel compilation on NOVA takes up to 3 minutes longer than on native hardware. KVM takes only two minutes longer, probably due to a smarter shadow paging implementation.

Paravirtualization

We also compare the performance of hardware-assisted virtualization with older paravirtualization approaches. For this, we run the benchmark on two paravirtualized systems, namely a 2.6.18 kernel running inside Xen's dom0 and an L4Linux 2.6.32 running on top of L4/Fiasco. Both had direct access to the hard disk as there is no efficient way to virtualize storage in these systems. For Xen, we also relied on a patched `libc`, which avoids the expensive segment reloads during every thread switch by not using segments for Thread Local Storage (TLS). Thanks to this optimization, Xen's paravirtualized domain can achieve 96.5% of the native performance. This is similar to the results from [BDF+03] and only 4 seconds slower than a faithfully virtualized domain on Xen. Without the changed `libc`, a paravirtualized Xen was as slow as a NOVA VM with shadow paging. L4Linux on the other hand needs one minute more than a native system to compile a Linux kernel. This corresponds to 88% or more than twice what was reported previously as paravirtualization overhead [HHL+97]. However, the current L4Linux implementation does not use the small address-space optimization anymore. This means additional address space switches have to be paid for every system call and every page fault. Nevertheless, both paravirtualized systems could beat hardware-assisted virtualization with shadow paging. These results are aligned with the observation that early hardware-assisted virtualization was slower than software techniques [AA06]. However, our measurements indicate that this does not hold true anymore on hardware with nested paging support.

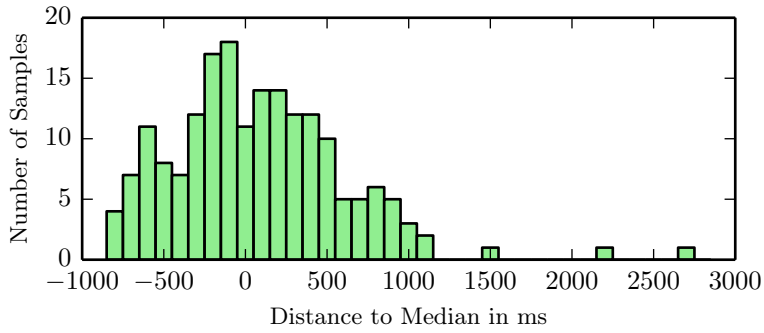

Figure 2.18: A histogram of the measured runtime relative to the median.

AMD and KVM-L4

To compare our system with KVM-L4, a port of KVM to the L4/Fiasco microkernel [PSLW09], we had to run the benchmark on an AMD machine because KVM-L4 was not supporting Intel CPUs at that time. The last set of bars in Figure 2.17 shows the results. Because it is not possible to compare the absolute times between the two different platforms, we also performed *Native* runs on this platform and normalized the results to these runs. For NOVA we measured 99.4% of the native performance. This is slightly better than what we measured on the Intel systems. This advantage is likely caused by the different nested page-table formats. AMD has reused the legacy x86 format, which has two levels whereas Intel defined the new EPT format, which requires four levels to describe the physical address space of the guest[40].

For KVM-L4 we measured 97.2%, which is similar to Xen but less than what we measured for KVM. This difference can be attributed to the additional layer of indirection introduced by KVM-L4. To resume a VM the KVM code running inside an L4Linux process will prepare the VM state. Instead of using this state directly, it has to be transferred to the kernel and validated by it. The additional kernel entry and the validation costs approximately 1% performance in a CPU and memory bound benchmark like kernel compilation.

2.6.3 Measurement Error

We spent considerable time to set up the benchmark in a way that the measurements are as accurate as possible. In the following, I will try to quantify the remaining measurement error.

Figure 2.18 shows a histogram of multiple benchmark runs for the *Direct* configuration. Most of the results are clustered around the median. However, there are a couple of outliers where the benchmark took significantly longer. The standard deviation of this distribution is approximately 500 milliseconds. The left side of Figure 2.19 shows that other configurations have even larger standard deviations. This means running the benchmark only once can lead to a measurement error of several seconds. Note that the *NOVA* numbers show that hardware virtualization itself does not necessarily increase this jitter.

We therefore executed all benchmarks numerous times and used the median to get a

[40] Large pages allow to skip the last level, which further increases AMDs advantage.

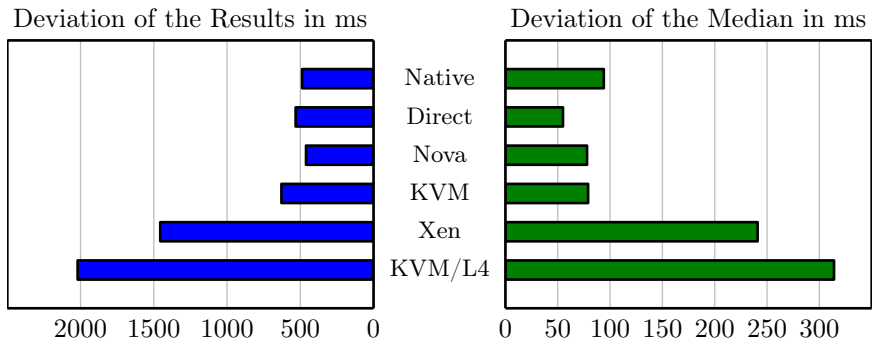

Figure 2.19: The jitter and the accuracy of the measurements expressed as the standard deviation of the results and the standard deviation of the median.

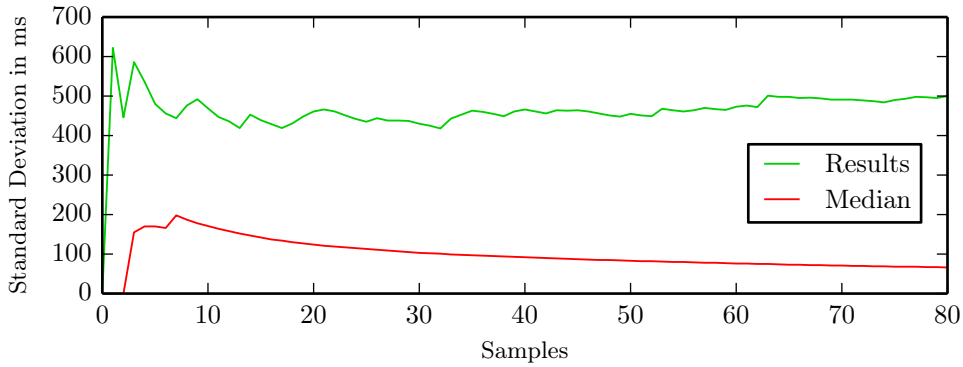

Figure 2.20: The jitter versus accuracy through a set of samples.

single result. We did not rely on the average because this would give outliers a larger impact. The accuracy of this approach can be measured as the standard deviation of the median over many runs. The right side of Figure 2.19 shows that the median of several measurements is at least five times more accurate than performing a single run only.

Figure 2.20 reveals that the standard deviation of a single run remains at its high level whereas the deviation of the median approaches zero if more samples are taken[41]. Thus the results will get more and more accurate over time. This observation can be used to calculate the number of measurements that have to be taken for a certain configuration to achieve a given accuracy.

The measured results should follow an inverse normal distribution because the system can be modeled as a set of jobs that have a lower bound on their runtime, but which are randomly delayed with a certain variance. One reason for such delays is the arrival time of hardware interrupts. A different arrival, which depends for instance on the exact disk head position, will lead to another process scheduling and therefore a different utilization of the CPU resources, which is less efficient than the optimum.

To simplify the calculations, a normal distribution is used as a first approximation. This would mean more than 95% of the measurements are contained within two times of

[41] Single runs should be completely independent from each other. More samples means outliers have a smaller effect on the median.

Event	EPT	vTLB
vTLB Fill		181,966,391
Guest Page Fault		13,987,802
CR Read/Write		3,000,321
vTLB Flush		2,328,044
PIO	540,680	723,274
INVLPG		537,270
Hardware Interrupts	174,558	239,142
MMIO	76,285	75,151
HLT	3,738	4,027
Interrupt Window	2,171	3,371
$\sum$	797,352	202,864,793
Injected Interrupts	*131,982*	*177,693*
Disk Operations	*12,715*	*12,526*
Runtime (seconds)	*471*	*645*
Events per second	*1,693*	*314,519*

Figure 2.21: Distribution of Virtualization Events during Linux Kernel Compilation

the standard deviation. Thus the results of *Native*, *Direct*, *NOVA*, and *KVM* are accurate within plus/minus two hundred milliseconds. This corresponds to 0.08% of the relative performance measurements.

2.6.4 Detailing the Overhead

While most measured systems can only be taken as a black box, our intimate knowledge of NOVA allows us to further breakdown its overheads. Figure 2.17 reported that Linux kernel compilation on NOVA is 0.8% or 3.87 seconds slower than on a native system. In the following, I will show where this time is spent.

Counting Events

To get an overview of the system behavior, we counted the number of hypervisor invocations and related system internal events such as the number of served disk requests during a benchmark run. Figure 2.21 shows the distribution of these events in NOVA with nested (*EPT*) and shadow paging (*vTLB*).

One can see that a shadow-paging setup causes two orders of magnitude more virtualization events than an EPT-based one. On average, there are 315 thousand events per second or one event every 8500 cycles. Given that any event needs at least 1000 cycles to transition between VM and hypervisor on the Core i7 920 CPU [SK10a] and that the hardware TLB is flushed on all of these transitions, the majority of the three minute overhead for the shadow paging configuration is already lost in hardware.

Note that the number of events could be further reduced with a more sophisticated virtual TLB implementation. However, the number of guest page faults and the number of control-register accesses will be the same, even with an implementation that can perfectly prefill the TLB. Furthermore, each guest page fault will lead to an additional virtual TLB fill event to synchronize the page table entry that was just created by the guest OS. These three types of events sum up to more than 35 times the number of events in the nested paging case. Thus, any TLB-heavy benchmark such as a parallel kernel compilation will tremendously benefit from nested paging hardware. I therefore concentrate the further analysis on this case only.

Name	Hardware Timer	Emulated Timer	Acknowledgment
PIC	11214	22198	4x PIO exits
APIC	7826	18740	1x MMIO exit
x2APIC	4216	13815	1x `wrmsr` exit
Auto EOI	2883	13629	delivery only

Figure 2.22: Cycles needed to generate a timer interrupt periodically, deliver it to a VM, and to acknowledge it. These numbers were measured inside a NOVA VM running on the Core i7 CPU.

Running the benchmark with EPT leads to approximately 1700 events per second. The guest system timer that periodically ticks with 250 Hz causes already 1250 of them[42]. The remaining events are the MMIO exits to access the disk and the host timer interrupts that are used by the hypervisor to schedule its threads.

There are also some `HLT` exits, which indicates that the virtual CPU was idle at certain times. The benchmark sometimes waits for I/O, even if we tried hard to keep the CPU busy by executing the make in parallel. Idle times occur especially at the beginning of the benchmark where the buffer cache is still empty. The measured 3738 `HLT` exits correspond to less than 15 seconds because any idle phase ends whenever the system timer fires or the outstanding disk job completes. On average, the guest CPU will be idle only half of that time because the `HLT` events should happen randomly within a timer period. In fact, a native Linux reported that the CPU was idle for 7.2 seconds during the benchmark. The very same time was measured within a NOVA VM. Thus `HLT` exits should have not influenced the NOVA overhead.

The high number of virtual TLB related events indicates that the benchmark will also put a large pressure on the nested paging hardware. The *Direct* measurements have already shown that nested paging is responsible for 2.51 seconds of the overhead. Thus only 1360 milliseconds or 35% of the NOVA overhead can be directly attributed to our implementation.

IRQ Overhead

Figure 2.21 showed that interrupt handling caused most of the events in a nested paging configuration. Interrupts are not only responsible for the direct effects such as exiting the virtual machine and injecting the interrupt but also for secondary events like emulation of the PIO to acknowledge and mask the IRQ at the virtual interrupt controller.

To more accurately estimate the influence of interrupt handling to the overall performance, I wrote a small application that can measure the overhead in isolation from memory and TLB effects. The application performs number crunching by simply incrementing a register in a tight loop. It does not touch any memory inside this loop. The loop is first executed without and then with interrupts enabled. The delta between these two runs can be attributed to the interrupt path. Interrupts are generated periodically by either using a hardware HPET that is directly assigned to the virtual machine or by using an emulated timer such as the PIT and the Local APIC. The number of loop iterations is calibrated beforehand so that a single run takes at least one second. This is enough for a couple of thousand interrupts to occur.

Figure 2.22 shows the results of this microbenchmark. By comparing the left column with the right, it is clear that between 9,000 and 11,000 cycles are needed to implement

[42]One event to inject the interrupt and four PIO events to program the interrupt controller.

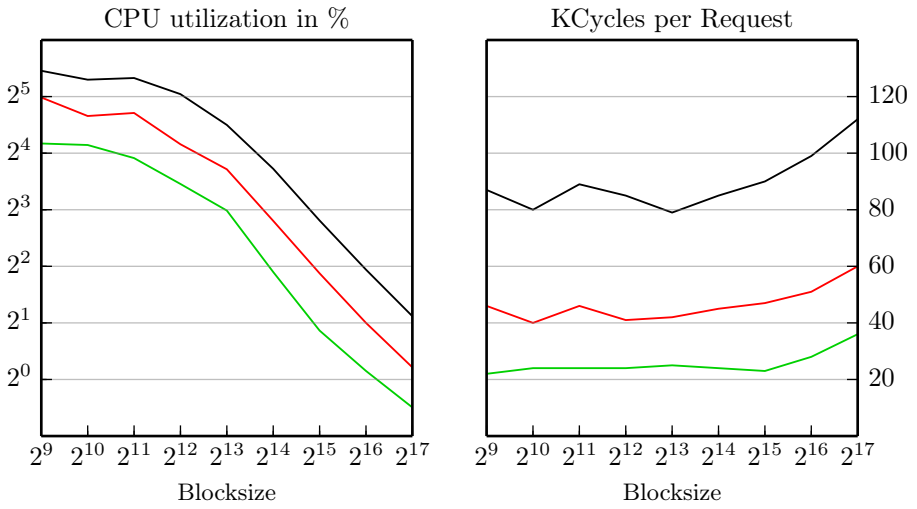

Figure 2.23: CPU utilization and corresponding cycles per disk request for a native (bottom), direct assigned (middle) and fully emulated (top) SATA disk on the Core i7 machine.

a virtual timer. This includes the time for calling the timeout service, programming the host HPET, forwarding the timeout to the VMM, and updating the device model. This path could be optimized by implementing timeouts directly in the hypervisor.

The left column shows that the overhead heavily depends on the employed interrupt-acknowledgement method. Using the PIC leads to the highest number of exits and the largest overhead. Relying on the Local APIC is a bit faster, even though its single MMIO exit is more heavyweight than a single PIO exit (5k vs. 2.1k cycles). The x2APIC is even faster than the Local APIC because its `wrmsr` interface does not need an instruction emulator. However, the fastest method measured was the virtual PIC in Auto EOI (End of Interrupt) mode. In this mode, no acknowledgement is required at all. Unfortunately, the used interrupt-acknowledgement method is chosen by the guest OS independently from the VMM.

In the kernel compilation benchmark, we measured the worst case by using the virtual PIT to generate 250 timer interrupts per second and deliver them through the i8254 PIC. By multiplying the runtime of the benchmark with the timer frequency and the overhead per timer interrupt and dividing this through the processor frequency, one can see that approximately 980 milliseconds can be attributed to the virtual timer handling.

Disk Performance

The MMIO exits are the third largest group of virtualization events in Figure 2.21. They are caused by the Linux disk driver, which issues disk requests through the emulated AHCI controller. To quantify the impact of the disk emulation speed to the overhead observed in the kernel compilation benchmark, we evaluated how the disk performs independently of the kernel compilation benchmark. We read from the 250 GB Hitachi hard disk sequentially and varied the block sizes. We relied on direct I/O to exclude the influence of the buffer cache.

Figure 2.23 shows the CPU utilization (left) and the cycles needed per disk request (right). The left part is reprinted for comparison from [SK10a]. The bottom curve shows

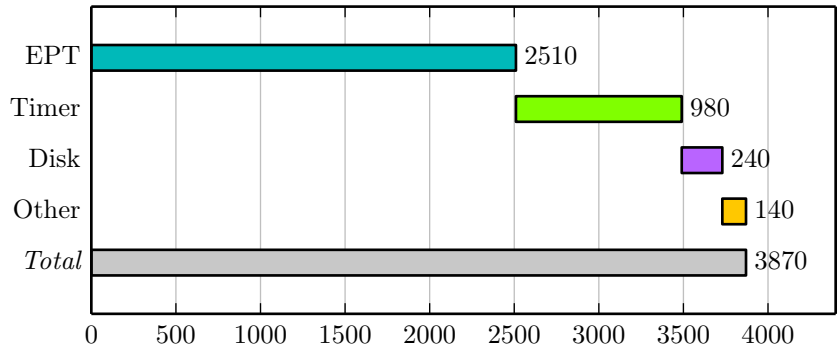

Figure 2.24: Breakdown of the NOVA overhead during kernel compilation in milliseconds.

the native hardware without any virtualization involved. The middle curve represents the direct assignment of the disk controller to the VM. Its overheads stems from the interrupt path and the cost of remapping DMA requests with the IOMMU. The fully emulated SATA disk (top) additionally includes the register emulation of the disk controller and the forwarding of the guest requests to the host disk driver. One can see a large performance gap between native and virtual I/O. The directly assigned disk already doubles the CPU utilization. The emulated SATA controller increases the CPU utilization by a factor of four compared to the native case.

The middle curve on the right shows that the cycles per request scales with the block size. This is expected as Linux usually programs one DMA descriptor per 4k page. This effect is further amplified in the emulated disk because an increasing number of DMA descriptors means not only more work of the disk driver, but also additional emulation overhead for fetching and translating these descriptors. The directly assigned disk reveals that approximately 20k cycles are needed for receiving, injecting, and acknowledging a disk interrupt. Compared to Figure 2.22, this seems to be a surprisingly high number. However, we have an idle system here where additional `HLT` exits will cause more overhead. The fully emulated numbers show that between 40k and 50k cycles have to be paid for every disk request above the interrupt handling. Approximately 30k of them are used to handle the six MMIO exits. The remaining cycles are needed to contact the disk driver and perform the physical disk access.

The 50k cycles for disk emulation and the 20k cycles for the interrupt path are an upper bound of the disk emulation overhead during the kernel compilation benchmark. The actual numbers are surely smaller because disk requests are not always issued with the maximum block size and the system is more often busy than idle. A lower bound would be to use the minimal measured disk emulation overhead (40k cycles) and the value from Figure 2.22 to handle a single hardware interrupt (11k cycles). These lower and upper bounds correspond to 240 and 340 milliseconds respectively.

Summary

Our intimate knowledge of the NOVA system allowed us to break down the overhead during the Linux kernel compilation benchmark into smaller parts as shown in Figure 2.24. We started with a total of 3.87 seconds more to compile a Linux kernel inside a NOVA VM. Of this total time 2.51 seconds can be attributed to the nested paging hardware.

The periodic guest timer accounts for 980 ms. The emulated disk costs at least 240 ms. This leaves less than 140 ms or 0.03% of the relative performance for the host scheduling timer and for secondary effects such as cache pollution and increased TLB pressure from the virtualization layer. I will not further analyze these overheads here because they are below the measurement accuracy of 160 ms.

2.6.5 Performance Outlook

The kernel compilation benchmark has shown that NOVA is virtualizing CPU and memory slightly faster than state-of-the-art virtualization stacks. Furthermore, we observed a significant performance gap between native and virtualized I/O. The superior CPU performance is especially surprising as we have to pay for our split architecture and a smaller codebase. Our performance optimizations seem to be enough to compensate these overheads.

However, the reported results provide only a snapshot of the performance from a certain point in time. It is unknown whether the ongoing development of the NOVA system will improve these numbers or not. I will therefore analyze the performance impact of new and upcoming features in this subsection.

There is a group of features that make the system slower because they increase either the CPU state that needs to be transferred between hypervisor and VMM or the number of virtualization events. For instance, virtualizing performance counters or debug registers, will result in additional state that has to be context switched by the hypervisor and provided to the VMM. Features such as an accurate TSC or virtual IOMMUs will lead to more events that need to be handled. Similarly, the introduction of the parent protocol has slowed down the backend connection as it requires an additional capability translation for any request.

There are some features that will be neutral with respect to the performance. Porting the system to 64-bit is such an example. A 64-bit VMM can benefit from more registers and a larger address space, but it will suffer from slower TLB fills and increased instruction bytes that have to fit in the caches. Furthermore, supporting 64-bit guests requires more memory accesses to parse the page tables during instruction fetch and more branches in the instruction emulator to distinguish the new operand size. Positively, it reduces the number of MMIO events because guests can use the faster `wrmsr` interface of the x2APIC for interrupt and timer handling.

Finally, there are many new features that would improve the performance of the system:

- An optimized instruction emulator that supports only `mov`-instructions and falls back to the current emulator for uncommon cases would improve the MMIO path.

- The ability to transfer only minimal state on an EPT fault would also accelerate MMIO emulation, even though this requires additional checks to detect when non-available state is accessed.

- Recalling the VCPU directly from the services would remove the helper threads and further shorten the I/O path.

- CPU-local timeouts provided by the hypervisor would minimize the overhead of timeout programming. They could be implemented with the *VMX preemption timer* or directly with the host scheduling timer.

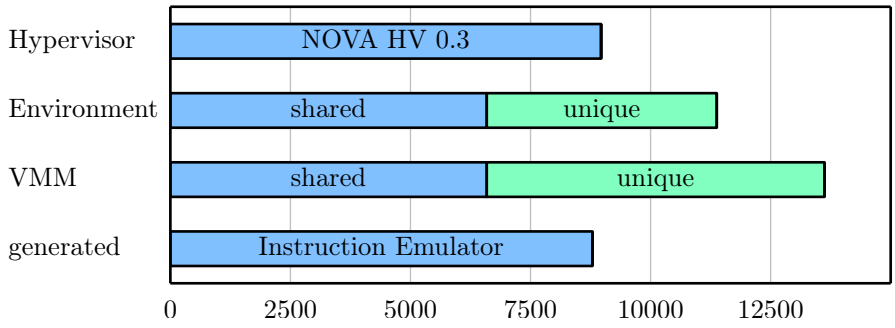

Figure 2.25: Size in SLOC of the NOVA system.

- Modeling newer devices that have queues and that can be emulated through polling would reduce the number of MMIO faults per request. Examples are the 82576 network model and the NVM Express interface for disks [NVM12]. This could make virtual network and disk I/O as fast as paravirtualized I/O without the need to modify the guest OS.

In summary, it should be possible to keep the near native performance for CPU and memory bound workloads but significantly improve the I/O performance of our system in the future.

2.7 Conclusions

With NOVA, we aimed to reduce the TCB impact for both unmodified guest operating systems running in virtual machines and applications not depending on virtualization at all. The size of the NOVA virtualization stack is shown in Figure 2.25. The NOVA user-level environment NUL, which multiplexes the physical resources between different VMMs, consists of 12 KSLOC. This includes around 7 KSLOC of code shared with the VMM. Most notably, the instruction emulator and multiple device models are reused by NUL to emulate VESA BIOS code. The Vancouver VMM consists of 14 KSLOC hand written and more than 8 KSLOC of generated code. The NOVA hypervisor contributes around 9 KSLOC to the TCB. In summary, virtual machines in NOVA are implemented within approximately 30 KSLOC. This is at least an order of magnitude smaller than existing virtualization stacks.

In this chapter, I have described my contributions to these results, centered around the implementation of the small Vancouver VMM:

- A software component design for the VMM and the user-level environment that enables the specialization of the codebase to the requirements of the guest as well as the host platform (§2.2).

- Newly developed device models to emulate a PC that are significantly smaller; mostly achieved by externalizing debugging and checkpointing functionality (§2.3).

- An x86 instruction emulator that relies on code-generation for a smaller size. I also suggested hardware support to speed up the emulation and introduced a simpler implementation approach based on physical CPU analysis (§2.4).

- A BIOS virtualized inside the VMM, which can even be emulated (§2.5).

Moreover, I have reevaluated the benchmarks, quantified the measurement error, and detailed the overheads we are causing (§2.6). This showed that NOVA is virtualizing CPU and memory slightly faster than state-of-the-art virtualization stacks. Finally, I estimated the impact of future developments on the performance of NOVA (§2.6.5).

2.7.1 Size Outlook

Will the current size advantage of NOVA remain in the future if new features are added and new platforms have to be supported? Or will NOVA grow similar to Linux or Windows as depicted in Figures A.4 and A.7?

In this thesis, I show that certain parts of the system functionality can be implemented outside of the TCB and that various techniques such as code generation or a heuristic allow to implement the same features within less code. This suggests that NOVA will not grow as fast as comparable systems, given the same rate of innovation. There are even a number of reasons indicating that the trend of an ever increasing codebase might be broken:

Microhypervisor The microkernel approach to the hypervisor design means that many features such as migration, can be implemented completely outside of it [Gal13]. Other features such as record/replay or replication that traditionally require kernel support, can be moved into a second hypervisor layer with the help of recursive virtualization [KVB11].

Deprecated hypervisor functionality can be traded against the code needed to run on newer platforms. It is, for instance, possible to remove 32-bit support after porting the system to 64-bit. Similarly the virtual TLB code could be removed, whenever a large enough fraction of the target platforms supports nested paging. Thus the size of the hypervisor should not increase significantly in the future.

Device Drivers Device drivers are typically the largest portion of monolithic kernels. Our ability to completely isolate these drivers at the user level means that only those device drivers, which are actually needed by a VM, have to be part of its TCB. Furthermore, running device drivers inside virtual machines (Driver VMs) ensures backward compatibility without the need to maintain legacy drivers and their support code inside the TCB.

Specialization The component architecture of the VMM and the user-level environment enables the specialization of the TCB to the usage scenario. Device models, drivers, and services that are not needed can be easily removed from the codebase.

Compatibility through Virtualization The code size increase for software compatibility reasons should be met with a consequent use of virtualization. Deprecated features should be removed as early as possible. Old NOVA applications, which are not adapted to the changing interfaces, should run inside a virtual machine within their original software environment.

In summary, NOVA has shown that the TCB of a virtual machine can be significantly reduced without inducing more overhead. Additional effort will be needed to retain this advantage in the future. However, the information available today indicates that there are many opportunities to further reduce the TCB.

Chapter 3

TCB-aware Debugging

> There are two ways to write error-free programs; only the third one works.
>
> A. Perlis in *Epigrams on Programming* [Per82]

Sophisticated debugging capabilities are crucial for the success of a system. They will not only reduce the time that is spent in bug hunting, they can also help external developers to write drivers and applications faster for the new system.

Debugging means more than single stepping an application or producing a core dump for a crashing component. It also includes inspecting a running machine to investigate any kind of unexpected system behavior, which could be caused by configuration errors, hardware failures, or even remote attacks.

Adding debug support to NOVA seems to be contrary to the goal of a small TCB (Trusted Computing Base), as debugging features usually lead to additional interfaces of trusted components and require additional code to implement them. To solve this issue, I researched how debugging support can be implemented in a TCB-aware way based on the previous results from the FULDA experiment [Kau07a] and Julian Stecklina's diploma thesis [Ste09].

Please note that simply removing debugging code during compile-time is not a viable solution, as to many programs depend on it. For instance, configuring a Linux kernel without a proc-filesystem is rarely done, even in highly secure environments. Loosing the ability to easily inspect a system just outweighs the reduction in TCB size. Instead, debugging features should be easily available even in a production system.

In this chapter, I discuss the requirements of a debugging infrastructure for NOVA (§3.1), the implementation of VDB (Vertical DeBugging) (§3.2), and how Firewire can be used to minimize the target driver (§3.3). Finally, I develop a minimal debug stub that can be injected into a running system (§3.4).

3.1 Requirements

3.1.1 Virtual Machines and Emulators

Virtual machines and emulators have been proposed multiple times as a debug platform for operating [GG74, Bel05, KDC05] and distributed systems [HH05]. The main advantage of using VMs for debugging is the full control over the machine: VMs can be halted,

restarted, replayed, and the machine state can be manipulated. Finally, adding debug support to a virtual environment is easier than adding it to a physical machine.

There are various arguments against this approach, besides the fact that a virtual machine for NOVA would need to support nested virtualization [BYDD+10]. Most importantly, virtual environments emulate only a limited set of hardware devices. Therefore only a fraction of the drivers, which are known to be the major cause of OS bugs [CYC+01], can be debugged. Qemu [Bel05], for example, models only 18 different network cards, which covers approximately 5% of the 349 network drivers of Linux 2.6.38.

Furthermore, the emulation of hardware is usually not perfect due to bugs, incomplete implementations, or performance reasons, which limits its debugging use. Qemu, for instance, does not check x86 segment limits because this would significantly increase its overhead. An OS, which violates these limits, will run on top of Qemu but would crash on a real machine.

Moreover, timing in a virtual environment can be very different, which may just hide certain bugs. [CCD+10] reports that measurements in PTLsim, a cycle-accurate x86 simulator, can deviate from the hardware by up to 30%. The timing in a general-purpose emulator may even be worse, as these implementations usually aim for speed but not cycle accuracy. Another example for timing issues is the synchronization of multiple CPUs. It is for example impossible to reproduce many race-conditions in multiprocessor workloads with Qemu because it strictly serializes the execution of guest CPUs.

In summary, relying on an emulator or a virtual machine for debugging NOVA is not feasible. Instead, the debuggee has to run on **real hardware**.

3.1.2 On-target versus Remote Debugger

A system debugger can reside in two locations: it can either be part of the target or it can run remotely on another machine. In the following, I discuss these options in detail.

Kernel Debugger A kernel or hypervisor internal debugger can directly access the hardware and kernel data structures without the need for a special debug interface. It can also include features for application debugging. Fiasco's JDB [MGL10], for example, allows an application developer to upload symbol information to beautify the backtrace of an application thread.

Nevertheless, kernel debuggers tend to be quite complex, as they suffer from the limited programming environment available at kernel level. Higher-level services and libraries such as graphical output or an USB (Universal Serial Bus) input driver are typically unavailable especially in small kernels. Furthermore, these debuggers are seldom online extensible. Every change will require a kernel recompilation and a time-consuming restart of the debugging session. Finally, if debugging is a compile-time option, it cannot be used to debug errors observed in the wild. However, if it is compile-in, debug code will significantly increase the TCB. JDB, for instance, adds more than 30 KSLOC to Fiasco's codebase [Ste09].

Remote Debugger The alternative to a debugger on the target is one running remotely on another machine with direct access to the kernel and the applications on the target machine. This has the advantage that such a debugger can manipulate all parts of the system and that its development can benefit from a richer programming environment. Compared to an in-kernel debugger, the side effects of the debug session to the target can

be limited and the additional lines of debug code in the TCB of an undebugged system can be minimized.

A remote debugger on the other hand requires a second machine and a fast transport medium to access the target. However these requirements are not a real problem anymore, as mobile devices are widely deployed and a wired or wireless network connection is available on nearly all of them.

Application Debugging Most developers and users will not debug the kernel itself but will target applications instead. In this case, it might be sufficient to implement the debugger as a service that runs on top of the target kernel. The development of such a debugger benefits from the richer programming environment as it can use existing libraries and frameworks. Furthermore, it can be implemented in any programming language available on the target. Finally, it would be relatively easy to replace. Nevertheless an application debugger would not support the most-critical and the hardest to debug code in a small system such as the kernel, the device drivers and all the services it depends on.

An application debugger has to rely on kernel debug interfaces[1] to control other applications and request information only available inside the kernel such as the list of threads, the layout of address spaces, and the installed capabilities. Such a debug interface does not exist in the NOVA hypervisor yet. Fortunately it will be very similar to what a remote debugger needs, except that a local implementation needs to restrict the set of protection domains a debugger can target. This keeps the isolation boundary of protection domains and would not taint applications that are uninvolved in the debugging session.

Summary A debugger that runs inside the hypervisor will be complex and will increase the TCB significantly. Furthermore, an application debugger alone is insufficient. I therefore designed and implemented a **remote debugger**. Such a remote debugger can run later directly on the system to debug applications without requiring a second machine.

3.1.3 Tracing and Interactive Debugging

Together with fail-stop debugging, tracing is one of the simplest non-interactive debugging tools of an operating-system developer. Tracing can sometimes be just a printf to a serial console, but may also be a tracebuffer in memory where entries can be added with little overhead. A tracebuffer may be dumped later to a remote machine or may be inspected directly on the target. Even though tracing is helpful, especially for debugging control-flow-related issues, it has a fundamental limitation: either all changes to the machine state are traced, which is often infeasible due to the sheer amount of data this produces, or critical events may be missed. Because it is often unknown at the beginning of a debug session, which events will lead to the bug, the tracing process has to be restarted multiple times with varying tracing points in the code. This can make debugging cumbersome or even infeasible in situations where it takes hours to reproduce the exact error condition. Furthermore, extensive tracing will influence the timing of the system and the so called *heisenbugs* will silently disappear [Gra85]. Thus, tracing cannot detect certain classes of bugs.

Interactive debugging on the other hand allows to inspect a remote system and even manipulate it. The debuggee is usually halted, to have a consistent view of the system. Halting may be requested by the programmer or may be the result of some CPU internal condition. The CPU, for instance, could have executed a certain instruction or modified

[1] Linux provides the `ptrace` system call and the `proc` filesystem for this purpose.

a certain variable. Because the developer can refine his view on the system without the need to rerun the debuggee, bugs can be found more easily.

Because tracing is often to simple to be helpful, I aim for a debugger that can **interactively** inspect and manipulate the whole system.

3.1.4 Special Requirements for NOVA

The NOVA architecture leads to additional requirements for a system debugger. The complexity of the system does not primarily originate from a single component because it may be tested and debugged in isolation. Instead the interactions between different components, the nesting of services and the legacy code in virtual machines are the main reasons for unexpected behavior. It is therefore necessary to be able to inspect and manipulate all layers of the system to understand and fix these bugs.

This can be achieved if the debugger provides a **vertical** view of the target system that includes the hypervisor, the applications, the user-level drivers, the virtual machine monitors, and the virtual machines itself. Even an application inside a virtual machine should be accessible by the debugger. Thus, applications written in different programming **languages** have to be supported as well.

Finally, NOVA is an open system where new components emerge and old ones are plugged together in unforeseeable ways. To support such configurations the debugger needs to be easily **extensible**.

3.2 The Vertical Debugger VDB

The last section showed that an interactive debugger is needed that provides a vertical view of a remote system. It should allow to manipulate programs written in different languages, which run on multiple CPUs in their own address spaces.

In the following, I explore whether GDB could be used for this task. I then describe the VDB (Vertical DeBugging) architecture and reveal some implementation details.

3.2.1 Reusing GDB?

The de facto standard for remote OS (operating system) debugging is the GNU Debugger (GDB) [SPS13]. There are multiple reasons for this: First, GDB is already known to many developers, which limits the time that is needed to get familiar with the debugging tool. Second, it has rich debugging features, which might need extensive work in a novel debugger. Especially data-type inspection, expression evaluation, and source-level debugging are commonly assumed as too complex to justify a new implementation. Finally, GDB supports many architectures, there are a lot of frontends to GDB, and the remote protocol is relatively simple. Relying on GDB as debugger therefore minimizes the development effort for an OS project.

However, the GNU Debugger has, due to its roots in Unix application debugging, a couple of drawbacks when utilized as a full system debugger:

Single address space Most importantly, GDB has no distinction of virtual and physical memory. It also lacks the concept of multiple address spaces. This makes it difficult to show physically addressed data structures such as page tables, impossible to handle unpaged areas in the virtual address space, and difficult to set breakpoints in one instance of a program but not in the other. However, multiple address spaces can be emulated

manually to some degree by using multiple GDB instances in parallel. Nevertheless this workaround is not a viable solution for NOVA, as it does not allow, for instance, to efficiently inspect a single variable in all instances of a program.

Unsupported hardware registers GDB has no support for various hardware registers. For example, on the x86 architecture it does not support the control registers, the shadow parts of the segment registers, and the model-specific registers (MSRs). Therefore, it cannot detect processor mode changes, which is especially important when debugging mixed 16-bit and 32-bit code. Furthermore, inspecting code that uses non-zero segment bases is cumbersome, as breakpoints and memory reads do not work as expected as the segment base has to be manually added to the corresponding debugger expressions. Because the segment bases and the MSRs are usually not accessible by GDB, developers have extended the remote protocol to get at least read-support from their targets. Unfortunately, their extensions are not integrated into GDB and have to be redeveloped for every new system.

Slow remote debugging protocol The remote protocol of GDB was designed for the slow serial port and its core commands roughly resemble the `ptrace()` interface. A lot of performance is already lost at this protocol level. For instance, memory reads are requested in small chunks, encoded as hexadecimal string and the protocol requires a round-trip before the next command can be issued. Thus, executing GDB scripts with a remote target can be really slow. For instance, displaying the run queue of the NOVA hypervisor with the GDB protocol without any intermediate caching may take multiple seconds [Ste09]. Such a low performance makes interactive debugging impossible.

Large Size GDB has grown to over 400 thousand lines since its debut in 1986. This number increases to over a million lines [LHGM09], if the supporting libraries are counted as well. Understanding, extending, and optimizing such a giant codebase is a tremendous task. An alternative and simpler approach seems to be preferable over fixing GDB.

GDB is insufficient In summary, GDB is *insufficient for systems debugging* [Ste09]. Instead a new debugger is needed that should natively support **address spaces** on multiple CPUs, give access to **all** available **hardware state**, and is **faster** then GDB.

3.2.2 Design

Due to the results of the previous investigation, I started to develop my own debugger named VDB (Vertical DeBugging). VDB is a portable and interactive systems debugger that aims for a small TCB footprint. Figure 3.1 depicts the key components of its architecture.

