Fakultät für Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Security in Telecommunications

Design and Provability of a Statically Configurable Hypervisor

vorgelegt von
Dipl.-Inform.
Jan Nordholz
geb. in Stadthagen

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Hans-Ulrich Heiß, Technische Universität Berlin
Gutachter:	Prof. Dr. Jean-Pierre Seifert, Technische Universität Berlin
Gutachter:	Prof. Fabio Massacci, Ph.D., Università di Trento
Gutachter:	Prof. Dr.-Ing. Stefan Jähnichen, Technische Universität Berlin

Tag der wissenschaftlichen Aussprache: 2. Juni 2017

Berlin 2017

Abstract

In this thesis we develop a novel, minimalist design for Type I hypervisors and present a fully working prototype for the ARMv7 and ARMv8 architectures. We introduce its key design paradigm, the Principle of Staticity, elaborate on its consequences for the resulting implementation, and discuss scenarios where not all dynamicity can be eliminated.

We also describe, implement, and evaluate the configuration framework necessary to build this specific kind of hypervisor. Even though the compiled hypervisor images are fully static, we show that the framework is powerful enough to support various system-on-chip target platforms, different address translation regimes, and even completely unrelated architectures. We further demonstrate its versatility by extending the framework to even support embedded processors which only contain a memory protection unit.

After performing a selection of benchmarks to substantiate the competitiveness of our implementation, we continue by subjecting our hypervisor to an analysis using a specialized form of symbolic model checking. We succeed in deriving integrity properties for the hypervisor as well as among virtual machines with surprisingly little effort, but show that complexity immediately ramps up once dynamic components like a shadow paging unit are added to our design.

Finally we demonstrate the extensibility of our design despite our postulated Principle of Staticity. As an optimization example, we introduce the concept of "lightweight VMs" and present a novel use case for a recent feature of the ARM architecture, thereby reducing the switch latency between virtual machines.

We conclude with an overview over the published research efforts based on or heavily inspired by our hypervisor and discuss future research questions.

Contents

1. Introduction

Microkernels had spent more than a decade on the sideline of academic research when Jochen Liedtke revived the idea by formulating his famous minimality principle[62] and creating the L3 microkernel and its successor L4. These embodiments of his principle became the ancestors of a large family of microkernels, and their success brought attention back to a seemingly uninteresting research field.

Since those days, the evolution of computing platforms has radically changed the circumstances. Contemporary personal computers easily run several virtual machines under the control of the host operating system, and even architectures with clear focus on the embedded devices market like ARM and MIPS have entered the virtualization business. The hypervisors, the pieces of control software that are in charge of managing virtualization, follow a design paradigm that is not completely unlike microkernels. Their history however is very different.

Due to the recent trend of consolidating disparate workloads onto a single compute platform, both designs increasingly find their way into highly critical environments, e. g. avionics, train control, and road vehicle electronics. All these environments require strict isolation of the individual workloads placed on the computing device. In addition to coding in special-purpose dialects like MISRA C and following safety-specific development processes like IEC 61508, many developers of microkernels and hypervisors have begun subjecting their existing designs to formal methods. Ideally these analyses result in proofs of formal properties of the corresponding implementation such as integrity, absence of covert channels, or even correctness—however endeavours like seL4[58] have demonstrated that generating such a proof is an arduous task which should not be underestimated.

The complexity of a proof is significantly influenced by the amount of dynamicity in the underlying system design. Dynamic subsystems like memory allocators, kernel object factories, balancing schedulers, and memory balloon mechanisms are in a way the only challenge to the creation of a formal model: each of these requires the specification of an invariant, a proof that the invariant holds under all possible operations and circumstances, especially the exhaustion case, and if global resources are being managed, additional reasoning that the mechanism cannot be abused to covertly transmit data. Algorithms which operate on static data sets on the other hand, like an iteration over a fixed number of elements of a linked list which is a priori guaranteed to be loop-free, can be easily expressed in formal terms.

We strongly believe that the path to a provably secure hypervisor implementation does not necessarily lead to the generation of a proof for one of the big existing general-purpose kernels or hypervisor cores. We argue instead that for most highly critical environments a far simpler implementation is sufficient. At the core of this simplification we propose to reduce dynamicity as much as possible by pregenerating data structures and choosing algorithmic parameters already at the compilation stage.

The reduced complexity is going to result in less code, retaining only the relevant device drivers and event handlers for the desired configuration, as well as in "simpler code" in terms of provability. This design paradigm does not even impose any versatility limitations on the resulting implementation, quite the contrary: we envision that

a hypervisor designed in this way can still support multiple platforms, even multiple different architectures, with the build process carefully selecting the desired pieces of functionality and generating the necessary corresponding data structures to form an easily provable and perfectly tailored code subset.

We claim that this approach is viable because embedded computing platforms such as those designed for highly critical environments are meticulously specified in terms of processing power, available memory resources, and peripheral devices, as opposed to the construction kit mentality of present-day desktop platforms. Likewise, the software components for such embedded devices usually have strict scheduling requirements and require a constant amount of CPU time to perform their tasks, whereas desktop computing is characterized by a vast variability of system utilization. We argue that the advantages of a static system configuration have not been fully recognized yet.

Another topic which poses extraordinary difficulty to formal methods is the correct and consistent modelling of multiprocessor interactions. Following our strategy of deliberately removing features that are unnecessary for embedded use cases, we claim that there is little incentive to implement VM migration, as this would only uproot the precalculated scheduling plans. If however there is no migration, there should be almost no system-global data which would require a classical multicore architecture with synchronization primitives such as spinlocks or a read-copy-update mechanism. Instead, we postulate that the desirable solution is to create multiple hypervisor instances, one per core, and to let them execute and schedule fully independently.

Based on these ideas we formulate a usable design paradigm, demonstrate and evaluate a fully-fledged multi-architecture and multi-platform hypervisor implementation with an accompanying compile-time configuration system, derive formal properties in a way that directly builds on the staticity of the system, and showcase how the basic implementation can be extended while preserving the applicability of our proof methodology. We release all the code written for this thesis as open source in order to broaden the field of system software research and to enable future researchers to conduct comparative measurements[1].