Target Driver and Debug Stub On the target side a *target driver* and a small *debug stub* are added to the hypervisor. The driver programs the hardware and provides the backend for the low-level transport-layer. The stub implements any higher-level protocol on top of it.

Usually remote debuggers have only a monolithic debug stub running on the target. Splitting them instead into two parts allows to customize them. For example, if a particular host debugger implementation will never issue remote I/O operations, the debug stub does not need to support them. Similarly, the target driver can be customized to

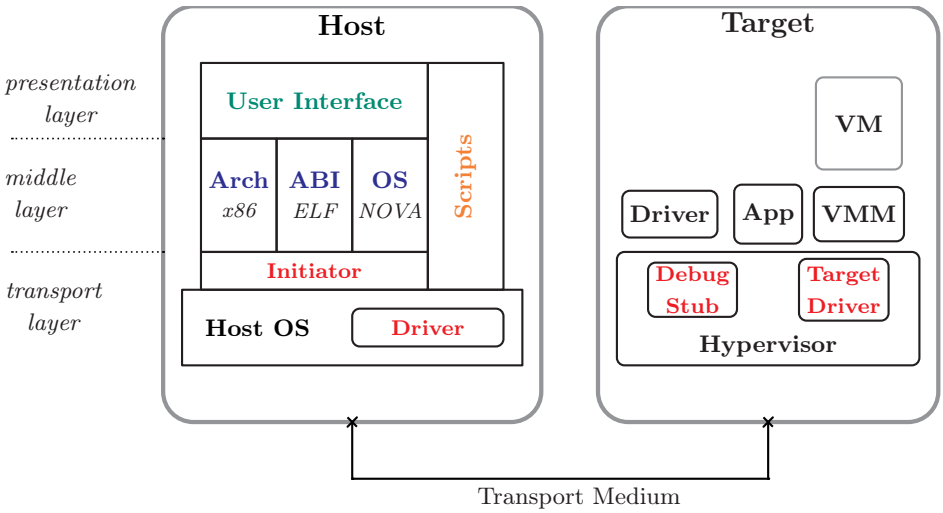

Figure 3.1: The VDB (Vertical DeBugging) architecture.

the particular platform. The driver can even be omitted as shown in the next section, if its tasks can be completely performed by hardware.

Transport Layer The transport layer abstracts from the actual hardware that is used to connect the two machines. It consists of an *initiator* on the host side that sends requests with the help of an *host driver* over the *transport medium* to a *target driver*.

The transport protocol is much simpler than the GDB remote protocol with its dozens of commands. Instead it supports only three operations: to read and write from physical memory as well as to signal the target machine. The simplicity of this protocol has a couple of advantages. First, it improves the portability as the protocol is independent of the target OS and the processor architecture. Second, it reduces the complexity of the target driver and allows to support a large set of transport mediums with a minimum of driver code. Third, its operations are sufficient enough so that higher level features such as CPU register access or halting the target machine can be efficiently implemented on top of it as shown in Section 3.4. Finally, most of the operations that inspect, analyze, and manipulate a remote machine can be already performed on this low level.

Middle Layer The middle layer is the core of the debugger. The *Architecture*-specific part abstracts from the machine architecture, for instance x86 or ARM. It understands hardware-specific data structures such as page tables and provides virtual address spaces to other debugger components. The *ABI*-specific part abstracts from the object file format such as Unix ELF (Executable and Linkable Format) or Windows PE (EXE), and the representation of debug information such as DWARF or PDB (Program Database) files. It parses program files and provides this information to other debugger components. The *OS*-specific part abstracts from the target OS that should be debugged. It provides the ability to stop and continue a target as well as to manipulate CPU registers. Moreover, it supports tasks that are frequently needed when debugging a particular target OS such as producing a process-list or listing the runnable threads.

By splitting the middle layer at its natural dependency boundaries, portability is

ensured. If, for example, the NOVA system is ported to ARM, only the architecture dependent debugger code needs to be adapted. Similarly, if the debugger should target a non-NOVA system, only the OS part is affected.

Presentation Layer The presentation layer visualizes the collected data and presents it to the developer. An example is a *user interface* for interactive debugging.

Scripting A powerful scripting language provides extensibility and flexibility, as it allows to develop new debugger components and plug them together in unforeseen ways. It can also be used to automate common debugging tasks.

3.2.3 Implementation

I have implemented a prototype of the VDB architecture in Python. In the following, I will describe some noteworthy details of it.

The implementation consists of approximately 3 KSLOC today. While the prototype misses certain features such as single stepping or a sophisticated GUI, it is already usable for many tasks such as:

- Getting access to the memory of a running OS via Firewire, the GDB protocol, or `ptrace`.
- Parsing symbols and types from DWARF debug information attached to ELF binaries [Eag07, DWA10, ELF95].
- Retrieving PDB files for Windows binaries from Microsoft symbol servers [Sch01].
- Evaluating data structures symbolically.
- Accessing deeply nested address spaces such as an application inside a VM.
- Viewing certain x86 data structures such as page tables, GDT, TSS, and the VGA text screen.

Python

I have chosen Python as programming language for VDB because of its excellent rapid-prototyping properties. Python programs are compact, easily extendable, and the test cycles are short.

An implementation in a scripting language will surely be slower than one written in a low-level language like C. Nevertheless, performance was seldom an issue. The only case where Python came to its limits was the decoding of DWARF debug information. While this code might be a good candidate for a reimplementation in C, I've chosen the simplest way out and just cached the results between different runs.

Python supports several ways to interface with code written in other programming languages. A simple approach is `os.popen` that executes an external program in another process. It establishes pipes to `stdout` and `stdin` that can be used to communicate with the process. I used this technique to demangle C++ symbols and for disassembling code with `objdump`.

A quite elegant way to use system libraries in Python programs provides the `ctypes` module. Figure 3.2 shows how it can be used to implement a `ptrace` binding within just

```
libc = ctypes.CDLL(ctypes.util.find_library("c"), use_errno=True)

def ptrace(command, pid, addr = 0, data = 0):
    if command == PTRACE_PEEKDATA:
        data = ctypes.byref(ctypes.c_long(data))
    if libc.syscall(SYSCALL_PTRACE, command, pid, addr, data):
        raise Exception("ptrace error %s"%(os.strerror(ctypes.get_errno())))
    if command == PTRACE_PEEKDATA:
        return data._obj.value
```

Figure 3.2: Implementation of a `ptrace(2)` binding in Python. This requires only a handful of lines when using the `ctypes` module.

eight lines of code. Note that I could not have used `libc.ptrace` directly because it overloads the error code with the result of PEEKDATA.

Finally, one can write extension modules for Python in C and dynamically load them at runtime. However, I have avoided this solution due to its complexity.

Abstractions

The VDB implementation is internally based on three abstractions: spaces, variables, and viewers.

A *space* allows to read and write to a region of memory. The space interface is provided by all transport layer implementations. Notable examples are the Firewire and `ptrace` target as well as a space that implements the GDB debug protocol. Page tables and caches are special spaces because they can be stacked. It is therefore possible to describe a space hierarchy of a Linux program in a VM (virtual machine) running on the NOVA hypervisor inside Qemu.

A *variable* associates a type such as `uint16` or `char` to some region in a space. Variables allow to evaluate expressions symbolically. While it is possible to synthesize types by hand, they are usually taken from the DWARF debug information [Eag07]. DWARF describes independent of the programming language the types and variables in a binary. Many compilers already generate DWARF output.

The possibility to freely and dynamically generate classes in Python allows to craft a local shadow object for every variable in a remote program. Members of the variable are attributes of the shadow object. Overloaded operators lead to a compact representation: the unary plus yields the address, the bitwise-negation the type and a call returns the value of a variable. See Figure 3.3 for an example how this can be used to evaluate a single linked list in a straightforward way.

A *viewer* renders a data structure in a human readable way by producing a stream of lines. I relied on the generator feature of Python [PEP] to implement viewers with a minimal latency. A viewer yields the output it produces line by line to the caller. This is similar to pipes on a shell, except that the data is transferred not byte- but line-wise. Furthermore, a generator in Python can be easily aborted and it can even throw exceptions.

There are viewers that are specific to the operating system or machine architecture such as the capability space viewer for NOVA or the page-table viewer of x86. Additionally there are generic viewers that for instance recursively show members of nested classes. Viewers can be stacked, which allows to decorate an anonymous disassembly with a set of symbols or to filter another viewer via regular expressions similar to `grep`.

```
def show_vma_list(space):
    start = +space.vma_head
    vma = space.vma_head.list_next()
    print "VMA in", (~space).name
    while True:
        print vma.node_base(), vma.node_order(),
        print vma.node_type(), vma.node_attr(), +vma
        vma = vma.list_next()()
        if +vma == start:
            break
```

Figure 3.3: Symbolic expression evaluation. The function prints the virtual memory area (VMA) list, which the NOVA hypervisor keeps for the memory and capability space of every protection domain.

User Interface

VDB has a simple console interface. Input is provided by `libreadline` with command history and tab completion of expressions. It supports shell-like tools such as `less`, `hexdump`, and `grep` to handle large output in a reasonable way. VDB also implements simple timer-based dumps. This can be used to show the state of a UTCB (User Thread Control Block) as it changes dynamically or to periodically monitor a remote VGA console.

Future Work

VDB was already useful to detect dozens of bugs. However it is surely not complete yet. Most importantly, VDB lacks a graphical user interface to inspect multiple data structures in parallel. This GUI could be similar to the Data Display Debugger [ZL96]. Second, the inspection of the thread state should be more detailed. Evaluating the corresponding DWARF expressions should allow to identify local variables on the stack even in nested calls. Third, the performance, especially when parsing DWARF information, needs to be improved. Finally, the support for Windows debugging is highly experimental. It should be fully integrated in the existing debugger core and it should be extended to speak the serial protocol of the windows kernel debugger [RS09] to be able to receive debug messages from a virtual machine running Windows.

3.3 Debugging without a Target Driver

A remote debugger needs a transport medium to access the state of a debugged machine. The transport medium chosen for this task does not only have an impact on the performance and cost of the resulting system, but also on the lines of code that are needed to drive the hardware. A fast transport medium supports use cases besides debugging. It may also be useful for trace extraction, remote booting and the implementation of remote services.

In this section, I will show that Firewire excels at these tasks and that a creative use of its features allows to reduce the TCB impact, so that in the best case, no driver code is required in the target after initializing the platform.

3.3.1 Choosing the Hardware

In the following, I discuss the advantages and disadvantages of choosing a serial line, Ethernet, USB, or Firewire as transport medium. Other options such as an In-circuit emulator (ICE), a self-built PCI (Peripheral Component Interconnect) card, IPMI Serial over LAN, and Intel's Thunderbolt technology can be easily dismissed due to their high costs and limited availability.

The Serial Line

The de-facto standard to connect to an OS development machine is the RS232 serial line, due to the following reasons: On one hand, programming a serial port is easy and no complicated software stack is needed. Thus it can be used early in the operating systems implementation phase to give feedback to an OS developer and help to debug the system. On the other hand, the hardware is cheap and was, until recently, already built into most main boards.

However, a serial port is not the best choice for debugging, tracing, and as a general support interface anymore. Most important, its relatively low speed (typically less than 1 Mbps) forbids certain use cases such as retrieving a large memory dump or performing a live analysis of a high-volume trace on another machine. Moreover, as serial ports are usually not built-in anymore, they have lost the advantage of the cheapest hardware. For these reasons, the serial line is not the right choice. The transport medium has to be **faster**.

Network Cards

NICs (network interface controllers) might be a good alternative to the serial port since fast and gigabit Ethernet as well as wireless networks promise at least two orders of magnitudes more bandwidth. Additionally, the hardware is already present on most platforms.

On the other hand, a large number of different NICs are deployed. Linux for example supports more than a hundred Ethernet adapters, including at least three dozen gigabit NICs. Unfortunately, most of them come with their own interface and require a dedicated driver. The large set of drivers makes it infeasible to use Ethernet as transport medium in a small OS project. In fact, solving the device driver problem is beyond the scope of this thesis and rather a part of future work.

A transport medium should require only **a small set of of drivers**.

USB

Another alternative is USB. With a nominal speed of 480 Mbps (USB 2.0) it is fast enough for many debugging scenarios. Furthermore, USB ports are readily available on every recent PC, thus expansion cards are seldom required. However, two hosts cannot be directly connected with standard cables, due to the master/slave architecture of USB. Instead a special link cable is needed. This makes a USB based solution more expensive than an Ethernet based one.

Theoretically, there are only a few drivers needed to support the most widely deployed USB chips (e.g. OHCI, UHCI, and EHCI). Nevertheless, Linux 2.6.34 contains approximately 50 KSLOC for the USB host drivers alone, which form the lowest layer of the much larger USB software stack. Several thousand lines of a USB stack are a significant burden for an OS aimed at a small TCB. Fortunately, there exists an extension to the

EHCI standard, namely the USB debug port which has a much simpler programming interface and does not require a full USB software stack. This minimizes the code, but also significantly limits the transfer speed to approximately 1 Mbps, which makes it too slow to be a real alternative.

USB would be a good choice if it had a **smaller software stack**.

Firewire

Firewire, officially known as IEEE-1394 [IEE08], is a high speed serial bus interface. Up to 63 devices can be connected to a single bus. Bridges allow to connect multiple busses. Firewire supports isochronous as well as asynchronous transfers with speeds between 100 and 3200 Mbps. The 400 Mbps and 800 Mbps versions are the most common ones. Thus Firewire is approximately three orders of magnitude faster than a serial line. See Section 3.3.5 for a performance evaluation of Firewire.

Most Firewire controllers follow the OHCI (Open Host Controller Interface) specification [OHC00]. This is an important advantage of Firewire, as only a single driver is required for all cards. Furthermore, the designers of OHCI moved most of the bus management into hardware. A Firewire/OHCI driver will therefore be much simpler than an USB/UHCI one.

Firewire does not employ a master/slave architecture like USB. Instead all nodes on a bus are equal. Therefore, multiple machines can be directly connected with standard cables without requiring special hardware. The only disadvantage of IEEE-1394 seems to be its falling popularity. New machines do not necessarily come with a Firewire port anymore. Nevertheless, this is not serious drawback as a Firewire expansion card is as expensive as a USB-link cable or a missing serial port.

In summary, Firewire is faster than a serial port, requires less driver code than USB or Ethernet, and the hardware costs are moderate. Consequently, I employed Firewire as transport medium to debug other machines.

3.3.2 Remote Access Without Runtime Code on the Target

In the following, I will discuss how the target driver can be reduced by using Firewire. First, I describe how remote memory can be accessed without involving a driver. Second, I show how this mechanism can be used to inject interrupts as well.

Memory Access

The programming model of Firewire/OHCI is similar to the one of a NIC:

1. The driver configures a RAM region for incoming requests.
2. If a request is received by the card, it transfers the data into these receive buffers.
3. The card notifies the driver by raising an IRQ (interrupt request).
4. The driver looks at the request, creates an answer and puts them into a reply queue.
5. The card asynchronously sends the reply.

Involving software on every request can be a bottleneck for high performance scenarios. Therefore, OHCI based controllers allow to handle physical DMA (direct memory access) requests completely in hardware. This feature makes all or parts of the main memory of a machine accessible to a remote node. Write requests that target this area are transferred

autonomously by the Firewire controller via DMA into main memory. Similarly, the controller replies to a read request with data directly from RAM.

Using physical DMA is especially elegant for a debugger because it gives access to a remote host without involving any target driver. This allows to inspect even a stuck system. Moreover, it is minimally intrusive because no interrupt disturbs the program flow and no driver code will pollute the CPU caches.

Allowing any node on a Firewire bus to autonomously access the physical memory of a machine is a security risk. Many publications, for instance [BDK05], have shown that this can be used to compromise the operating system and to steal private data. The impact of this threat is quite low in our case, as we usually debug our system in a development environment and therefore need to have access to physical memory anyway. However, in a production system it might be necessary to restrict debugging access to a limited set of users. With OHCI's features to selectively allow certain nodes to use physical DMA and to run handler code for received messages, it is possible to implement a challenge response protocol between two Firewire nodes. A handler for these messages would only enable physical DMA for a particular node after successful authentication.

Interrupt Injection

Having access to remote memory is necessary but not sufficient for a fast debugger, as any higher level debugging protocol between target and host would have to use polling to detect incoming messages. An implementation that aims for better performance needs a way to signal a new message to the target.

In 2007 I discovered [Kau07a] that Firewire DMA can be used to trigger MSIs (Message Signaled Interrupts). MSIs are 4-byte messages that target a certain address range[2] [Bal07, SDM13]. They can be send as a normal DMA write transaction by any device on the PCI bus. The PCI host bridge will forward these messages to the system bus where all CPUs will receive it. A CPU interprets an MSI as a new interrupt vector. MSIs can also trigger a NMI (non-maskable interrupt), which is especially important for a debugger that wants to examine interrupt-disabled kernel code paths. MSIs have been supported for a couple of years now since they were introduced on the x86 platform with Intel's P4 and AMD's K8 CPUs.

Thus, the remote CPUs can be directly signaled with Firewire through MSIs. Polling on requests and interrupt forwarding to other CPUs is not required.

3.3.3 Surviving a Bus-Reset

Unfortunately, physical DMA stops working as soon as the Firewire bus is resetted. A bus-reset occurs, if a new device is attached or an old one leaves the bus. They will happen often in an OS debugging environment where machines are constantly rebooted.

During the bus-reset phase, the new topology of the bus is detected and unique node identifiers are assigned to the devices. These identifiers are later used to target requests to particular nodes on the bus. If a reset occurred, an OHCI driver needs to clear the bus-reset indicator in an hardware register, before physical DMA will work again[3]. Thus, target driver code has to run after a bus-reset, which negates the benefit of Firewire to work without runtime code. Furthermore, a debugger would need to be written very

[2] On x86 they are mapped to the standard Local APIC (Local Advanced Programmable Interrupt Controller) address range of 0xFEExxxxx.

[3] Drivers usually initialize the physical request filters as well. This step can be omitted, if accesses from non-local busses are permitted and the requests are sent with a different bus number.

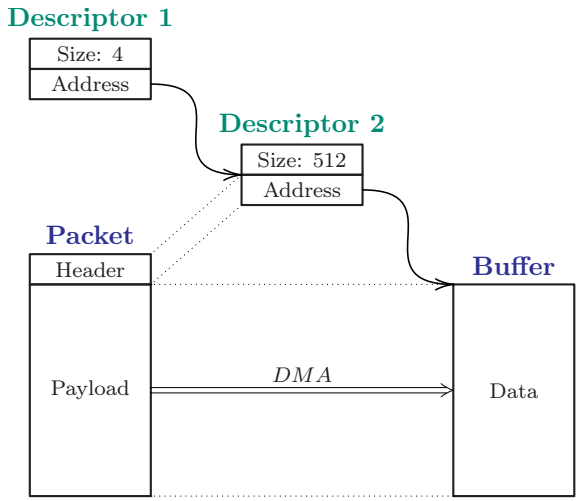

Figure 3.4: Self-modifying DMA. The first DMA descriptor specifies that the header of the incoming packet should be moved in the address field of the second descriptor. The second DMA descriptor defines that the payload of the packet should be moved to a buffer residing at this address.

carefully, to not disturb the driver during a bus-reset and loose the connection to the remote machine.

In the following, I will describe a novel technique that allows remote DMA even after a bus-reset without involving a driver anymore.

Self-modifying DMA

While searching for a solution to the bus-reset problem, I made two observations:

1. Firewire supports not only asynchronous transfers but also isochronous ones. Isochronous data is broadcasted over one of 64 channels and is not addressing a particular node on the bus. It can therefore be received independently of a bus-reset.

2. A Firewire/OHCI card transfers received packets autonomously via DMA into main memory. DMA descriptors are memory structures that define the destination address for these transfers.

The fact that the DMA descriptors reside itself in physical memory, can be exploited in the following way: If the header of the packet overwrites the address of the next DMA descriptor, which is afterwards used to transfer the body, the data in the packet itself chooses its destination. I call this technique *self-modifying DMA*, in short *SMD*, as the first part of a DMA program modifies the destination of the second part. See Figure 3.4 for an example.

Please note that SMD is independent of Firewire and can be implemented on top of any device that supports DMA, reads DMA descriptors from main memory, and receives external packets such as USB or Ethernet. The implementation of SMD on Firewire/OHCI cards is especially elegant due to the following four reasons. First, DMA programs can be never-ending because the DMA descriptors are implemented as a linked list instead of a circular buffer. Thus, a loop in the descriptors can be made by pointing to a previous

descriptor in the list. Reprogramming the card to recycle already used descriptor slots is therefore unnecessary. Second, the DMA engines on OHCI cards can perform scatter/-gather DMA with byte granularity. This makes the technique robust as only the necessary parts such as the address of the next DMA descriptor can be overwritten through SMD. Third, the packet data can be received in raw mode without the need to strip off any header or footer. Finally, there is a special receive mode, called *packet fill mode* where a fixed number of DMA descriptors is used to define the destination of a single packet. Descriptors that are not needed for a small packet, are simply ignored. This allows to receive variably sized packets without any alignment restrictions.

SMD provides write-only access to a remote machine; memory cannot be directly read with it. However, read-access can be implemented on top of it by using SMD to modify a running isochronous transmit DMA program. To distinguish answers from different nodes and to detect outdated messages, the node replies to isochronous read requests not only with the data but also with its Global Unique ID and the requested address.

Currently, SMD on Firewire can access only the first 4 GB of main memory because the addresses fields in DMA descriptors used by OHCI v1.1 are only 32 bits wide. The whole 48 bit address space of physical DMA can therefore not be used. Nevertheless, the technique is sufficient to inject code, trigger MSIs and recover the physical DMA path after a bus-reset occurred.

3.3.4 Implementation

The target Firewire driver is quite simple. It consists of approximately 200 SLOC (source lines of code) and runs once at boot-time just before the hypervisor gets control. It searches for Firewire cards on the PCI bus and initializes them to allow physical remote DMA. It also programs the isochronous DMA engines for self-modifying DMA.

If an IOMMU (I/O Memory Management Unit) is used by the hypervisor, a Firewire card cannot freely access memory. Disabling the IOMMU in the debugged system is not an option, as this would make features such as pass-through PCI devices infeasible. Instead an exception needs to registered at the IOMMU. This issue can be easily solved by extending the DMA Remapping ACPI table (DMAR), which the BIOS (Basic Input/Output System) uses to announce its own DMA regions to the hypervisor. The hypervisor uses this table as a white list during initialization time and will program the IOMMU accordingly.

Similarly, interrupt injection of a Firewire card can be inhibited by the hypervisor. Requesting the corresponding interrupt vector from the hypervisor would be the natural choice, however the NOVA interface does not allow to assign the NMI vector to any device. Changing the interrupt redirection table via remote DMA is not an option as the IOMMU may cache entries. Because interrupt redirection is currently only a security feature, but has no additional functionality yet[4], we can simply disable it in the hypervisor.

3.3.5 Firewire Performance

The performance of the debugger depends on the speed of the transport medium. Firewire has a nominal bandwidth of up to 3200 Mbps. However, protocol overhead and implementation issues will limit the available application throughput. Since the literature misses a consistent study on the performance that can be reached with Firewire, I have benchmarked different low-cost 400 Mbps cards (S400).

[4]This does not hold true anymore with the introduction of the x2APIC.

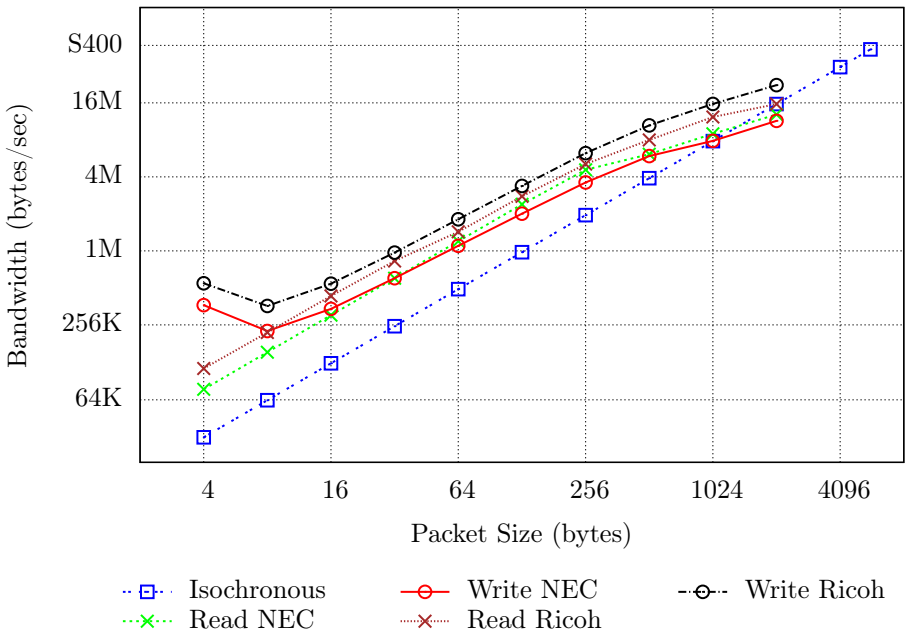

Figure 3.5: Firewire S400 Throughput

S400 Results

The microbenchmark transfers a large amount of bulk data between two directly connected nodes. To minimize the influence of driver and OS, I excluded scenarios where the hardware could not fulfill the requests on its own. I measured asynchronous reads and writes into the physical DMA region and isochronous transfers where DMA engines where programmed beforehand. I varied the packet size during the measurements and ran the benchmarks multiple times with half-dozen different controllers. The results for all chips are quite similar. However, in general newer chips achieve a better performance. The results for the controller with the best (Ricoh 1180:e832) and the worst (NEC 1033:00e7) performance are shown in Figure 3.5 and Figure 3.6.

From these figures the following points are worth a detailed look:

Isochronous Transfers The nominal bandwidth of S400 is approximately 400 Mbps This corresponds to 6144 bytes that can be sent every period of 125μs. The IEEE 1394 specification defines that 80% of the bandwidth are guaranteed to be available for isochronous transfers. This would be 320 Mbps or one packet of 4915 bytes every period. The actual implementation limit for isochronous traffic is higher and depends on the bus topology.

Over a single point-to-point link I was able to successfully transfer packets with any size between 4 and 5700 bytes. This is equivalent to 364 Mbps or 93% of the nominal speed of S400.

Controller Dependencies The throughput of asynchronous transfers depends on the Firewire controller. The fastest S400 controller can sustain writes with nearly 190 Mbps whereas the worst of the measured controllers barely reaches 100 Mbps.

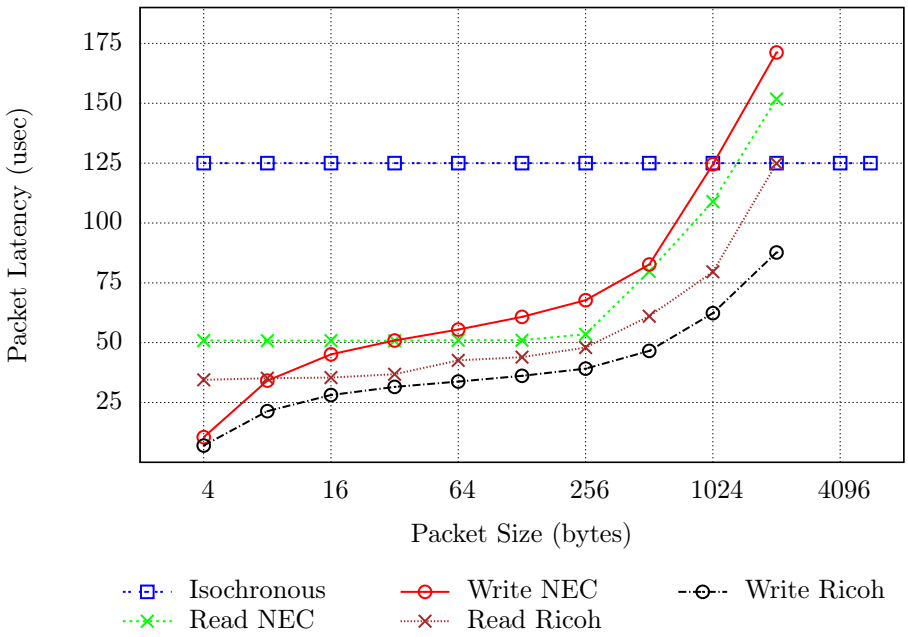

Figure 3.6: Firewire S400 Average Burst Latency

Similarly, the asynchronous read throughput varied between 130 and 100 Mbps. Isochronous transfers on the other hand do not dependent on the controller.

Isochronous vs. Asynchronous The bandwidth of isochronous transfers is nearly two times higher than what can be achieved with asynchronous transfers. One reason is that isochronous requests have the priority on the Firewire bus, to ensure their bandwidth guarantees. Asynchronous transfers on the other hand have to use a slow arbitration protocol. A controller has to wait quite a long time before it can take bus ownership and send asynchronous data over the wire. Finally, chip designers might have tweaked their chip for isochronous transfers to be specification compliant.

Reads vs. Writes Asynchronous reads are usually slower than writes. The NEC controller was the only exception to this rule. A reason for this might be that most Firewire controller implement *posted writes* to buffer the write request and send the reply before the data reaches main memory. This allows to piggyback the reply on the acknowledgement. A read request, on the other hand, cannot be acknowledged and replied in one step because the controller has to wait for the data to arrive from main memory. Even if the controller has received the data, it cannot immediately reply with it. Instead it has to wait at least an arbitration gap to gain bus ownership again.

4-byte writes It may look surprising that 4-byte writes are faster than 8-bytes ones. However, they are send as *quadlet* instead of block-write requests over the bus. This allows a more efficient encoding and requires less data to be transferred per request.

Beyond S400

Unfortunately, I couldn't get the hands on Firewire cards that are faster than 400 Mbps. However, the newer Firewire standards up to S3200 are already approved and hardware for S800 is purchasable now. Even if I was not able to measure them, the specifications allow to estimate the effect of the higher nominal speeds.

The 80% bandwidth guarantee of isochronous transfers holds for all Firewire versions. Thus, S800 guarantees up to 600 Mbps (75 MB/s) and S3200 up to 2400 Mbps (300 MB/s) for isochronous traffic. This is implemented by keeping the very same timing constraints (a period is still 125 μs long) but increasing the maximum size of a packet that can be send in a single period. Similarly, the maximum size of asynchronous packets are doubled with every version.

Furthermore, S800 introduces a new arbitration protocol. This protocol allows to perform bus arbitration in parallel to the data transfer by relying on an additional cable pair. This significantly reduces the transfer gaps and thereby the overhead of asynchronous traffic. I therefore expect that S800 more than doubles the achievable asynchronous bandwidth.

Summary

Isochronous transfers can fully use 80% of the nominal bandwidth. This corresponds to 320 Mbps for the 400 Mbps version of the Firewire standard. Between 100-200 Mbps can be achieved with asynchronous transfers, depending on the used Firewire controller. Finally, the newer Firewire standards will improve the performance significantly.

3.3.6 Related and Future Work

Firewire was used with various existing debuggers in the past. The Windows kernel debugger for instance can use serial ports, USB debug cables and Firewire to access a remote machine [Gre04]. Similarly, Fireproxy [Kai08] on Linux allows to tunnel the GDB remote protocol over Firewire as an alternative to the serial port of KGDB [KGD]. On FreeBSD the `dcons` console driver allows to use GDB remotely [Shi08]. However, in these projects, Firewire was solely employed to tunnel an existing remote debug protocol over a faster medium. Firewire's ability to directly access the machine without any protocol was not used. One reason might be that the ability to send MSIs over Firewire is largely unknown and that no solution to the bus-reset problem was previously known.

Firewire is useful in a development system for other scenarios besides debugging. I use Firewire regularly for booting and retrieving large amounts of trace output from remote machines because it is faster than 100 Mbps Ethernet. Furthermore, I plan to tunnel service requests over Firewire to give a development machine network and remote disk access when the corresponding hardware drivers are missing.

3.4 Minimizing the Debug Stub

In the previous section, I showed that remote memory can be accessed with Firewire without a target driver in the TCB. This leaves the debug stub on the target system. Unfortunately, it cannot be omitted because it is necessary for debug requests that require the cooperation of the debugged CPU[5]. Examples of such requests are:

[5]Relying on additional hardware such as an In-circuit Emulator would also solve the problem, however this is to expensive to be employed in the large scale.

- Stopping and restarting a CPU,
- Reading and writing CPU registers,
- Flushing cached data to memory, and
- Accessing platform devices.

Currently, debugger developers would add a large debug stub directly to the OS kernel that supports all needed requests. However, linking the stub to the kernel makes it part of the TCB even in a system that is never debugged. Moreover, this approach is inflexible because it does not allow to add new features in a backward compatible way. Old kernel versions will never support new debugging features.

I developed another technique instead: by injecting a small stub into a running system, it will not be part of the TCB and can be specialized to the debug problem. Such stubs need only a handful of instructions. They can be independent of the target OS, which minimizes the implementation and testing effort.

Note that Julian Stecklina already presented in his master thesis how a monolithic debug stub can be injected from a remote machine into the NOVA hypervisor [Ste09]. I significantly improved his work by splitting the stub in multiple pieces and reducing the complexity of them. I also simplified the injection procedure by performing more steps in the debugger and not in the stub itself.

3.4.1 Design of a Halt and Resume Stub

In the following, I will show that a very small debug stub is enough to halt and resume a CPU. More functionality can be provided on top of it by injecting additional debug plugins into the target. At the end of this section, I describe what is necessary to debug the NOVA hypervisor with this stub.

The debug stub will be triggered by a NMI. It first saves the CPU registers to memory to make them available to the remote debugger. It then halts the CPU until the next NMI is received. Finally, it loads the CPU registers from memory to resume execution.

I have chosen an NMI instead of a normal IRQ to be able to debug kernel code that runs with disabled interrupts. While this increases the applicability of the stub, it also leads to a serious problem. The NMI can interrupt the system in a state where not enough stack is available to save the previous CPU state, especially the current stack and instruction pointer. The NOVA hypervisor for instance uses the execution context (EC) as stack during kernel entry for a couple of instructions to push the user-level registers directly into the EC. If an NMI hits one of these instructions, it can easily corrupt the EC data structure.

The NMI handler should therefore execute on its own stack. A TSS (task state segment) can be used for this purpose in 32-bit mode[6]. An NMI leads then to a hardware-task switch, which saves the current CPU registers in the previous TSS (RUN-TSS) and loads new register values from the NMI-TSS including a new stack and instruction-pointer. The general-purpose registers need not to be explicitly saved and restored in the stub.

The CPU will be resumed at the previous state by switching back to the RUN-TSS with `iret`. Unfortunately, a hardware-task switch does neither save the CR3 (control register #3) nor the LDT (Local Descriptor Table) register, but just loads them from the destination TSS. It is therefore necessary to fix these two fields in the RUN-TSS

[6] In 64-bit long mode, where the hardware-task switch is not supported, an interrupt stack can be used instead.

```
halt:   hlt                     // wait for an NMI
        jmp halt                // repeat on the next invocation

nmi:    dec  %esi               // count NMIs
        js   doiret             // filter NMIs that would resume too early

        xchg %eax, 32(%ebp)     // xchg EIP in the running TSS
        xchg %ecx, 36(%ebp)     // xchg EFLAGS
        xchg %edx, 76(%ebp)     // xchg CS
        xchg %ebx, 80(%ebp)     // xchg SS

doiret: iret                    // return to the running TSS to halt or resume
        jmp nmi                 // repeat on the next NMI
```

Figure 3.7: A debug stub to halt and resume a CPU can be implemented within ten instructions. The NMI code is executed on its own TSS and switches between running and the halt loop. The registers and the resuming CR3 are initialized from the debugger.

before the previous state can be resumed. The fix-up could be directly done from the NMI handler. However, this would surely complicate the stub, as this code is highly OS dependent. The debugger therefore corrects these values from the outside, just before it sends the NMI to resume the CPU.

Halting the CPU

Halting the CPU while running on the NMI-TSS is not possible because NMIs and normal interrupts need to be blocked while this TSS runs. Thus, the debugger would not be able to signal continuation to the debug stub except through shared memory. However, polling on memory wastes a lot of CPU resources and consumes power unnecessarily.

I figured out that the RUN-TSS can be reused for halting the CPU[7]. This means the NMI handler just flips the instruction pointer and segments in the RUN-TSS with new values, to let it point to the `hlt`-loop. It then returns to the RUN-TSS. A second NMI then flips these values back. The NMI handler switches in this way between the halt and run state.

Please note that an unconditionally switch between these two states is not robust against spurious NMIs generated by watchdogs or overflowing performance counters. To filter these unwanted wakeups out, the number of received NMIs can be counted and a resume can be aborted if this value is below a threshold.

Implementation

A 32-bit stub code implementation is shown in Figure 3.7. The control flow is the following:

1. The stub code is injected into the target machine by writing it into remote memory. The NMI-TSS is initialized and the interrupt descriptor table points to it.

2. The CPU is signaled with an NMI to halt execution.

[7]Previous work required another TSS for this purpose [Ste09].

3. The CPU performs a hardware-task switch to the NMI handler. The NMI handler flips the EIP, CS, SS, and EFLAGS in the RUN-TSS to the `hlt`-loop. It then returns with another hardware-task switch to the RUN-TSS.

4. The CPU executes `hlt` in a loop with interrupts disabled. This effectively stops the CPU and saves power. The debugger can inspect and manipulate the system.

5. The debugger fixes the CR3 as well as the LDT values in RUN-TSS and triggers an NMI to break out of the `hlt`.

6. The CPU performs a hardware-task switch to the NMI-TSS. The NMI handler flips the previous values of EIP, CS, SS, and EFLAGS in the RUN-TSS back. The CPU is resumed at the same place it was interrupted before by returning with a hardware-task switch to the RUN-TSS.

The stub code largely benefits from the hardware-task switch, as this already saves the general-purpose registers in the RUN-TSS and thereby makes them available to the debugger. Initializing the data structures and performing the fix-up from the outside simplifies the stub further. Consequently, it consists of only ten instructions that compile to 20 bytes compared to approximately 250 SLOC for an injectable stub in the previous version [Ste09].