We begin in Chapter 2 by delving into past and concurrent designs of microkernels and hypervisors, highlighting key insights which influenced our work and putting our ideas into perspective. In Chapter 3 we then define our core design principle and discuss how it affects each core component of a hypervisor implementation. Finally, we introduce Phidias, the second incarnation of the principle we have just developed, and analyze it in terms of code base minimality. In Chapter 4 we complement the hypervisor design by introducing our corresponding configuration framework, Schism, and discuss the complexity shifts from runtime kernel to build-time configuration we have incurred.

The next two chapters focus on the analysis of our design. In Chapter 5 we conduct a performance evaluation of our hypervisor and compare our results against recent measurements on other state-of-the-art hypervisors. In Chapter 6 we finally subject our hypervisor to a customized form of symbolic execution in order to prove that our implementation maintains integrity and prevents direct unintended interaction between virtual machines.

Chapter 7 explores the extensibility of our design by enhancing one of the core operations of a hypervisor, the VM switch. We describe the necessary implementation

[1] Further information including URLs to the source code repositories can be found in Appendix B.

changes, provide measurement data which demonstrates the reduction in overhead, and discuss the compatibility of the optimization with our symbolic execution proof.

We conclude by presenting publications directly based on our hypervisor as well as related research papers in Chapter 8 and by pointing out future research directions and follow-up questions in Chapter 9.

2. Background

As our hypervisor draws inspiration from lessons learnt throughout the development history of microkernels, yet builds on the technological experience of forty years of virtualization, we begin by analyzing the evolution of these two concepts from their inception to their present state. Based on this we then discuss commonalities in the light of the technological advancements of the last ten years and set the stage for our own design.

2.1. A Brief History of Hypervisors

The concept of a virtualization component between the physical machine and the operating system kernel dates back to the late 1960s when IBM developed CP/CMS, the Control Program for the Cambridge Monitor System. This early design already exhibited many features we expect from our commodity desktop virtualization solutions today: every virtual machine was presented with its independent simulated view of system hardware, and any operating system capable of running on a bare System/360 mainframe would also run unmodified as the guest OS inside a CP/CMS container.

This was possible as the System/360 supported two execution modes, privileged and unprivileged (or "system" and "problem" at the time, respectively); CP/CMS took advantage of this by occupying privileged mode itself and moving guest operating systems down into unprivileged mode, emulating those instructions that were illegal when executed unprivileged. This concept of trap-and-emulate hypervisor behaviour was later formalized by Popek and Goldberg in their seminal paper on architecture virtualizability[78]. The strategy of deprivileging the guest operating system and emulating its privileged instructions is still the established way of implementing virtualization on present-day hardware that does not offer built-in virtualization capabilities, e. g. by providing an additional privilege level or specific instructions for the management of virtual machines.

IBM's original CP-40 hypervisor has evolved over the decades into the hypervisor component of current z/VM incarnations. But apart from the mainframe business segment, virtualization did not become a viable opportunity until much later. By the mid-1980s, personal computers had finally become sufficiently affordable that the general public was now able to use their own computing resources instead of sitting at terminals hooked up to big time-sharing systems. And with the rise of the desktop PC we saw new use cases for virtualization. In contrast to the mainframes, where isolating different users' sessions from each other was one of the key features, these new devices were effectively single-user systems. Thus the focus was no longer on separation, but rather on ease of setup and use, and also increasingly on high-performance and easily configurable sharing of resources between the host operating system and its guests.

This desire also resulted in the creation of the first hypervisors which are nowadays known as "Type II". This new class of hypervisors is fully integrated into a host

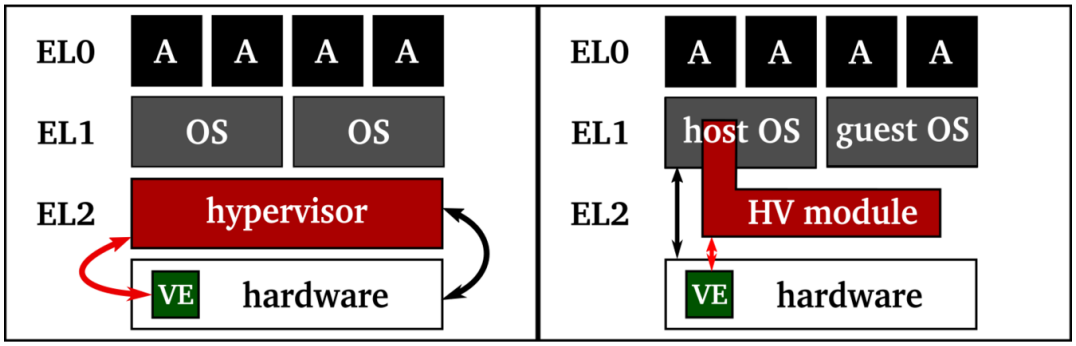

Figure 2.1.: Type I (left) vs. Type II (right) hypervisor driving the Virtualization Extensions (VE, red arrow), using its own or the host operating system's device drivers to communicate with platform peripherals (black arrow).

kernel and able to use its driver infrastructure. It switches between execution of the host kernel, which runs with full privileges and still has direct access to the platform hardware, and guest operating systems, whose hardware access is mediated by the hypervisor. Type I hypervisors on the other hand treat all guest operating systems as equally deprivileged (cf. Figure 2.1).

As the execution environment for the host kernel is not altered when a Type II hypervisor module is added, it was easy for users of desktop PCs to retrofit their systems with such a module and benefit from the added capabilities without impacting their base experience. Early iterations of virtualization solutions for the now-prevailing x86 architecture had to rely on binary translation[2] or to manually port guest kernels to a paravirtualized interface[10], as the architecture's instruction set was not fully virtualizable.[1] Soon though the required architectural extensions were added, and efficiency and featurefulness of virtualization solutions skyrocketed. In order to increase portability and promote interchangeability of virtualization solutions, standards for device virtualization like VirtIO were created[81, 73].

The third wave of virtualization hit the mobile device segment. Over the last decades, mobile phones have evolved from the size and weight of the proverbial brick to palm-sized, but fully featured miniature computers. With this miniaturization and gain in power, the OS landscape for these devices changed as well. Where early so-called "feature phones" ran special-purpose developments like Symbian, modern "smartphones" are now usually equipped with a mobile port of a general-purpose operating system: the Linux-derived Android, MacOS-derived iOS, or Windows Mobile. The academic community and industry reacted to these changes and started porting their hypervisor implementations to mobile platforms—among them solutions like VMware's Mobile Virtualization Platform[11] and *Xen* on ARM[47], which both stem from a rich development history in the x86 desktop segment.

The ARM architecture these phones are based on has its origins in the embedded devices market, but its history with respect to virtualization is not unlike that of x86. As the ARM ISA was not virtualizable in its initial form as well, the increase in

[1] The completely unrelated technique of application separation through mere OS containers such as Linux VServer[88] does not add an additional security layer and thus does not provide any resilience against kernel-level exploits. We are therefore going to disregard this development branch entirely.

processing power of these chips first led to paravirtualization efforts[47]. Again, not much later these were followed by the announcement of the corresponding hardware extensions for the then-current architecture version ARMv7.

The advent of this new technology for the mobile devices market enabled usage scenarios already known from the desktop world like honeypots[69] and operating system debugging environments[52] as well as novel ones like dual-persona smartphones, which can have one of their Android compartments integrated into the infrastructure of the employer[2] while leaving the other compartment free for private use.

As processing power grew further, even smaller platforms like embedded devices were able to benefit from virtualization[44]. This development brought this technology to the attention of two fields which had not touched the subject before: automotive and avionics. In the past, automotive circuit designers had favoured a multitude of decentralized electronic control units (ECUs) and sensor nodes with low processing power connected over a common bus, e. g. CAN or LIN. By their very nature, these components belong to vastly different criticality levels: total failure or even missing a deadline poses lethal dangers if the ECU in question is controlling the brakes, the engine, or the airbag deployment mechanism, whereas a similar malfunction of the climate control or the car entertainment system would only be considered a nuisance.

With the capabilities of virtualization it has recently become possible to consolidate several functional pieces into the same computing unit, thus reducing power consumption and production costs while keeping efficiency high. The requirements imposed on hypervisor implementations for those platforms are therefore extraordinarily complex: they have to cope with different scheduling expectations, guaranteeing hard real-time semantics to those workloads requiring them while assigning the remaining processing power to other partitions that expect fair-share scheduling; they must ensure proper execution of critical software components even in the face of other less critical components misbehaving; and they have to maintain overall strict isolation and integrity between components except for explicitly configured communication channels.

All these requirements have been collected into corresponding industry standards such as ISO 26262 for automotive safety and ARINC 653 for avionics. The analysis by VanderLeest et al.[92] provides an insightful example for how the APIs mandated by ARINC 653 could be combined into existing hypervisor implementations.

Therefore in this fourth wave we see the resurgence of the principles of strong temporal and spatial separation—and these requirements have to be taken to a new level of assurance, as failure to meet them, no matter whether due to a misconfiguration, a programming error, or an attack on the compute platform, may put lives in danger.

2.2. A Brief History of Microkernels

Microkernels entered the academic discussion with the introduction of *Mach*[1] in 1986. Their key distinguishing element is the focus on modularization: all subsystems and services which do not require to be executed in privileged mode are split off and moved into individual "server" processes in userspace. Applications using those

[2] This trend to integrate privately owned devices into the company network has been coined "bring your own device" (BYOD). As this integration might lead to sensitive company data being transferred to the employee's device, strong isolation is a prerequisite—a problem which virtualization is perfectly suited to solve.

services are thus required to engage in horizontal communication, i. e. inter-process communication, instead of performing vertical system calls into the kernel.

The immediate benefits over monolithic operating system designs, which were the prevalent form then as they are now, are fault containment and seamless error recovery. In a monolithic kernel all components share the same privilege level and address space, so an erroneous memory write operation caused by one component could affect any other component and thus easily crash the whole kernel. In a modularized system such as a microkernel, a programming error in one driver can only cause a crash of the particular server process which encapsulates that driver; and if crashed drivers are restarted by a central watchdog module, these crashes only cause a temporary lack of availability.

The increase in robustness that *Mach* introduced came at a high price though. The cost of a system call—two switches in privilege level, one into the kernel and one out again—was orders of magnitude lower than that of its microkernel equivalent, the message-based remote procedure call (RPC). One RPC required two system calls, one each for the transmission of request and reply, and additionally two context switches, from the calling process to the desired service provider process and back. The context switches were particularly expensive, as this involved saving and loading the full set of general purpose registers, reloading the page table base register to activate the new address space, and it incurred—depending on the architecture—an implicit flush of the translation lookaside buffer.

Mach thus suffered from a performance loss that seemed insurmountable, as it was widely believed to be inherent to the microkernel approach. However, Liedtke proved with his seminal L3 design[61] that the core problem of RPC overhead could be overcome by carefully crafting and relentlessly optimizing the hot path: the transmission of short messages of a few machine words combined with an address space ("task" in L3/L4 parlance) switch.

Liedtke based his design on synchronous inter-process communication (IPC), which requires sender and receiver to rendezvous[3] to exchange a message. In this case the kernel is able to directly copy the message from the sender's to the receiver's buffer. The key optimization however was the invention of the "short IPC" path: very short messages were transported in the general purpose registers themselves, saving both the buffer copy operation and a portion of the state save/restore. L3 and its successor design L4 integrated this hot path even into the scheduler: as these IPC messages were in fact synchronous calls, the message from a client would wake the server process, whose reply would again cause a direct switch back to the client, as the server went back to sleep; for this common case, even the modification of the list of blocked tasks in the microkernel scheduler was expendable.

With this Liedtke was able to ultimately reduce the costs of IPC calls by two orders of magnitude[63]. Due to the much more competitive performance of his microkernel, he could achieve an even higher level of minimality than *Mach*, pushing additional components out into userspace processes. He also coined the oft-cited "principle of minimality"[62]:

[3] The communication party arriving (issuing its system call) first would block in the kernel, waiting for the other party to issue its matching system call. Additional timeout parameters were added to solve deadlock situations.

> The determining criterion used is functionality, not performance. More precisely, a concept is tolerated inside the μ-kernel only if moving it outside the kernel, i. e. permitting competing implementations, would prevent the implementation of the system's required functionality.

The original L4 design was later forked into many different branches, among them the proven *seL4* microkernel[58] developed at NICTA, the Australian government's ICT research lab, the commercial *OKL4* branch developed by Open Kernel Labs, a NICTA spinoff, the open source *Fiasco.OC*[75] maintained at TU Dresden, and the commercial *PikeOS*[51], developed by SysGo.[4]

However, all these designs were not built to facilitate reuse of existing driver code. Microkernels are by design an OS replacement optimized for fast IPC between user-space processes, but device driver code is always embedded into the framework of the OS it has been written for and thus not easily transferable. Reworking a system built on a monolithic Linux kernel with a dozen device drivers into a microkernel-based setup with a dozen user-space driver processes would require porting the system activities (e. g. memory allocation, interrupt registration) of those kernel driver modules onto the microkernel API and converting their mutual interaction from function calls to RPC.