Adding More Features

The debug stub does very little itself, except stopping and resuming a CPU. However, it solves one of the major problems. It puts the CPU in a known state where the system can be inspected and safely resumed afterwards. Any additional code can then be directly executed in the NMI-TSS context. The following steps show how other debug features can be supported by letting a halted CPU execute a so called *debug plugin*:

1. Save a copy of the NMI-TSS

2. Inject the plugin

3. Modify the NMI-TSS to point the EIP to the plugin and set the input parameters

4. Trigger an NMI and wait for completion

5. Read result registers from NMI-TSS

6. Restore the saved NMI-TSS values

Examples for debug plugins can be found in Figure 3.8. Note that all CPU registers can be made accessible in this way. In fact, all debugging requests, mentioned at the beginning of this section, should be implementable within such simple code.

3.4.2 Debugging a NOVA system

Three things are necessary to use the debug stub within a kernel or hypervisor such as NOVA. First, memory has to be found and allocated in the kernel address space to hold the injected code as well as all the needed data structures. Second, the code as well as the data has to be injected and the necessary pointers have to be set, such that an NMI invokes the stub code. Finally, the OS-specific fix-up code in the debugger has to be written that corrects the CR3 and LDT fields in the TSS.

```
// save debug registers
mov %dr1, %eax
mov %dr2, %ebx
mov %dr3, %ecx
mov %dr4, %edx
mov %dr6, %esi
mov %dr7, %edi
iret
```

```
 // save the FPU state
clts
fnsave (%eax)
iret
// flush the current VMCS to memory
vmptrst (%eax)
vmclear (%eax)
iret
```

Figure 3.8: Three debug plugins that save the debug registers, the FPU (Floating Point Unit) state and flush the VMCS (VM Control Structure) to memory. They can be injected into a target and executed as NMI handler from a CPU halted with the debug stub.

Free Space

The easiest way in the NOVA hypervisor to gain free memory is to reserve it in the linker script. Unfortunately, this requires changes to the source code, which are unlikely to be merged upstream. Another solution would be to remotely allocate memory from one of the kernel allocators. However, this would require to synchronize with running CPUs and to have intimate knowledge of the corresponding data structures. Instead, I observed that there is enough slack space in the layout of the hypervisor address space. The CPU-local memory region for instance leaves currently 788 bytes empty. Similarly, the kernel stack resides on a 4k-page, but only a fraction of it is actually used during execution.

A new TSS can be allocated in any of these areas. Unfortunately, this would also require a new GDT (Global Descriptor Table) because the current one does not have any free entry to point to it. Additionally, the CPU would have to execute an `lgdt` instruction before the new table can be used.

Instead I favored a simpler approach and reused the double-fault TSS to run the NMI handler. The double-fault TSS previously printed only a panic message on the screen when an unrecoverable error occurred. Reusing it for the debug stub has no functional consequence except that it would halt the CPU in these cases instead of printing a message.

I also discovered that the stack pointers in the TSS, which are required for interrupt handling, will never be used in the NMI-TSS because the NMI handler always runs with IRQs disabled. Interestingly, these 24 bytes are enough to hold the instructions of the whole debug stub. It is therefore unnecessary to allocate any memory in the NOVA hypervisor, thanks to the small size of the stub.

Extending VDB

Putting the debug stub in the NMI-TSS avoids not only changes in the source code, but also simplifies the injection procedure. In fact only the two TSS need to be accessible in the debugger to inject and use the debug stub. Both reside in the hypervisor address space in a region that is reserved for CPU-local data. The physical address of this region can be found by using the boot page tables to read the `PD::root` data structure, which points to per-CPU page directories, which include the CPU-local memory. The DWARF support in VDB helps to get the addresses for the symbols and to traverse the pointers as well as the PD data structure. There is no need to change the IDT (Interrupt Descriptor Table), as the NMI entry already points to the double-fault TSS. In summary, injecting the stub code and initializing the NMI-TSS requires less than 20 lines of code in VDB.

The TSS fix-up code for the NOVA hypervisor is quite simple as well, due to two reasons. First, the LDT feature of the CPU is not used in NOVA. Instead, the LDT field is always zero and need not to be corrected. Second, the previous CR3 can be recovered from the `Pd::current` variable. The fix-up code is therefore only a single line in VDB.

3.5 Summary

I showed in this chapter how an operating system like NOVA can be debugged in a TCB-aware way. First of all, I analyzed the requirements and found that only a remote debugger would support most use cases without increasing the TCB. Second, I presented the design and implementation of VDB, a remote debugger that allows symbolic debugging of operating systems and its applications. Third, I described how Firewire can be leveraged to get high-speed access to the memory of a remote machine without involving any runtime code. Finally, I developed a minimal stub that can be injected into a production system to debug it without increasing the TCB beforehand.

Chapter 4

Simplifying the Compiler

Concentrate on the essential, purge the rest.

N. Wirth and J. Gutknecht in *Project Oberon* [WG92]

In the previous chapters, I have shown how the lines of code of different parts of an operating system can be minimized. In the following, I will look at the compilation process that generates binaries from source code and describe how the size of the compiler as well as the build tools can be reduced.

The compiler is usually assumed to be trusted in projects aiming for a small TCB (Trusted Computing Base), even though any defect in it will have a large security impact. Thompson for example explained in his famous Turing Award lecture [Tho84] that a subverted compiler can easily omit security checks in programs it compiles. Security assumptions, which are perfectly valid on the source-code level, may not hold anymore after the program was compiled to machine code. It is therefore strictly necessary to include the compiler in the TCB.

Unfortunately, the size of current build environments can easily surpass the size of the trusted software it produces. For instance building the OSLO bootloader [Kau07b] with a GCC toolchain [GCC] involves approximately 2.4 MSLOC as shown in Figure 4.1. This is three orders of magnitude more code than the 1534 lines of OSLO. Thus, the enormous size of state-of-the-art compilers seems to nullify any advances in TCB reduction. Furthermore, compilers are far from bug free. GCC for instance received more than 3000 bug reports for every major version since 4.0. I therefore studied how compiler, assembler, and linker can be made much smaller.

In the following section, I will introduce B1, a new programming language specially tailored for low-level systems programming. In the second section, I describe the implementation of a corresponding toolchain that compiles B1 source code to x86-32 machine code. This is followed by a performance evaluation of B1 programs.

4.1 Design

Not all C compilers are as large as GCC. LCC [FH91] or more recently TinyCC [Bel02] need only tens of KSLOC to compile a C file to a non-optimized binary. These promising developments are still an order of magnitude larger than OSLO. The most likely reason for this gap is that the C language is to complex to be easily compiled. Thus it seems to be not sufficient to improve the implementation alone to get a toolchain as small as

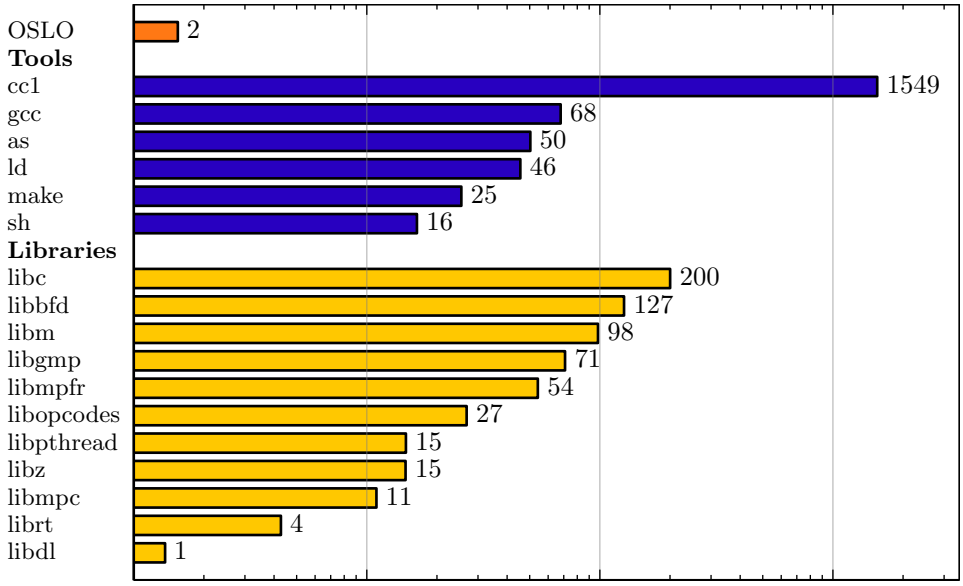

Figure 4.1: More than two million lines of code are needed to compile OSLO with a GCC toolchain. The numbers are estimates from the binary sizes as described in Section A.1.

OSLO.

Instead I propose to simplify the programming language as well because it eases writing the compiler and reduces the TCB of the whole system. Compiler verification [Ler09] will also benefit from a simpler language, as this reduces the effort to specify and prove the semantics of the language. Finally, a language simpler than C increases the likelihood of different toolchain implementations, which is a prerequisite for detecting a subverted compiler [Whe05].

In this section, I will present a new programming language called B1 and discuss the design decisions that have lead to it. Please note that I have not aimed for the simplest possible system programming language. Instead I carefully balanced language features against compiler complexity. I thereby deliberately added *syntactical sugar* if it was easy to implement in the compiler and if it would make B1 programs less complex.

4.1.1 The Syntax

Instead of developing a new syntax for B1 from scratch, I reused the syntax of Python v3.2 [Pyt] and defined a new semantics for it. This allows Python source code to be statically compiled to machine code. A subset of Python is sufficient for this purpose. Yet enough unused syntax elements remain for future language extensions.

Reusing an existing syntax for a new language has the advantage that programmers need not learn another programming language and source code can even be shared between the two languages. Furthermore, the compiler implementation can be simpler as shown in the next section. A disadvantage is that the changed semantics of the very same operations might confuse programmers that use both languages in parallel. For example arithmetic operations in B1 silently overflow similar to C instead of being converted to a longer integer type as in Python.

In the following, I will describe the semantics of different language elements.

Name	Size in Bytes	Signed	Byte-Swapped	Type Code
Mword	2,4,8	no	no	0
Uint8	1	no	-	1
Int8	1	yes	-	–1
Uint16	2	no	no	2
Int16	2	yes	no	–2
Uint16s	2	no	yes	3
Int16s	2	yes	yes	–3
Uint32	4	no	no	4
Int32	4	yes	no	–4
Uint32s	4	no	yes	5
Int32s	4	yes	yes	–5
Uint64	8	no	no	6
Int64	8	yes	no	–6
Uint64s	8	no	yes	7
Int64s	8	yes	yes	–7

Figure 4.2: Type codes for dereferencing pointers and for structure members in B1. Types larger than the `Mword` are not supported by a platform.

4.1.2 Variables and Constants

Variables are `mwords`

Variables in Python are dynamically typed. Consequently there is no syntax for declaring the type of a variable, even though a compiler needs to know this information to choose the corresponding machine code. To still compile B1 code statically, one could annotate variables for instance by reusing the `is` operator or by relying on Python's parameter annotations. Another option would be to let the compiler automatically infer the types, as done for instance in PyPy [RP06].

I have chosen the simplest solution and used the machine word (short: `mword`) as the only data type for variables[1]. An `mword` is an unsigned number large enough to fit into the integer registers on the target machine. It is basically the same as an `unsigned long` when compiling C code with GCC on x86.

If all variables and expressions are `mwords`, type declarations are unnecessary. This simplifies the compiler, as it does not have to parse type definitions, propagate types, and check for type compatibility in assignments.

Accessing Other Types

A B1 program needs to read and write other types besides `mwords` from memory. Examples are characters from a string or 16-bit hardware registers when programing a device driver. To support these use cases, any `mword` value can be used as a pointer. A rich dereference operator `[index:type]` reads and writes shorter integer types from a pointer and converts them to or from an `mword`. All types that fit into an `mword` can be accessed this way, including signed, unsigned, and byte swapped types. See Figure 4.2 for a list of types defined in B1.

To make B1 programs easier to write, I added the following syntactical sugar:

- The type can be omitted in the dereference operator if mwords are accessed.

[1] The language is called B1 because having only a single data type makes it similar to the predecessor of C, namely the programming language B [Tho72].

```
LinkedList = ["next",   Mword,  "prev", Mword]
Union      = [0, ["u8", Uint8,  "u16",  Uint16]]
Element    = ["value",  Union, "list", LinkedList]

def getValue16(element):
    return element.Element.value.u16
```

Figure 4.3: Structures in B1 are defined as lists, which combine a name with a type code. Casts and member access is performed via qualified names.

- The index in the dereference is scaled with the size of the access type to enable fast iteration over arrays.
- Structure members are accessed from a pointer with dotted names. This avoids to specify the right index and type on every dereference.
- Structure definitions are lists that combine a name with a type code. A type code is a small integer such as 1 for `uint8`. See Figure 4.2 for a list of type codes.
- Structures can be nested. If zero is used as name, the definition becomes a union.
- Every structure has the underscore (_) as implicit last member, which can be used together with the address operator to get the size of a structure and to navigate through an array of structures.
- The last occurrence of a name in a structure is used when multiple members share the very same name. This provides member overriding, which eases emulating of C++ style classes.

An example of struct and union definitions in B1 is given in Figure 4.3.

Translating the Constants

Python supports a rich set of constants namely booleans, integers, floating point and complex numbers, strings, bytes, lists, sets, dictionaries, and tuples. B1 defines operations only on a subset of them, to keep the compiler sufficiently simple. However constants of these types are made available to B1 programs. This will significantly ease the implementation of library code that likes to operate on them.

Basic constants such as booleans, integers and floating-point numbers are converted to `mwords`, if they occur in global variables and expressions. These constants can then emitted by the assembler as immediates in the machine code.

Other constants are put into the data section of the binary. The size of the constant is prepended, to make a `sizeof` operator unnecessary. Strings are null terminated and decoded as UTF8 before they are converted into an array of bytes. Variable-sized longs are reduced to the smallest array of bytes they fit into including their sign bit.

Lists of booleans, integers, floating point and complex numbers are converted to arrays of fixed-size integers. This allows to define char and word arrays and even arrays with elements larger than `mwords`. Sets and dictionaries are converted to sorted lists. Nesting of lists, sets and dictionaries is implemented by converting the child and using the reference to it as `mword` element in the parent.

Because tuples are basically the same as lists in Python, except that they are read only, they are freely available for special purposes. I use them in B1 to specify inline

assembler, to allocate additional stack space, and to communicate alignment as well as section names to the linker.

Local vs. Global Variables

Local variables are declared inside a function body by assigning a value to them. They are allocated on the stack and hold `mword` values including pointers. Global variables are declared outside function definitions by initializing them with a constant. They may hold basic constants such as integers or immutable pointers referencing complex constants such as lists or functions. Additionally, the general purpose CPU registers are available as global variables. This eases lowest-level programming for instance when implementing binary interfaces.

4.1.3 Operators, Functions and Control Structures

Operators

Operators are only defined for `mwords` because all variables, constants, and pointer dereferences return this type. The arithmetic (`+,-,*,//,%`) and bitwise (`«,»,&,|,^,~`) operators are the same as in C, except that the double slash is used for integer division. Similarly, the address of local variables and of structure members is returned by the unary plus instead of the ampersand operator. This makes B1 code more similar to Python code and eases code reuse.

Assignments can be used to update local variables (`x="foobar"`) and CPU registers. Global variables cannot be directly modified because they are either basic constants or immutable pointers referencing complex constants. Arbitrary memory locations can be written by assigning a value to a pointer dereference such as `x[3:Uint8] = 0`.

The following syntactical sugar is inherited from Python to shorten B1 programs:

- In-place assignments for any binary arithmetic operator. `x += y` is the same as `x = x+(y)`.
- The three boolean operators: `or`, `and`, `not`. The expression `x or y` is equal to `x ? : y` with GCC, `x and y` is equal to `x ? y : 0`, and `not x` is equal to `!x`.
- The ternary if: `y if x else z` corresponds to `x ? y: z` in C.

Inline Assembler

Assembler instructions can be directly embedded as constant tuples into B1 source code. This provides low-level access to processor features otherwise not available through the core programming language. See Figure 4.4 for an example.

A program may also register new instructions at the assembler. This is useful for instance to add floating-point instructions when implementing a corresponding library. Finally the general processor registers are accessible as global variables in B1.

Functions

Currently, only function definitions and global variables are handled as B1 code by the compiler. Global code and class definitions that are found in a source file are simply ignored.

```
def memcpy(dst, src, count):
   for i in range(count):
       dst[i: Uint8] = src[i: Uint8]
```

```
def memcpy(dst, src, count):
   ("mov", dst,   EDI)
   ("mov", src,   ESI)
   ("mov", count, ECX)
   ("code", 0xf3, 0xa4)
```

Figure 4.4: A simple memcpy implementation in B1 native code (left) and x86 inline assembler (right).

```
def divmod(a, b): return a//b, a%b
def get(which):
     return which and (lambda x,y: (x*y, 0)) or divmod
```

Figure 4.5: Functions in B1 are defined with `def` and `lambda`. They can return multiple values including function pointers.

Functions are defined with the `def` keyword or anonymously with a `lambda` expression. Functions are the main way to structure a B1 program. See Figure 4.5 for an example. Nested function definitions are possible. Closures are not.

Functions can be called with a variable number of arguments, similar to `variadic` functions in C. Arguments are `mwords` and have the same semantics as local variables inside a function. Default values for arguments and Python's keyword-argument feature are not supported, yet.

A function may return up to four `mwords` in registers[2]. Returning more values or even supporting Python's generators would require memory allocation in B1, which I avoided for simplicity.

Control Structures

B1 supports loops for structured programming. There are `while` loops and counting loops of the form `for i in range(...)`. In contrast to Python, it is not possible to directly iterate over arbitrary lists or arrays with the `for` loop because only the size of constant arrays are known at compile time.

The statement `break` can be used to jump behind a loop and `continue` to start the next iteration prematurely. The optional `else` statement of a loop allows to execute code, if the loop was not aborted with a `break`. This will lead to efficient code without requiring the `goto` statement, for example when searching an element in a list [Knu74].

Furthermore, B1 supports the conditional statement (`if`/`elif`/`else`). Compared to C there is no `switch` statement. It can be substituted either by multiple `elif` constructs or by calling through a dispatch table, if a sufficiently continuous range should be distinguished.

Labels and `goto` directives are not directly available in B1. Inline assembler can be used for this purpose if required.

Exceptions can be thrown via `raise` and caught with `try`/`except`/`finally`. This will make error handling in B1 programs much simpler than in C. The `assert` statement is provided as a shortcut. It will raise an AssertionError if some condition does not hold.

[2]The register EAX, EDX, ECX, and EBX hold the return values on x86.

4.1.4 Discussion

B1 is a new programming language powerful enough to write low-level system code with it. I have moved seldom required language features such as floating-point support from the core of the language into external libraries. Furthermore, I reduced the number of syntactic and semantic checks that are performed by the compiler. These steps have led to a toolchain smaller than before. Compared to Hoare's now classical hints on programming-language design [Hoa73], B1 shines on simplicity, readability, and (potentially) fast compilation but lays the security of the program in the hand of the developer. In the following, I will compare B1 to other languages.

Comparison with B and C

The semantics of B1 is very similar to B [Tho72] the predecessor of C. B1 does not have labels, declaration of local variables, and the switch statement. Instead, it includes access to non-`mword` memory and more powerful loops to make the `goto` statement redundant. Furthermore complex constants, inline assembler, and exceptions make programming in B1 much easier than in B.

The most important difference to C is that B1 only defines operations on `mwords`. While this makes most B1 programs shorter and surely simplifies the compiler, it also means the following C features are unavailable:

Type Checks A programmer has to be careful when converting between pointers and values or between different pointer types, as this does not require an explicit cast and no type checks are performed in the compiler. Static code analysis tools should be employed to find bugs in such code.

Signed Operations There is no support in the core language for multiplying, dividing, shifting, or comparing signed integers. If an algorithm needs this functionality, it should be provided by library code. Using assembler instructions and inlining those functions can limit the overhead of this design decision.

Floating-Point Operations Only floating-point constants but no operations on them are supported in B1. This limitation comes from the observation that a lot of kernel and low-level code does not need floating-point support at all.

Instead, floating-point support for B1 programs should be provided by library code with the help of inline assembler. This allows to choose the fastest available instruction set extension on the particular machine such as SSE (Streaming SIMD Extensions) or AVX on x86 without complicating the compiler for all the programs that do not require floating-point code.

There are several less important C features not supported in B1. Most of them can be efficiently emulated on top of it such as bit fields, enums, the `switch` statement, and casting of expressions. Others are seldom used in low-level systems code, but removing them simplifies the compiler. For instance omitting `alloca()` means that a frame pointer is never needed.

However B1 is not a strict subset of C. It supports additional features to make B1 programs easier to write than programs written in plain C. Examples are:

- Complex constants such as dictionaries or variable-sized longs;
- Byte-swapped types for big and little-endian access;

- Types such as `uint8` are constant numbers and structure definitions are ordinary lists available to the B1 program. This can be used for type introspection and to implement dynamic casts.
- Raising and catching exceptions for error handling, including `final` sections to execute cleanup code even if an exception occurred.

Other System Languages

Numerous programming languages were proposed for systems programming, either by introducing a new language, extending an application level language with lower-level primitives, or by changing the semantics of an existing interpreted language so that it can be compiled to machine code. In the following, I will compare B1 to a few of them.

Oberon The most similar approach to B1 is probably the Oberon Project by N. Wirth and J. Gutknecht [WG92], which tried to *reduce the complexity of programming languages* [Wir04]. Oberon was derived from Modula-2 by removing less important features such as unsigned and complex numbers. B1 can be seen as a next step in that direction. In addition to Oberon, I have removed floating-point numbers as well as garbage collection and strongly checked types from the language. Besides low-level features and certain syntactical sugar, B1 adds only exceptions to the language. All removed features can be emulated on top of B1, except for the static type checks, which would require external tool support.

Go An example for a language designed from scratch is `Go` [GO] that tries to combine the ease of programming of a scripting language with the performance of a compiled language like C. Unlike B1 it has a full-featured type system and special support for concurrency. While Go supports exceptions, the authors discourage to use them. A Go toolchain is very similar to a C toolchain both in size and complexity. In fact, the first Go compiler was derived from the Plan9 C compiler suite [Tho90]. Other compiler implementations are just frontends to GCC or LLVM.

House A different approach was taken by the House project [HJLT05], which extended Haskell with low-level primitives. While a functional language like Haskell might ease verification [KEH+09], a system based on Haskel has not only the compiler but also a large runtime of several hundredth KSLOC in its TCB.

Small Java Small Java [Fre09] is a subset of Java that can be fully compiled to machine code. A special class called MAGIC provides low-level hardware access and allows to embed machine code into the binary. The corresponding compiler is self-hosting, but with 38 KSLOC more than an order of magnitude larger than the B1 toolchain.

Python Syntax B1 is not the first language that shares the syntax with Python. The PyPy Project [RP06] for instance defined the RPython language [AACM07] to implement a Python interpreter with just-in-time compilation support. By forbidding certain language constructs such as modifying class definitions during runtime and deducing the type of variables at compile time, RPython can be compiled statically to machine code. Similarly, the Boo [BOO] language has a Python-inspired syntax with static types for the .NET environment. A B1 compiler can be simpler than these two examples because it does not need to perform any type interference.

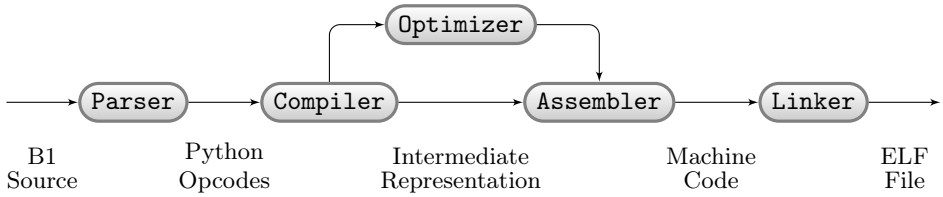

Figure 4.6: Data flow through the B1 toolchain to produce ELF files from B1 source code.

Component	SLOC
Parser	25
Compiler	361
Optimizer	121
Assembler	294
Linker	137
Misc	42
Sum	980

Figure 4.7: The size of the B1 toolchain in lines of Python3 code.

4.2 Implementation

In the previous section, I presented the syntax and semantics of B1. In this section, I discuss the implementation of a B1 toolchain that compiles B1 programs to static x86-32 binaries for Linux, NOVA, and the multiboot environment [MBI10]. It can also be used to generate ELF (Executable and Linkable Format) object files to be linked with code written in other programming languages.

The toolchain consists of five components as depicted in Figure 4.6:

1. The `Parser` reads B1 source code and outputs Python bytecode,
2. The `Compiler` translates them to an intermediate representation (IR),
3. The optional `Optimizer` improves the IR,
4. The `Assembler` produces x86-32 machine code from the IR, and
5. The `Linker` generates ELF files.

The five components were implemented in less than 1000 Lines of Python. This is at least an order of magnitude smaller than TinyCC and 2–3 orders smaller than GCC (See Figure 4.7). In the following, I will describe noteworthy implementation details that lead to this small size.

4.2.1 Implementation Language

While many compilers are implemented in a low-level language like C, I have chosen Python v3.2 instead. Python turned out to be *very suitable for this purpose* [Ern99] because it provides several features that simplify the implementation of a compiler:

- Dictionaries, Lists, and other *high-level data structures* allow lookup, iteration, and data manipulation in a consistent and compact way.
- A lot of small objects are constantly created and destroyed during compilation. *Garbage Collection* takes care of memory management without cluttering the code with `malloc` and `free` calls.
- Several transformation passes can be pipelined through *Generators* without producing large intermediate results.
- Range checks are often unnecessary because overflows raise *Exceptions* that are more easily handled in a central place.

Using Python has also some drawbacks. First, a compiler written in Python might not be as fast as an implementation in a lower-level language. This is only an issue with very large projects. Caching already compiled and optimized code should significantly improve the performance of the toolchain.

Second, implementing the toolchain in Python means a Python interpreter needs to be part of the TCB. The PyMite [PyM] project has shown that such an interpreter can be quite small. Own experiments indicate that approximately 2 KSLOC are required for a simple implementation in B1, which would make the current toolchain self-hosting.

This leaves the problem of bootstrapping the whole process because another B1 compiler is required to build this interpreter, similarly how a C compiler is needed to build TinyCC. Writing a B1 toolchain in B1 itself and manually converting it to machine code is one approach to solve this issue[3].

4.2.2 Let Python Parse

Sharing the syntax with Python allows us to reuse the parser and code generator of Python. This has the following advantages:

First, it significantly reduces the implementation effort, as not only the parsing, but also most of the compiler error handling comes for free. The parser just imports a B1 file as a Python module and converts the Python bytecode of the global functions to Python opcode tuples. This can be done in merely 25 lines of code. Given the experience from Tiny-IDL [Kau06], where a simple IDL parser required only 200 SLOC (source lines of code), I estimate that a standalone B1 parser can be written in less than 1 KSLOC.

Second, the Python interpreter can execute any global code in the source file it loads. Thus Python can be used to generate B1 source code. One can for instance compute a B1 function as a Python string and call `compile()` on it. Compared to a preprocessor like CPP, Python is much more powerful as metalanguage, as one can also freely transform Python bytecode. This can be used for instance to apply optimizations like loop unrolling and constant propagation.

Third, Python's module and package support can be reused to structure B1 projects because import statements are evaluated at compile time.

Finally, global variables can be used to configure the toolchain. This feature is exploited for instance to register new instruction encodings at the assembler. A developer needs just to define a global `B1_ENCODING_` variable.

However reusing Python's bytecode may lead to slower programs, as Python operates on a virtual stack machine, whereas physical CPUs are usually register based. I will show later that the optimizer can recover most of the performance lost in this detour.

[3] As done for instance for WEB by Knuth [Knu84].

Type	Instructions
Arithmetic	`add, sub, mul, div, neg, lea`
Logical	`xor, or, and, not`
Shift	`shl, shr`
Conditional	`cmp, jz, jnz, set{b,be,z,nz,a,ae}`
Control Flow	`jmp, call, ret`
Memory Access	`mov, push, pop, lsubscr, ssubscr`
Special	`data, reloc, local`

Figure 4.8: Intermediate Representation (IR) Instruction as emitted by the B1 compiler.

```
lambda r: r[0][0] == "STORE_GLOBAL" and (1, ("asm", "pop", r[0][1]))
```

Figure 4.9: A compiler transformation rule. It replaces a store to a global variable in Python bytecode with a `POP` to the very same name in IR. This is later translated by the assembler to an x86 `pop` instruction and a relocation entry to address the named variable.

4.2.3 Compiling to an Intermediate Representation

The B1 compiler does not directly generate machine code as opposed to TinyCC. Instead it translates Python opcode tuples to an intermediate representation (IR). Using an IR is required to split the toolchain into compiler, optimizer, and assembler. Furthermore it improves portability, as the compiler and optimizer can be platform independent. Only the assembler needs to be changed to support a new target.

IR instructions are tuples in Python consisting of a *name* and a variable number of *operands*. Many of them share the semantics with their x86 counterpart. This allows the assembler to translate IR instructions easily to x86 machine code. Porting the toolchain to other platforms might require to relax this strong relationship in the future.

The IR is also used to communicate relocation entries (`reloc`) and binary blobs (`data`) to the linker. Figure 4.8 lists all IR instructions emitted by the compiler.

The transformations in the compiler, optimizer, and assembler are performed on a stream of tuples, instead of the usual tree-like data structure. This means compact transformation rules can be used, which are written as a small function or as lambda expression. Such a rule typically checks the current and possibly former instructions for a certain pattern. If the pattern matches it rewrites one or more tuples in the instruction stream. See Figure 4.9 for an example rule.

The compiler uses the following four passes over the input stream to gradually translate Python opcodes into IR instructions:

Pass 0 compile-time constants, unpack tuples before return
Pass 1 constant folding, for loops, inline assembler
Pass 2 structs, nested functions, exceptions
Pass 3 generate IR instructions

Using multiple passes is a simple way to express the precedence of translation rules, even though it may slowdown the compilation process as the IR has to be inspected multiple times.

4.2.4 Optimizing the Compiler Output

The compiler translates Python opcodes directly into corresponding IR code without eliminating any superfluous stack access. Operations just pop their arguments into registers

and push results onto the stack. While this leads to valid intermediate and machine code, it surely costs performance to access the CPU stack unnecessarily. I therefore added a standalone optimizer to improve the intermediate code.

The optimizer is currently based on 38 translation rules with different complexity. There are simple peephole-like optimizations that among others remove arithmetic instructions operating on zeros, replace a multiplication by a power of two with a corresponding shift instruction, or detect a bitwise rotation in a series of shift and bitwise `or` instructions. There are also more complex rules that reduce stack operations by keeping values in registers, replace memory references to read-only data with immediate values, and merge arithmetic instructions. The most effort is needed to optimize function calls. This rule rearranges the stack layout so that the function address is not pushed before the arguments to the stack but made directly available as immediate to the call instruction.

I will show in the evaluation that these rules can compensate most of the overhead caused by translating Python bytecode in a straight forward way. However, the optimizer does not implement many optimizations found in GCC and LLVM such as an efficient register allocator or the elimination of common subexpressions. B1 binaries therefore pay for the simplicity of the toolchain with a certain runtime overhead. I will quantify the effect of this size vs. performance tradeoff in the evaluation section (§4.3.1).

4.2.5 Generating Machine Code

The assembler produces machine code in three steps. First, it calculates the stack depth for every instruction. This is needed to reference local variables relative to the stack pointer without requiring a frame pointer. This step also eliminates any unreachable code that was created by the optimizer for instance by replacing always-taken conditional jumps with unconditional ones. Second, the assembler translates IR tuples to x86-32 instructions. This is done twice so that shorter encodings can be chosen for near jumps. Third, relocations to local labels are resolved and nested data structures are flattened recursively.

The assembler supports 14 different instruction formats such as binary instructions with a Mod/RM encoding[4]. It uses only 78 encodings of 31 different instructions to translate intermediate to machine code. This is more than an order of magnitude less than the 1475 instructions that the `binutils-2.22` know for x86. Interestingly, this resembles an old observation from [AW75] that has lead to the development of RISC processors [PD80].

There are several reasons why this small set of instructions is sufficient for B1. Most notably omitting floating-point operations in the core language removes all floating point and SSE instructions. Furthermore, operating only on machine words means byte, word, and signed encodings are unnecessary. Finally, restricting the instruction set to the ones required by the B1 compiler allows to omit all system instructions such as `cli` or `sysenter`.

The assembler can be extended from B1 source code by defining new instruction encodings through `B1_ENCODING_` variables. Thus, floating-point libraries and system code can be written in B1, even though certain instructions are not directly available in the core language.

[4] See Section 2.4 for the x86 instruction format.

4.2.6 Linking the Binary

The linker takes the machine code as well as the data produced by the assembler and writes it into an ELF object file [ELF95]. This file can be linked to code written in other programming languages. The linker can also generate a static ELF binary by linking the assembler output to some address and resolving all relocations.

Two design choices make the B1 linker smaller than other linkers. First, it supports only ELF as output format. This means a large compatibility layer such as the `libbfd` is not required. If other executable formats such as Windows PE (Portable Executable) are needed for instance when generating UEFI (Unified Extensible Firmware Interface) programs, they should be converted from the ELF files with the help of external tools. Second, the B1 linker does not support linker scripts. Due to its size, it is easier to modify the source code instead of adding another special-purpose language to make it more configurable. Please note that the linker is largely target independent. So it is unlikely to grow if additional platforms are supported.

4.2.7 Implementing the Standard Library

The B1 toolchain produces standalone binaries. Thus it cannot link against external libraries such as the `libc`. Instead B1 programs use their own standard library.

Implementing `libc`-like functionality in B1 such as `stdio`, Linux system calls, or the string functions (`memcpy`, `memset`, `strlen`, ...) was a straight forward task. Python's package feature allows to nicely structure the code and separate generic from x86 specific code.

The B1 compiler does not depend on a standard library per se. In contrast to GCC, it will never emit calls to string routines or to 64-bit arithmetic functions. The B1 compiler still relies on library support if a program needs variadic functions or exception handling. In the following, I will describe how I implemented these two features in the standard library.

Variadic Functions

Variadic functions such as `printf` use the first argument to know how many parameters will follow. Unfortunately, function parameters are evaluated from left to right in Python, which is the reverse of the C calling convention. This means the first argument in B1 is stored above all other parameters on the stack and can only be accessed by the callee if the overall number of arguments is known beforehand.

There are various solutions to this issue: One could either force the programmer to manually switch the parameters of variadic function calls, or one could reverse the argument order at the bytecode level. Both solutions unnecessarily complicate either B1 programs or the compiler. Another option would be to put the number of transferred parameters in a register. However, this would slow down every function call for a case that is seldom required.

Instead I observed that it is sufficient to check the instruction at the address where the function will return to. If arguments are transferred, this instruction will be an `ADD` of the stack pointer to let the caller drop the previously pushed parameters. The immediate part of the instruction reveals the number of `mwords` previously pushed to the stack. This is equal to the arguments of the function for direct calls. It will be off by one for indirect calls due to the additional function pointer. To distinguish these two cases, I let the

optimizer replace the `ADD` instruction with a `SUB` of the negative value when it generates direct calls.

Instruction-based detection works as long as the optimizer does not merge or move the cleanup instruction, which can easily be assured. Furthermore, it can be implemented efficiently because only two instruction encodings occur in actual applications[5]. Finally, the technique makes variadic functions robust against stack underflows, as it ensures that only parameters actually pushed by the caller are accessed by the called function.

Exceptions

Exceptions should be a lightweight mechanism and not cause any runtime overhead if they are not taken. This excludes a direct translation of Python's block stack where exception frames are linked during runtime. It also disqualifies the `setjmp/longjmp` approach used in some C++ exception implementations [HMP97].

Instead, the B1 compiler emits a call to the `raise_exception` function, whenever a `raise` or `assert` statement occurs. This function needs to unwind the stack and jump to the right exception handler. Consequently there is no runtime overhead of a `try/except` block if no exception occurred apart from an unconditional jump over the exception handler code.

The `raise_exception` function needs to know the stack depth at every function call to unwind the stack. But if asynchronous events such as UNIX signals or CPU interrupts should be converted to exceptions as well, the stack depth has to be available for every instruction. Please note that the assembler does not use a frame pointer register to address the top of the current stack frame. Another way has therefore to be found to recover the stack depth.

One option would be to recover the stack depth from the instruction stream by scanning the next instructions until a function return is reached. However the complexity of the x86 instruction encodings makes this option quite slow. Furthermore, the possibility to extend the assembler with previously unknown encodings means this approach is likely to fail. On a RISC processor like ARM, one would surely choose this solution as it does not require any additional memory.

Instead, the B1 assembler stores the calculated stack depths in the special `.b1info` section in the binary. This section includes markers to identify function boundaries as well as the beginning and the end of `try` blocks. The `.b1info` section is compressed with a run-length encoding to reduce its size. Additional function information and the function names are recorded in the `.b1func` and `.b1name` section respectively. The three sections are typically one third of the x86 code in the ELF file. They are not only used to handle exceptions, but also for backtraces, profiling, and to ease debugging. I will evaluate the performance of B1 exceptions in Section 4.3.2.

4.2.8 Summary

In this section, I described the implementation of the B1 toolchain and presented a couple of reasons that lead to its small size, namely implementing it in Python, reusing Python's bytecode, working on an intermediate representation, splitting the optimizer from the compiler, emitting only a small number of instructions and supporting a single output format.

[5] These are `add` and `sub` with a byte immediate on x86.

```
def fib(x):
    if x < 2: return 1
    return fib(x-2) + fib(x-1)

from lib.linux import *
main = lambda: fib(40)
```

```
int fib(int x) {
    if (x < 2) return 1;
    return fib(x-2) + fib(x-1);
}

int main() {  return fib(40); }
```

Figure 4.10: A recursive implementation of Fibonacci numbers in B1 (left) and in C (right).