All major L4 branches tried to accommodate the problem by developing their own deprivileged Linux derivative, such as *L4Linux* by the Dresden group[43]. However, the API provided by a microkernel differs greatly from a native execution environment. The following list illustrates the most striking items:

- Native virtual memory is managed by creating page tables, an architecture-specific tree-shaped data structure; a microkernel API abstracts from the hardware and provides a generic map/unmap interface to request changes in address translation.
- Native faults and exceptions are posted through an architecture-specific trap entry sequence, which may involve a privilege level change, a stack switch, and possibly the creation of a trap frame on the new stack; microkernels signal the occurrence of such events in one thread by contacting the corresponding "exception handler" thread, i. e. by delivering a special fault IPC message—the handler thread's reply can then include a newly mapped page to resolve the fault condition or alter the faulting thread's state to sidestep the offending instruction.
- Native interrupts are configured at the platform's resident interrupt controller (e. g. at an x86 LAPIC/IOAPIC or at an ARM GIC), and interrupt delivery is comparable to native fault delivery; the microkernel API again provides a hardware-agnostic interface, with delivery masqueraded as special IPC messages.
- Native creation of new address spaces and threads is seamless, the OS simply creates a new page table and registers a new thread control block with its scheduler; on microkernels both operations require an explicit call, one to register a new address space with the map/unmap subsystem (so it is a valid destination of later map() calls, and execution threads can be attached to it) and one to register a new thread with the microkernel scheduler.

[4] The recent survey by Elphinstone and Heiser on the family tree of L4 derivatives[32] provides a more elaborate overview.

The mentioned ports of Linux therefore required invasive changes to make them compliant with the respective microkernel API. Especially the virtual memory subsystem had to be ripped out completely and replaced with a map/unmap system. The last item of the above list additionally hints at a second major point of conflict when paravirtualizing an OS on top of a classical microkernel. As microkernels usually do not support any form of more fine-grained scheduling beyond their own understanding of threads, the scheduler of such a paravirtualized OS has to be effectively disabled, surrendering all scheduling decisions to the (often much less versatile) microkernel scheduler.

2.3. Converging Concepts

Despite their different legacy, the research community also recognized the structural similarities among the two concepts. Shortly before the introduction of hardware virtualization extensions for the x86 architecture, an academic debate ensued over the comparability of microkernels and virtual machine monitors (VMMs). Hand et al.[41] saw the *Xen* hypervisor as the superior implementation of the common goals and principles that both concepts shared. Heiser et al.[46] refuted those claims and pointed out the different interfaces and sets of primitive operations the two concepts offered.

However, the provided analysis is both inaccurate and by now outdated. Firstly, microkernels offer more functionality than bare IPC: even though many operations can be superficially made to look like IPC, and page fault and exception handling is usually built on top of existing IPC mechanisms, the actual `map` operation that adds a physical page to an address space can only be performed by the microkernel itself.

Secondly, the list of purported primitives offered by VMMs quickly diminishes under scrutiny. We find that five of the ten items are only required for a paravirtualizing hypervisor, as was necessary at the time; two describe an inter-VM communication mechanism equivalent to microkernel-based IPC; and two more define resource allocation operations which are not part of the mandatory VMM functionality set[5]. The last property, the availability of common devices like network interfaces and disk drives, owes its presence on the list to the conflation of the terms hypervisor and VMM: the former is nowadays usually used to refer to the virtualization module that has to run with highest hardware privileges, while the latter is a deprivileged component that handles fault handling, device emulation, and other maintenance tasks which do not require elevated privileges.

As a further substantiation to the dwindling differences, many microkernels which originally only supported the execution of native deprivileged threads atop their custom API were retrofitted to also allow the execution of virtual machines. *Fiasco.OC* was extended into such a hybrid design, allowing both native L4 API threads and virtual machines; it left all VM management tasks like faithful device emulation to a VMM that was implemented as a native L4 thread, handling only the hypervisor duties like configuration and activation of the processor's virtualization extensions inside the kernel.

Open Kernel Labs, the company behind *OKL4*, opted for another viable choice: instead of allowing native threads and virtual machines to coexist in hybrid configu-

[5] In fact, the mentioned "page-flipping" property strongly indicates that the inclusion was directly aimed at *Xen*.

rations, the *OKL4* microvisor[45] was created, a separate VM-only alternative—but not a superseding replacement—to the *OKL4* microkernel.

We conclude from all these observations that the difference between microkernels and hypervisors on platforms with virtualization extensions is merely a choice of interface. We illustrate this with Figure 2.2: Given an operating system kernel, which exposes a POSIX API to its applications in our example (top left), we can add a microkernel underneath and componentize untrusted drivers into separate processes using two distinct approaches. The first option is paravirtualization (top right), which implies making extensive changes to the low-level code of the OS kernel in order to port it to the API exposed by the microkernel. The other option is the leveraging of hardware virtualization extensions (bottom left), with a userspace VMM taking over the duty of translating between the hardware-like interface exposed to the VM and the API offered by the microkernel. This setup is however only one step away from a true hypervisor: it merely requires replacing the remaining occurrences of the custom API with the raw machine interface (bottom right) and componentized drivers to be converted to "driver VMs".

Finally, even hypervisor solutions do not strictly require virtualization extensions, as paravirtualization can also be performed while staying faithful to the hardware interface by simply replacing all sensitive unprivileged instructions in the guest kernel. However, the only difference that remains then is the mentioned choice of interface and the amount of context per scheduling entity, as a virtual CPU abstraction requires replication of the full privileged and unprivileged register file, whereas a native thread has only access to the unprivileged set. Thus we ultimately face a trade-off between context switch cost and implementation effort.

We place our hypervisor in this area of convergence, building on and extending Liedtke's principle of minimality, but restricting ourselves to virtual machines as our only supported kind of activity atop the hypervisor and—which is going to be our key advantage for our formal verification efforts—heavily limiting the set of operations we offer to VMs.