I have already used the B1 toolchain to implement a couple of Linux tools used in the Evaluation, the bootlets described in Section 5.2 and other low-level programs. Nevertheless, it waits to be employed in larger projects and ported to other architectures. In the next section, I evaluate the performance of programs written in B1.

4.3 Evaluation

I will start the performance evaluation of B1 applications with two microbenchmarks to reveal the influence of the optimizer and to quantify the cost of exceptions. Furthermore, I have rewritten four Linux tools in B1 that are typical instances of I/O bound (`dd`) and CPU bound (`wc`, `gunzip`, `sha1sum`) workloads. While these programs will not have all the features of their Linux counterpart like working with multiple input files, they should show the current performance of B1 and reveal where further work is necessary.

All measurements were done with Linux 3.6 on a Lenovo T420 equipped with an Intel Core-i5 2520M at 3.2 Ghz. Comparison is done against GCC v4.7.1, TinyCC v0.9.25, Python 3.2.3, Java from OpenJDK-7-u3, GNU core utilities v8.13 and BusyBox v1.2.

4.3.1 The Influence of the Optimizer

With the first microbenchmark, I will evaluate the influence of compiler optimizations on the performance of B1 programs. I have chosen a recursive implementation of Fibonacci numbers as benchmark because the recursion will amplify the costs of any superfluous instruction not removed by the optimizer.

The source code of the benchmark application for both B1 and C are given in Figure 4.10. The versions are very similar, except that B1 does not need to define types and that it uses indentation instead of curly brackets. Furthermore it can use a lambda expression to define `main()` in a single source-code line. Finally the entry and exit functions of the B1 program are not defined by a linker script as in C but imported from a library.

Figure 4.11 shows the time needed to run the benchmark with different compilers and compiler options. I have compiled the C program with GCC for the first eight bars. One would expect that enabling more optimizations would always result in faster code. However the first two bars show that using no optimizations at all (`gcc O0`) produces faster code then optimizing for code size (`Os`). The binary is even larger when optimizing for code size. This unexpected result can be explained by the fact that `Os` relies on a frame pointer to access local variables whereas `O0` does not use one. Optimizing for size but disabling the frame pointer gives the expected result as shown in the third bar (`Os -FP`).

The first five bars are measured without tail-call optimization (TCO)[6] because B1 does not implement this particular optimization yet. The next three bars have TCO enabled.

[6]By using the option `-fno-optimize-sibling-calls` on the command line.

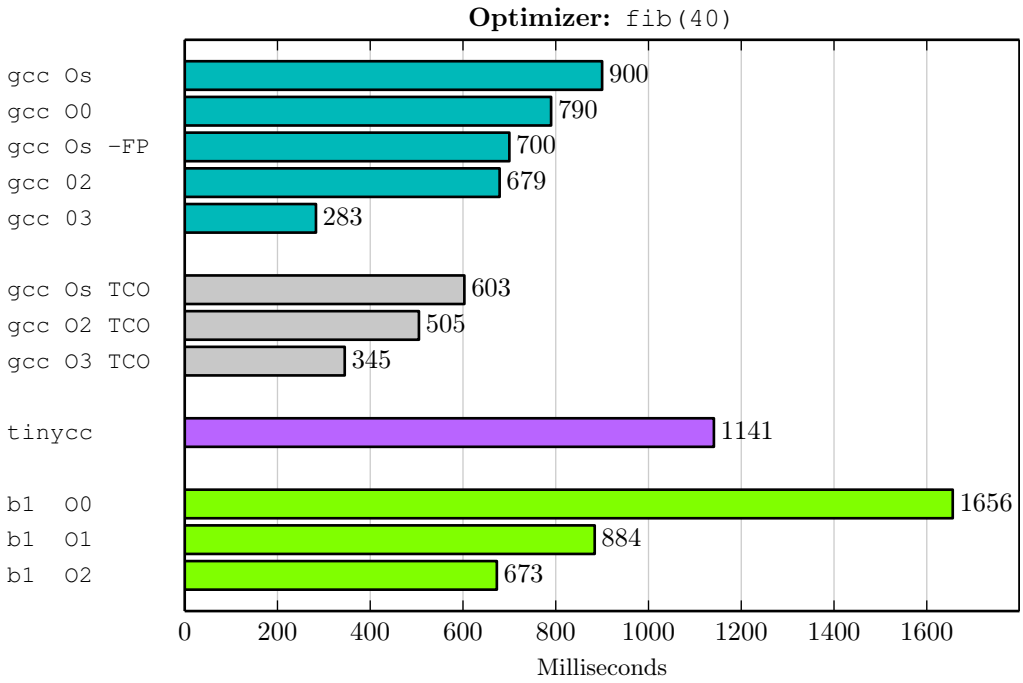

Figure 4.11: The time to recursively calculate fib(40) written in B1 and C with different levels of optimizations.

TCO optimizes a call at the end of a function by reusing the stack frame and replacing the call instruction with a faster jump. Interestingly the TCO versions are always faster, except in the `O3` case where GCC additionally unrolls the recursive loop. This shows that some optimizations are not compatible to each other and just using all of them may not lead to the best performance.

The next bar reveals that TinyCC produces a binary that is always slower than GCC. This can be explained by the use of a frame pointer and another temporary register that is saved and restored on every `fib()` call.

The last three bars give the performance of `fib` written in B1. The `B1 O0` case is the slowest of all measured versions. This can be attributed to the direct translation of Python's stack-machine based bytecode to x86 instructions without performing any optimizations. For the `O1` and `O2` bars the optimizer was used once or twice, respectively. The first optimizer pass gives already a twofold speedup, by removing a large number of redundant stack operations such as pushing intermediate results to the stack and popping them directly afterwards. Running the optimizer twice, additionally merges constants and omits redundant comparisons. This improves the performance by another 30%. Running the optimizer more then twice has no effect here, but still improves more complex code. The speed of B1 function calls is comparable to `gcc-O2` with disabled tail-call optimization.

In summary, the optimizer can compensate the overhead caused by translating Python bytecode in a straight forward way. Nevertheless, recursive functions in B1 are not as fast as in GCC yet because the optimizer misses loop unrolling and tail-call optimization.

```
def test(frames):
    if frames <=2:
        raise 0
    test(frames-1)
def main():
    for i in range(10**6):
        try:
            if STACK_FRAMES > 1:
                test(STACK_FRAMES)
            else:
                raise 0
        except:
            pass
```

Figure 4.12: Benchmark program to evaluate the cost of exceptions in B1. The number of STACK_FRAMES is varied between 1 and 10.

4.3.2 Exception Handling

With the second microbenchmark I will quantify the cost of raising and catching exceptions in B1, Python, Java, and C++. Because the performance of exceptions depends on the depth of the call stack that has to be unwind by the exception handling machinery, I use a recursive function that calls itself a number of times before it raises an exception. See Figure 4.12 for the B1 source code of the benchmark. All benchmark implementations are quite similar except for Java where a statically allocated Exception object is used. The results are presented in Figure 4.13.

I have implemented three ways to handle exceptions in B1. The simplest implementation labeled *B1 linear* in the figure searches linearly in the `.b1info` section for the stack depth of a given instruction pointer. Because the function information contains an offset to the stack information, unrelated functions can be skipped in logarithmic time. The complexity of this approach is linear with the size of the function. It does not need additional memory during runtime.

A faster implementation calculates two lookup tables that combine the address with the stack depth and exception block number. Binary search can now be employed. The time complexity of this approach is logarithmic with the code size of the binary. It needs three times the memory of the linear version or approximately one byte for any x86 instruction byte. The line called *B1 binary* shows that the binary search speeds up exception handling by 50–100%.

The fastest method uses three tables with one entry per instruction byte. These tables allow to lookup stack depth, function number, and exception-block number in constant time. The line labeled *B1 constant* in the Figure 4.13 shows that this improves exception handling by an order of magnitude. Exception handling is now faster than an uncached memory access. However this approach needs seven times more memory then the *binary* version when using 16-bit per entry. Please note that exception handling in B1 is not tied to the compiler but implemented in the standard library. An application developer can freely choose, which of the three versions fits his needs.

Handling a C++ exception within code compiled with `g++` from GCC requires several thousand CPU cycles. It is even slower if non-call exceptions are enabled[7]. Exceptions

[7]Normally only call-instructions may raise exceptions. The `-fnon-call-exceptions` compiler flag allows all trapping instructions to raise exceptions as well. This makes g++ more similar to B1, which

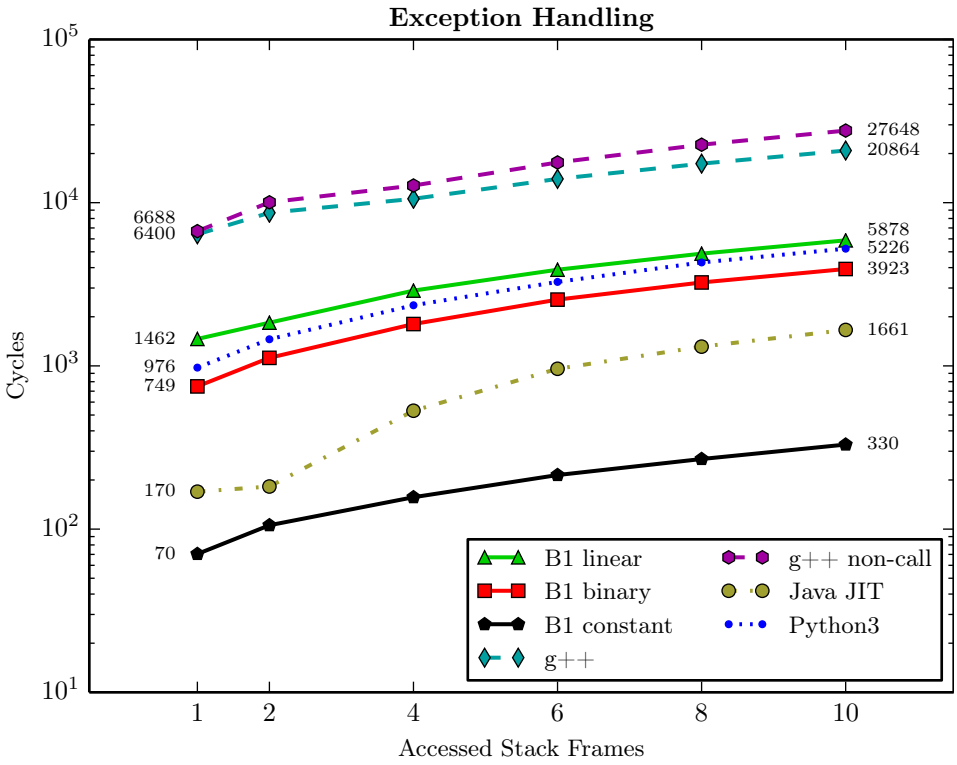

Figure 4.13: CPU cycles to handle a single exception in C++, Python, Java, and B1.

in B1 are between three times and two orders of magnitude faster. There are various reasons for the high cost of C++ exceptions: The encoding of the stack unwind information might have the highest impact. The GCC runtime uses DWARF bytecode for this purpose [Eag07]. Decoding this format requires a Turing-complete interpreter, which is much slower than recovering the stack depth from the run-length encoded `.b1info` section in B1. Another source of overhead is the *personalities* function, which is called on every stack frame to let exceptions propagate through code written in different programming languages.

Exceptions in Java can be two to three times faster then in the *B1 binary* version if just-in-time compilation (JIT) is used as shown in the *java jit* line. If a JIT is unavailable this advantage is lost and exceptions take as long as in the *B1 binary* case. This indicates that Java uses a logarithmic search as well.

Exceptions in Python are even slower than in Java because Python creates a traceback object on every exception. This was explicitly avoided in the Java benchmark by throwing an existing object.

In summary, exceptions in B1 can be faster than in C++, Python, or even Java because B1 trades CPU cycles against memory and neither deletes stack variables, creates a traceback object, nor works across language boundaries in an application. B1 exceptions are basically a fast way to divert the control flow.

4.3.3 System Calls: `dd`

The first macrobenchmark looks at applications that are I/O bound and spend most of their CPU time outside the application code. The `dd` tool is a good representative for this class of programs, if it is used to copy bytes from one file to another without doing any data conversion.

Figure 4.14 shows the throughput of the `dd` tool written in B1 relative to a 32-bit C version from the GNU core utilities when copying data from `/dev/zero` to `/dev/null` with increasing block sizes and different kernel-entry methods.

One would expect that the performance of `dd` running on Linux does not dependent on the programming language or the compiler, as very little calculation is done inside the application. However a `dd` written in B1 has copied data in small blocks twice as fast as a 32-bit C version from the GNU core utilities. It turned out that the B1 version entered the kernel with the fast `vsyscall` method, whereas the C version relies on the particular libc implementation, which used the legacy `int 0x80` way to enter the kernel. However if both binaries are using the same kernel-entry method, the performance difference is below the measurement accuracy.

In summary, I/O bound applications that will not spend much time in user level, should be as fast in B1 as in C.

4.3.4 Simple Calculation: `wc`

The second macrobenchmark compares a `wc` implementation in B1, with the C version from the GNU core utilities. `wc` is a small program to count the number of bytes, words, and lines in a file. It can also measure the length of the longest line. In addition to the previous benchmark this adds buffer scanning and simple calculations to the workload.

I use multiple input files for the benchmark because the performance of word-count depend on its input, as one can for instance tweak an implementation for consisting of

supports exceptions from all instructions.

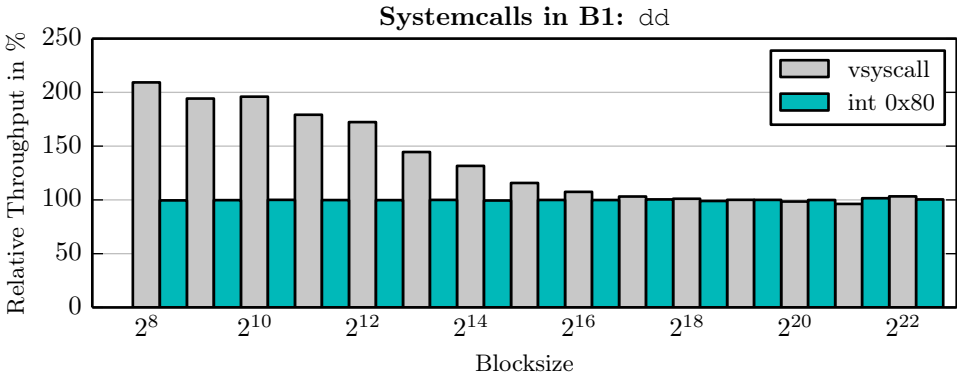

Figure 4.14: Throughput of the `dd` tool written in B1 relative to a 32-bit C version from the GNU core utilities using `int 0x80`. The B1 tool is using either the `vsyscall` or the `int 0x80` method to enter the Linux kernel.

zero characters. Figure 4.15 lists the properties of the four input files. Furthermore, I measure different operation modes, as programmers will typically provide optimized code for the following four special cases with increasing complexity:

Bytes Counting the number of bytes without touching the data.

Lines Additionally counting the number of lines by finding the newline characters in the input data.

Words Additionally counting words by distinguishing spaces from printable characters.

Maximum Line Additionally counting the printable characters in each line.

Figure 4.16 presents the results of the different runs, normalized to the throughput of the C version.

The first set of bars exposes no measurable difference between C and B1 when counting bytes alone. This resembles the results from the `dd` benchmark because only little calculation is necessary here and both implementations use the same system calls to either seek to the end of the file or to read blocks from a pipe.

The second set of bars show that B1 needs up to three times longer to search for newlines. This large overhead can be attributed to the implementation of the `memchr(3)` function, which scans a block of memory for a certain character. While the B1 version checks four bytes at a time, the libc implementation relies on SSE2 instructions to check

Name	Words	Lines	Average Line	Max Line	Description
Zero	0	0	0	0	200 MB from /dev/zero
Random	820829	4623664	255.5	498	generated from /dev/urandom
Long	8119625	33740753	25.8	318	pdftotext from a manual
Short	22153081	22153082	9.5	23	dictionary, one word on a line

Figure 4.15: Four different files of 200 MB were used for evaluating word-count performance.

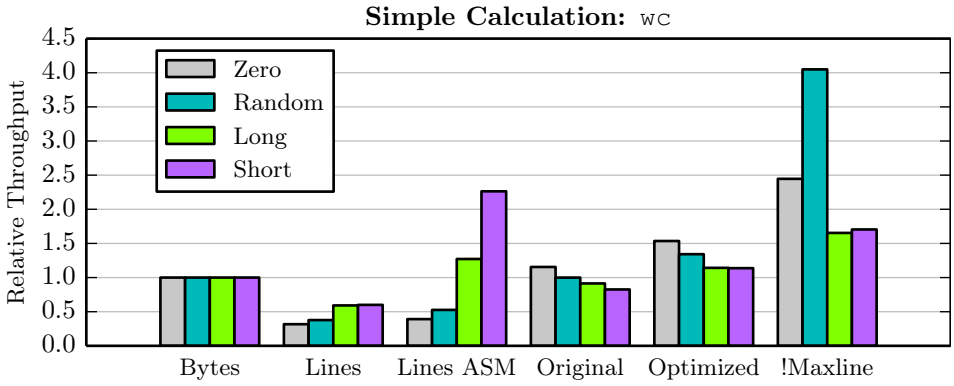

Figure 4.16: Throughput of the word-count (`wc`) tool written in B1 relative to the C version from the GNU core utilities for different input files.

16 bytes in parallel. This overhead is not a limitation of the B1 compiler but caused by the design decision of the library implementer to not use SIMD instructions.

To still get acceptable performance I have implemented a line counting algorithm for B1 in x86 inline assembler without using any SSE instructions. The third set of bars shows that this implementation is 2.3 times faster if there are enough newlines in the input but 2.5 times slower on zero files. On average it is 10% faster then the SIMD version. This disparity is caused by the use of a different algorithm. Implementations based on `memchr` are optimized for the absence of newlines, whereas my assembler implementation works equally fast on all inputs. This was achieved by omitting all branches in the inner loop.

For the fourth set of bars I reimplemented the original algorithm from the GNU core utilities in B1. Because B1 neither provides a switch nor a goto statement, I replaced them with corresponding `if/elif` constructs. The slightly different results for the four input files reveal that I did not exactly mimic the order of conditional jumps. It is therefore 17% and 9% slower on short and long texts respectively. However it is also 15% faster on zero files and shows comparable performance on random input. On average the B1 version is 4% slower then the C implementation.

A deeper analysis of the GNU algorithm revealed that it is far from ideal and severely limited by the number of non-predictable branches. I therefore optimized the implementation by changing the branch order and using arithmetic instructions instead of branches if possible. The fifth set of bars (*Optimized*) shows that such an optimization improves the throughput by 13% to 53% with an average of 27% compared to the original C implementation.

Finally, I observed that the version from the GNU core utilities does not implement specialized code when words have to be counted but the line length is not requested. However this particular case can be more efficiently implemented than the general one because tracking the line length and looking for instance for TAB characters becomes unnecessary. Furthermore, it is the default when invoking `wc` without parameters. The last set of bars (*!Maxline*) shows that such a specialized implementation will lead to an average speedup of 2.4x.

In summary, this benchmark has shown that simple calculations in B1 can be nearly as fast as in C. Furthermore, optimizing the algorithm and carefully using inline assembler will lead to much larger improvements than what was previously lost due to the simpler compiler.

	High	Middle	Low	Small-Files
Number of Files	1	1	1	11690
Uncompressed Size in MB	145.9	43.5	54.2	442.2
Compressed Size in MB	25.6	11.1	54.2	117.6
Compression Ratio	5.7	3.9	1.0	3.8

Figure 4.17: Files with low, middle, and high compression ratio and a large set of small files were used as input for the gunzip benchmark.

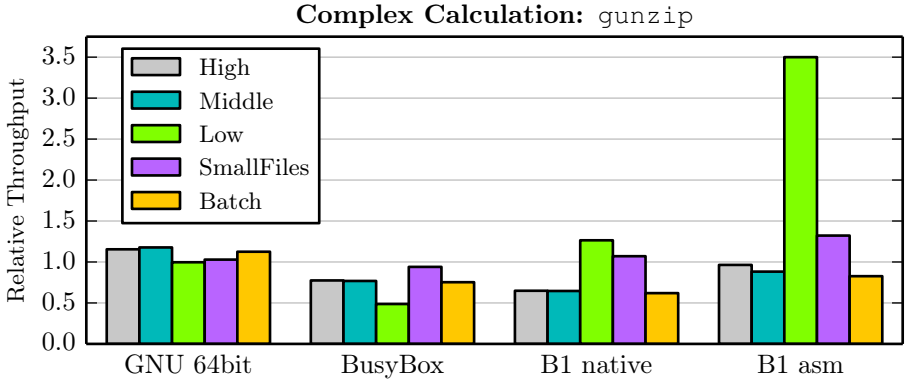

Figure 4.18: Throughput of `gunzip` implementations relative to the 32-bit GNU version.

4.3.5 Complex Calculation: `gunzip`

The third macrobenchmark compares a `gunzip` implementation in B1 with two C versions from GNU gzip and BusyBox. `gunzip` unpacks a compressed file. In contrast to the previous benchmark the gzip algorithm [Deu96] is quite complex and a mixture of bit fiddling, table lookups, and CPU intensive calculation.

I evaluate four different inputs because an implementation can be tweaked for special cases. There are three files with low, middle, and high compression ratio that will exercise different parts of the implementation. Furthermore there is a set of more than ten-thousand small files found in a Linux installation that have a compression factor similar to the middle case. This dataset should reveal the impact of initialization code to the performance. See Figure 4.17 for the input properties.

The results of the benchmark runs are given in Figure 4.18 relative to 32-bit GNU gzip, with bars for each of the four input datasets. The fifth bar in each set labeled *Batch* shows the throughput when the small files are given in batches to the application instead of creating a new process for each input file.

I measured a 32-bit and a 64-bit version of GNU gzip v.1.5. The former is used as the baseline in the figure because B1 uses 32-bit as well. The later shows the benefit of an increased register set. Having 15 instead of 7 general purpose registers available increases the throughput by approximately 10%. Especially the performance of compute intensive inputs improves. Whereas 64-bit does not help in the *Low* compression case because this uses only simple code paths. Similarly the results for *SmallFiles* get only marginally better. This case seems to be dominated by the process startup time.

To put the GNU results in relation to another C implementation, I also evaluated gunzip from BusyBox v1.2. While this code was derived from GNU gzip and shares most of its structure, it was later tweaked for embedded systems. This process made the

BusyBox version significantly slower. I measured only 50% throughput on low-compressed files and 75% in the other cases.

For the *B1 native* bars all code is written in B1 except in two cases where inline assembler is required to shuffle registers for system calls and exception handling. B1 achieves 65% of the baseline with the *High* and *Middle* input files. It is 25% faster in the *Low* case. This can be explained by the use of `memcpy` when handling uncompressed blocks instead of doing a byte-wise data transfer as other implementations. Furthermore, gunzip in B1 is approximately 7% faster when operating on many small files, but 60% slower when they are given to it in a large batch. This shows that the small program size allows B1 gunzip to start much faster than the C implementations. The native B1 implementation gets 85% throughput on average compared to the GNU gzip version written in C.

One can see the limits of the simple optimizer here: For instance inlining functions should significantly improve the performance. Furthermore using more than three registers to evaluate complex expressions should give a speedup similar to the 64-bit case. Nevertheless, gunzip in B1 is on average faster than the BusyBox version. This shows that a better compiler cannot fully compensate for unoptimized source code.

To improve the performance of the B1 implementation, I implemented three functions in inline assembler. Using assembler for `memcpy` and `memset` is a fairly standard procedure. However I have also used assembler to more efficiently calculate the CRC32 checksum. This optimization was not employed in any of the C implementations yet. Altogether these three functions are approximately 70 lines or less than 10% of the whole `gunzip` code.

Using inline assembler gives a performance boost of more than 20% as shown in the *B1 asm* bars. The *Low* case is three times faster then before. Most of this improvement can be attributed to the CRC function. The importance of the checksum to the overall gunzip performance has been underestimated in other implementations. The *High* compression case is now nearly as fast as the baseline. The 83% for the *Batch* and 88% for the *Middle* case leave room for further improvements. The B1 assembler implementation is 50% faster on average than the baseline due to the optimized checksum implementation and its shorter startup time.

In summary, a B1 application with complex calculations is expected to be slower than an equally optimized C program. Especially tiny loops and large expressions show the limitations of the simple compiler. However, optimizing the source code and writing critical parts of the software in inline assembler can make a B1 implementation significantly faster than one solely written in C.

4.3.6 Number Crunching: `sha1sum`

The previous benchmark was based on a complex instruction mix. I will now look at a "number crunching" scenario, where raw CPU performance counts and any additional instruction in the so called "inner loop" will have a significant performance impact.

I used different implementations of `sha1sum`, a small tool that calculates the SHA-1 hash of its input files. SHA-1 [SHA95] is a popular hash function that relies on 32-bit unsigned arithmetic to produce a 160-bit hash from arbitrary sized input. This workload should be representative for many number crunching algorithms that only depend on the speed of the CPU without relying on the memory or cache subsystem.

Figure 4.19 shows the result of the different benchmark runs. A native B1 program needs 74.7 cycles per byte to calculate the SHA-1 hash. This is the slowest of the measured implementations. The very similar C program compiled with TinyCC is nearly twice as

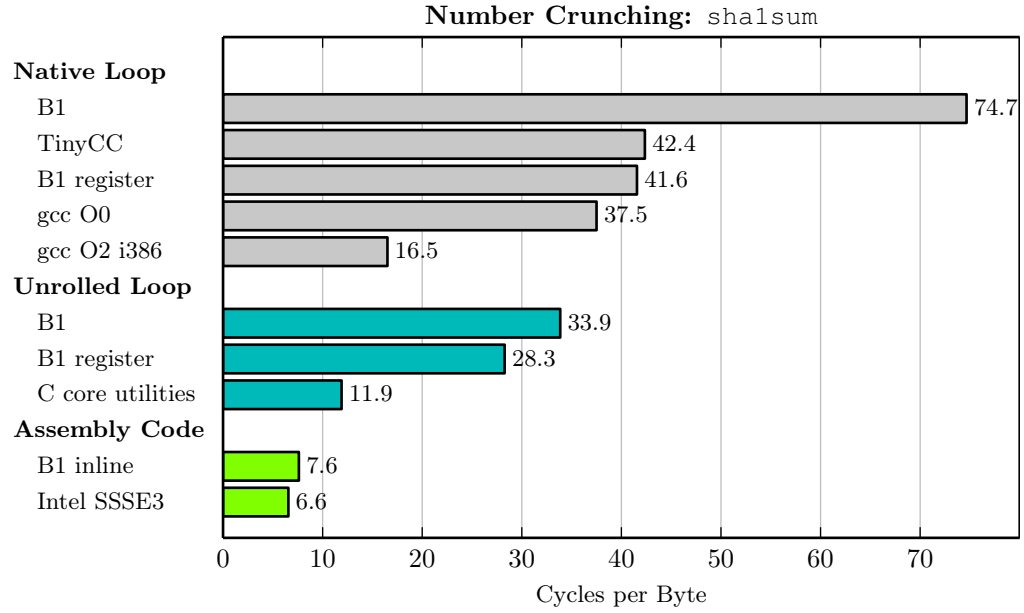

Figure 4.19: The performance of the SHA-1 hash function written in B1, C, and hand-optimized assembler code.

fast as shown in the second bar. The most likely reason is that the B1 compiler uses only three of the seven general purpose registers to evaluate expressions.

The third bar shows that sha1sum in B1 will need 41.6 cycles per byte, if some of the local variables are put manually in registers[8]. This is slightly faster then C code compiled with TinyCC and approximately 90% of the performance of GCC without optimizations (`gcc -O0`). Enabling optimizations for i386 in GCC produces a binary that needs only 16.5 cycles per bytes as shown in the fifth bar. This is more than twice as fast as the unoptimized version. Interestingly, optimizing instead for newer CPUs such as the `i586` tends to be 10% slower[9].

To help the compiler to generate faster code, I also measured the throughput of `sha1sum` with the 80 rounds of SHA-1 fully unrolled in the sources. This not only saves the loop overhead, but also removes branches and instructions to swap variables. The sixth and the seventh bar in Figure 4.19 show a 2.2x performance boost for the native B1 implementation and a 1.5 times improvement for the B1 register version. To compare against C code, I used the `sha1sum` implementation from the GNU core utilities. This version needs only 11.9 cycles per byte to calculate the SHA-1 hash. This is approximately 40% faster than a C version without unrolling. The fastest C versions are on average 2.4x faster than the corresponding B1 register versions.

To set these results in relation to the speed of the processor, I also measured two SHA-1 implementations in assembly code as shown in the last two bars of the figure. The first version consists of B1 inline assembler code and relies solely on 32-bit arithmetic instructions. With only 7.6 cycles per byte it is 2.2x faster than the native C version and

[8]This is done by naming the variable after the register and defining it as a global variable. The effect is similar to the register keyword in C.

[9]The compiler emits a `rol` instruction, even though the `shld` instruction is faster in rotating registers on the SandyBridge CPU.

1.6x faster than SHA-1 in C with the loop unrolled. The second assembler version was implemented by Intel [Loc10] and uses 64-bit registers as well as SSSE3 instructions to achieve another 15% improvement. This is likely the fastest implementation on the Core-i5 CPU. However none of the compilers can generate such efficient code from a high-level description yet.

In summary, number crunching code compiled with the B1 compiler should be as fast as C code compiled with TinyCC, if registers are manually allocated. It will be approximately two times slower than C code optimized with GCC. Even though a production-quality C compiler such as GCC produces faster binaries than B1 today, number crunching is still significantly slower in C than with hand-tuned assembler code.

4.3.7 Summary

A small experimental system such as the B1 compiler will not be as efficient as a production compiler like GCC that was tweaked for more than 20 years. B1's overheads stem from the simple optimizer, which lacks various optimizations commonly found in a compiler. For example, it does not support tail-call recursion, loop unrolling, function inlining nor uses it all general purpose registers to evaluate expressions. Nevertheless, B1 programs tend to be at least as fast as C programs compiled with TinyCC. Cases that are I/O bound or that need only lightweight calculation might even be as fast as GCC, but number crunching code is significantly slower. Finally, exceptions in B1 can be exceptionally fast by trading memory against CPU cycles.

4.4 Conclusions and Future Work

Systems aiming for a small TCB need compilers that are significantly smaller than what is common today. In this chapter I used a holistic approach to show how such compilers can be build. I not only looked at the implementation but also at the semantics of the programming language. To simplify the compiler I defined a new programming language called B1, which is simpler than C, but still powerful enough for systems development.

To show that the B1 language can be easily compiled I developed a corresponding toolchain from scratch. Instead of following the usual approach and writing a new frontend for an existing compiler infrastructure such as GCC or LLVM, I have used the high-level language Python to ease the toolchain implementation. This has not only led to a working compiler within two man months, it also enabled different design tradeoffs in the toolchain implementation. This made for example exceptions in B1 extremely fast. Most importantly it resulted in a working toolchain within 1000 lines of code. This is more than four times smaller than LCC [FH91], TinyCC [Bel02], or OBC [Spi], even when including another 3000 lines for a parser and Python interpreter for a self-hosting version. In summary, using Python as much as possible proved to be an excellent choice. It might be useful to implement other specialized compilers in Python as well.

I also evaluated the performance of programs written in B1. Application code produced by the B1 toolchain tends to be as slow as a less-optimized C version, or as C code compiled with a non-optimizing C compiler. While B1 is not ready for number-crunching scenarios yet, it is already fast enough for many use cases.

Nevertheless, there is still enough work to be done before B1 is ready for production use:

- Writing B1 programs could be made easier by supporting more features from Python

such as classes, optional and named parameters, or the `with` statement[10].

- Porting the toolchain to other targets like ARM or x86-64 should stabilize the intermediate representation and prove that only very little platform specific code has to be written.
- A type interference tool should be developed that can distinguish values from different types of pointers to detect bugs in B1 programs automatically.
- Generating DWARF debug information would ease bug hunting.
- Caching intermediate results would speedup compilation by not translating the whole code all the time.
- More optimizations could narrow the performance gap to C.
- Bootstrapping a B1 compiler from scratch would remove the dependency to Python.

[10] In the meantime classes were added to B1 by utilizing the `metaclass` feature of Python. This required approximately 50 SLOC in the toolchain and another 100 SLOC in the standard library.

Chapter 5

Shrinking the Boot Stack

Security's worst enemy is complexity.

N. Ferguson and B. Schneier in [FS03]

In the previous chapters I described how an OS (operating system) with a small TCB (Trusted Computing Base) can be implemented, debugged, and compiled. In this chapter, I show how it can be booted without inflating its TCB with a large boot stack.

Booting an operating system on a x86 machine involves a surprisingly large software stack. The code needed for booting (i.e. firmware, bootloader, and ACPI interpreter) sums up to at least 120 KSLOC (§*A*.4). Unfortunately, this huge code base is part of the TCB of any operating system because it fully controls the platform and runs before and within SMM (System Management Mode) even below the OS.

In this chapter, I present three approaches to make the x86 boot stack significantly smaller, thus reducing the TCB of an OS. The first part (Section 5.2) focuses on the bootloader. I show that a decomposed design and several implementation improvements can decrease its size by more than an order of magnitude. A few thousand lines of code are now sufficient to boot an OS. In Section 5.3 I analyze whether trusted computing techniques will lead to a smaller and more secure boot stack. I will show that a secure loader like OSLO can remove around 35 KSLOC from the TCB. Finally, I describe in Section 5.4 how the TCB impact of the ACPI (Advanced Configuration and Power Interface) interpreter can be minimized. Using a heuristic can reduce the code by more than two orders of magnitude down to a few hundred SLOC (source lines of code).

Parts of this chapter were previously presented in [Kau07b, Kau09a].

5.1 Background: Booting an OS on a PC

Booting an OS on the PC platform is a complicated process. It starts with the firmware initializing the platform, followed by the execution of several bootloader stages and ends with the OS running its own initialization routines. In this section, I describe the x86 boot process in detail so that a reader not familiar with it may better understand the technical details mentioned throughout this chapter.

5.1.1 The Firmware

The x8 CPU fetches its first instruction after a reset or power-up occurred from a fixed address, which is 16 bytes below 4 GB. This memory range is mapped by the chipset to the firmware EEPROM that contains either BIOS (Basic Input/Output System) or UEFI (Unified Extensible Firmware Interface) instructions. In the following, I will use the term firmware for both of them.

The first task of the firmware is to initialize the CPU cores and the chipset. It enumerates the available cores, loads microcode updates and enables processor caches. It detects the installed memory and trains the DRAM controllers so that it can further execute from DRAM and is not constrained to the CPU caches anymore. Finally, it configures the chipset and assigns PIO (Port I/O) as well as MMIO (Memory Mapped I/O) regions to PCI (Peripheral Component Interconnect) and other configurable devices.

The firmware initializes devices that might be needed by the bootloader and a legacy operating system, like DOS. This includes devices necessary to fetch the bootloader such as disk and network controllers but also platform (PIC, PIT, I/O APIC), input (keyboard), and output (graphics, serial) devices. The necessary drivers can be built into the firmware, fetched from device EEPROMs (the so called Option ROMs), or loaded from disk (UEFI modules).

The firmware registers the services it provides in the Interrupt Vector Table (IVT) and in specially tagged memory regions. These services can be called by later running software, for instance to load blocks from disk, output characters on the graphics card, or get the available memory in a platform-agnostic way. This significantly simplifies the writing of initial bootloader and OS code.

The firmware also initializes system management mode (SMM), a x86 processor mode invoked through a SMI (System Management Interrupt). An SMI suspends the running OS and transfers control to a hidden memory area. SMM effectively allows the firmware to use the CPU behind the OS to perform power management tasks like fan control and handle fatal errors such as unrecoverable DRAM faults. OEMs (Original Equipment Manufactures) also rely on SMM to emulate legacy or workaround faulty hardware.

Finally, the firmware writes several tables containing platform specific information to memory. Examples are the ACPI, SMBIOS, and PnP tables but also the BIOS Data Area (BDA). It then loads the bootloader code and jumps to it.

5.1.2 The Bootloader

Whereas all bootloaders will load the OS and transfer control to it, their implementation depends on the feature set, the location they are loaded from and the OS they are booting. In the following, I describe how GRUB2, the most widely used Debian Linux bootloader [DPC], behaves when starting Linux.

The BIOS loads only the first sector of the boot disk, the so called Master Boot Record (MBR). These 512 bytes contain the initial boot code, but also the partition table, and in some cases even filesystem parameters. Consequently, many bootloaders are split into several stages to cope with the limited space.

The MBR code of GRUB2 loads the second bootloader stage from an unallocated disk area just before the first partition. This area, which can be as small as 62 sectors, is barely enough to hold the core, the disk driver, and the filesystem code of a feature-rich bootloader like GRUB2. Additional functionality is later loaded on demand from the disk. This includes a graphical user interface, read-only access to many filesystems

even on RAID and encrypted partitions, the support for multiple OS boot protocols and scripting languages.

The OS boot process is controlled by a script. It decides from what source the Linux kernel and the initial ramdisk is loaded and what command line is given to it. GRUB2 extracts the boot code from the Linux kernel image, patches certain data structures and finally transfers control to Linux.

5.1.3 The OS

As modern operating systems are initialize in parallel and start functionality on demand, the initialization- and runtime-code cannot be easily distinguished anymore. Even if OSes behave differently, the following steps are done by nearly all systems during startup.

The most important task of the OS boot code is to collect all the platform knowledge that cannot be accessed later[1]. It will ask the firmware, for instance, for the memory layout of the platform and the geometry of the disks. Furthermore, it searches for the firmware tables and detailed information about installed legacy devices like floppy or PS/2 controllers. An OS also checks whether it was started on a supported platform to notify the user early about any incompatibilities.