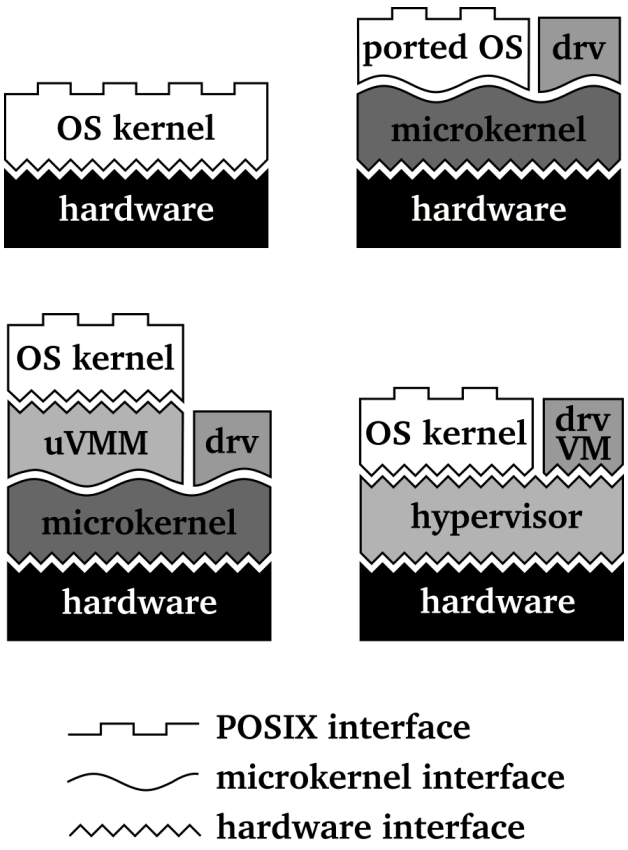

Figure 2.2.: Microkernel- and hypervisor-based virtualization choices. Line-by-line from top left: native execution, paravirtualized port to a microkernel (e. g. *L4Linux*), microkernel virtualization with userspace VMM, Type I hypervisor virtualization.

3. Structural Design

Based on the our analysis of the hypervisor and microkernel landscape and development history we now set out to construct our own Type I hypervisor. We begin by defining our central design principle and studying its implications for all necessary functional components.

3.1. Principle of Staticity

The idea at the heart of our concept is to ease provability and reduce runtime complexity as well as memory footprint by eliminating dynamic elements completely or—if such elimination is found to be impossible—by reducing or constraining their dynamic behaviour as much as possible. We see this as a consequent extension of Liedtke's pursuit of minimality that is specific to the field of embedded computing, where its adverse effects are inconsequential.

Let us succinctly phrase this principle:

> **A hypervisor implementation for an embedded system can and should be meticulously tailored to its usage scenario. Thus every element of dynamicity that does not constitute a mandatory part of runtime functionality is to be transformed into a compile-time configuration tool. The retained runtime code is then purely static, its actions determined by previously generated and externally verifiable configuration data.**

This rejection of dynamic elements applies to the whole spectrum of software design, from basic features of the programming language to algorithmic constructs.

One example of a language feature we consider unnecessary and thus expendable for our endeavour are modifiable function pointer variables. The Linux kernel in particular makes extensive use of vtable-like function pointer structures in order to introduce a limited form of polymorphism for its subsystem boundaries, e. g. the file system API. Other designs like the *Fiasco.OC* microkernel are written in a language that supports polymorphism (C++) and thus intrinsically contain writable vtables as part of their runtime objects. In both cases it is extremely difficult to model the possible values of these function pointers and thus the invocation targets of these indirect branches. Read-only function pointers and jump offset tables on the other hand can be handled just like jump instructions with immediate offset; the compiler even emits such tables to speed up the execution of large switch/case blocks.

Our most prominent candidate for an expendable dynamic runtime module is the memory allocator. The allocator plays an integral role in current general-purpose operating system kernels and is heavily optimized, as it is used virtually everywhere, e. g. in platform probing (allocation of per-device state structures), task creation and destruction (allocation of task control block, kernel stack etc.), and handling dynamic userspace activities (stack and heap growth). We solve this by disallowing probing

in general, and instead require that the integrator for a particular platform fully specifies the list and accessibility data (memory ranges, I/O ports, interrupt lines) of available peripheral devices. We discuss the further implications of the omission of an allocator in Subsection 3.2.1, whereas we explain the details of this build time specification framework in Chapter 4.

The only dynamic hypervisor element we find to be irreplaceable in certain hardware configurations is the shadow paging module. We describe our efforts of reducing its dynamicity in Subsection 3.2.8.

3.2. Applying the Principle of Staticity

We now examine each core component of a minimal hypervisor implementation in turn and discuss the consequences of applying our Principle of Staticity.

3.2.1. Memory Management

The deliberate exclusion of an allocator, whether small-scale or large-scale[1], mandates that not only the details of the platform, but also the properties of all virtual machines are fully specified at compile time. With these pieces of information the configuration framework is then able to check whether the requested memory allocations and alignment constraints for the virtual machines are actually satisfiable by the available memory of the selected target platform, and if so, assign fixed physical addresses to each of those allocations. The resulting metadata is finally emitted as C structure declarations and compiled into the bootable hypervisor image, as parsing the data from e. g. a textual representation at hypervisor startup would also require an allocator.

As a consequence of the memory layout being decided so early, we can directly generate all necessary page tables from the memory placement data. For the hypervisor itself we generate one page table per instance, i. e. per physical CPU (for our multicore design decision, see Section 3.3.1), and one common "init page table" which is used transitorily during bootup.

Besides the page tables, which contain the translation between virtual and physical addresses in an architecture-mandated format, we also need to have the list of mapped memory ranges available to software. In particular, the trap handler in the hypervisor must be able to distinguish the different kinds of page faults raised by a virtual machine. In order to do so, the hypervisor has to identify the nature of the guest-physical address the virtual machine tried to access.

- The address might belong to an emulated device; in this case, the hypervisor has to identify the device and pass control to the appropriate emulation driver;
- if shadow paging is active, the address might belong to the virtual machine's range of allotted memory; in this case, the hypervisor has to pass control to the shadow paging module to create the corresponding shadow page table entry;
- otherwise, the virtual machine has attempted an illegal memory access and should be terminated.

[1] The Linux kernel equivalents are `kmalloc()`, which is used to allocate small internal structures, and `get_free_pages()` for whole pages, e. g. to grow the heap or stack of a userspace process.

For this reason, we also generate lists of memory areas, each area denoting one contiguous mapping between virtual and physical address ranges with associated permission flags. In order to bring lookup operations in these tables on par with page table walks, we additionally create binary lookup trees on top of these lists. It has also proven beneficial to link areas which map the same physical address range with an additional pointer—this makes it easier for the hypervisor to inspect a trapped guest instruction, as it has to reference it by its hypervisor-virtual address instead of the guest-virtual or guest-physical address received through the trap handler.

All these data structures are compiled into the bootable image—we examine this in greater depth in Chapter 4.

3.2.2. Virtualization Interface

As the primary motivation for a microkernel was always to replace a regular operating system kernel, not implementing a virtualization layer, each microkernel exposes some form of fabricated API. As we have elaborated in Chapter 2, when support for Linux guests was introduced in the L4 family of microkernels, this was achieved by shoehorning the Linux kernel into the tasks-and-threads model mandated by the L4 API[91].

As our design is a true hypervisor, we have chosen to present a "CPU-like" interface to its guests that mimics native hardware semantics as close as possible. The state of these so-called virtual CPUs (VCPUs) includes the full register set of the simulated CPU, i. e. the full privileged and unprivileged register banks, as opposed to regular threads which would only contain unprivileged state. In these VCPUs we can then execute unmodified guest operating system code, while the hypervisor traps and emulates certain instructions to maintain the illusion. We uphold this paradigm even for those architectures where hardware virtualization is unavailable: although this requires changing the guest OS kernel by replacing sensitive unprivileged instructions with explicit traps, we nevertheless still abstract at the VCPU interface instead of introducing a custom virtualization API, as the necessary changes are much less intrusive.

From the interface perspective, we treat multi-core VMs, i. e. guests that consist of more than one simulated CPU, no different than single-core VMs. Each VCPU is allocated its state save area, and VCPUs of the same multi-core VM "happen" to execute on shared memory. The hypervisor merely has to pay attention to architecture-specific intricacies of memory coherency. To give an ARM-specific example, VCPUs may be executing on different elements of the physical CPU topology than their virtual CPU identifiers suggest; in this case, these VCPUs might issue memory barriers and cache flush operations with a smaller visibility than necessary, which the hypervisor needs to upgrade to maintain transparent coherency. Luckily, ARM provides such architectural features at the hypervisor level.

3.2.3. Scheduler

The debate concerning whether to include a scheduler and scheduling policy decisions in a microkernel is still unsettled. Opponents claim that this would violate Liedtke's minimality principle, as scheduling decisions could easily be implemented outside the microkernel. Proponents counter by citing the additional overhead spent in address space switches and privilege level transitions; in fact, no satisfying microkernel-based system with a userspace scheduler has been presented[32].

This latter argument is exacerbated when we consider hypervisors instead of classical microkernels. Instead of switching to a different address space, we would have to perform a VM switch to invoke an external scheduler, which entails saving and restoring much more state—and the privilege level transition in our case is a world switch. Both operations are far more expensive than their microkernel counterparts.

We have therefore chosen to include a scheduler in our implementation. For our prototype we have opted for a simple single-priority round-robin policy. This concept requires two data structures: a variable holding the currently executing VCPU, and a queue of VCPUs which are ready to continue execution. As we cannot create and destroy queue elements at runtime, we decided to implement the singly-linked queue as pointer members inside the VCPU state structure itself. Thus dynamicity is reduced to the technical minimum, as the execution sequence of VCPUs depends on runtime events and therefore cannot be determined beforehand.

We note that due to the individual instantiation of our hypervisor on each physical CPU, neither locks nor other additional variables are required. As we have chosen not to provide migration[2], no cross-CPU modification of data structures is necessary, which greatly simplifies our model. The variable holding the current VCPU is thus also local to each CPU.

Adding more scheduling classes or even moving to a multi-stage scheduler is a straightforward extension. All that is required are additional pointer members inside the VCPU structure for the new queues. If the added classes require some storage for bookkeeping (e. g. priority values, elapsed budget, period), these can be included in the generated data structures as well. Classes with hard constraints could even employ the compile-time data generation engine to check the permissibility of the given configuration, such as whether all scheduling deadlines can be met, and even compute a valid parameter set (hyperperiod, scheduling plan). As devising useful VM scheduling policies is a complex task in itself (see for example the analysis of the then-available schedulers for the *Xen* hypervisor in [23]), we leave these endeavours as future work.

3.2.4. Interrupts

Operating system implementations which do not target the embedded devices market usually design their interrupt dispatch subsystem with a dynamic core: once a driver has completed probing and configuring its assigned peripheral device, it will register an interrupt service routine for the determined interrupt line and then configure the interrupt controller to enable the line.

On generic platforms like the x86 PC, which are built around the concept of extensibility, this is a sine qua non: devices on the PCI bus (even whose presence has to be determined through bus enumeration first) allow generous customization of address ranges and interrupt lines, the standard interrupt controllers themselves are freely relocatable in physical memory by writing a model-specific register (MSR), and management of advanced platform functionality like suspend is impossible without parsing the ACPI tables, which itself requires a highly dynamic kernel.

For embedded devices however we consider this generalization unnecessary. These platforms usually have a fixed memory map as well as an unchangeable assignment of interrupt lines to peripheral devices. This insight led us to use static interrupt dispatch

[2]The benefits of migration on overall resource utilization in e. g. data center scenarios are well-known[19], but these advantages do not apply for our embedded platform use cases, as resource utilization is of very little concern in safety-critical environments.

tables, one table per supported platform. As it is placed in read-only memory, we can reason about interrupt dispatch invocations just like any function call, removing possible races between deregistration of a handler and arrival of the last interrupt (which might then lead to calling a NULL pointer). Interrupt pass-through to a VM is possible by having the configuration system generate a second read-only table which identifies the VM target for each interrupt line.

If such pass-through is in use for a given configuration, we do require dynamic masking of lines in the interrupt controller. This follows from the problem that the hypervisor has to prevent a pending interrupt that is configured as pass-through from being continuously signalled as long as the target VM has not quiesced the device yet. The only possibility to achieve this is to temporarily mask the line until the target VM reports the device as handled[3].

3.2.5. Emulated Devices

We require all desired device emulations to be specified as part of the scenario configuration. Every such device usually needs a modifiable data structure to maintain the emulated runtime state, and is accompanied by a set of guest-physical memory ranges to which the device will respond[4]. These ranges are compiled into a list and—strikingly similar to the tree we generate from the list of address translation ranges in Subsection 3.2.1—into an overlay binary tree, as the guest fault handler has to perform address-based lookups in order to locate the corresponding emulation driver for an attempted memory access.