The OS then uses the firmware for the last time in the boot-process to access devices. Linux on the one hand switches to graphics mode here. Windows on the other hand uses the ability to access the disk without a dedicated driver, for loading the kernel and the boot modules as late as possible. The boot code then extracts and jumps to the main OS kernel, which initializes its data structure and the runtime environment.

The OS later resumes the hardware detection by discovering the devices and the resources they are using. This step typically involves an ACPI interpreter to collect the interrupt routing, legacy device information, and resource allocations.

Finally, the kernel executes the initial process, which in turn mounts filesystems, invokes boot scripts, and starts background processes like network services, GUI, and the login program. Additional services may be started later on demand.

5.2 The Bootloader

Originally, each operating system came with its own bootloader. This has changed with the advent of common boot protocols like Multiboot [MBI10]. A bootloader can now start different operating systems, understand multiple filesystems, and include device drivers that are not already provided by the firmware. Contemporary bootloaders consists of at least 15 KSLOC, whereas a feature-rich bootloader like GRUB2 can be as large as 300 KSLOC [GRU14].

Because the bootloader fully controls the platform, it will be part of the TCB of any code running after booting is completed. The size of contemporary bootloaders is a burden for any TCB constrained system. Booting, for instance, the NOVA OS described in Chapter 2 with a stripped-down GRUB2 already doubles the TCB. Additional code is added to the TCB of a guest OS if another bootloader is employed inside the virtual machine. Significantly reducing the size of the bootloader is therefore an important step towards a general purpose OS with a small TCB.

In the following, I analyze the required feature set needed for a bootloader, present a decomposed design, and describe noteworthy implementation details that will reduce the

[1] Linux implements this functionality in the `boot/` subsystem whereas Windows relies on `ntdetect.com`.

TCB impact of the bootloader by at least an order of magnitude while keeping enough functionality to start multiple OSes.

5.2.1 Features

The feature set has probably the largest impact on the bootloader size. A minimal loader like [Plo12] that only loads a single OS from a fixed location on the disk can be much smaller than a bootloader like GRUB2, which understands different filesystems, provides an interactive GUI, and supports multiple operating systems. We aim for a general-purpose bootloader with the following features:

Boot Protocols It should be able to start both host and guest systems in a NOVA setting. It therefore needs to support multiple boot protocols. At the minimum it has to start Multiboot compliant OSes [MBI10], Linux kernels, and legacy systems via chainloading.

Drivers The bootloader should rely on the firmware for device access. Dedicated device drivers are purely optional.

Filesystems Loading OSes from a fixed list of disk blocks is too inflexible, especially on dual-boot installations where different operating systems may act independently of each other. If one of the OSes optimizes the disk layout or just updates a file, the bootloader may get out of sync. The bootloader has to be able to read files from a centralized `boot` partition as present in many Linux and UEFI-based systems. It therefore has to understand a limited set of filesystems, most importantly EXT2, FAT, and ISO9660.

Dual-Boot The user should be able to switch between different boot configurations. The bootloader should at least be able to start an emergency system to recover from a failed OS update. Supporting a graphical user interface where the user can interactively modify existing boot configurations is optional.

Configuration A simple configuration file is sufficient to describe the different boot configurations. A Turing-complete scripting language as present in GRUB2 is not required.

Legacy Support for legacy firmware and hardware interfaces inflates the codebase[2]. Such features should be dropped if possible or at least made optional.

In summary I aim for a general purpose bootloader with a feature set less than that of GRUB2. However, the feature reduction alone will not be enough to significantly reduce the TCB. The decomposed design described in the next subsection together with a careful implementation are required as well to achieve this goal.

5.2.2 Design

A modern bootloader should run on a wide range of machines, support multiple operating systems, understand different filesystems and may even provide an interactive GUI. This requires a large feature set, which is in conflict with a small TCB.

In a previous experiment I reused components of the NOVA user-level environment (§2.2.3) and extended them with boot specific functionality to get a small bootloader

[2]Examples: CHS vs. LBA addressing, floppy support, A20 mask, e820 memory map, firmware bugs.

called BootSAK. Its design resembled a swiss army knife, where many small tools are glued together. Unfortunately, this approach could not cope with the high variability required from a bootloader because every customization required a recompilation of the binary. Moreover, a core of more than 1,000 SLOC could not be removed easily. In summary, the BootSAK experiment has shown that a monolithic design cannot provide a fine-grained enough TCB.

Instead, I apply the divide-and-conquer principle and decompose the bootloader into many small independent programs, which I call **bootlets**. This architecture enables the administrator to choose an *application-specific TCB* [HHF+05, FH06] by selecting only the subset of the bootlets that are required for a certain scenario. Bootlets have the following properties:

Single Task Bootlets perform a single task of a monolithic bootloader and they should do it well. There is for instance one bootlet that loads a Linux kernel from disk and another one that starts it.

Having one tool for each task keeps the complexity low and makes the bootloader highly configurable. This follows the Unix maxim: "Make each program do one thing well." [MPT78].

Independent Bootlets are independent binaries, which do not share any runtime code. Thus, bootlets can be implemented in different programming languages. The runtime overhead of the duplicated code should be negligible. The maintenance effort can be kept low if shared code is collected in a common repository.

Alternative approaches like dynamic linking are too complex, as shown by GRUB2, which needs more than 1,000 SLOC to implement it. Similarly, using a system-call interface as in SYSLINUX [SYS13] includes a growing runtime environment into all bootlets, even into those that will not benefit from the additional lines of code in their TCB.

Specialization Bootlets can be specialized to cope with legacy interfaces and diverging platform requirements.

Multiple implementations for the same feature can coexist: There might be for example a smaller bootlet that works on most platforms and a larger but more generic version that supports all of them.

Chainloading Bootlets are chainloaded. Most bootlets are Multibooted and start the next bootlet, after performing their duty, in the Multiboot way as well [MBI10]. The remaining ones initiate a new chain from other interfaces. The final bootlet then starts the OS.

Multiboot turned out to be especially suited as chainloading interface for the following reasons:

- The ability to specify multiple modules allows the construction of the chain at the start and enables the manipulation of it at each intermediate step.
- Multiboot enables the parameterization of each bootlet through a command line.
- A bootlet can use memory beyond its lifetime by reserving it in the memory map.
- Existing Multiboot-compliant loaders and tools can be reused.

`mbr_hdd`	Master boot record (MBR) code loads initial bootlets from disk.
`init_disk`	Extract bootlets from the disk blocks loaded.
`mod_load`	Load additional bootlets and OS files from the disk.
`mem_mmap`	Request the memory map from the BIOS.
`ui_select`	Skip parts of the chain according to keys pressed by the user.
`start_linux`	Start a Linux kernel.

Figure 5.1: Six bootlets are needed for the dual-boot scenario where a user can choose to start either a Multiboot OS like NOVA or a Linux kernel from disk.

- The Multiboot interface is simple enough so that chainloading can be implemented within 100 SLOC.

Resume Chainloading favors a linear execution flow: one bootlet is executed after the other. However, a bootlet can also suspend its execution by adding itself anywhere into the chain. If the bootlet is invoked again, it can resume its execution from the very same state as before.

Suspend/Resume eases the implementation of user interfaces like a shell or GUI. It can also be used as a plugin-like extension mechanism to remove seldom needed code from a bootlet. Finally, one could even implement cooperative multitasking with it.

In summary, bootlets are independent programs that perform a subtask of a monolithic bootloader. They are Multiboot chainloaded but a resume-style execution is possible as well. An administrator can freely combine multiple bootlets to produce a specialized bootloader with an application-specific TCB.

5.2.3 Implementation

Altogether, I implemented more than 30 bootlets that start a Multiboot chain, load files from disk, inspect the platform, boot different OSes, or just reboot the system. Only the six bootlets shown in Figure 5.1 are needed for a dual-boot scenario, where a user can choose to start either a Multiboot OS like NOVA or a Linux kernel from a disk. In this section, I explain noteworthy implementation details of the six bootlets that contribute to the reduction in bootloader size by an order of magnitude.

Programming Language

The bootlets are implemented in B1 as described in Chapter 4 due to the compactness of the language and the ability to directly include assembler code. Furthermore, many helper functions did not had to be written and could be used directly from the B1 standard library.

For the bootlets I choose a programming environment richer than the one in OSLO [Kau07b]. This has eased programming but made the TCB slightly larger. The B1 `printf` implementation, for instance, is around 80 SLOC larger than the minimal output functions from OSLO.

The B1 import feature, inherited from Python, proved to be crucial to reuse code from the standard library and even across bootlets. A bootlet can thus be easily specialized by importing a more general bootlet and overwriting small parts of it. A few extra code lines are usually sufficient for this purpose. For example, the `mod_load_ext2` bootlet removes

```
B1_ENCODING_OUTB = ["outb", "n", [0xee]]
def OUTB(ch, base):
    B1_ENCODING_OUTB
    ("mov", base, EDX)
    ("mov", ch,   EAX)
    ("outb",)
```

```
def ADD64(lo, hi, im):
    ("mov", lo,  EAX)
    ("mov", hi,  EDX)
    ("add", im,  EAX)
    ("adc",  0,  EDX)
    return EAX, EDX
```

Figure 5.2: The extensible B1 assembler makes low-level hardware access and 64-bit arithmetic functions straightforward. The left shows that a new instruction decoding like `outb` can be defined, referenced, and used within three lines. The right shows that using the add-with-carry instruction (`adc`) eases a 64-bit addition.

all filesystems except `ext2` from `mod_load`. Similarly, the `mem_mmap` bootlet removes all legacy features from `mem_bios` and just uses the most recent `e820` BIOS interface to retrieve the memory map.

Implementing the bootlets in B1 was also beneficial to the language, toolchain and the standard library for the following reasons:

- The newly introduced classes feature simplified the filesystem and disk driver implementation through abstract interfaces as well as inheritance.
- Generalizing the instruction encodings made writing 16-bit realmode code much easier. Furthermore, lowercasing the inline assembler statements makes larger functions more readable.
- The standard library was extended with collections, allocators, and dozens of helper functions.

Minimizing the Assembler Code

Low-level inline assembler code is required only in the few cases where B1 does not provide access to low-level hardware features like PIO. Similarly, assembler code was used for routines that would be either much slower, like `memcpy()` already shown in Figure 4.4, or would be much more complicated in B1, like working with 64-bit numbers. The extensible B1 assembler makes the implementation of such code straightforward as shown in Figure 5.2.

Inline assembler is also needed for code that runs outside the normal 32-bit protected mode. In the following, I will shortly describe how I minimized the 16-bit assembler code used for BIOS access and for initiating a new Multiboot chain from the Master Boot Record (MBR).

BIOS Access The bootlets rely on the BIOS for disk and keyboard access. While this makes any driver code in the bootloader obsolete, it also means bootlets have to call into 16-bit realmode. Note that alternatives to realmode code execution exist. However, they are either not as compatible (vm86 mode), too complex (instruction emulation), or not available everywhere (hardware-assisted virtualization).

Calling into realmode is implemented as a library function that switches to realmode, loads new register values from an address given as argument to the function, calls the destination, saves the modified registers back and returns to the caller in protected mode. Altogether 50 inline assembler instructions are sufficient to invoke the BIOS or jump to any other realmode routine from a bootlet.

Two assumptions make the code slightly simpler than comparable implementations. First, reloading the stack pointer is unnecessary if the realmode and the protected mode code use the same stack. Second, segment fix-up code can be omitted, if the stack, as well as the instructions, are located within the lowest 64k of the address space. The position independence and the small size of around 150 bytes makes it feasible to copy the code to a trampoline page before invoking it. This also allows to transparently use BIOS services from bootlets linked to an address above one megabyte – an area that is normally inaccessible from 16-bit realmode code.

MBR The `mbr_hdd` bootlet is loaded by the BIOS from the first sector of the hard disk, commonly called the master boot record (MBR). This code starts in 16-bit realmode and performs four tasks: i) It enables the A20 gate[3] to make all memory addressable. ii) It initializes the Multiboot information structure (MBI). iii) It loads a fixed set of blocks from the disk. iv) It switches to 32-bit protected mode and jumps to the loaded code in the Multiboot way.

To shrink this code down to 30 instructions, which is approximately 15% of the startup assembler code in GRUB2, I made the following simplifications:

- The code assumes that the BIOS implements contemporary interfaces correctly. There is neither support for CHS[4] addressing, nor for sophisticated A20 handling. This functionality has to be provided either by a second implementation or by later running bootlets if required for an ancient platform.

- The MBR does not extract binaries, add modules to the MBI, or request the memory map. These tasks are left to later bootlets, where they can be implemented much easier in high-level B1.

- The MBR directly includes the disk address packets (DAP) to map disk sectors to memory regions as dictated by the BIOS interface. There is only a modest size gain when generating this format at runtime from a dense data structure whereas the additional instructions complicate the code.

- There is no need for a second loader stage to cope with the limited space in the MBR. Since the DAP region is located at the end of the MBR, any loaded disk block can extend it. This enables loading of an unlimited number of disk sectors. Nevertheless, the relative low speed of the BIOS drivers and a 1 MB address barrier of most BIOS implementations limits this feature.

Finally, the `mbr_hdd` bootlet is self describing, so that installation code can modify the DAP region or the destination entry point, without having to recompile the code or consult a manifest. If the bootlets that should be loaded fit into 254 sectors, the MBR does not need to be changed. Installation then is as simple as writing `mbr_hdd`, `init_disk`, and all other bootlets together with their command line at the first sectors of the disk. The `init_disk` bootlet will then populate the Multiboot Information Structure from this densely packed format.

[3]The famous A20 gate can force the 20th address line of the CPU to zero to enable an address wraparound at 1 MB. This is a legacy feature of the PC platform for i8086 and DOS compatibility. Enabling A20 early allows loading above 1M on platforms with a particular EDD extension.

[4]CHS is an ancient addressing mode for disks where the caller has to specify the cylinder, head, and sector number. The modern linear block addressing (LBA) uses a single number instead.

Chainloading via Code Generation

Multiboot chainloading is essential to split the bootloader into smaller bootlets. The idea and with it the first implementation dates back to OSLO, where it was used to externalize features that need not be part of the TCB in every scenario [Kau07b]. Optional features like hashing of command lines were moved into separate tools that are Multiboot loaded from OSLO and that would after finishing their task start the next tool in the Multiboot way.

The original chainloading implementation was limited to non-overlapping binaries. Each tool had to be linked to different addresses, as the decoder, which unpacks the ELF (Executable and Linkable Format) files to memory, would otherwise overwrite its own code, which in turn could crash the machine. This restriction was acceptable as long as the number of binaries was small. However, as more and more tools, like `morbo`, `montevideo`, or `santamonica`[5] relied on Multiboot chainloading, it became clear that this simple approach would not scale to hundreds of different binaries produced by multiple vendors.

In January 2010 Julian Stecklina and I discovered that code generation can solve this issue elegantly. Our *fancy chainloader* does not call `memcpy` and `memset` anymore to extract the sections from the ELF file. Instead, it emits x86 code that performs the exact same task including the final jump to the target entry point. Since the generated code is position independent, it will fit into a small scratch area outside of the target binary. Furthermore, as the code neither requires the stack nor any other part of the original binary, it can fully overwrite any memory the old binary is using. Lastly, the approach is quite simple: 100 SLOC are sufficient to implement it.

In summary, code generation allows the bootlets to be linked at any address. This makes Multiboot chainloading an universal extension mechanism for bootloaders.

Specialization

A significant TCB reduction can be achieved by implementing a specialized solution for the common case, albeit with reduced functionality. Besides the specialization of whole bootlets already mentioned before, the config language as well as the user interface are two examples where this approach was successfully employed.

Config Language The `mod_load` bootlet retrieves OS binaries as well as bootlets from disk and adds them to the MBI structure. The files to load are specified at the command line or in config files expressed in a simple language inspired by Pulsar [Ste13][6]:

- Each line starts with a four character command followed by optional parameters.
- Unknown commands and short lines are ignored for forward compatibility reasons.
- The character `#` starts a comment that reaches until the end of the line.

Parsing and executing the eight commands shown in Figure 5.3 requires only 100 SLOC, which is at least twenty times less than what GRUB2 needs to interpret its Turing-complete scripting language. Yet, this simple language is powerful enough for the common case, specifically to load multiple boot configurations from different filesystems. For the

[5] `morbo` initializes a Firewire controller [Ste09], `montevideo` switches the graphics mode with the help of VBE, and `santamonica` gunzips Multiboot modules. The last two are extensions to the OSLO codebase.

[6] The implementation is not fully compatible to Pulsar yet, as `exec` does not adjust the module load address with the end address of the retrieved ELF file.

Name	Params	Description
`exec`	PATH CMDLINE	Prepend a file to the list of modules.
`load`	PATH CMDLINE	Append a file to the list of modules.
`conf`	PATH	Load and interpret another script.
`root`	PATH	Set the directory for relative paths.
`addr`	ADDR	Change the load address.
`fsnr`	NR	Select an FS (filesystem) to load files from.
`uuid`	PATTERN	Select an FS through a unique identifier.
`bdev`	[NR]	Set the bootdevice in the MBI to the current FS or to NR.

Figure 5.3: The bootlet configuration language. The `mod_load` bootlet understand eight different commands.

Layer	B1	SYSLINUX	Reduction	GRUB2	Reduction
`vfs`	260	1100	4.2	710	2.7
`ext{2,3,4}`	180	600	3.3	780	4.3
`fat{12,16,32}`	200	700	3.5	660	3.3
`iso9660`	150	260	1.7	880	5.9
Sum	790	2660	3.4	3330	4.2

Figure 5.4: Size of read-only filesystem code in SLOC. The B1 implementation is more than three times smaller than the SYSLINUX and GRUB2 implementations written in C.

few use cases where this is not enough, a more complex bootlet should be loaded that either implements the required functionality directly in B1 or provides it through an existing scripting language like LUA or Python.

UI Interactively changing the boot configuration is seldom needed. In the vast majority of cases, the default entry or one of a few predefined configurations is booted. However, when implementing only the selection step and deferring the interactive editing to a more sophisticated UI that is chainloaded from it, the complexity of the user interface can be drastically reduced.

The `ui_select` bootlet implements this simpler task by evaluating the BIOS keyboard state and skipping a certain number of bootlets in the chain, depending on the keys that were pressed. Holding for instance the `shift` key could boot a second OS, whereas pressing the `space` key could load a rescue system. The bootlet is only 40 SLOC large, whereas an interactive shell might require several hundred SLOC, especially if graphic drivers are required as well.

Filesystem Access

Filesystem code is one of the largest parts of a modern bootloader. GRUB2 for instance needs more than 20 KSLOC to read two dozen file and filesystem formats. The feature analysis in Section 5.2.1 has already revealed that only a few filesystems have to be supported. Moreover, neither encryption nor RAID support is needed, which are exceptionally costly to implement. Getting read-only access to a boot partition on disks, USB (Universal Serial Bus) stick, and optical media is sufficient. Thus, supporting EXT2, FAT, ISO9660, and their extensions is basically enough.

Figure 5.4 compares the size of the read-only filesystem code in the `mod_load` bootlet with the corresponding code in SYSLINUX and GRUB2. A fair comparison with more

codebases is difficult, as they do not support all of these filesystems (ELILO, U-Boot, OpenBIOS), reuse Linux headers (EMILE), or ignore file metadata (GRUB-legacy).

The B1 implementation is on average 3.4x smaller than the filesystem code in SYSLINUX, even though it has a slightly larger feature set by supporting *UUID* in all filesystems and *extents* in `iso9660`. The GRUB2 code is even larger, due to the following reasons: i) GRUB2 implements more features like filesystem labels and the Joliet as well as the RockRidge extension for `iso9660`[7]. ii) GRUB2 does not use a buffer cache to simplify the code but accesses the disk directly. iii) The code contains definitions of structures and constants that are never used.

Two advantages make the B1 implementation smaller than their C counterparts. First, B1 code tends to be more compact. Defining structures as lists and returning up to four values per function saves lines. Moreover, the recently added classes feature allows to inherit and overwrite methods easily. Emulating this behavior in C by manually defining a structure requires more code.

Second, B1 uses a filesystem interface optimized for size. It does not use file descriptors nor `open()` or `close()` functions. Instead a file is identified through a 64-bit number that references the *inode* in a POSIX filesystems like `ext2` or the directory entry in non-POSIX filesystems like `fat`. A read-only filesystem has to implement only four functions: `mount()` to identify and initialize the filesystem, `walk()` to recursively visit directory entries, `read()` to read blocks from a file, and `stat()` to retrieve metadata like the file size. A generic `translate()` function, which converts paths to *inode* numbers, is already provided by the VFS layer. A filesystem can still overwrite this function if a faster way to translate a path exists.

In summary, interface as well as implementation improvements make the filesystem access code at least three times smaller than in other bootloaders.

5.2.4 Evaluation

The decomposed architecture has no visible influence to the overall boot time. The bootlets start an OS basically as fast as any other monolithic bootloader that relies on the BIOS for hardware access. I therefore skip a detailed performance evaluation here in favor of evaluating the code size improvements.

Even though I implemented more than 30 bootlets, I will only evaluate the six bootlets from Figure 5.1, as they are enough for a dual-boot scenario that enables the user to choose between a Linux and a *Multiboot* OS like NOVA.

Figure 5.5 reveals that the bootlets are between 80 and 1800 SLOC large, with `mbr_hdd`, which is programmed in assembler, being the smallest and the `mod_load` bootlet, which includes the filesystem code, being the largest one. The six bootlets together consist of 2050 SLOC. This is around 60% of the size of the individual bootlets combined because the bootlets share a significant portion of their code, like chainloading, output, and string routines.

The 2050 SLOC can be reduced further by an administrator by choosing only the necessary feature set. As the combinations are nearly unlimited, I will concentrate on the most likely ones as listed in Figure 5.5:

- If a scenario fits in the sectors loadable by the MBR, the `mod_load` bootlet and with it 1275 SLOC can be omitted.

[7]The `fat` numbers were taken before EXFAT support was added.

Bootlet	SLOC
`mbr_hdd`	80
`init_disk`	180
`mod_load`	1800
`mem_mmap`	250
`ui_select`	470
`start_linux`	540
Overall	2050

Removal	SLOC
`mod_load`	-1275
`mod_load_ext2`	-360
`mbr_hdd+init_disk`	-100
`ui_select`	-40
`start_linux`	-60
`mem_mmap`	-30

Figure 5.5: The six bootlets are between 80 and 1800 SLOC large. Overall they are build from 2050 SLOC (left). This number can be reduced by removing unnecessary features. For example a scenario that does not require `mod_load` reduces the size by 1275 SLOC down to 775 SLOC (right).

- If the files are only retrieved from an `ext2` filesystem, the specialized `mod_load_ext2` can be employed, which saves 360 SLOC.
- If the MBR code is not needed, for instance within a Vancouver VM, 100 SLOC for the `mbr_hdd` and `init_disk` bootlets can be removed.
- If boot selection is unnecessary, `ui_select` with its 40 SLOC need not to be included.
- If only Multiboot OSes are booted, 60 SLOC from the `start_linux` code can be saved. Alternatively, if one needs to boot only Linux kernels, a memory map need not be provided by `mem_mmap` and the TCB becomes 30 SLOC smaller.

When comparing this to the example scenario measured in Appendix (§*A*.4.2), only 1690 SLOC are required to start Linux or a Multiboot OS from an `ext2` disk. In contrast GRUB2 needs 30 KSLOC or around 17x more for the same task.

5.2.5 Summary

In this section, I showed how a significantly smaller bootloader can be constructed by employing the following techniques:

1. Decomposing it into smaller *bootlets* to enable an application-specific TCB.
2. Reducing the required feature set and moving legacy features from the critical path.
3. Specializing bootlets to reduce the TCB in the common case.
4. Relying on firmware interfaces instead of own drivers.
5. Using Multiboot chainloading as flexible extension mechanism.
6. Choosing B1 - a programming language more compact than C.
7. Using a few implementation tricks, like an extensible DAP region, a simpler config language, and code generation.

In summary, a bootloader does not need to be ten- or even hundred-thousand lines large. A few thousand lines will be sufficient for most scenarios.

5.3 Trusted Computing

The boot stack of the PC platform has been under attack since the *Brain* virus was released in 1986 [Szo05]. It is currently easy for an adversary to replace the bootloader [KCW+06], circumvent firmware security checks [Kau07b], take over SMM [DEG06], or hide itself in ACPI tables [Hea06]. Various techniques were developed to make such attacks on the boot stack more difficult. In this section, I concentrate on Secure Boot and Trusted Computing techniques that aim to improve the security of the whole boot stack including the firmware and bootloader. I analyze whether they can be used to shrink the amount of code that needs to be trusted and describe security threats to show the current limits of these approaches.

5.3.1 Secure Boot

Secure Boot is probably the most widely deployed technique to harden the boot process. It protects the Xbox and millions of iOS installations from running unauthorized applications [Ste05, iOS14]. It has reached the PC platform as well since hardware certified for Windows 8 has to support UEFI Secure Boot [Mic14].

The idea underlying Secure Boot is to allow only known and hopefully secure software to execute, by checking that each program is digitally signed by a trusted authority. Additionally a revocation list ensures that vulnerable programs cannot be used any longer.

Secure Boot improves the platform security by raising the bar for attackers: simple approaches to takeover the boot stack, like patching the bootloader, will fail. However, various Secure Boot implementations have not withstood sophisticated attacks:

- Microsoft could not defend the Xbox secure boot system against determined attackers [Ste05].
- Apple seems not to be able to protect any iOS version longer than a couple months against jailbreaks [MBZ+12].
- Various attacks against UEFI Secure Boot are known. The technique is still "too young to be secure" [BSMY13].

There are various reasons why Secure Boot systems are vulnerable:

Bug Free The whole boot stack needs to be bug free because a single defect can be enough to circumvent its protection. The huge size of current implementations makes this an impossible task.

Updates The long-term security of a platform depends on timely updates of the vulnerable program and the extension of the revocation list. However, as [BKKH13] has shown, a compromised platform can effectively avoid those updates while telling the user that everything is well.

Code Signing The firmware, external modules, bootloaders, the OS kernel and any other code that has full control over the platform needs to be signed. However, verifying the integrity of even a single program is a costly task. Obfuscation can be used to bypass automated security checks [HKY+13].

Root of Trust The certificate checking forms a trust chain, where each component in the boot process is trusted to correctly verify the signature of the next part it boots. The first software in this chain, called the Root of Trust for Measurement

(RTM), can only be protected by hardware mechanisms. However, considering its importants in the chain, this protection is often not as strong as it should be [Ste05, Kau07b, BSMY13].

Even though Secure Boot can be implemented relatively easily, protecting such a system against determined attackers is a much harder task.

Finally, adding Secure Boot functionality to an existing software stack increases its complexity and with it the TCB. Secure Boot does not reduce the size of the boot stack.

5.3.2 Trusted Computing with a Static Root of Trust for Measurement

Secure Boot suffers from a fundamental limitation: There is no way to reliably tell what software is running or what updates were installed on a particular machine. This restricts trust to a binary value: either a certain software was signed by one of the trusted authorities or not. A finer grained trust relationship seems not to be implementable with Secure Boot.

Trusted Computing solves this issue by adding a smartcard-like chip called TPM (Trusted Platform Module) to the platform [Pea02, Gra06]. Instead of checking a signature on the program to load, a hash of it is added to a so called PCR (Platform Configuration Register) that resides in the TPM. PCRs are tamper proof and cannot be directly modified. Instead they can only be `extended` by storing a hash over the previous value of the PCR and the value to add. Whereas this additional layer of indirection changes little from a security point of view[8], it allows to store a large number of hashes within the limited memory of a small chip like the TPM.

In the initial implementation of Trusted Computing, the PCRs could only be set to zero by rebooting the whole platform. At this point trusted firmware gets control over the platform and can hash itself into a PCR. This code is called the Static Root of Trust for Measurement (SRTM) because it is always invoked at a fixed point in the machine life cycle. I will later describe the newer Dynamic RTM approach, where a new hash chain can be started at anytime after booting.

The TPM can digitally sign the PCR values with a private key to attest to a third party, which software stack is currently running (*authenticated booting*). The PCR values can also be used by the TPM to make security-relevant data only available to a particular OS version (*sealed memory*). This feature protects data against reboot attacks for instance in disks encryption systems like Bitlocker [Mic09].

Attacks on SRTM systems

There are two security assumptions in the SRTM design. First, all software layers have to hash the next layer before they are giving control to it, starting from the first firmware code until the OS takes over control. Second, the TPM and the platform have to reset at the very same time. In [Kau07b] I described several attacks targeting all layers of a TPM-protected system that violate these assumptions:

Bootloader The *trusted* bootloaders turned out to be the weakest link in the system. Some bootloaders did not hash all the code they execute. Others can be tricked into executing different code than what was hashed before.

[8] It enables new cryptographic attacks, like reversing one `extend` or reversing all previous `extends`. However, these are generally believed to be mathematically harder than collision attacks on the hash function that would already break the cryptographic protection.

Firmware The BIOS in a TPM-enabled platform could be fully modified as no update protection was implemented. This includes the SRTM code that extends the PCRs for the first time. As a patched BIOS will not extend PCRs anymore but leaves them in their reset state, an attacker can fake any PCR configuration. The lack of platform certificates means that this bug will affect the remote attestation of all machines with the same TPM chip.

TPM Firmware A "feature" of a particular firmware allowed driver code to reset the TPM independently of the platform. The remote attestation with the affected TPM version cannot be trusted any longer. This attack revealed that the security of the system also depends on hidden firmware.

Hardware Attack TPMs are usually attached to the Low Pin Count (LPC) bus. A single wire connecting the reset line of this bus to ground will reset the TPM independently of the platform.

Fixing the bootloader and the firmware bugs is a tremendous task, especially as those large software stacks were never written with security in mind. Moreover, as trusted computing based on an SRTM cannot protect against "simple hardware attacks" [Gra06], it is unsuitable for many scenarios like kiosk computing [GCB+08], the Digital Rights Management scenario [RC05], theft protection [CYC+08], or whenever physical modifications for instance through an "evil maid" [RT09] cannot be excluded.

In summary, Trusted Computing based on a SRTM can easily be compromised. It cannot reduce the TCB of the system but adds new code including the TPM firmware to it.

5.3.3 Trusted Computing with a Dynamic RTM

In 2006 I implemented the Open Secure LOader (OSLO), the first publicly available bootloader that relies on a Dynamic Root of Trust for Measurement (DRTM) to use Trusted Computing even in the face of a "resettable TPM, an untrusted BIOS, and a buggy bootloader" [Kau07b]. A DRTM is a hardware extension invoked through a special instruction that puts the CPU into a known good state, stores a hash of a small piece of code called the secure loader into a new PCR and transfers control to it. The secure loader can then verify that the platform is configured securely before extending a PCR with the hash of the OS it boots.

Intel and AMD provide different DRTMs. Intel uses the `senter` instruction to invoke an `SINIT` module, a secure loader that has to be signed by the platform manufacturer. AMD provides the `skinit` instruction to start any piece of code as secure loader. The advantage of Intel's approach is the ability to ship platform verification code that relies on undocumented chipset features, whereas AMD's approach is much simpler to implement in hardware as no signatures have to be checked. OSLO works only with AMD machines. However, the `tboot` project provides similar functionality for Intel platforms [TBO].

A DRTM counteracts the aforementioned attacks on the SRTM system because resetting the TPM is not enough anymore. Newly introduced PCRs are not set to zero on a TPM reset anymore. Instead, they are initialized to `-1`. Only the special LPC bus cycles induced by the DRTM can force them to zero.

TCB Reduction with OSLO

Starting the trust chain after the firmware and a legacy bootloader have finished their tasks allows removing their code from the TCB. Removing the ACPI interpreter from

the TCB as well is tricky because validating the ACPI tables, the only source of certain platform specific knowledge, seems to be unfeasible. Instead, one can use the solution presented in Section 5.4 to shrink the ACPI code.

Replacing the firmware and bootloader with the 1583 SLOC required for OSLO reduces the TCB significantly. However, one needs to trust more than just OSLO in such a scenario:

Verification Code OSLO does not include code to verify that the platform is configured securely. Whereas no such code seems to exist for AMD platforms yet, one can surely assume that it will be as complex as on Intel chipsets. A SINIT module for the Intel Haswell platform, for instance, has an estimated size of 9 KSLOC [TBO].

TPM The TPM has to be trusted to store the hashes and the signing keys safely. Measuring the TCB impact of a TPM directly seems to be impossible, as neither the firmware nor the hardware are freely available. The best available estimate are the 19 KSLOC of a TPM emulator [SS08].

External Device One needs an external device to receive and check the PCR values. A simple microcontroller based implementation requires around 4 KSLOC [VCJ+13].

New Hardware Additional hardware is required to use OSLO. A platform not only needs a TPM but also an IOMMU (I/O Memory Management Unit) for DMA (direct memory access) protection. Furthermore, the CPUs as well as the chipset have to securely forward the hash of the secure loader from to the TPM.

Requiring new hardware features excludes legacy and due to additional costs also many embedded systems. Furthermore, it enables new ways to attack the system. Finally, validating new features is costly.

Unfortunately, the additional complexity induced by Trusted Computing cannot be easily expressed in SLOC because hardware implementations and validation procedures are usually kept confidential by the manufacturers.

In summary, relying on trusted computing with a DRTM can replace the 70 KSLOC of firmware and bootloader with approximately 35 KSLOC for the secure loader, TPM, platform verification, and external device.

5.3.4 Security Challenges

Using a secure loader like OSLO alone, is not sufficient to construct a secure system. Most importantly the booted OS needs to be able to protect itself against various attacks. It must use the IOMMUs, hash any code that is loaded into the kernel and protect the kernel interfaces that can compromise its security, may it be from the administrator of the machine or not [VMQ+10]. Having an OS with a small TCB like NOVA significantly simplifies this task.

The following hardware and software challenges have to be met as well to gain a secure DRTM system:

Firmware One has to ensure that the untrusted firmware cannot get control over the platform again after the secure loader was executed. Sandboxing BIOS code and limiting the impact of ACPI code as explained in Section 5.4, removes two ways how malicious firmware can takeover the system.

A still open attack vector is SMM [WR09a]. This means the firmware has to be trusted to install bug free SMM code and protect it against various attacks [DEG06, DLMG09, WR09b]. However, an adversary with physical access can avoid that the firmware runs at all, by booting from an untrusted PCI device[9].

The ideal solution would be to completely disable system management mode. However, this excludes several legitimate use cases and requires OS as well as firmware changes [Man09]. Running the malicious SMM code instead in a VM, as proposed by Intel and AMD [SVM05, UNR+05], will further increase the complexity of the virtualization layer and with it the TCB of the system.

TPM The large distance to the CPU, the complexity of the interface, and the low cost make the TPM a primary target:

- Tarnovsky could break the TPM hardware protection and expose its secret keys [Tar10]. Even though it is infeasible to mount his invasive attack in the large, it revealed that TPMs are not as tamper-resistant as believed before.
- Winter and Dietrich manipulated the LPC frame signal with an FPGA to turn normal requests on the LPC bus into DRTM messages. They conclude that special cycles on the LPC bus provide "little to no protection against adversaries with physical hardware access" [WD12]. Thus, Trusted Computing with a DRTM is not resilient against "simple hardware attacks" [Gra06].

Implementing a reduced TPM feature set directly inside the CPU as proposed for instance by Intel SGX [MAB+13], would make hardware attacks harder. Nevertheless, the challenge of bootstrapping trust into such a platform remains [Par08].

Verifier OSLO does not include the verification code to ensure for instance that the chipset is initialized correctly, security-critical registers are locked, the DRAM controller is trained properly, and any platform backdoor is disabled. Writing such code is complex and error prone, especially as chipset details are often undisclosed.

Moreover, securely verifying the DRAM configuration seems to be impossible on the x86 platform because the only reliable source for this information, the EEPROM on the DRAM, is only accessible through the slow and untrusted SPD (Serial Presence Detect) bus, which can be easily manipulated.

Memory The ability to replace DRAMs in a running or suspended system will compromise even an otherwise invulnerable OS. This so called Cold-Boot attacks [HSH+09] are a threat to Trusted Computing, especially as the code and data integrity is checked only during startup. Once the OS is hashed and running, it should be able to protect itself.

Intel SGX relies on a memory encryption engine [MAB+13] to remedy these classes of attack, even though this increases the complexity of the hardware and costs some performance.

Similarly, cache attacks [OST06, DLMG09] limit the confidentiality a system can provide. Keeping valuable secrets in CPU registers [KS09] and implementing encryption as well as random number algorithms directly in the CPU [SDM13], counteracts cache attacks with additional hardware.

[9] Intel Southbridges have a debugging feature publicly documented since 2005 (ICH6). They consult the Boot BIOS functional straps to decide whether to load the initial code from EEPROM or from the PCI bus. An attacker can enable these pins with two simple wires.

Backdoors Contemporary platforms have rich management and debugging capabilities to cope with the high system complexity. Unfortunately, these features may also be used as backdoor into the system:

- The Intel AMT Management Engine (ME) can access memory and control devices like network cards, graphics controllers, and USB. An attacker able to exploit a defect in its firmware could remotely takeover the machine.
- The embedded controller in the AMD SB700 southbridge resides on the LPC bus together with the TPM. Malicious firmware code on this device could generate special cycles and thereby break the DRTM without requiring any physical modification.
- There are various debug interfaces like JTAG or PECI that allow to read and modify CPU as well as chipset internal state. If accessible in production hardware, they could be used to compromise the OS.

Finally, malicious CPUs can be built easily but detecting them is hard [KTC+08]. Thus, proving for a particular platform that all backdoors are closed remains an open research challenge.

5.3.5 Summary

In this section, I analyzed whether Secure Boot and Trusted Computing can be used to shrink the trusted boot code. I found that Trusted Computing based on a DRTM can remove as much as 35 KSLOC from the TCB.

I also show that various security challenges have to be met before Trusted Computing will offer the security guarantees it promises. Without a solution to these issues the "additional security of current secure bootstrap facilities is minimal" [HvD04].

A common pattern one can observe in the solutions that were already proposed, is that security-critical functionality is moved to the hardware. However, this will not reduce the overall TCB of the system, but just trade software against hardware complexity. Instead, one should also account the hardware to the TCB and apply reduction techniques to it as well. Ideally, the hardware of a secure platform should be verifiable independently of the vendor.

5.4 ATARE: Parsing ACPI Tables with Regular Expressions

Fully supporting the ACPI [ACP] in an OS is a task many developers want to avoid because it requires the understanding of a complex specification and the implementation of a lot of code, most notably for an ACPI Machine Language (AML) interpreter.

Unfortunately, an OS needs to know how the interrupt lines of PCI devices are routed to global system interrupts (GSI). This information is platform specific as it depends on the chipset as well as on the board wiring. However, drivers need the routing information to listen to the right interrupt vector for device interrupts. If the OS uses an incorrect routing, no interrupts or even an interrupt storm may be received by the driver. An OS might not successfully boot in this case.

The only authoritative source for this information are ACPI tables that are typically evaluated by an AML interpreter. As the traditional approach of porting such an interpreter to a new OS adds tens of thousand lines to the TCB, I searched for a new approach.

Name	Path	SLOC
Linux 2.6.27.4	drivers/acpi	50295
	include/acpi	7325
FreeBSD head	contrib/dev/acpica	44621
	dev/acpica	12878
ACPICA 20080926	binary: 98.2k	73840

Figure 5.6: Lines of Code of ACPI implementations.

Because platform resource discovery and fine-grained power management need not to be based on ACPI, only a fraction of the data available through AML is actually required. In fact, the only mandatory information is the routing of PCI interrupt lines to global system interrupts (GSI) [SA99]. In this section, I therefore discuss how this particular information can be deduced from AML. Other platform-specific knowledge such as the S3 sleep state parameters can be retrieved in the same way. Related approaches that consume this information, like [Sch12], are orthogonal to this work whereas I focus on the retrieval step.

By extending the results from [Kau09a] with a C++ implementation and evaluating it on a recent and much larger ACPI table collection, I show in the following that regular expressions can extract the interrupt routing in nearly all known cases with significantly less code then a full ACPI interpreter.

5.4.1 Background

ACPI standardizes the interaction between hardware, firmware, and operating system. It allows an OS to enumerate platform resources, change resource allocations, and perform power-management tasks.

The ACPI specification is quite large. Version 3.0b [ACP06], for instance, consists of more than 600 pages, defines 13 ACPI tables, and references 19 externally defined ones. ACPI tables, which map a name like `RSDT` to a binary blob, either have a fixed layout that can be easily understood by the OS or they consists of AML bytecode that needs a complex interpreter. The DSDT and SSDTs fall into the later category.

AML is a domain specific language with many features: It knows named objects, methods that could be called on them, and control structures such as `if`, `while`, and `return`. It supports various data types, for instance numbers, strings, and variable sized packages of other objects. Different operations can be performed on these types, like arithmetic and bitwise operations on numbers. In summary, AML is a Turing-complete and domain-specific language specially tailored for platform configuration and power management.

Table 5.6 gives lines of code for the ACPI implementations in Linux, FreeBSD, and in the reference implementation ACPICA [ACA]. The numbers for ACPICA also include various tools, which are not strictly needed in an OS like an AML compiler. These numbers show that the traditional approach of porting ACPICA will add more than 50 KSLOC to the OS.

Furthermore, interpreting AML code has security implications, as shown by Heasman [Hea06] who was able to hide a root kit in AML. The main issue stems from the fact that the AML code needs to access platform resources to perform its duty. However an interpreter in the OS cannot decide whether a certain access should be allowed or whether it will open a backdoor without the knowledge that is only available in AML. For example, Windows XP blocks access to some I/O ports [Mic03], but just warns if

AML code accesses kernel memory. In summary, if the AML code is not sandboxed, it will be part of the TCB as well.

Alternatives to ACPI

The easiest available source for the interrupt routing is the PCI configspace where the BIOS puts the IRQ (interrupt request) numbers for DOS compatibility reasons. Unfortunately, these numbers are only correct when using the legacy PICs (Programmable Interrupt Controllers) [PIC88] but not with the newer I/O APIC (I/O Advanced Programmable Interrupt Controller) interrupt controller [Int96]. If an OS relies on this older mode, it gives up most of the gains from the newer I/O APIC architecture such as reduced IRQ sharing and fine-grained IRQ routing to different processors [Mic01]. Furthermore, acknowledging an IRQ can be quite slow on the slave PIC. Finally, certain interrupt lines may not be wired to the PIC at all. Solely relying on the PIC can therefore not be an option for an OS.

MSI (Message Signaled Interrupt) [Bal07] are an alternative to interrupt controllers and dedicated IRQ lines. A PCI device with MSI support can be programmed to directly send the interrupt vector to a CPU. IRQ routing is not an issue with MSIs anymore as all interrupt controllers in the chipset are bypassed. Unfortunately, MSI support is not available on many devices and even broken on some platforms. Linux 2.6.27.4 for instance disables MSI in 24 cases [LIN08]. MSIs are mandatory for PCI express devices and clearly the future, but an OS cannot solely rely on them today.

There are other sources known for IRQ routing. One could, for example, rely on the MP configuration table, as defined in the Multi-Processor Specification [MP97]. It consists of a format that is simpler to parse than AML. However, many BIOSs do not provide an MP table anymore, as it is deprecated since the first ACPI specification has been published in 1999. Another solution would be to build an IRQ routing database into the OS. However, collecting this dataset would be a tremendous task due to the large number of different motherboards.

In summary, there is no real alternative to ACPI as the authoritative source for the IRQ routing of a platform.

IRQ Routing Representation

IRQ routing information is returned in ACPI by evaluating the `_PRT` method of a PCI bridge. This returns a package of IRQ mappings for devices on its bus. An IRQ mapping consists of the following fields:

1. PCI device address,
2. IRQ pin (0 - `#IRQA`, 1 - `#IRQB`,...),
3. A name of a PCI IRQ router,
4. A GSI (global system interrupt) number if no IRQ router was given.

5.4.2 Pattern Matching on AML

There is a pattern in many DSDTs that can also be seen in Figure 5.7: The IRQ mappings returned by the `_PRT` method in I/O APIC mode, use a fixed GSI value and not an IRQ router. This means finding the IRQ mappings in the AML code is enough to know what

```
Method (_PRT, 0, NotSerialized) {
    If (GPIC)
        Return (Package (0x01)
            Package (0x04) {
                0x0014FFFF,
                0x02,
                0x00,
                0x12})
    else
        Return (Package (0x01)
            Package (0x04) {
                0x0014FFFF,
                0x02,
                \_SB.PCI0.LPC0.LNKC,
                0x00})
}
```

Figure 5.7: A slightly modified real-world `_PRT` method. Depending on the value of the `GPIC` variable, it returns different IRQ mappings. If the OS runs in I/O APIC mode the PCI IRQ pin is routed to GSI 0x12 whereas in PIC mode the pin is mapped to an IRQ router named `LNKC`.

```
SEG    = [A-Z_][A-Z_0-9]{3}
NAME   = (\ | ^*) (SEG | (\x2e SEG SEG) | (\x2f. SEG*))
PKGLEN = [\x00-\x3f] | [\x40-\x7f]. | [\x80-\xbf].. | [\xc0-\xff]...
DATA   = [\x00\x01\xff] | \x0c.... | \x0b.. | \x0a.

METHOD = \x14 PKGLEN NAME
DEVICE = \x5b\x82 PKGLEN NAME
SCOPES = \x10 PKGLEN NAME
DEFINE = \x08 NAME ( DATA | \x12 PKGLEN )
IRQMAP = \x12 PKGLEN \x04 DATA{4}
```

Figure 5.8: The Regular Expressions needed to extract IRQ mappings from AML.

GSI is triggered through a PCI interrupt line. However, as not all DSDTs follow this pattern the corresponding search algorithm cannot be exact. It will be an heuristic.

Note that the accurate parsing of AML elements is complex and impractical. To get the start of the `_PRT` method in the bytecode stream, all the unnecessary elements have to be skipped. Unfortunately, there is no clear hierarchy between different element types. It is possible that a calculation of a variable is directly followed by a definition of a new method. To find the start of a method, one has to know every AML element type, its length and whether it would contain other elements or not.

A look into the ACPI specification reveals that AML is densely packed binary data but still contains enough "space" in its encoding. In fact, distinct binary ranges are used for different elements. This allows to apply regular expressions to search for specific AML elements. Furthermore all variable sized objects such as `METHOD` or `DEVICE` directly specify their length. Whether one object is contained in another one is therefore easily decidable.

Figure 5.8 lists all the regular expressions that are needed to find the required AML objects. The first four definitions are just shortcuts to simplify the later expressions. The last five are used for the search. Name segments, for example, are always four characters long and start with an uppercase letter or an underscore. The last three chars may also contain digits. Another example would be the data that is found in the four element IRQ mapping packages. It consists either of one of the characters {0,1,255} or it starts with

	Files
good	456
no I/O APIC	270
compile error	108
IRQ router	17
Sum	851

Figure 5.9: DSDTs tested with the Python prototype.

one of {12,11,10} followed by a {4,2,1} byte-wide integer respectively. The pattern named `DATA` in Figure 5.8 reflects this case.

5.4.3 The Search Algorithm

The regular expressions listed in Figure 5.8 allow searching for names, methods, devices, scopes, definitions of variables, and packages with four elements that look like IRQ mappings. A higher level search algorithm can be built upon these basic primitives. To avoid false positives, the search algorithm has to make sure that IRQ mappings are returned from a `_PRT` method. IRQ mappings can be directly defined within the method. Furthermore, DSDT writers sometimes put the list of IRQ mappings outside the actual `_PRT` method and just reference them directly or indirectly via name. To cope with those cases all references have to be followed.

The search algorithm can therefore be described as follows:

1. Search for a `_PIC` method and fail if it is not present. The `_PIC` method is needed to tell ACPI that the OS switched from the default PIC to the I/O APIC mode. The absence of it is a good indicator that the ACPI table is too old to have support for an I/O APIC.

2. Search for the next `_PRT` method. If nothing is found, finish.

3. Search for the `_SEG`, `_BBN`, and `_ADR` methods of the enclosed bridge device. These methods return the PCI segment, bus, and device address respectively. They specify to what PCI bridge the IRQ mappings belong.

4. Search for IRQ mappings directly contained within the `_PTR` method.

5. If nothing was found, search recursively within every object referenced by the `_PTR` method.

6. Goto step 2.

5.4.4 Evaluation

Python Prototype

The search algorithm was first prototypically implemented in Python by using its regular expression library to match the patterns depicted in Figure 5.8. This prototype was used to evaluate the heuristic against 851 real-world DSDTs from a discontinued DSDT repository [SFA]. The oldest entries in this repository date back to 2003 whereas the latest ones were added in November 2007.

```
bool match_seg(char *res) {
  for (unsigned i=0; i < 4; i++)
      if (!((res[i] >= 'A' && res[i] <= 'Z')
           || (res[i] == '_')
           || (i && res[i] >= '0' && res[i] <= '9')))
        return false;
  return true;
}
```

Figure 5.10: The simplicity of the regular expressions made a C++ implementation easy. This function matches the SEG regular expression from Figure 5.8.

Figure 5.9 shows the results. From the original 851 DSDTs 13% or 108 had compile errors because they were submitted in the wrong format such as hex dumps or C-code. Around 32% of the DSDTs had *no I/O APIC*. They either didn't include a `_PIC` method or they simply ignored it. This left 456 *good* cases, and 17 ones where the heuristic failed. Please note that the database contains duplicates as users were uploading updated versions. From the 456 DSDTs only 190 have a unique vendor and version number.

A manual inspection of the 17 failed ones showed that these cases rely on PCI IRQ routers to forward IRQs not only to the PIC but also to the I/O APIC. This is possible by connecting an unused router pin to the I/O APIC pins that are normally unavailable in PIC mode. Fortunately the BIOS configures the IRQ routers for PIC mode to be DOS compatible. Thus by falling back to the PIC interrupt numbers from the PCI configspace, these 17 cases can be handled as well. Because the routers are not used in this case, more interrupt lines have to be shared.

In more than 95% of the cases the heuristic extracts the IRQ routing successfully from AML. A simple fallback results in a quality degradation for the remaining cases but not in a failure. In summary, the approach is successful in all tested cases.

C++ Version

Because the prototype proved the feasibility of the approach, a C++ version was implemented using roughly 300 lines of code. Compared to the prototype, it does not rely on a regular expression library anymore. Instead it implements the pattern matching directly in C++ as depicted in Figure 5.10. Furthermore, it resolves object references more accurately and searches for IRQ mappings recursively. Finally, it can extract the values needed to put the platform into S3 sleep states.

To evaluate the heuristic also with more recent DSDTs, various sources were crawled. Most notably are the Ubuntu bug database[10] and the Bugzilla of the Linux kernel[11]. Furthermore several forums were scanned for submitted DSDTs[12]. These sources increased the number of available DSDTs to over ten thousand. The collection now includes 2845 unique DSDTs[13] covering the last 15 years. Figure 5.11 shows how the C++ implementation handles these cases. I omitted percentage values in the figure because the collection is not a good representative for the machines a general purpose OS will run on. Instead

[10] `http://launchpad.net`
[11] `http://bugzilla.kernel.org`
[12] E.g. `http://tonymacx86.com`
[13] Two ACPI tables are considered the same if they share the OEMID, OEM Table ID, and the OEM Revision.

	Files	Unique
good	8587	2293
IRQ router	664	227
no I/O APIC	645	298
multiple	207	50
broken	155	63
Sum	10258	2845

Figure 5.11: The C++ implementation classifies more than ten thousand files from the DSDT collection.

	Number	Percent	Unique
good	82,666	87.6	2,097
IRQ router	8,566	9.1	178
no I/O APIC	2,146	2.3	185
multiple	711	0.8	28
broken	321	0.3	41
Found	94,405	100.0	2,525
Missing	15,113	16.0	4,563
Sum	109,518		7,088

Figure 5.12: An unbiased distribution of DSDTs were taken from Ubuntu bug reports. The C++ implementation handles most of the cases correctly.

it is biased towards older (*no I/O APIC*) and defective DSDTs (*multiple*, *broken*) due to the sources these files were retrieved from.

The Ubuntu bug database available through `http://launchpad.net` proved to be a valuable source for an unbiased distribution of current hardware. The Linux boot messages, which are attached to hundred thousands of bug reports, also mention the DSDT the platform is using. Figure 5.12 shows how the C++ implementation handles the DSDTs referenced this way. Even if the DSDT collection covers only a third of the seven thousand unique tables mentioned in the boot messages, around 85% of the entries could be successfully classified.

The heuristic extracts the IRQ routing in seven out of eight cases (*good*). In 9.1% an *IRQ router* was used whereas 2.3% of the cases had *no I/O APIC* support. Both cases can be handled by falling back to the PIC interrupt numbers from the PCI configspace, as mentioned in the prototype evaluation. The DSDT defines *multiple* entries for the same IRQ line in 0.8% the cases. To use these defective tables as well, an OS can implement the same workaround as Linux and just ignore all but the first entry.

Altogether the heuristic and the two workarounds successfully return the IRQ routing of a platform in 99.7% of the cases. The remaining 0.3% (*broken*) cases are caused by 41 different DSDTs, which rely on AML features not detected by our heuristic. These cases fall into three categories:

1. Include superfluous entries in `if (Zero)` sections or reference the routing through an `Alias`.

2. Select the IRQ routing information through the ID of a PCI device or through an ACPI variable overridden by an SSDT.

3. The IRQ routing depends on the BIOS revision or a proprietary MMIO or PIO

register.

Whereas the heuristic could be extended to cover the first and potentially even the second category, it would surely fail to handle the third one. However, as the number of broken DSDTs is relatively low[14], an OS could just include a few quirks to support all the known cases.

5.4.5 Summary

Previously, there were only two ways of handling ACPI in an OS: either ignoring it or using a full featured AML interpreter. I developed a third approach between these two extremes. By using a heuristic to extract only the information needed from AML, the complexity of the implementation can be drastically reduced by more than two orders of magnitude from approximately 50 KSLOC to 300 lines of C++ code. The evaluation has shown that the heuristic together with a small number of quirks can provide the platform specific routing of PCI interrupts on all known platforms.

5.5 Conclusions

TCB constraint systems need a boot stack much smaller than today. In this chapter, I have presented three approaches to reduce the TCB impact of the boot stack:

1. I decomposed the bootloader. A better design and several implementation improvements reduced its size by more than an order of magnitude.

2. I showed that Trusted Computing based on a DRTM can remove around 35 KSLOC from the TCB. However, several software and hardware challenges need to be solved until this technique is as secure as it aims to be.

3. I introduced a heuristic to reduce the ACPI table parsing code by more than two orders of magnitude.

These results can be employed in two ways to shrink the TCB impact of the x86 boot stack from 120 KSLOC to approximately 35 KSLOC: One can use the platform initialization code from Coreboot (See Appendix A.4.1) together with the disk and input drivers from the NOVA user-level environment (§2.2.3) to run the decomposed bootloader developed in Section 5.2. Alternatively, one can keep the normal firmware as well as the legacy bootloader but use Trusted Computing with a DRTM to limit its TCB impact. Relying on the ACPI heuristic is useful in both cases.

To further reduce this size one should improve the platform initialization as well as the verification code and reduce the complexity of the TPM. Both tasks are left to future work.

[14] Linux v3.10 ships with 99 PCI quirks on x86.

Chapter 6

Conclusions

In this work I aimed to increase OS (operating system) security by significantly reducing the security-critical part of it, commonly called the TCB (Trusted Computing Base). As I argued in the introduction, a single defect in the OS exploited by an attacker can give full control over all the software running on the machine. Unfortunately, current operating systems consist of several million lines of code. For example, a minimal cloud-computing scenario requires 1.2 MSLOC, whereas the typical case will be around 14.5 MSLOC. These numbers indicate that the OS is too large to be defect free, as even "good quality software" comes with "1 defect per every 1,000 lines of code" [Cov14]. Reducing the size of the OS and with it the number of defects in it, therefore increases the overall security of a platform.

By following a holistic approach and improving several system layers I could shrink the TCB below 100 KSLOC. We started with a new OS called NOVA, which can provide a small TCB for both newly written applications but also for legacy code running inside virtual machines. Virtualization is thereby the key technique to ensure that compatibility requirements do not increase the minimal TCB of our system. The main contribution of this work is to show how the VMM (virtual machine monitor) for NOVA was implemented with significantly less code without affecting the performance of its guests. Additional contributions towards a smaller TCB were made in debugging, compiling, and booting the operating system.

Figure 6.1 compares the size of our system with the *minimal* and *typical* configuration measured in Appendix A. The largest improvements could be achieved in the OS and VMM layer, compared to the minimal configuration from Appendix A. The compiler is

Name	Thesis	Minimal	Factor	Typical	Factor
OS	21	830	40	9400	448
VMM	9	220	24	2000	222
Debugging	3	30	10	1000	333
Compiler	4	20	5	1800	450
Firmware	33	55	2	150	5
Bootloader	2	15	8	175	88
Sum	72	1170	16	14525	202

Figure 6.1: The TCB in KSLOC of a virtual machine when combining all achievements into a unified system compared to the minimal and typical configuration shown in Appendix A.

on the top spot when considering the typical case instead. The smallest reduction was possible at the firmware level. It now accounts for around half of the code in the TCB even though it is only needed to initialize the platform. Future research should tackle this area.

Compared to the 1.2 MSLOC for the minimal configuration, a sixteenfold TCB reduction could be achieved on average. This increases to two-hundred fold if the typical case is considered instead. A unified system including all improvements needs only 72 thousand lines of code for the TCB of a virtual machine when assuming that all layers can be compiled with our own B1 compiler. However, rewriting the whole system including hypervisor, VMM, and firmware in the B1 programming language is left to future work.

6.1 Techniques

As predicted by [Bro87], I have not found the *silver bullet*, which will reduce the TCB on all layers. Instead, several design and implementation techniques had to be employed to achieve the desired reduction. Most notably techniques are:

Architecture The NOVA OS Virtualization Architecture reduces the TCB for both applications and virtual machines by following the microkernel approach. It separates the hypervisor and VMM, uses one VMM per virtual machine, and runs device drivers in a decomposed user-level environment (§2.2.1).

Component Design Building the user-level environment as well as the VMM from loosely coupled software components allows to specialize the TCB to the requirements of the hardware and the virtualized OS. Furthermore, it enables component sharing between independent programs (§2.2).

Code Generation A large portion of the instruction emulator was generated automatically thereby reducing the amount of hand-written code (§2.4).

BIOS Virtualization Virtualizing the BIOS inside the VMM is simpler and faster than emulating it inside the virtual machine (§2.5).

Debugging Memory access and signaling is sufficient for sophisticated debugging of physical and virtual machines, which minimizes the debug code in the OS (§3.2). Moreover, Self-Modifying DMA over Firewire provides this low-level interface without any additional runtime code (§3.3).

Compiling Simplifying the programming language, reusing an existing syntax, and implementing it in a high-level language reduces the TCB impact of the compiler (§4).

Boot One can use the platform initialization part of the firmware together with a decomposed bootloader to reduce the TCB of the boot stack (§5.2). Alternatively, one may use Trusted Computing based on a DRTM to remove legacy bootloaders and most of the firmware from the TCB (§5.3).

Heuristic The ACPI code in the OS can be minimized with a heuristic (§5.4).

6.2 Lessons Learned

In the course of this work I have learned several lessons that can be useful when designing and implementing other systems with a small TCB:

- The key principle towards a smaller system is **divide-and-conquer**. One should breakup large software in manageable entities that are as small as the runtime overhead permits. The NOVA OS Virtualization Architecture (§2.2.1), the design of the VMM (§2.2.4), debugger (§3.2.2), and bootloader (§5.2.2) are four cases where this principle was successfully applied.

- One can rely on fine-grained **software components** to reuse functionality multiple times and to specialize the TCB to the requirements of the application and hardware platform. This technique was most prominently used in the VMM (§2.2.4) and in the NOVA user-level environment (§2.2.3) to encapsulate device models and device drivers. The bootlets implementing a subset of traditional bootloader functionality in a replaceable component are another example (§5.2.2).

- One should follow a **holistic approach** and look beyond the single entity to improve. The VMM would not have the same impact on the TCB without running them on the microhypervisor (§2.2.1). Similarly, the compiler would be larger when preserving the semantics of the language (§4.1).

- One should favor **specialized solutions** over general ones. The read-only filesystem code in the bootlets is less complex than a read-write implementation (§5.2.3). Similarly, extracting only the necessary information from the ACPI tables allows to use a simple heuristic instead of a full-featured AML interpreter (§5.4).

- One should **not port** existing code that was written with different goals in mind. A reimplementation will often lead to a much smaller codebase (§2.3.2).

- One should not only improve the implementation but also the **interfaces**. Having a smaller debugging (§3.2) as well as a simpler filesystem interface (§5.2.3) reduced the necessary lines of code to implement them.

- New **hardware features** can significantly lower the software complexity. For example, hardware support for CPU virtualization made a large and complex binary translation system unnecessary. Similarly, nested paging removed the need for a more efficient shadow paging implementation (§2.2.2). Moreover, having a platform with Firewire connectivity allows to debug the system without runtime code (§3.3). However, moving functionality from the software to the hardware level will not reduce the TCB per se (§5.3).

- Finally, **novel ideas**, like reverse engineering the CPU (§2.4) or virtualizing the BIOS inside the VMM (§2.5), might be necessary to reduce the TCB below a certain size.

6.3 Future Research Directions

Besides the points already mentioned in the previous chapters, future research should target the following areas:

- The smallest TCB reduction could be achieved at the firmware level, which remains the largest part of the TCB. Future research should investigate how a firmware like *coreboot* can be further improved.
- Recursive virtualization can counteract the steady software inflation. Even though we have already proposed an efficient design in [KVB11], an evaluation of this technique is still missing.
- NOVA currently depends on a small set of platform drivers, which allows it to run on many x86 systems but surely not on all of them. With dedicated driver VMs we could reuse legacy device drivers and thereby minimize the required development effort. One has to solve the minimization and packaging issues to make this technique feasible and deploy drivers together with their original OS in a single image of a reasonable size.
- The inherent complexity of the OS could be further reduced by improving the hardware, namely CPUs, chipsets, and I/O devices. One should research how the combined complexity of hardware and software can be reduced while keeping the necessary functionality and performance.

Finally, combining the achievements of this thesis with previous work such as [SKB+07, Fes09, Wei14] into a general-purpose operating system with a small TCB would be a great research direction.

Appendix A

TCB of Current Systems

Publications aiming for a small TCB (Trusted Computing Base) usually report sizes only for certain parts of a system. The literature therefore misses a consistent overview of the size of a contemporary operating system, which supports virtual machines. In the following, I aim to close this gap by quantifying the TCB size of a typical operating system that is needed for a cloud-computing setting, including its kernel/hypervisor, user-level environment, VMM (virtual machine monitor), debugger, boot stack, and compiler. I will also try to report lower bounds since it is often assumed that a particular software package like the Linux kernel can be made significantly smaller through careful configuration [TKR+12, SPHH06, SK10a].

I use SLOC (source lines of code) as the measurement unit for the TCB size. A SLOC is basically a single non-empty line in a source file, which is not a comment [Whe]. Even if there are programs that are highly obfuscated [IOC] or that might be explicitly circumvent this measurement, most programs are written in a straight forward way and SLOCCount reports useful results. Furthermore, the number of SLOC corresponds to the lines a developer has to study to understand how the program works and whether or not it is correct. Finally, measuring the lines of code of a program might not be the perfect code-complexity indicator. It is nevertheless a very intuitive one.

For completeness reasons I sometimes include closed source programs as well. If possible I rely on numbers reported in the literature. As a last resort I developed an heuristic to estimate the SLOC from the binaries.

A.1 Estimating Lines of Code from the Binary

Getting accurate SLOC numbers for closed-source programs is often impossible. Either this information is kept private by the vendor or the binary is so old that the software repository was lost long ago. Estimating the size from the binaries might be the only way to get an approximate result in these cases.

It is common belief that different architectures, programming languages, compilers, and its optimizations as well as the individual style of the programmer will make such an estimation very inaccurate. Consequently, there is very little literature about this technique. The only exception seems to be [Hat05] where Hatton tries to estimate the size of Windows from a small sample application of 10 KSLOC. Unfortunately, his approach seems to be too simple and his surprisingly accurate results lead to more questions than giving answers.

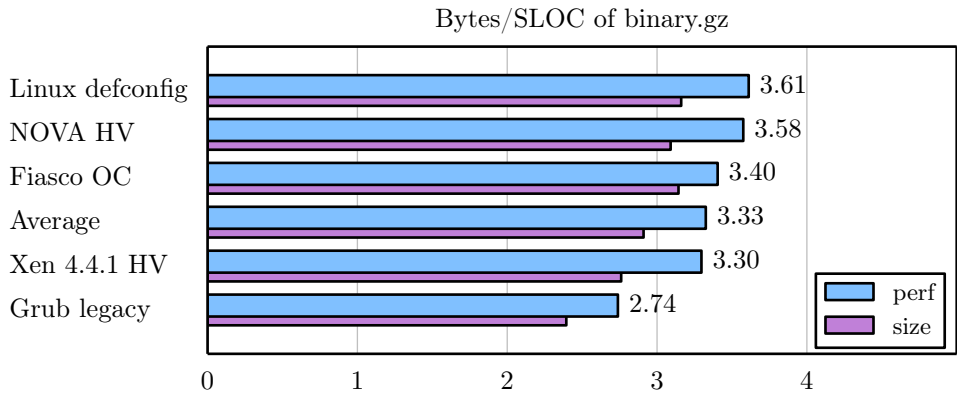

Figure A.1: Bytes per SLOC in a gzip compressed binary for different low-level software.

Idea

I developed a new approach to predict the code size of closed-source programs with sufficient accuracy, by refining an idea from [Kau07b]. I use the size of the compressed binary to approximate the Source Lines of Code that were used to build them. To increase the accuracy I used the following set of restrictions to exclude or at least minimize the influence of architecture, programming language, and compilers:

- The binaries are written in similar languages such as C and C++.
- The binaries are compiled for a particular architecture with similar compiler options.
- The binaries are developed for the same environment and perform similar work. This excludes many apples with oranges comparisons. For instance estimating the size of a Windows application from a bootloader or device driver.
- The binaries are stripped to make sure that symbols and debug information are removed. Enabled debugging options can more than double the binary size.
- The binaries are compressed to reduce the effect of code alignment, padding, and other optimizations such as loop unrolling.

These restrictions cannot fully exclude programmer related deviations. However, programming style is not an issue here, as the prediction does not try to reveal the exact source code size of a project. Instead, it should return how many lines of code would be needed if a certain program would be written in a style similar to existing software.

Evaluation of the Approach

In the following, I will shortly evaluate the approach by applying it to five different open-source system software projects. I compiled two different versions for each binary with gcc-4.6, namely a size optimized version (`-Os`) and a performance optimized one (`-O2`). This should reveal how large compiler optimizations influence the code size. I measured the size of the compressed binaries and compared it with the SLOC that were used to produce it. I relied on `gzip` for compression. However I observed later that a better compressor such as `xz` will result in even smaller deviations.

Figure A.1 shows that every SLOC leads on average to 3.33 gzip-bytes for the performance optimized version. The low deviation of +9% and -18% is remarkable here. This shows that low-level system code in open-source projects tend to have the same code density. The size optimized versions are approximately 14% smaller. This clearly reveals that compiler optimizations have a small but significant influence. However, the effect of optimizations could be taken into account because a size-optimized compilation can be relatively easily detected from the disassembled binary and the numbers can be adjusted accordingly. These results reveal that, under certain assumptions, binaries can be used to estimate the number of SLOC with an error rate that is below 20%. A more detailed comparison might allow to relax some of the restrictions in the future. In the following, I will use the result of this short experiment, namely the 3.33 gziped-bytes per SLOC, to estimate lines of code for closed-source binaries.

A.2 Virtualization

A.2.1 Hypervisor

Figure A.2 shows the size of different hypervisor implementations. I either measured the SLOC myself, if the source was available (Xen, KVM), got the number from the literature (ESX 2.5 [ESXa], Hyper-V [LS09]), or I estimated it from the binaries (Hyper-V, ESX 5.0).

This estimation is sufficiently accurate. According to the literature [LS09] the Hyper-V hypervisor consists of 100 KSLOC of C code and 5000 lines of assembler. The approximation from the compressed binaries is only 20% below this number. It would be even more accurate if the code for AMD-V processors could be counted as well, which is not included in the Intel version of the binaries.

The measured hypervisors are between 30 and 750 KSLOC large. The wide spectrum mainly results from different system architectures. KVM as the smallest hypervisor heavily relies on its host operating system. It reuses for example drivers as well as resource management code from Linux and can therefore be much simpler than other stand-alone hypervisors. Hyper-V, the second smallest HV, supports Windows in a paravirtualized `dom0` to run its device drivers and to implement management software. Xen has a paravirtualized dom0 as well but also supports paravirtualized user domains (domU). This feature makes it more complicated than Hyper-V. Finally, ESX implements operating system functionality inside the hypervisor to run drivers and applications on top of it. Whereas ESX is the largest hypervisor, it does not require a Linux or Windows instance as support OS (operating system). Its size is not as large as it seems to be. In fact, ESX is only 40% larger than the 530 KSLOC of KVM and a minimal Linux, which is the lower bound for a hypervisor and support OS altogether.

Another result from the hypervisor size measurements is the observation that virtualization software also grows over time. ESX v5.0 is 3.5 times larger than ESX v2.5 six years ago. Similarly, Xen has quadrupled its size in the same time frame, as shown at the right of Figure A.2. This development is similar to the increase of operating system software as observed on Linux and Windows.

A.2.2 VMM

I measured the size of three virtual machine monitors, namely Qemu, ESX, and VirtualBox. The numbers are presented in Figure A.3.

Name	binary.gz	SLOC
KVM/Linux 3.0.4		28,670
Hyper-V 2.0 Intel	271 KB	83,000
Xen 4.1.1		182,935
ESX 2.5		200,000
ESX 5.0 Kernel	2,420 KB	744,000

Version	Date	SLOC
Xen 2.0.7	Aug 2005	48,286
Xen 3.0.4	Dec 2007	90,504
Xen 3.2.0	Jan 2008	127,648
Xen 3.3.0	Aug 2008	138,892
Xen 3.4.0	May 2009	152,787
Xen 4.1.1	Jun 2011	182,935
Xen 4.3.1	Oct 2013	214,114
Xen 4.4.0	Mar 2014	220,425

Figure A.2: Hypervisor Size in SLOC.

Name	binary.gz	KSLOC
Qemu x86_64		221
Qemu Repository		582
Qemu x86_64 w/ libs	6,192 kB	1,900
Debian qemu-kvm	15,228 kB	4,573
VirtualBox Repository		1,798
ESX 5.0 VMM	6,998 kB	2,150

Figure A.3: Size of the x86 VMM.

Qemu is the smallest VMM. The repository, which includes support for different architectures and platforms, consists of 580 KSLOC. Because not all of this code contributes to a binary for a certain architecture on a given platform, I measured only the part that was used to compile a binary on Linux for x86_64 processors. I relied on the access time (*atime*) feature of the filesystem to decide what files contributed to the binary. This revealed that only 220 KSLOC from the 580 KSLOC were actually referenced when omitting other architectures and many optional features. However, the compilation returned a dynamically linked binary, which needs 49 external libraries to run. If I also take the dynamically linked libraries such as the `libc`, `libglib`, and `libsdl`, into account, Qemu needs more than 6 MB of compressed binaries. This corresponds to nearly 2 MSLOC. A similar increase can be observed when installing `qemu-kvm` and the 63 packages it requires onto a basic Debian 7.0 *Wheezy*. This operation adds around 15 Mb of compressed binaries or 4.5 MSLOC to the system.

The VirtualBox repository consists of 1.8 MSLOC. This is much larger than the Qemu sources due to a couple of reasons. First, the tree includes hypervisor, VMM, and management functionality, which cannot be easily separated. Second, a large part (700 KSLOC) of the source accounts for guest additions. These are paravirtualized libraries to reduce virtualization overhead. Thus, less than 1 MSLOC will be part of the VMM, if we do not count any external libraries. Finally, VirtualBox allows to run virtual machines on CPUs without hardware support for virtualization.

The SLOC for the VMM of ESX 5.0 cannot be directly measured, as no source code is publicly available. From the compressed binaries I have estimated a size of 2.1 MSLOC. ESX has the biggest VMM, most likely because it implements the largest feature set among the three VMMs. It virtualizes for instance graphics with 3D acceleration and allows to migrate virtual machines between different hosts. Furthermore, the ESX binary, which includes library code, is not much larger than Qemu plus its external libraries.

In summary, a x86 VMM is between 220 KSLOC and 4.5 MSLOC large. The typical VMM consists of approximately 2 MSLOC.

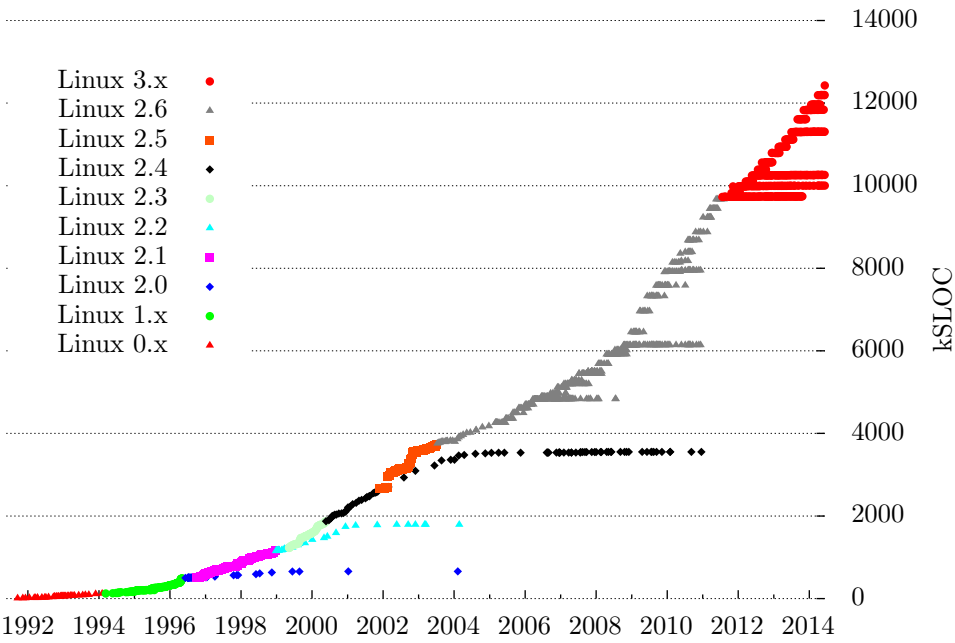

Figure A.4: Evolution of the Linux kernel size between September 1991 (v0.0.1) and June 2014 (v3.15).

A.3 Support OS

Linux is used as Host OS in most virtualization projects. First, I show how the code size of the Linux kernel evolved over time. I then answer the question how small a minimal Linux kernel can be today. Finally, I estimate the size of a Linux distribution and put these numbers in relation with the size of Windows.

A.3.1 Evolution of an OS

I measured the size of the whole source tree for all releases of the Linux kernel from the first release in September 1991 up to the latest version (3.15) from June 2014 [LX, LXO]. The sizes of around 1500 different releases are shown in Figure A.4 where each version is represented by a single dot. Most importantly the code base of Linux has grown from eight thousand in 1991 to more than twelve million lines today. One can see from the graph that the slope of the curve increased in 2008 with the introduction of the new development life cycle. Furthermore, one can distinguish the development and the maintenance phase for each major version. For example Linux 2.4 was rapidly developed between 2001 and 2004. After that very little code was back ported from newer versions. The kernel was just maintained for another seven years.