However, generating the required state structures and faithfully emulating device functionality has proven to be difficult for multi-core VMs. We expect more complex device emulations to be provided by dedicated "driver VMs", but even the three basic device types offered by the hypervisor (UART, timer, and interrupt controller) cover all flavours of device locality: fully VCPU-local (timer), partially shared (interrupt controller, which contains a VCPU-local interface to manage the local interrupts and a shared distributor interface), and shared (UART). We take care of these differences in VCPU emulation state sharing when we design the multicore model for our hypervisor in Section 3.3.1.

Finally, we face another problem regarding faithful device emulation which is not specific to our design: the emulation of atomic read/write registers. The ARM Generic Interrupt Controller contains several groups of registers which allow setting (or clearing) bits atomically, i. e. writing value v to a register with current value r results in the new value $r \mid v$ (or $r \& \bar{v}$, respectively). As these operations are atomic on the actual hardware device, they are race-free even if several CPUs perform concurrent writes to these registers. In order to replicate such behaviour in a lock-free manner, we have to resort to the corresponding architectural instruction pattern for race-free memory updates (x86 `cmpxchg`, ARMv8 `ldaxr`/`stxr`).

3.2.6. Events and Timers

As we cannot make any assumptions concerning the number of available hardware timers on a particular platform, we have decided to design our event queue based on

[3] On ARM there is the additional possibility of keeping the line in the *active* state instead of masking and deactivating it.

[4] Devices may also be accessible through other architectural mechanisms than physical memory addresses, such as x86 IO ports or ARM coprocessor registers

a single hardware timer. Similar in spirit to the scheduler design, we define a singly-linked list of queued events, sorted by deadline, and a global variable identifying the earliest of these events.

Correspondingly, we also have to pregenerate all possible event structures which might be inserted into this list. One of these is the timer event that the hypervisor uses to signal the end of a VCPU's timeslice. Additionally, we preallocate one timer event structure for every emulated timer device; as one of our timer emulation options, the ARM standard timer peripheral SP804, actually consists of two independently programmable timer units, the emulation state structure for these devices has to contain two such event structures.

This construction models the hardware semantics as close as possible. Complexity is slightly increased by the interaction of guest-programmed timers with hypervisor scheduling and hardware support for virtualized timers. We support the following two modes of operation:

- emulated mode: The timer device accessible by a guest is purely virtual, its device state is fully managed by the hypervisor, and programmed timer events are always managed by the hypervisor's central event queue.
- virtualization mode: If the platform contains a timer chip with separate timers for guest and hypervisor, the guest is allowed to freely use the guest timer during its timeslice. Programming and delivery of events is handled by the hardware (and the hypervisor's interrupt delivery module). If a guest has programmed a timer event that has not expired yet when that guest is to be descheduled, the virtualized timer is disabled and the programmed deadline is carried over into the hypervisor's main timer; otherwise the virtualized timer would cause an interrupt to a different guest.

3.2.7. Inter-VM Communication

We also have to provide a mechanism to allow VMs to interact with each other. While communication between the VCPUs of a VM is usually provided by the emulated interrupt controller and thus part of the hypervisor's faithful device emulation, communication from one VM to another is an extension of the hardware model, just as IPC on a microkernel extends the model of isolated userspace processes. In a virtualization setting, this feature is usually provided by two complementary mechanisms: signalling and shared buffers.

The relatively unconstrained possibilities of defining memory ranges and attaching them to one or several VMs (see the definition of our configuration toolkit in Chapter 4) already allow for the creation of communication buffers between VMs. As permissions can be assigned asymmetrically, these buffers can even be configured as one-way channels, limiting the flow of information.

We have chosen to implement the signalling channel through virtual, i. e. software-induced interrupts. VCPUs can issue hypercalls, specifying the intended channel number, which causes the delivery of an interrupt to the configured destination VCPU of that channel. As we give out channel identifiers in linear order, we do not have to generate any special lookup structure to speed up the hypercall—a bounds check and an array member dereference is sufficient.

According to our principle, the creation of new channels at runtime is prohibited, so this mechanism is completely devoid of dynamicity. We use the established concept

of capabilities[87], i. e. opaque tamper-proof handles which represent internal objects of a higher security context and which can be invoked to trigger an action using that object. In our case, the represented objects are software interrupt gates that specify a target VM and an interrupt number, and the VM holding the capability is able to activate the defined interrupt.

We implemented channels as a communication means between VMs instead of individual VCPUs, as the permission to raise a virtual interrupt should not be tied to a particular VCPU of the sending VM. Equally, the destination VM is able to configure its (emulated) interrupt controller to deliver the incoming interrupt to a VCPU of its choice.

Transmission and delivery of said virtual interrupts, whether among VCPUs of a single VM or between VMs, may require crossing a physical CPU boundary. We explain the details of our multikernel design and the consequences for collaboration across CPUs in Subsection 3.3.1.

3.2.8. Virtual TLB

The virtual translation lookaside buffer (VTLB) is one of the cornerstones of paravirtualized execution of VCPUs. Its modus operandi has been described extensively, e. g. by Kivity et al.[56], so we only recapitulate it here briefly before we move on to our implementation in the light of our Principle of Staticity.

On platforms without a two-stage memory management unit (MMU), the effective page table for execution of a guest has to be maintained by the hypervisor[5]. The VTLB is put into action whenever a memory access attempted by a guest causes a fault, as this indicates that the effective page table is lacking an entry for the desired address. In order to handle the fault, the "walker" component of the VTLB inspects the guest page table. If the guest does not have an entry for the faulting address either, the fault is real and injected into the guest. If on the other hand the guest page table does contain an entry, the fault is a shadow fault, incurred by the additional indirection. In this case, the VTLB checks the target address of that guest page table entry and the associated permissions, applies the hypervisor-controlled second stage of translation to the address and potentially also restricts the permission set, and invokes its "pager" component to add the resulting combined two-stage translation to the effective page table.

As the core operation of the VTLB is dynamic management of a page table, it runs counter to our Principle of Staticity. On paravirtualization platforms the VTLB is an indispensable component however, so we have to make concessions and try to contain the dynamic elements as far as possible. Most importantly, we have to pay attention that allocation and deallocation of page table directories do not introduce covert channels across VMs (see Section 8.2.2).