A steady growth of the code base seems to be a characteristic of many actively developed system projects. Debian, Windows, and Xen exhibit the same behavior as shown in the Figures A.6, A.7, and A.3. In comparison I discuss in Section 2.7.1 how the size of the NOVA system might evolve in the future.

Config	SLOC	%	Binaries	Description
allnoconfig	362,771	3.7	762 kB	All config options are disabled.
alldefconfig	471,030	4.8	1,151 kB	The default for all config options.
defconfig	1,330,650	13.7	4,693 kB	The default config for x86.
oldconfig	1,622,700	16.7	5,565 kB	T420 laptop config w/ 161 modules.
debian	5,983,958	61.5	31,588 kB	Debian config w/ 2774 modules.
allyesconfig	8,266,554	85.0	51,989 kB	All config options are enabled.
whole tree	9,729,144	100.0		

Figure A.5: SLOC and binary sizes of different configurations for Linux 3.0.4.

A.3.2 Minimal Linux

The previous measurement considered the whole source code of Linux. It does not reveal how small a minimal Linux can be. In [SK10a] we assumed that Linux can be reduced to 200 KSLOC through careful configuration. Because this was only a very rough estimate, I'd like to give a more accurate figure here.

The feature set compiled into the kernel has probably the most impact on the code size. Thus, I distinguish six different configurations: Two minimal (*allnoconfig*, *alldefconfig*), three typical (*defconfig*, *oldconfig*, *debian*), and the maximum configuration (*allyesconfig*).

I measured the latest kernel version at that time, namely Linux 3.0.4, on the x86 architecture. Whereas older versions will be smaller, one will likely need the newest drivers and the newest kernel features to get the best performance from the latest machines. To decide what code was used during kernel compilation, I relied on the access time (*atime*) feature of the filesystem. I simply assumed that all files accessed during compilation have contributed to the kernel size. Note that this assumption leads to slightly overestimated results because all code in $\#ifdef$ sections, normally filtered out by the preprocessor, is counted as well[1].

Figure A.5 shows the results. The minimal configuration of Linux, which consist only of the core functionality such as the system calls or the block layer, already accounts for more than 360 KSLOC. A slightly larger configuration, which additionally includes platform drivers such as ACPI, is already 110 KSLOC larger. However, this does not include any network protocol, disk driver nor a single filesystem. A typical Linux configuration, which includes these important features, is around 1.5 MSLOC large. A distribution kernel like the one from Debian that enables almost all drivers, is four times larger. An *allyesconfig* that enables all config option includes 85% of the whole Linux tree. The remaining code is architecture dependent.

In summary, a minimal Linux configuration to support virtual machines will not be smaller than 500 KSLOC without large modifications to the source code. The typical case will be three times this size.

A.3.3 Linux Distribution

The kernel alone is not sufficient to run a Linux system, a user-level environment is needed as well. Using BusyBox [Wel00] statically compiled with uclibc [And] leads to a minimal environment. Debian ships such a BusyBox binary that supports 219 commands within approximately 300 KSLOC.

Measuring a typical system is difficult, as it highly depends on the required features. To get a lower bound, I bootstrapped different Debian releases and measured the size of

[1] Using `strace` to trace the `open` system calls led to the same results albeit with more overhead.

Version	Date	Packages	Download	Binary.gz	KSLOC
2.2 Potato	08-2000	81	17,340 kB	6,740 kB	2,024
3.0 Woody	07-2002	100	21,412 kB	8,836 kB	2,653
3.1 Sarge	06-2005	122	33,796 kB	13,694 kB	4,112
4.0 Etch	04-2007	124	39,116 kB	16,688 kB	5,011
5.0 Lenny	02-2009	119	43,836 kB	19,596 kB	5,885
6.0 Squeeze	02-2011	116	50,004 kB	21,675 kB	6,509
7.0 Wheezy	05-2013	135	56,228 kB	26,023 kB	7,815
8.0 Jessie	TBD	159	52,080 kB	30,717 kB	9,224

Figure A.6: Lines of code for a basic Debian installation.

the installed programs and libraries[2]. The employed `debootstrap` tool installs only those packages tagged as required[3] or important[4]. On average 120 packages were installed, which corresponds to less than a percent of the available packages.

The results of these measurements are shown in Figure A.6. Interestingly, the size of a basic Debian system increases around 10% every year, which is significantly less than the "doubling in size every 2 years" of the whole Debian distribution reported in [GBRM+09]. Debian 2.2 *Potato* required only 2.0 MSLOC for it. The latest Debian 7.0 *Wheezy* release needs already 7.8 MSLOC or approximately two percent of whole release [Bro12]. The yet to be released *Jessie* is already 18% larger, even if it has reduced the download size by compressing the packages with `xz` instead of `gzip`.

In summary, a minimal Linux environment is 300 KSLOC whereas a typical Debian installation will be larger than 7.5 MSLOC.

A.3.4 Windows

Not all virtualization environments use Linux as support OS. Most notably Hyper-V relies on Windows for this purpose. In [SK10a] we assumed that a Windows Server 2008 would be larger than 500 KSLOC. In the following, I will show that this number is at least an order of magnitude to small.

I could not directly measure the size of Windows, due to its undisclosed source code. Fortunately, the lines of code for some Windows versions were reported in the literature. Figure A.7 shows the numbers presented in [Mar05]. However, these numbers have to be taken with a grain of salt. First, they are probably line counts and therefore higher than what would be measured with SLOCCount. Second, they are likely to cover the whole codebase of Windows, including helper programs such as `regedit.exe` that are not needed in a size-optimized installation. The 50 million lines of code for Windows Server 2003 can still be taken as an upper bound.

A lower bound can be found in a recent paper [PBWH+11]. The authors have built a Windows-compatible library OS to run unmodified Windows applications such as Excel or Internet Explorer. They reused 5.5 million lines of code from Windows. This number is a lower bound, as it does neither include a filesystem nor any device drivers.

In summary, a Windows based support OS for a virtualization environment will consists of 5.5 to 50 million lines of code. This is similar to a typical Linux installation, which

[2]Linux Distribution especially optimized for size like AlpineLinux or DamnSmallLinux include more functionality and are consequently larger.

[3]These are the binaries one needs to trust when using Debians package system. This includes programs like `bash, perl, gpg, dpkg, coreutils` and basic libraries like `libc, libstd++, libncurses`.

[4]This includes packages one can assume on a Linux system like `syslog, cron, traceroute`, and an editor like `nano`.

Released	Name	Size
1993 Jul	NT 3.1	4.5 M
1994 Sep	NT 3.5	7.5 M
1995 May	NT 3.51	9.5 M
1996 Jul	NT 4.0	11.5 M
1999 Dec	Win 2000	29 M
2001 Oct	Win XP	40 M
2003 Apr	Server 2003	50 M

Figure A.7: Lines of Code for different Windows versions according to [Mar05].

Name	SLOC	Description
TinyBIOS	6,478	Incomplete BIOS w/o ACPI and VGA support.
SeaBIOS	26,882	BIOS for Qemu 0.15 w/ limited platform init.
CoreBoot T60	29,492	Platform initialization code for a Lenovo T60.
Bochs BIOS	89,819	65% assembler. Includes Bochs VGABios.
Award BIOS	150,000	Assembler code from 1999.
HP nx6325	150,986	Estimated SLOC from the binary sizes [Kau07b].
CoreBoot v2	290,712	Open-source BIOS for more then 230 mainboards.
AMI UEFI	444,000	UEFI firmware from 2013.

Figure A.8: Size of the PC Firmware (BIOS or UEFI) in SLOC.

needs more than 9 MSLOC for kernel plus user-level environment.

A.4 Boot and Toolchain

A.4.1 PC Firmware

The size of most BIOS (Basic Input/Output System) implementations cannot be directly measured, as they are usually closed source programs. The only exceptions seem to be an old AWARD BIOS from 1999 and an AMI UEFI implementation that was leaked in 2013. Furthermore the source code of open-source BIOS implementations such as CoreBoot (formerly known as LinuxBIOS) or TinyBIOS and virtual machine BIOSs such as SeaBIOS or Bochs BIOS are available as well. To also report the size for a recent closed-source BIOS, I estimated the lines of code for an HP BIOS from the binary size previously reported in [Kau07b]. This result fits nicely into the measurements of the open-source implementations.

Figure A.8 lists the measured sizes. Note that the three smallest implementations are not feature complete. SeaBios for instance does not initialize the DRAM controller whereas CoreBoot does not include any BIOS runtime service. However, they provide enough functionality if they act together. In summary, a complete BIOS or UEFI implementation will be between 55 and 450 KSLOC large, depending on the feature set and the number of supported platforms. A typical BIOS like the one from HP consists of 150 KSLOC.

A.4.2 Bootloader

There are many bootloaders that can be used to load an operating system from disks or over the network. Their size heavily depends on the implemented feature set. Some bootloaders retrieve only hard-coded binaries whereas others allow the user to interactively

Name	SLOC
Minimal Linux	154
ELILO 3.14	15,259
LILO 23.2	21,428
GRUB2 2.02	302,592
kernel	11,037
linux16+multiboot	30,004
default	174,718
GRUB Legacy 0.97	27,576
Syslinux 6.02	366,359

Figure A.9: Bootloader Size in SLOC.

change its configuration. Some provide a graphical user interface whereas others print only diagnostic messages. Some bootloaders include device drivers, understand different filesystems, and speak different boot protocols whereas others simply call BIOS functions to get files from the network.

Figure A.9 shows that the size of x86 bootloaders can differ by three orders of magnitude. The *Minimal Linux* Bootloader [Plo12] needs only 154 Lines of assembly to retrieve the Linux kernel from fixed locations of the disk. However, its functionality is severely limited.

More advanced bootloaders, which understand the filesystem layout like *ELILO* [ELI12] or that let the user select different boot configurations like *LILO* [LIL11] are already two orders of magnitude larger. Unfortunately, both just boot Linux and cannot start a multiboot OS like NOVA.

GRUB2 [GRU14], the bootloader with the largest feature set and the most popular bootloader on Debian[5] consists of more than 300 KSLOC. However this shrinks to 30 KSLOC, when counting only the GRUB2 *kernel* and the seventeen modules required to boot both Linux and *multiboot* OSes. This is comparable to the 28 KSLOC of the older *GRUB Legacy* codebase, even though this includes support for more filesystems [GRU05]. A *default* GRUB2 installation on x86 consists of 175 KSLOC within 250 modules.

The *Syslinux* [SYS13] codebase, a collection of multiple bootloaders and tools, is even larger than GRUB2 as this number includes 165 KSLOC for the network bootloader GPXE. The size of the other parts cannot be as easily separated. However, its *core* alone is larger than 18 KSLOC.

In summary, a general purpose bootloader is between 15 and 300 KSLOC large. The 175 KSLOC for the default GRUB2 installation would be typical for a Linux system.

A.4.3 Debugger

The complexity and with it the size of a debugger highly depends on its architecture and functionality. See Figure A.10 for the size of different system debuggers. A special-purpose implementation inside a microkernel like Fiasco's JDB is around 30 KSLOC large whereas a Unix process debugger that also understands DWARF debug format like Path64 needs 50 KSLOC. Similarly, the Java debugger is reported to be 85 KSLOC large.

A general-purpose implementation that supports multiple architectures and even remote targets like GDB or LLDB will be more than 400 KSLOC large. The GDB size increases to over a million lines, if the required `bfd` and `opcodes` libraries are taken into account as well [LHGM09]. LLDB is in the same region if one counts the parts of

[5]According to the Debian Popularity Contest GRUB2 is installed in over 85% of the machines [DPC]

Name	KSLOC	Source
Fiasco JDB	30	from [Ste09]
Path64	50	`https://github.com/path64/debugger`
Java Debugger	85	from [LHGM09]
GDB	420	from [LHGM09]
GDB w/ libs	1,020	from [LHGM09]
LLDB	440	`http://llvm.org/git/lldb.git`
Valgrind	640	`svn://svn.valgrind.org/valgrind/trunk`
Frysk	335	`git://sourceware.org/git/frysk.git`

Figure A.10: The debugger can be as large as 1 MSLOC.

Name	KSLOC	Notes
LCC	18	v4.2 w/o tests [FH91]
Plan9	20	from [Tho90]
TinyCC	30	v0.9.5 w/o win+tests [Bel02]
LLVM	1,670	`http://llvm.org/git/{clang,llvm}.git`
GCC	4,300	v4.6.3 repository [GCC]
Open64	4,400	v5.0 w/ GCC 4.2 frontend [O64]

Figure A.11: A C compiler is between 18 KSLOC and 4.3 MSLOC large.

the LLVM (820 KSLOC) and CLANG repositories (850 KSLOC) is depends on. Other projects that follow a different debugging approach like Valgrind or Frysk, are also several hundred KSLOC large.

In summary, 30 KSLOC seem to be a lower bound for a special-purpose debugger whereas the 1 MSLOC of GDB will be the typical case in most systems.

A.4.4 Compiler

A contemporary operating system will ship programs written in different programming languages. However, most of the code is still written in C [Bro12]. I have therefore examined only C compilers. Figure A.11 shows the sizes of several small C compilers and of multi-purpose compiler projects like GCC and LLVM.

The GCC repository is around 4.3 MSLOC large. A significant portion to this huge size can be attributed to the support of multiple languages like JAVA, ADA, and Fortran. To get a more accurate number I have build the OSLO bootloader [Kau07b] with a GCC toolchain and estimated the size from the binaries. This involves tools that are approximately 1.8 MSLOC large as shown in Figure A.12. Another 600 KSLOC are needed for several libraries. The Open64 compiler, which includes the GCC v4.2 frontend, has a similar size. The LLVM repositories for the C compiler frontend (*clang*) and code generation/optimization (*llvm*) are with 1,670 KSLOC less than half their size.

LCC is the smallest compiler for ANSI C. It is around two orders of magnitude smaller than one of the multi-purpose compilers. The Plan9 C compiler has a similar size. TinyCC is around 50% larger due to many C99 features it supports.

In summary, a minimal C compiler can be as small as 18 KSLOC whereas the 1.8 MSLOC of GCC will be the typical case.

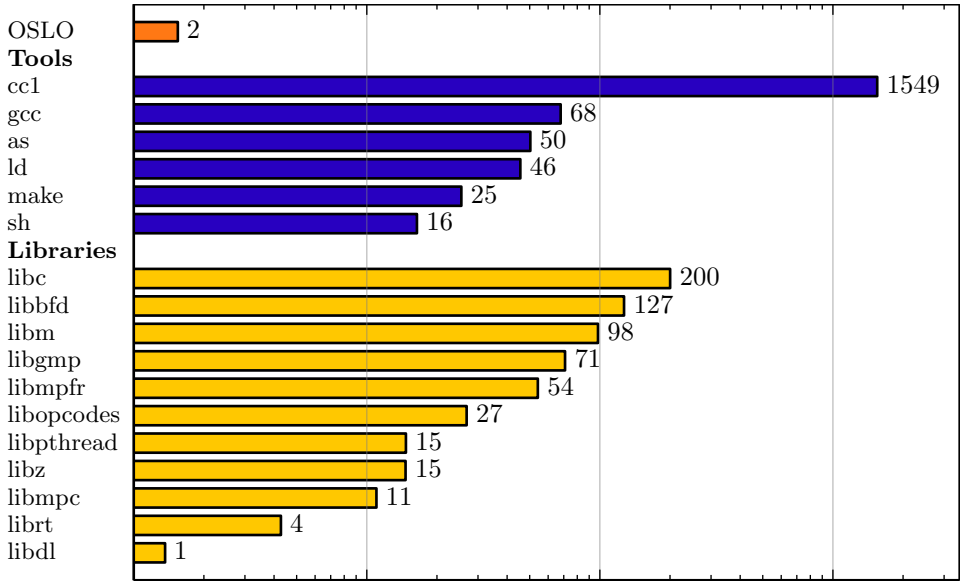

Figure A.12: Tools and Libraries with 2.4 MSLOC are needed to compile a small bootloader like OSLO with a GCC toolchain. The numbers are estimates from the binary sizes.

A.5 Summary

I measured the TCB size of a contemporary operating systems that supports hardware assisted virtual machines on x86. Figure A.13 presents the sizes for all system layers for a minimal and a typical configuration. It shows that the TCB for a virtual machine in current implementations will be between 1.2 MSLOC and 14.5 MSLOC large. This is at least two times more than what we assumed in [SK10a]. Furthermore, the constant software growth, which can be observed nearly on all of these layers, will increase the TCB beyond these values in the future.

Name	Minimal	Typical
BIOS	55	150
Bootloader	15	175
HV+Kernel	530	1,600
Environment	300	7,800
VMM	220	2,000
Compiler	20	1,800
Debugger	30	1,000
Sum	1,170	14,525

Figure A.13: Overall TCB size in KSLOC for an OS that supports virtual machines in a minimal and typical configuration.

Appendix B

Glossary

ACPI short for Advanced Configuration and Power Interface [ACP] standardizes the interaction between hardware, firmware, and operating system. It allows an OS to enumerate platform resources, change resource allocations and perform power-management tasks. 16, 17, 49, 52, 67, 69, 135–137, 147, 149, 150, 152–159, 162, 163

AHCI short for Advanced Host Controller Interface, is a standardized interface for a device driver to access SATA devices. 48, 49, 53, 73, 74, 82

Binary Translation allows to run a virtual machine on a non-virtualizable architecture by translating the instruction stream before it is executed. On x86 superseded by hardware-assisted virtualization. 20, 22–24, 163, 178

BIOS short for Basic Input/Output System, is the traditional firmware of a PC. After being loaded from an EEPROM, it initializes the platform including the memory controller and all the devices that are necessary to run a bootloader as well as the operating system. A *virtual BIOS* performs the same task for a virtual machine. A BIOS can be extended with Option ROMs. 16, 20, 23, 33, 38, 39, 41, 53, 54, 67–71, 73, 85, 86, 100, 136, 141, 142, 144, 145, 149–151, 154, 157, 158, 162, 163, 172, 173, 180, 182, *see* UEFI

CR3 short for control register #3, is a special purpose register of the x86 CPU pointing to the current pagetable hierarchy. 104–106, 108

Device Model is one part of a VMM to provide the illusion to the guest that it runs alone on the platform, by emulating the behavior of their physical counterpart. 16, 20, 23–26, 32, 38, 39, 41–44, 46–51, 53–56, 69–71, 85, 86

DMA short for direct memory access, is the ability of a device to directly write and read data from main memory. This is used for instance by modern network cards to store received packets in RAM without involving the CPU. The risk that untrustworthy drivers can takeover the host OS via DMA can be mitigated by an IOMMU. 15, 27, 28, 32, 39, 42, 53, 54, 56, 64, 83, 97–101, 150, 162, 179

DRAM short for Dynamic Random Access Memory, is volatile memory in a computer that needs to be periodically refreshed. 54, 70, 72–74, 136, 151, 172

DWARF is a standardized data format that helps a debugger to understand a compiled program by describing its symbols, data types, and even the stack layout at each instruction pointer [Eag07, DWA10]. 48, 92–95, 107, 127, 134, 173

EEPROM short for Electrical Erasable Programmable Read-Only Memory, is non-volatile memory keeping stored data even if the system is powered down. 136, 151, 177, *see* DRAM

ELF short for Executable and Linkable Format, is a file format used by many operating systems to hold executable machine code [ELF95]. 37, 60, 92, 93, 117, 121, 122, 143

Firewire also known as IEEE-1394, is a serial bus interface that connects 63 devices with up to 3200 Mbps [IEE08]. 11, 17, 56, 87, 93–103, 108, 143, 162, 163, 179

Firmware is software embedded in a device to initialize it or to provide its external interface, for instance in a printer or a network card. 16, 21, 135–138, 146–153, 159, 162, 164, 177, 180, 181, *see* BIOS & UEFI

GDT short for Global Descriptor Table, is a data structure consulted by the x86 CPU to map code and data segments to protection rings. It may also include TSS descriptors and call gates to securely transfer control between rings. 93, 107, 179

Guest An operating system running inside a VM is called a guest of the hypervisor. 3, 15, 16, 20, 22, 23, 25–27, 29, 30, 38–44, 46–54, 57, 64–71, 73, 74, 76–78, 80–85, 88, 168, 177–182

GUI is an acronym for graphical user interface. 29, 53, 56, 93, 95, 137, 138, 140

Hardware-Assisted Virtualization is a platform virtualization approach that neither modifies the guest OS (paravirtualization) nor uses binary translation. Instead it relies on a CPU extension such as Intel VT-x and AMD V [UNR+05, SVM05] to add the necessary traps from the guest environment to the hypervisor to an otherwise unvirtualizable processor. 23–25, 75, 77, 141, 177, 180

Hardware-Task Switch can be invoked on a x86 CPU via an interrupt, an exception or special `CALL` and `JMP` instructions. The CPU saves the current state in the active TSS and loads the next processor state from the invoked TSS. 57, 104, 106, 181

HPET short for High Precision Event Timer, is a platform timer device present in recent PCs [HPE04]. 52, 56, 73, 81, 82, 180

Hypervisor runs in kernel mode to multiplex the host machine so that multiple guest operating systems and in the case of NOVA also native applications can execute concurrently. 15, 20, 21, 23–33, 35–37, 41, 43, 48, 49, 71, 76, 77, 80–82, 84–86, 88–91, 94, 95, 100, 104, 106–108, 162, 165, 167, 168, 178, 180, 182, *see* VMM

I/O APIC short for I/O Advanced Programmable Interrupt Controller, is replacing the PIC on the mainboard to provide multi-processor interrupt management for typically 24 interrupt sources. Recent implementations just send MSIs to the Local APICs. 51, 54, 136, 154–157, 179, 180

IDE short for Integrated Device Electronics, is an older parallel interface to mass-storage devices, nowadays replaced by SATA. 42, 49, 53, 73

IDT short for Interrupt Descriptor Table, is a data structure that defines interrupt and exception handler of a x86 CPU. 70, 107

IOMMU short for I/O Memory Management Units were recently added to AMD and Intel chipsets to translate as well as filter DMA requests originating from devices for security and legacy reasons [BYMX+06]. IOMMUs also remap DMA requests from *directly assigned* devices so that legacy device drivers will run unmodified inside a VM. Furthermore IOMMUs can sandbox untrustworthy device drivers and restrict the set of MSIs a device can trigger. 15, 21, 27, 30–32, 39, 42, 56, 83, 84, 100, 150, 177, 179, 180

IPC short for Inter-Process Communication, is an OS feature to pass messages between different threads. 27, 30–32, 35–37, 40, 41

IRQ short for interrupt request, signals to the CPU that a device needs attention because a previously programmed device operation completed. 15, 37, 51, 54, 81, 97, 104, 107, 154–158, 179, 180

KSLOC are 1,000 SLOC (source lines of code). 14, 16, 26, 27, 30, 38, 68, 71, 85, 88, 93, 96, 109, 116, 118, 135, 137, 144, 146, 150, 152, 153, 159, 161, 165, 167, 168, 170–175

LDT short for Local Descriptor Table, is the task-local version of the GDT. 104, 106, 108

Local APIC short for Local Advanced Programmable Interrupt Controller, manages the interrupt pins of a x86 processor and prioritizes IRQs sent from other CPUs, I/O APICs, or MSI-capable devices. 25, 51, 52, 55, 70, 73, 81, 82, 98, 178, 180, 182

MMIO short for Memory Mapped I/O, is used by drivers to access device registers via a special memory region. 30, 31, 37, 49, 57, 65, 70, 80–85, 136, 158, 182, *see* PIO

MSI short for Message Signaled Interrupt, is an interrupt raised by a device, for instance on the PCI bus, by sending a write transaction to a special address range [Bal07]. An MSI can be a vectored interrupt but also an NMI or SMI. MSIs can be remotely triggered via Firewire and blocked with an IOMMU. 51, 98, 100, 103, 154, 178–180

MSLOC are 1,000,000 SLOC (source lines of code). 13, 38, 109, 161, 162, 168, 170–172, 174, 175

NCQ short for Native Command Queuing, allows a disk controller to execute several commands in parallel to increase the performance. 47, 53, 74, 76

nested paging is a CPU feature obsoleting shadow paging by autonomously translating memory accesses from a VM through both the guest and the host pagetables. 21, 23, 26, 32, 71, 76, 77, 80, 81, 83, 86, 181

NIC is an acronym for network interface controller, denoting, for instance, Ethernet and Infiniband adapters. 20, 38, 42, 43, 48, 53, 54, 72, 96, 97

NMI short for non-maskable interrupt, is an interrupt, which cannot be inhibited by clearing the interrupt flag via `cli` on x86. A NMI is blocked until the previous one was handled. It can be filtered with an IOMMU and disabled at the device that causes them. 98, 100, 104–107, 179, 181

NOVA short for NOVA OS Virtualization Architecture, denotes the OS as well as its novel architecture consisting of microhypervisor, user-level environment and one VMM per VM. See Section 2.2. 16, 19, 20, 24–37, 41, 42, 44, 46, 56, 60, 70, 71, 73, 74, 76–78, 80, 81, 83–91, 93–95, 100, 104, 106–108, 117, 137, 138, 140, 145, 161–164, 169, 173, 178

Option ROM contains firmware code from an adapter card that extends the BIOS. Typical use cases for Option ROMs are graphic mode switching or booting from otherwise unsupported devices such as network cards or SCSI drives. Option ROMs are sometimes called BIOS itself such as the VGA BIOS or the VESA BIOS. 23, 67, 136, 177

OS short for operating system. 3, 13–16, 19–22, 25, 26, 28–34, 38, 40–43, 46, 48–52, 55, 63, 65, 67–69, 71–73, 80, 82, 85, 90, 92, 93, 96, 98, 101, 104–106, 135–140, 143–159, 161, 162, 164, 167, 169, 171, 173, 175, 177–182

Paravirtualization is a technique to run multiple OSes on non-virtualizable architectures by patching sensitive but non-trapping instructions with calls to the hypervisor. Still used on systems supporting hardware-assisted virtualization to speedup virtualized I/O by injecting enlightened drivers and libraries into the guest OS. 21, 22, 24–26, 42, 43, 56, 68–71, 74, 75, 77, 85, 167, 168, 178, *see* binary translation

PCI short for Peripheral Component Interconnect, is a family of computer busses including Conventional PCI, PCI-X, and PCI Express, which connect a peripheral device to a CPU. 20, 37, 39, 47, 51, 53, 96, 98, 100, 136, 151–159, 179

PCR short for Platform Configuration Register, is a 160-bit register in a TPM to store a chain of hashes of the platform configuration and the executed programs. See Section 5.3.2. 68, 148–150, 181

PIC short for i8259 Programmable Interrupt Controller, is a chip on the mainboard to prioritize IRQs from multiple devices [PIC88]. Introduced with the original IBM PC, it is nowadays replaced by I/O APIC and MSI. 26, 40, 46, 47, 51, 54, 73, 82, 136, 154–158, 178

PIO short for Port I/O, are special instructions on x86 used by drivers to access device registers via a dedicated I/O address space. 31, 37, 39, 40, 49, 53, 54, 65, 80–82, 136, 141, 158, *see* MMIO

PIT short for i8254 programmable interval timer, is a chip on the mainboard able to trigger interrupts periodically after a certain time has passed [PIT94]. Introduced with the original IBM PC, it is nowadays replaced by HPET and Local APIC timer. 26, 33, 50, 52, 54, 73, 81, 82, 136

PS/2 short for Personal System/2, was a hardware platform by IBM standardizing, for instance, the PS/2 keyboard and mouse interface. 33, 37, 39, 52, 54, 137, 181

RTC short for real-time clock, is a chip on the mainboard introduced with the IBM PC-AT. The RTC keeps wall-clock time and configuration values in battery-buffered memory. It can also trigger periodic interrupts with a high frequency and used as alarm watch. 24, 46, 48, 50, 52, 54

SATA short for Serial AT Attachement, specifies a high-speed serialized data link interface to attach mass storage devices to a platform. 25, 53, 56, 73, 82, 83, 177, 179

Shadow Paging is a implementation technique usable by CPUs without nested paging support which allows the hypervisor to combine the pagetable hierarchies defined by the guest and host OS. 77, 80, 163, 179

SLOC is a unit of source code size as measured by the SLOCCount tool [Whe]. SLOCs (Source Lines of Code) are basically all non-empty lines in a source file, which are not a comment. 44, 53, 55, 62, 71, 85, 100, 106, 118, 134, 135, 139, 140, 143–146, 150, 153, 165–168, 170, 172, 173, 179

SMI short for System Management Interrupt, switches the CPU into SMM. 48, 136, 179, 181

SMM short for System Management Mode, is a special operation mode of the x86 CPU with higher privileges than kernel mode. If the CPU accesses a certain I/O port or receives an SMI, it will switch transparently to SMM mode where it executes SMM code provided by the firmware. SMM is used to emulate legacy devices such as serial port or a PS/2 keyboard on legacy-free hardware. It is also employed to handle power management and platform error events. 28, 135, 136, 147, 151, 181

SSE short for Streaming SIMD Extensions, are several additions to the x86 architecture supporting Single-Instruction Multiple-Data operations on packed integer and floating-point numbers. 23, 115, 120, 128, 129, 133

TCB short for Trusted Computing Base, consists of all the software that can influence the security of a program. The TCB of two applications in a traditional OS share the kernel and all the services such as device drivers or filesystems that both of them directly or indirectly use. 3, 14–17, 19, 20, 24–30, 33, 38, 42–44, 47, 49, 65, 67–69, 71, 85–89, 91, 95, 96, 103, 104, 108–110, 116, 118, 133, 135, 137–140, 143, 146, 148–152, 154, 159, 161–165, 175

TLB short for Translation Lookaside Buffer, is a CPU cache holding pagetable entries. 30, 32, 63, 71, 72, 76, 77, 80, 81, 84, 86

TPM short for Trusted Platform Module, is a security chip on a motherboard with PCRs and digital keys to sign them. See Section 5.3.2. 56, 68, 148–152, 159

TSC short for Time-Stamp Counter, is a special register of the x86 processor incremented with CPU clock rate. 52, 56, 72, 84

TSS short for task state segment, is a data structure provided by the OS that describes the x86 CPU state loaded during a hardware task switch. This includes all CPU registers and the stacks that are used to handle traps from lower privilege levels. A TSS also points to an I/O permission bitmap giving a task fine grained I/O port access. Events that may occur at unpredictable times in an OS such as double-faults or NMIs are typically handled with a TSS. 93, 104–108, 178, 181

UEFI short for Unified Extensible Firmware Interface starts to replace the traditional BIOS as the first program that runs in a PC. It also initializes the PC and has to load the first bootloader. Compared to the BIOS it offers richer interfaces to bootloaders and OSes. 121, 136, 138, 147

USB short for Universal Serial Bus, is a fast and dynamically attachable interface for peripheral devices on the PC and many other platforms. 38, 56, 88, 96, 97, 99, 103, 144, 152

User-Level Environment are the OS services like filesystem or network stack but also the management tools that run in the user mode of the CPU. 14–16, 20, 29, 30, 32, 33, 36, 41, 85, 86, 138, 159, 162, 163, 165, 170, 172, 180

VDB short for Vertical DeBugging, is a novel debugging architecture described in Section 3.2. 48, 49, 87, 90–95, 107, 108

VESA short for Video Electronics Standard Association, which standardized a BIOS extension to switch a graphics card into high-resolution modes. 53, 54, 67, 70, 85, 180

VGA short for Video Graphics Array, is an older standard for graphics card and BIOS interfaces originally defined by IBM. 23, 53, 67, 69, 70, 93, 95, 180

VM short for virtual machine, is an execution environment for an OS provided by hypervisor and VMM. A VM is similar to but not necessarily the same as the underlying physical machine. 3, 15–17, 19–30, 32, 33, 38, 40, 41, 47–49, 53, 54, 56, 67–73, 76–78, 80, 81, 83, 85–88, 90, 94, 95, 161, 162, 165, 168, 170, 172, 175, 178–180

VMM short for virtual machine monitor, provides the illusion to a guest OS that it runs on its own machine by emulating at least one virtual CPU and the necessary peripheral devices such as timer and interrupt controller. 3, 15, 16, 20–27, 29, 30, 32, 33, 38–44, 47–55, 57, 64, 65, 67–71, 73, 82, 84–86, 90, 161–163, 165, 167, 168, 175, 177, 180, 182

x2APIC is a novel operating mode of the Local APIC using a faster `wrmsr` interface instead of MMIO. 51, 82, 84, 100

Appendix C

Bibliography

[AA06] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *ASPLOS'06: 12th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, San Jose, October 2006. 23, 77

[AACM07] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *DLS'07: Dynamic Languages Symposium*, pages 53–64, Montreal, October 2007. 116

[AAD+09] J. Arnold, T. Abbott, W. Daher, G. Price, N. Elhage, G. Thomas, and A. Kaseorg. Security Impact Ratings Considered Harmful. In *HOTOS'09: Workshop on Hot Topics in Operating Systems*, Monte Verità, May 2009. 13

[ABB+86] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Summer 1986 Conference*, pages 93–113, Atlanta, July 1986. 21

[ACA] ACPICA - ACPI Component Architecture project. `http://acpica.org`. 153

[ACP] ACPI - Advanced Configuration and Power Interface. `http://acpi.info`. 152, 177

[ACP06] *ACPI Specification - Revision 3.0b*, October 2006. 153

[AFK+11] D. Amelang, B. Freudenberg, T. Kaehler, A. Kay, S. Murrell, Y. Ohshima, I. Piumarta, K. Rose, S. Wallace, A. Warth, and T. Yamamiya. STEPS Toward Expressive Programming Systems. Technical Report TR-2011-004, Viewpoints Research Institute, October 2011. 14

[AGSS10] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam. The Evolution of an x86 Virtual Machine Monitor. *ACM SIGOPS Operating Systems Review*, 44(4):3–18, December 2010. 22, 24

[Aig11] R. Aigner. *Communication in Microkernel-Based Operating Systems*. PhD thesis, TU Dresden, Germany, January 2011. 27

[AMD09] AMD. *SimNow Simulator 4.6.1 - Users Manual - Revision 2.13*, November 2009. 20, 24

[And] E. Andersen. uClibc: A C library for embedded Linux. `http://www.uclibc.org`. 170

[ANW+10] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *CCS'10: 17th Conference on Computer and Communications Security*, pages 38–49, Chicago, October 2010. 26, 28

[APM13] AMD. *AMD64 Architecture - Programmer's Manual - Revision 3.20*, May 2013. 59, 61

[AW75] W. G. Alexander and D. B. Wortman. Static and Dynamic Characteristics of XPL Programs. *Computer*, 8(11):41–46, November 1975. 120

[Bal07] J. H. Baldwin. PCI Message Signaled Interrupts. In *BSDCon'07: The Technical BSD Conference*, Ottawa, May 2007. 98, 154, 179

[BDF+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP'03: 19th Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, October 2003. 22, 24, 44, 77

[BDK05] M. Becher, M. Dornseif, and C. N. Klein. FireWire: all your memory are belong to us. In *CanSecWest'05: 6th Applied Security Conference*, Vancouver, May 2005. 98

[BdPSR96] F. Barbou des Places, N. Stephen, and F. D. Reynolds. Linux on the OSF Mach3 Microkernel. In *1st Conference on Freely Distributable Software*, Cambridge, February 1996. 15, 22

[BDR97] E. Bugnion, S. Devine, and M. Rosenblum. DISCO: Running Commodity Operating Systems on Scalable Multiprocessors. In *SOSP'97: 16th Symposium on Operating Systems Principles*, pages 143–156, Saint Malo, October 1997. 21, 22

[Bel02] F. Bellard. TinyCC - Tiny C Compiler. `http://www.tinycc.org`, 2002. 109, 133, 174

[Bel05] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX'05: USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, Anaheim, April 2005. 20, 22, 23, 43, 44, 87, 88

[Bie06] S. Biemüller. Hardware-Supported Virtualization for the L4 Microkernel. Master's thesis, University of Karlsruhe, Germany, September 2006. 25

[BKKH13] J. Butterworth, C. Kallenberg, X. Kovah, and A. Herzog. Problems with the Static Root of Trust for Measurement. In *BlackHat USA*, Las Vegas, July 2013. 147

[BOO] R. B. de Oliveira. BOO Programming Language. `http://boo.codehaus.org`. 116

[Bro87] F. P. Brooks Jr. No Silver Bullet - Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, April 1987. 14, 162

[Bro12] J. E. Bromberger. Debian Wheezy: US$ 19 Billion. Your price . . . FREE! `http://blog.james.rcpt.to/2012/02/13/debian-wheezy-us19-billion-your-price-free/`, February 2012. 13, 171, 174

[BSMY13] V. Bashun, A. Sergeev, V. Minchenkov, and A. Yakovlev. Too Young to be Secure: Analysis of UEFI Threats and Vulnerabilities. In *FRUCT'13: 14th Conference of the Open Innovations Association*, pages 11–24, Helsinki, November 2013. 147, 148

[BSSM08] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *ASPLOS'08: 13th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–35, Seattle, March 2008. 23, 32

[BUZC11] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel Symbolic Execution for Automated Real-World Software Testing. In *EuroSys'11: 6th European Conference on Computer Systems*, pages 183–197, Salzburg, April 2011. 14

[BYDD+10] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *OSDI'10: 9th Symposium on Operating Systems Design and Implementation*, pages 423–436, Vancouver, October 2010. 29, 88

[BYMX+06] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. Van Doorn, J. Nakajima, A. Mallick, and E. Wahlig. Utilizing IOMMUs for virtualization in Linux and Xen. In *Linux Symposium*, pages 71–86, Ottawa, July 2006. 15, 179

[CC10] V. Chipounov and G. Candea. Reverse Engineering of Binary Device Drivers with RevNIC. In *EuroSys'10: 5th European Conference on Computer Systems*, pages 167–180, Paris, April 2010. 43

[CCD+10] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In *EuroSys'10: 5th European Conference on Computer Systems*, pages 27–40, Paris, April 2010. 20, 24, 64, 88

[CDE10] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI'08: 8th Symposium on Operating Systems Design and Implementation*, pages 209–224, San Diego, October 2010. 14

[CFH+05] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *NSDI'05: 2nd Symposium on Networked Systems Design and Implementation*, pages 273–286, Boston, April 2005. 24, 28, 47, 77

[CGL+08] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *ASPLOS'08: 13th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, Seattle, March 2008. 15, 19, 28

[CMW+11] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *ApSys'11: Asia-Pacific Workshop on Systems*, Shanghai, July 2011. 13

[CNZ+11] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *SOSP'11: 23rd Symposium on Operating Systems Principles*, pages 189–202, Cascais, October 2011. 14, 26, 27

[COM95] Microsoft and DEC. *The Component Object Specification Model - Version 0.9*, October 1995. 40

[Cov14] Coverity. *Coverity Scan: 2013 Open Source Report*, April 2014. 13, 14, 161

[CVE] MITRE. Common Vulnerabilities and Exposure (CVE). `https://cve.mitre.org/`. 13

[CYC+01] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An Empirical Study of Operating Systems Errors. In *SOSP'01: 18th Symposium on Operating Systems Principles*, pages 73–88, Banff, October 2001. 88

[CYC+08] D. Challener, K. Yoder, R. Catherman, D. R. Safford, and L. van Doorn. *A Practical Guide to Trusted Computing*. IBM Press, January 2008. ISBN: 978-0132398428. 149

[DEG06] L. Duflot, D. Etiemble, and O. Grumelard. Using CPU System Management Mode to Circumvent Operating System Security Functions. In *CanSecWest'06: 7th Applied Security Conference*, Vancouver, April 2006. 147, 151

[Deg12] U. Degenbaev. *Formal Specification of the x86 Instruction Set Architecture*. PhD thesis, Saarland University, Germany, February 2012. 66

[Dep85] Department of Defense. *DoD 5200.28-STD: Trusted Computer System Evaluation Criteria*, December 1985. 14

[Deu96] P. Deutsch. *GZIP file format specification version 4.3*, May 1996. RFC 1952. 130

[DGT13] T. A. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *SOSP'13: 24th Symposium on Operating Systems Principles*, pages 33–48, Farmington, November 2013. 37

[Dij68] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, July 1968. 31

[Dik01] J. Dike. User-mode Linux. In *5th Annual Linux Showcase & Conference*, pages 3–14, Oakland, November 2001. 22, 43

[DKC+02] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M.-C. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual - Machine Logging and Replay. In *OSDI'02: 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Boston, December 2002. 28

[DLMG09] L. Duflot, O. Levillain, B. Morin, and O. Grumelard. Getting into the SMRAM: SMM reloaded. In *CanSecWest'09: 10th Applied Security Conference*, Vancouver, March 2009. 151

[DPC] Debian. Popularity Contest. `http://popcon.debian.org`. 136, 173

[DPSP+11] T. Distler, I. Popov, W. Schröder-Preikschat, H. P. Reiser, and R. Kapitza. SPARE: Replicas on Hold. In *NDSS'11: 18th Network and Distributed System Security Symposium*, pages 407–420, San Diego, February 2011. 19

[DSA] Debian Security Advisories. `http://www.debian.org/security/`. 13

[DWA10] DWARF Debugging Information Format - Version 4. `http://www.dwarfstd.org`, June 2010. 93, 178

[Eag07] M. J. Eager. *Introduction to the DWARF Debugging Format*, February 2007. 93, 94, 127, 178

[EH13] K. Elphinstone and G. Heiser. From L3 to seL4 - What Have We Learnt in 20 Years of L4 Microkernels? In *SOSP'13: 24th Symposium on Operating Systems Principles*, pages 133–150, Farmington, November 2013. 14, 30, 31

[ELF95] Executable and Linking Format (ELF) Specification - Version 1.2, May 1995. 93, 121, 178

[ELI12] ELILO - The EFI Linux Loader - Version 3.14. `http://elilo.sf.net`, June 2012. 173

[Ern99] T. Ernst. TRAPping Modelica with Python. In *CC'99: 8th Conference on Compiler Construction*, pages 288–291, Amsterdam, March 1999. 117

[ESXa] VMware. ESX Server Virtual Infrastructure Node Evaluator's Guide. `http://www.vmware.com/pdf/esx_vin_eval.pdf`. 167

[ESXb] VMware. VMware ESXi. `http://www.vmware.com/esx/`. 24

[Fai06] T. Faison. *Event-Based Programming: Taking Events to the Limit*. Apress, May 2006. ISBN: 978-1590596432. 40

[FBB+97] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *SOSP'97: 16th Symposium on Operating Systems Principles*, pages 38–51, Saint Malo, October 1997. 15, 40, 44

[FC08] B. Ford and R. Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *USENIX'08: USENIX Annual Technical Conference*, pages 293–306, Boston, June 2008. 66

[Fes09] N. Feske. *Securing Graphical User Interfaces.* PhD thesis, TU Dresden, Germany, February 2009. 15, 164

[FH91] C. W. Fraser and D. R. Hanson. A Retargetable Compiler for ANSI C. *ACM SIGPLAN Notices*, 26:29–43, October 1991. 109, 133, 174

[FH06] N. Feske and C. Helmuth. Design of the Bastei OS Architecture. Technical Report TUD–FI06–07, TU Dresden, December 2006. 33, 34, 36, 40, 139

[FHL+96] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernel Meet Recursive Virtual Machines. In *OSDI'96: 2nd Symposium on Operating Systems Design and Implementation*, pages 137–151, Seattle, October 1996. 34

[FHN+04] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *OASIS'04: Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, Boston, October 2004. 25

[Fre09] S. Frenz. SJC - Small Java Compiler. `http://www.fam-frenz.de/stefan/compiler.html`, 2009. 116

[Fri06] T. Friebel. Übertragung des Device-Driver-Environment-Ansatzes auf Subsysteme des BSD-Betriebssystemkerns. Master's thesis, TU Dresden, Germany, March 2006. 15, 44

[FS03] N. Ferguson and B. Schneier. A Cryptographic Evaluation of IPsec. *Counterpane Internet Security, Inc*, December 2003. 14, 135

[FSLM02] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller. Think: A Software Framework for Component-based Operating System Kernels. In *USENIX'02: USENIX Annual Technical Conference*, pages 73–86, Monterey, June 2002. 40

[Gal13] J. Galowicz. Live Migration of Virtual Machines between Heterogeneous Host Systems. Master's thesis, RWTH Aachen, Germany, October 2013. 48, 53, 86

[GBRM+09] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, June 2009. 171

[GCB+08] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Trustworthy and Personalized Computing on Public Kiosks. In *MobiSys'08: 6th Conference on Mobile Systems, Applications, and Services*, pages 199–210, Breckenridge, June 2008. 149

[GCC] Free Software Foundation: The GNU Project. GCC - The GNU Compiler Collection. `http://gcc.gnu.org`. 109, 174

[GG74] S. W. Galley and R. P. Goldberg. Software Debugging: the Virtual Machine Approach. In *ACM'74: Annual ACM Conference*, pages 395–401, San Diego, November 1974. 21, 87

[GJGT10] I. Goiri, F. Julia, J. Guitart, and J. Torres. Checkpoint-based Fault-tolerant Infrastructure for Virtualized Service Providers. In *NOMS'10: Network Operations and Management Symposium*, pages 455–462, Osaka, April 2010. 47

[GJP+00] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *EW'00: 9th ACM SIGOPS European Workshop*, pages 109–114, Kolding, September 2000. 14, 22, 30, 40

[GLV+10] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *OSDI'08: 8th Symposium on Operating Systems Design and Implementation*, pages 309–322, San Diego, October 2010. 77

[GO] The Go Programming Language. `http://golang.org`. 116

[Gol73] R. P. Goldberg. Architecture of Virtual Machines. In *Workshop on Virtual Computer Systems*, pages 74–112, Cambridge, March 1973. 21, 30

[Gol74] R. P. Goldberg. Survey of Virtual Machine Research. *Computer*, 7(6):34–45, June 1974. 21

[GR03] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *NDSS'03: 10th Network and Distributed System Security Symposium*, pages 191–206, San Diego, February 2003. 19, 70

[Gra85] J. Gray. Why Do Computers Stop and What Can Be Done About It? Technical Report 85.7, Tandem Computers, June 1985. 89

[Gra06] D. Grawrock. *The Intel Safer Computing Initiative - Building Blocks for Trusted Computing*. Intel Press, January 2006. ISBN: 978-0976483267. 148, 149, 151

[Gre04] T. Green. 1394 Kernel Debugging Tips and Tricks. In *WinHEC'04: Windows Hardware Engineering Conference*, Seattle, May 2004. 103

[GRU05] Free Software Foundation: The GNU Project. *GRUB Legacy - Version 0.97*, May 2005. 173

[GRU14] Free Software Foundation: The GNU Project. *GRUB 2*, March 2014. 137, 173

[GSB+99] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble Component–Based Operating System. In *USENIX'99: USENIX Annual Technical Conference*, pages 267–282, Monterey, June 1999. 30, 40

[GSJC13] R. Garg, K. Sodha, Z. Jin, and G. Cooperman. Checkpoint-Restart for a Network of Virtual Machines. In *Cluster'13: International Conference on Cluster Computing*, Indianapolis, September 2013. 48

[Hat05] L. Hatton. Estimating Source Lines of Code from Object Code. `www.leshatton.org/Documents/LOC2005.pdf`, August 2005. 165

[Hea06] J. Heasman. Implementing and Detecting an ACPI Rootkit. In *BlackHat Europe*, Amsterdam, February 2006. Invited Talk. 147, 153

[HEB01] W. C. Hsieh, D. R. Engler, and G. Back. Reverse-Engineering Instruction Encodings. In *USENIX'01: USENIX Annual Technical Conference*, pages 133–146, Boston, June 2001. 62

[HF98] J. Helander and A. Forin. MMLite: A Highly Componentized System Architecture. In *EW'98: Workshop on Support for Composing Distributed Applications*, pages 96–103, Sintra, September 1998. 40

[HH05] A. Ho and S. Hand. On the Design of a Pervasive Debugger. In *AADEBUG'05: 6th Symposium on Automated Analysis-Driven Debugging*, pages 117–122, Monterey, September 2005. 87

[HHF+05] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *CollaborateCom'05: 1st Conference on Collaborative Computing: Networking, Applications and Worksharing*, San Jose, December 2005. 15, 139

[HHL+97] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The Performance of μ-kernel-based Systems. In *SOSP'97: 16th Symposium on Operating Systems Principles*, pages 66–77, Saint Malo, October 1997. 15, 22, 25, 77

[HJLT05] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A Principled Approach to Operating System Construction in Haskell. In *ICFP'05: 10th International Conference on Functional Programming*, pages 116–128, Tallinn, September 2005. 116

[HKY+13] J. Han, S. M. Kywe, Q. Yan, F. Bao, R. Deng, D. Gao, Y. Li, and J. Zhou. Launching Generic Attacks on iOS with Approved Third-Party Applications. In *ACNS'13: 11th Conference on Applied Cryptography and Network Security*, pages 272–289, Banff, June 2013. 147

[HMP97] M. Hof, H. Mössenböck, and P. Pirkelbauer. Zero-Overhead Exception Handling Using Metaprogramming. In *SOFSEM'97: 24th Seminar on Current Trends in Theory and Practice of Informatics*, pages 423–431, Milovy, November 1997. 122

[Hoa73] C. A. R. Hoare. Hints on Programming Language Design. Technical Report STAN-CS-73-403, Stanford University, December 1973. 115

[HPE04] Intel. *IA-PC HPET (High Precision Event Timers) Specification - v1.0a*, October 2004. 52, 178

[HSH+09] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold-Boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5):91–98, May 2009. 151

[HUL06] G. Heiser, V. Uhlig, and J. LeVasseur. Are Virtual-Machine Monitors Microkernels Done Right? *ACM SIGOPS Operating Systems Review*, 40(1):95–99, January 2006. 30

[HvD04] J. Hendricks and L. van Doorn. Secure Bootstrap is Not Enough: Shoring Up the Trusted Computing Base. In *EW'04: 11th ACM SIGOPS European Workshop*, Leuven, September 2004. 152

[HWF+05] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? In *HOTOS'10: Workshop on Hot Topics in Operating Systems*, Santa Fe, June 2005. 30

[IBM90] IBM. *Personal System/2 - Hardware Interface Technical Reference*, October 1990. 52

[IEE08] IEEE 1394 Working Group. *1394-2008, Standard for a High-Performance Serial Bus*, October 2008. IEEE Computer Society. 97, 178

[Int96] Intel. *I/O Advanced Programmable Interrupt Controller (I/O APIC)*, May 1996. Order Number: 290566-001. 51, 154

[IOC] The International Obfuscated C Code Contest. `http://www.ioccc.org`. 165

[iOS14] Apple. iOS Security. `http://images.apple.com/ipad/business/docs/iOS_Security_Feb14.pdf`, February 2014. 147

[JWX07] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. In *CCS'07: 14th Conference on Computer and Communications Security*, pages 128–138, Alexandria, October 2007. 19

[Kai08] B. Kaindl. *Using physical DMA provided by OHCI-1394 FireWire controllers for debugging*, January 2008. 103

[Kau05] B. Kauer. L4.sec Implementation - Kernel Memory Management. Master's thesis, TU Dresden, Germany, May 2005. 41

[Kau06] B. Kauer. TinyIDL: How to make an IDL compiler simple? DROPSCon Talk, TU Dresden, November 2006. 118

[Kau07a] B. Kauer. Fulda: Next Generation OS Debugging via Firewire. DROPSCon Talk, TU Dresden, May 2007. 87, 98

[Kau07b] B. Kauer. OSLO: Improving the Security of Trusted Computing. In *16th USENIX Security Symposium*, pages 229–237, Boston, July 2007. 68, 109, 135, 140, 143, 147, 148, 149, 166, 172, 174

[Kau09a] B. Kauer. ATARE: ACPI Tables and Regular Expressions. Technical Report TUD–FI09–09, TU Dresden, August 2009. 135, 153

[Kau09b] B. Kauer. RTC polling mode broken, September 2009. qemu-devel mailing-list. 46

[KCW+06] S. T. King, P. M.-C. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *S&P'06: 27th Symposium on Security and Privacy*, pages 314–327, Oakland, May 2006. 147

[KDC05] S. T. King, G. W. Dunlap, and P. M.-C. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX'05: USENIX Annual Technical Conference*, pages 1–15, Anaheim, April 2005. 28, 87

[KEH+09] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *SOSP'09: 22nd Symposium on Operating Systems Principles*, pages 207–220, Big Sky, October 2009. 14, 116

[KGD] KGDB: Linux Kernel Source Level Debugger. `http://kgdb.linsyssoft.com`. 103

[KKL+07] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: The Linux Virtual Machine Monitor. In *Linux Symposium*, pages 225–230, Ottawa, June 2007. 25, 44

[Knu74] D. E. Knuth. Structured Programming with `go to` Statements. *ACM Computing Surveys*, 6(4):261–301, December 1974. 114

[Knu84] D. E. Knuth. Literate programming. *Computer*, 27(2):97–111, May 1984. 118

[Kor09] K. Kortchinsky. Cloudburst - Hacking 3D and Breaking out of VMware. In *BlackHat USA*, Las Vegas, July 2009. 19

[KS08] P. A. Karger and D. R. Safford. I/O for Virtual Machine Monitors: Security and Performance Issues. *IEEE Security & Privacy*, 6(5):16–23, September 2008. 19

[KS09] E. Käsper and P. Schwabe. Faster and Timing-Attack Resistant AES-GCM. In *CHES'09: Cryptographic Hardware and Embedded Systems Workshop*, pages 1–17, Lausanne, September 2009. 151

[KTC+08] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and implementing malicious hardware. In *LEET'08: Workshop on Large-Scale Exploits and Emergent Threats*, pages 1–8, San Francisco, April 2008. 152

[KVB11] B. Kauer, P. E. Verissimo, and A. Bessani. Recursive Virtual Machines for Advanced Security Mechanisms. In *DCDV'11: Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments*, Hong Kong, June 2011. 21, 86, 164

[KZB+91] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A Retrospective on the VAX VMM Security Kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1163, November 1991. 21, 25

[Law96] K. P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, 1996(29es), September 1996. 20, 23, 43

[LCFD+05] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y.-T. Shen, K. Elphinstone, and G. Heiser. User-Level Device Drivers: Achieved Performance. *Computer Science and Technology*, 20(5):654–664, September 2005. 15

[LCM+05] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI'05: 2005 Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, June 2005. 24

[LCWS+09] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *EuroSys'09: 4th European Conference on Computer Systems*, pages 1–12, Nuremberg, March 2009. 77

[Ler09] X. Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, July 2009. 14, 110

[LHGM09] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley. Debug All Your Code: Portable Mixed-Environment Debugging. In *OOPSLA'09: 24th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 207–226, Orlando, October 2009. 91, 173, 174

[Lie95] J. Liedtke. On μ-Kernel Construction. In *SOSP'95: 15th Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, December 1995. 22, 30, 31

[LIL11] LILO - The Linux Loader - Version 23.2. `http://lilo.alioth.debian.org`, April 2011. 173

[LIN08] Linux 2.6.27.4 PCI quirks. `http://lxr.linux.no/linux+v2.6.27.4/drivers/pci/quirks.c`, 2008. 154

[Loc10] M. Locktyukhin. *Improving the Performance of the Secure Hash Algorithm (SHA-1)*, March 2010. 133

[LPH+10] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and Kitten: New High Performance Operating Systems For Scalable Virtualized and Native Supercomputing. In *IPDPS'10: International Symposium on Parallel and Distributed Processing*, pages 1–12, Atlanta, April 2010. 26, 43

[LPL10] S. Liebergeld, M. Peter, and A. Lackorzynski. Towards Modular Security-conscious Virtual Machines. In *RTLWS'10: 12th Real-Time Linux Workshop*, Nairobi, October 2010. 26, 43, 72

[LS09] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM'9: 16th Symposium on Formal Methods*, pages 806–809, Eindhoven, November 2009. 14, 25, 167

[LSS04] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable Virtual Machines Enabling General, Single-node, Online Maintenance. In *ASPLOS'04: 11th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–223, Boston, October 2004. 47

[LUC+05] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-Virtualization: Slashing the Cost of Virtualization. Technical Report 2005-30, Universität Karlsruhe, November 2005. 22

[LUSG04] J. LeVasseur, V. Uhlig, J. Stöß, and S. Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *OSDI'04: 6th Symposium on Operating Systems Design and Implementation*, pages 17–30, San Francisco, December 2004. 30

[LW09] A. Lackorzynski and A. Warg. Taming Subsystems: Capabilities As Universal Resource Access Control in L4. In *IIES'09: Workshop on Isolation and Integration in Embedded Systems*, pages 25–30, Nuremberg, March 2009. 33, 36

[LX] Linux Kernels. `ftp://ftp.kernel.org/pub/linux/kernel/`. 169

[LXO] Old Linux Kernels. `ftp://nic.funet.fi/ftp/pub/Linux/PEOPLE/.Linus.old/`. 169

[MAB+13] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *HASP'13: Workshop on Hardware and Architectural Support for Security and Privacy*, June 2013. 151

[Mac79] R. A. MacKinnon. The Changing Virtual Machine Environment: Interfaces to Real Hardware, Virtual Hardware, and Other Virtual Machines. *IBM Systems Journal*, 18(1):18–46, March 1979. 21

[Man09] K. Mannthey. System Management Interrupt Free Hardware. *Linux Plumbers Conference*, September 2009. 151

[Mar05] V. Maraia. *The Build Master: Microsoft's Software Configuration Management Best Practices*. Addison-Wesley, October 2005. ISBN: 978-0321332059. 171, 172

[MBI10] Multiboot Specification - Version 0.6.96. `http://www.gnu.org/software/grub/manual/multiboot/`, January 2010. 117, 137, 138, 139

[MBZ+12] C. Miller, D. Blazakis, D. D. Zovi, S. Esser, V. Iozzo, and R.-P. Weinmann. *iOS Hacker's Handbook*. John Wiley & Son, May 2012. ISBN: 978-1118228432. 147

[MCE+02] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moested, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, February 2002. 24

[Meh05] F. Mehnert. *Kapselung von Standard-Betriebssystemen*. PhD thesis, TU Dresden, Germany, July 2005. 28

[MGL10] F. Mehnert, J. Glauber, and J. Liedtke. *Fiasco Kernel Debugger Manual - Version 1.1.6+*, September 2010. 88

[Mic01] Microsoft. *Key Benefits of the I/O APIC*, December 2001. 154

[Mic03] Microsoft. *Windows XP: I/O Ports Blocked from BIOS AML*, January 2003. 153

[Mic09] Microsoft. *BitLocker Drive Encryption: Technical Overview*, July 2009. 148

[Mic12] Microsoft. *Hypervisor Top-Level Functional Specification: Windows Server 2008 R2 - Version 2.0a*, May 2012. 25

[Mic14] Microsoft. *Windows Hardware Certification Requirements*, April 2014. 147

[MMH08] D. G. Murray, G. Milos, and S. Hand. Improving Xen Security through Disaggregation. In *VEE'08: 4th Conference on Virtual Execution Environments*, pages 151–160, Seattle, March 2008. 14, 26

[MMP+12] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators. In *ASPLOS'12: 17th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 337–348, London, March 2012. 50

[Mor10] J. P. Morrison. *Flow-based programming.* CreateSpace, May 2010. ISBN: 978-1451542325. 40

[MP97] Intel. *MultiProcessor Specification - Version 1.4*, May 1997. Order Number: 242016-006. 154

[MPRB09] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU Emulators. In *ISSTA'09: 18th International Symposium on Software Testing and Analysis*, pages 261–272, Chicago, July 2009. 23, 50, 66

[MPT78] M. D. McIlroy, E. N. Pinson, and B. A. Tague. Unix Time-Sharing System: Forward. *The Bell System Technical Journal*, 57(6, part 2):1899–1904, July 1978. 139

[MQL+10] J. M. McCune, N. Qu, Y. Li, A. Datta, V. D. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *S&P'10: 31st Symposium on Security and Privacy*, pages 143–158, Oakland, May 2010. 29

[MSB] Microsoft. Security Bulletin. `https://technet.microsoft.com/security/bulletin/`. 13

[NS07] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, June 2007. 24

[NUL] NOVA UserLevel Environment - Repository. `https://github.com/TUD-OS/NUL/`. 33

[NVM12] Intel. *NVM Express - Revision 1.1*, October 2012. 85

[O64] The Open64 Compiler Infrastructure. `http://www.open64.net`. 174

[OHC00] *1394 Open Host Controller Interface Specification - Release 1.1*, January 2000. 97

[OPSS93] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In *SOSP'93: 14th Symposium on Operating Systems Principles*, pages 58–68, Asheville, December 1993. 40

[Orm07] T. Ormandy. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. In *CanSecWest'07: 8th Applied Security Conference*, Vancouver, April 2007. 19

[OST06] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA'06: RSA Conference, Cryptographers' Track*, pages 1–20, San Jose, February 2006. 151

[Par08] B. J. Parno. Bootstrapping Trust in a "Trusted" Platform. In *HOTSEC'08: Workshop on Hot Topics in Security*, San Jose, July 2008. 151

[Par13] M. Partheymüller. Adding SMP Support to a User-Level VMM. Master's thesis, TU Dresden, Germany, November 2013. 56

[PBWH+11] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *ASPLOS'11: 16th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–304, Newport Beach, March 2011. 171

[PD80] D. A. Patterson and D. R. Ditzel. The Case for the Reduced Instruction Set Computer. *ACM SIGARCH Computer Architecture News*, 8(6):25–33, October 1980. 120

[Pea02] S. Pearson. *Trusted Computing Platforms: TCPA Technology in Context.* Prentice Hall, August 2002. ISBN: 978-0130092205. 148

[PEP] Python Enhancement Proposal (PEP) 255 - Simple Generators. `http://www.python.org/dev/peps/pep-0255/`. 94

[Per82] A. J. Perlis. Special Feature: Epigrams on Programming. *ACM SIGPLAN Notices*, 17(9):7–13, September 1982. 87

[PG74] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, July 1974. 21

[PIC88] Intel. *8259A - Programmable Interrupt Controller (PIC)*, December 1988. Order Number: 231468-003. 51, 73, 154, 180

[Pik00] R. Pike. Systems Software Research is Irrelevant. Invited talk, University of Utah, April 2000. 15

[PIT94] Intel. *82C54 - CHMOS Programmable Interval Timer (PIT)*, October 1994. Order Number: 231244-006. 73, 180

[PKB+08] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn. Efficient Guaranteed Disk Request Scheduling with Fahrrad. In *EuroSys'08: 3rd European Conference on Computer Systems*, pages 13–25, Glasgow, March 2008. 38

[Plo12] S. Plotz. Minimal Linux Bootloader. `http://sebastian-plotz.blogspot.de/2012/07/1.html`, July 2012. 138, 173

[Poe09] H. Poetzl. i8259 defaults wrong?, August 2009. qemu-devel mailing-list. 46

[PSLW09] M. Peter, H. Schild, A. Lackorzynski, and A. Warg. Virtual Machines Jailed: Virtualization in Systems with Small Trusted Computing Bases. In *VTDS'09: Workshop on Virtualization Technology for Dependable Systems*, pages 18–23, Nuremberg, March 2009. 25, 27, 44, 72, 78

[PyM] D. Hall. Python-on-a-Chip: PyMite. `http://www.pythononachip.org`. 118

[Pyt] Python v3.2.3 Documentation. `http://docs.python.org/3.2/`. 110

[RC05] J. F. Reid and W. J. Caelli. DRM, Trusted Computing and Operating System Architecture. In *AusGrid'05: Australasian Workshop on Grid Computing and e-Research*, pages 127–136, Newcastle, January 2005. 149

[RCK+09] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic Device Driver Synthesis with Termite. In *SOSP'09: 22nd Symposium on Operating Systems Principles*, pages 73–86, Big Sky, October 2009. 43

[REH07] T. Roscoe, K. Elphinstone, and G. Heiser. Hype and Virtue. In *HOTOS'07: Workshop on Hot Topics in Operating Systems*, San Diego, May 2007. 15, 19

[RI00] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *9th USENIX Security Symposium*, pages 129–144, Denver, August 2000. 21

[RP06] A. Rigo and S. Pedroni. PyPy's Approach to Virtual Machine Construction. In *OOPSLA'06: 21st Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 949–953, Portland, October 2006. 111, 116

[RS09] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals*. Microsoft Press, June 2009. ISBN: 978-0735625303. 95

[RT09] J. Rutkowska and A. Tereshkin. Evil Maid goes after TrueCrypt. *The Invisible Things Lab's blog*, October 2009. 149

[Rus08] R. Russell. virtio: Towards a De-Facto Standard For Virtual I/O Devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, July 2008. 22, 68

[SA99] T. Shanley and D. Anderson. *PCI System Architecture*. Addison-Wesley, May 1999. ISBN: 978-0201309744. 153

[San09] R. Sandrini. VMkit: A lightweight hypervisor library for Barrelfish. Master's thesis, ETH Zürich, Switzerland, September 2009. 26

[SAT09] L. M. Silva, J. Alonso, and J. Torres. Using Virtualization to Improve Software Rejuvenation. *IEEE Transaction on Computers*, 58(11):1525–1538, November 2009. 28

[SBK10] U. Steinberg, A. Böttcher, and B. Kauer. Timeslice Donation in Component-Based Systems. In *OSPERT'10: Workshop on Operating Systems Platforms for Embedded Real-Time Application*, Brussels, July 2010. 31, 36

[Sch01] S. B. Schreiber. *Undocumented Windows 2000 Secrets - A Programmer's Cookbook*. Addison-Wesley, May 2001. ISBN: 978-0201721874. 93

[Sch12] A. L. Schüpbach. *Tackling OS Complexity with Declarative Techniques*. PhD thesis, ETH Zürich, Switzerland, December 2012. 153

[SDM13] Intel. *Intel 64 and IA-32 Architectures - Software Developer's Manual (SDM)*, March 2013. Order Number: 325462-046US. 59, 61, 98, 151

[SET+09] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. In *VEE'09: 5th Conference on Virtual Execution Environments*, pages 121–130, Washington DC, March 2009. 19, 28

[SFA] Linux/ACPI Project - DSDT repository. `http://acpi.sf.net/dsdt/`. 156

[SHA95] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard (SHA-1)*, April 1995. 131

[Shi08] H. Shimokawa. *dcons (4) - dumb console device driver*. The FreeBSD Project, January 2008. 103

[SK10a] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *EuroSys'10: 5th European Conference on Computer Systems*, pages 209–222, Paris, April 2010. 16, 19, 29, 32, 72, 80, 82, 165, 170, 171, 175

[SK10b] U. Steinberg and B. Kauer. Towards a Scalable Multiprocessor User-level Environment. In *IIDS'10: Workshop on Isolation and Integration for Dependable Systems*, Paris, April 2010. 36

[SKB+07] L. Singaravelu, B. Kauer, A. Böttcher, H. Härtig, C. Pu, G. Jung, and C. Weinhold. BLAC: Enforcing Configurable Trust in Client-side Software Stacks by Splitting Information Flow. Technical Report GIT-CERCS-07-11, Georgia Tech, November 2007. 15, 164

[SKLR11] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *CCS'11: 18th Conference on Computer and Communications Security*, pages 401–412, Chicago, October 2011. 26, 28

[SLQP07] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP'07: 21st Symposium on Operating Systems Principles*, pages 335–350, Stevenson, October 2007. 19, 28

[SPHH06] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB Complexity for Security - Sensitive Applications: Three Case Studies. In *EuroSys'06: 1st European Conference on Computer Systems*, pages 161–174, Leuven, April 2006. 15, 165

[Spi] M. Spivey. OBC - Oxford Oberon-2 Compiler. `http://spivey.oriel.ox.ac.uk/corner/Oxford_Oberon-2_compiler/`. 133

[SPS09] M. Sand, S. Potyra, and V. Sieh. Deterministic High-Speed Simulation of Complex Systems Including Fault-Injection. In *DSN'09: 39th International Conference on Dependable Systems and Networks*, pages 211–216, Estoril, June 2009. 26, 43

[SPS13] R. M. Stallman, R. Pesch, and S. Shebs. *Debugging with GDB: the GNU Source-Level Debugger*. GNU Press, May 2013. ISBN: 978-0983159230. 90

[SS08] M. Strasser and H. Stamer. A Software-Based Trusted Platform Module Emulator. In *Trust'08: 1st Conference on Trusted Computing and Trust in Information Technologies*, pages 33–47, Villach, March 2008. 68, 150

[SSF99] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A Fast Capability System. In *SOSP'99: 17th Symposium on Operating Systems Principles*, pages 170–185, Charleston, December 1999. 30

[Ste02] U. Steinberg. Fiasco μ-Kernel User-Mode Port. Study thesis, TU Dresden, Germany, December 2002. 43

[Ste05] M. Steil. 17 Mistakes Microsoft Made in the Xbox Security System. In *22C3: 22nd Chaos Computing Congress*, Berlin, December 2005. 147, 148

[Ste09] J. Stecklina. Remote Debugging via FireWire. Master's thesis, TU Dresden, Germany, April 2009. 87, 88, 91, 104, 105, 106, 143, 174

[Ste11] U. Steinberg. *NOVA Microhypervisor - Interface Specification*. TU Dresden, October 2011. 16, 20, 30

[Ste13] U. Steinberg. Pulsar PXE Loader - Version 0.5. `http://hypervisor.org/pulsar/`, June 2013. 143

[Ste15] U. Steinberg. *NOVA: A Microhypervisor for Secure Virtualization*. PhD thesis, TU Dresden, Germany, est. 2015. *Work in Progress*. 16, 20, 30

[SVL01] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *USENIX'01: USENIX Annual Technical Conference*, pages 1–14, Boston, June 2001. 24

[SVM05] AMD. *Secure Virtual Machine Architecture - Reference Manual - Revision 3.01*, May 2005. 23, 30, 151, 178

[SYS13] The Syslinux Project - Version 6.02. `http://www.syslinux.org`, October 2013. 139, 173

[Szo05] P. Szor. *The Art of Computer Virus Research and Defense*. Pearson Education, February 2005. ISBN: 978-0321304544. 147

[Tar10] C. Tarnovsky. Hacking the Smartcard Chip. In *BlackHat DC*, Washington DC, January 2010. 151

[TBO] Trusted Boot (TBOOT). `http://tboot.sf.net`. 149, 150

[THB06] A. S. Tanenbaum, J. N. Herder, and H. Bos. Can We Make Operating Systems Reliable and Secure? *Computer*, 39:44–51, May 2006. 15

[Tho72] K. Thompson. *User's Reference to B*, January 1972. 111, 115

[Tho84] K. Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–763, August 1984. 16, 109

[Tho90] K. Thompson. A New C Compiler. In *UNIX – The Legend Evolves: Summer 1990 UKUUG Conference*, pages 41–51, London, July 1990. 116, 174

[TKR+12] R. Tartler, A. Kurmus, A. Ruprecht, B. Heinloth, V. Rothberg, D. Dorneanu, R. Kapitza, W. Schröder-Preikschat, and D. Lohmann. Automatic OS kernel TCB Reduction by Leveraging Compile-Time Configurability. In *HotDep'12: Workshop on Hot Topics in System Dependability*, Hollywood, October 2012. 165

[UNR+05] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *Computer*, 38(5):48–56, May 2005. 23, 30, 151, 178

[VCJ+13] A. Vasudevan, S. Chaki, L. Jia, J. M. McCune, J. Newsome, and A. Datta. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *S&P'13: 34th Symposium on Security and Privacy*, pages 430–444, San Francisco, May 2013. 150

[VMQ+10] A. Vasudevan, J. M. McCune, N. Qu, L. Van Doorn, and A. Perrig. Requirements for an Integrity-Protected Hypervisor on the x86 Hardware Virtualized Architecture. In *Trust'10: 3rd Conference on Trusted Computing and Trust in Information Technologies*, pages 141–165, Berlin, June 2010. 150

[VMw11] VMware. *Timekeeping in VMware Virtual Machines*, December 2011. 52

[Vog08] W. Vogels. Beyond Server Consolidation. *ACM Queue*, 6(1):20–26, January 2008. 19

[Wal75] E. J. Walton. The UCLA Security Kernel. Master's thesis, UCLA, USA, 1975. 21

[Wal02] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI'02: 5th Symposium on Operating Systems Design and Implementation*, pages 181–194, Boston, December 2002. 24

[Wat08] J. Watson. VirtualBox: Bits and Bytes Masquerading as Machines. *Linux Journal*, 2008(166), February 2008. 25, 44

[WD12] J. Winter and K. Dietrich. A Hijacker's Guide to the LPC Bus. In *EuroPKI'11: 8th European Conference on Public Key Infrastructures, Services, and Applications*, pages 176–193, Leuven, September 2012. 151

[Wei14] C. Weinhold. *Reducing Size and Complexity of the Security-Critical Code Base of File Systems*. PhD thesis, TU Dresden, Germany, January 2014. 15, 164

[Wel00] N. Wells. BusyBox: A Swiss Army Knife for Linux. *Linux Journal*, 2000(78), October 2000. 170

[WG92] N. Wirth and J. Gutknecht. *Project Oberon - The Design of an Operating System and Compiler*. Addison-Wesley, November 1992. ISBN: 978-0201544282. 109, 116

[WH08] C. Weinhold and H. Härtig. VPFS: Building a Virtual Private File System with a Small Trusted Computing Base. In *EuroSys'08: 3rd European Conference on Computer Systems*, pages 81–93, Glasgow, March 2008. 27, 36

[Whe] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`. 165, 181

[Whe05] D. A. Wheeler. Countering Trusting Trust through Diverse Double-Compiling. In *ACSAC'05: 21st Annual Computer Security Applications Conference*, pages 33–48, Tucson, December 2005. 110

[Wir95] N. Wirth. A Plea for Lean Software. *Computer*, 28(2):64–68, February 1995. 13, 14

[Wir04] N. Wirth. *Programming in Oberon - A derivative of Programming in Modula-2*, October 2004. 116

[WJ10] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *S&P'10: 31th Symposium on Security and Privacy*, pages 380–395, Oakland, May 2010. 26, 28

[Woj08] R. Wojtczuk. Subverting the Xen Hypervisor. In *BlackHat USA*, Las Vegas, August 2008. 19

[WR09a] R. Wojtczuk and J. Rutkowska. Attacking Intel Trusted Execution Technology. In *BlackHat DC*, Washington DC, February 2009. 151

[WR09b] R. Wojtczuk and J. Rutkowska. Attacking SMM memory via Intel CPU cache poisoning. *Invisible Things Lab*, March 2009. 151

[WSG02] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical Report 02-02-01, University of Washington, February 2002. 22

[WSG10] J. Wang, A. Stavrou, and A. Ghosh. HyperCheck: A Hardware-Assisted Integrity Monitor. In *RAID'10: 13th Conference on Recent Advances in Intrusion Detection*, pages 158–177, Ottawa, September 2010. 26, 28

[WWGJ12] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating Commodity Hosted Hypervisors with HyperLock. In *EuroSys'12: 7th European Conference on Computer Systems*, pages 127–140, Bern, April 2012. 26, 27, 72

[WWJ13] C. Wu, Z. Wang, and X. Jiang. Taming Hosted Hypervisors with (Mostly) Deprivileged Execution. In *NDSS'13: 20th Network and Distributed System Security Symposium*, San Diego, February 2013. 26, 27, 72

[YHLR13] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing. In *SC'13: Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, November 2013. 64

[You07] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *ISPASS'07: International Symposium on Performance Analysis of Systems and Software*, pages 23–34, San Jose, April 2007. 24

[ZC10] C. Zamfir and G. Candea. Execution Synthesis: a Technique for Automated Software Debugging. In *EuroSys'10: 5th European Conference on Computer Systems*, pages 321–334, Paris, April 2010. 14

[ZCCZ11] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *SOSP'11: 23rd Symposium on Operating Systems Principles*, pages 203–216, Cascais, October 2011. 28

[ZL96] A. Zeller and D. Lütkehaus. DDD - a free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, January 1996. 95