We solve this issue by restricting each VTLB instance to draw memory from a dedicated set of pools instead of letting them operate freely on the parts of memory that have been left unassigned during configuration. This decision yields several benefits:

[5] We recognize that research has suggested that selective use of shadow paging may be a worthwhile strategy even for platforms where a MMU capable of two-stage translation is available[95, 38]. As the VTLB violates our core design principle, we only consider platforms where its presence is mandatory.

- Covert channels between VTLBs and thus possibly VMs are prevented, as starvation in one VTLB cannot be observed from any other. Eviction has to be handled locally.
- The explicit assignment of memory allows the creator of a configuration to allocate resources unevenly, e. g. configuring substantially larger memory pools for an Android VM than for its companion Linux driver VM.
- Giving each level of the VTLB its own pool further reduces implementation complexity. Thus the configuration system can determine at build time whether the alignment constraints of each page table layer are met, structural errors like references to the same directory at different depths are trivial to determine (both at runtime and by a prover), and allocator fragmentation at runtime is impossible[6].

As the VTLB is effectively an additional level of caching between the source (the guest's page tables) and the sink (the hardware page table walker inside the MMU), an optimal implementation strives to retain as much information as possible, thereby reducing the impact of the VTLB to a minimum. Our experiments on the efficiency of sharing VTLBs across VCPUs of the same VM as opposed to running individual instances have so far been inconclusive[7]. We therefore opted for the less complex and more faithful implementation that keeps the VTLBs of each VCPU separate from all others.

Platforms which are not even capable of basic register virtualization require an even more complex form of VTLB. As there are only two available privilege levels on these devices, the guest operating system kernel and the userspace applications both have to run in the same unprivileged mode. In order to still provide proper separation between these two, the VTLB has to split each address space in two, and switch between them on each transition between guest kernel and guest userspace (cf. Figure 3.1).

Furthermore, there is no world switch operation that is atomically invoked whenever guest execution is interrupted. Therefore the VTLB has to ensure that at least the low-level hypervisor trap handlers for interrupts, faults, and system calls are always mapped and protected from modification by unprivileged code. As a consequence, the VTLB has to reserve a certain fixed window of the virtual address range for these handlers.

This in turn means that faults on addresses in this window cannot be resolved by adding a page table entry, even if the guest page table contains an appropriate entry, because doing so would shadow the trap handlers. If it is desired to allow VMs to access these addresses, the hypervisor would have resort to emulation—however, it is usually far simpler to gently modify the guest operating system not to use a certain portion of the virtual address range.

After developing our own prover toolkit and using it to derive integrity properties for our hypervisor core, we attempt to extend those invariants to the VTLB (cf. Section 6.6). Yet, the dynamic nature of the VTLB already foreshadows that we might

[6] While legacy x86 paging uses 4kB level-1 page directories as well as 4kB level-2 page tables, ARMv7 "short paging" uses 16kB level-1 directories and 1kB level-2 tables. Allocating these from the same pool would cause fragmentation and needlessly increase complexity.

[7] While it may seem beneficial at first glance to share the cache-like VTLB across VCPUs, the necessary synchronization in the hypervisor might outweigh the advantages and add—regardless of actual performance gains—undesired complexity to our implementation

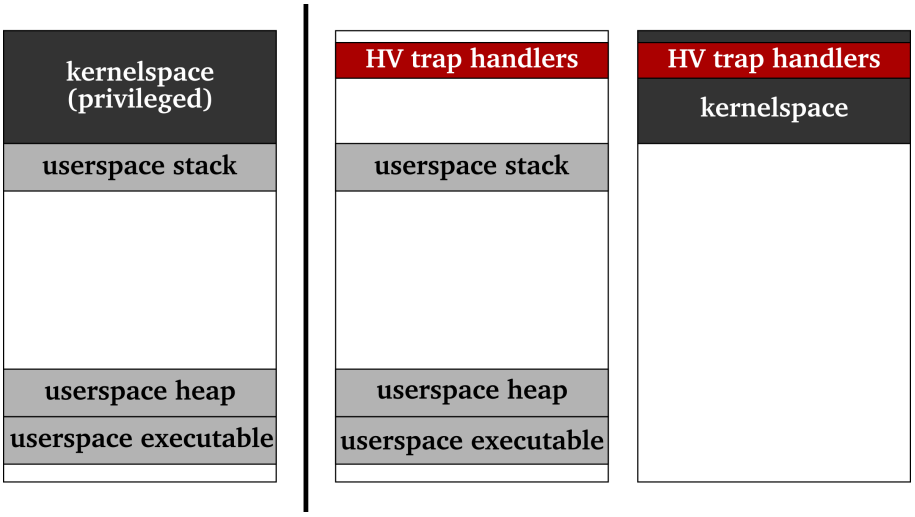

Figure 3.1.: Address spaces created by the VTLB on platforms with (left) and without (right) register virtualization and third privilege level. On latter systems, VM kernel and its applications have to be segregated in different address spaces, as both run in the same unprivileged execution mode, and the hypervisor trap handlers must always be mapped.

find our prover toolkit, which draws its power and simplicity from our core design principle, to be ill-suited for the task in its present state.

3.3. Related Decisions

With our core hypervisor components aligned with the imposed Principle of Staticity, we now turn to several ancillary decisions. While these are not directly mandated by the principle, they nicely complement the characteristics established above to complete the picture.

3.3.1. Multicore Architecture

The design spectrum for multicore-capable operating systems ranges from "giant lock" kernels, which place all execution in kernel mode under the umbrella of a single spinlock and thus effectively disable all concurrency inside the kernel, to "shared nothing" multikernels, a design idea proposed by Baumann et al.[14], which completely removes shared memory regions (and therefore the need for any synchronization primitives) and instead has its kernel instances communicate with each other through message passing, much like a distributed system.

The original motivation for the multikernel design was comparing the scalability of implicit coherency protocols for shared memory as opposed to explicit message passing for state replication on manycore platforms. As our targets are embedded devices instead of large-scale server platforms, the given reasoning does not apply. Still we have decided to adopt this model, albeit for a different architectural reason: in our design, there is literally nothing to share. Let us analyze the list of hypervisor data structures whose equivalents would be protected by one giant or individual fine-grained locks in common kernels like Linux:

- Data structures which contain VM configuration data (number and placement of memory allocations, list of communication channels, etc.) are relevant for all hypervisor instances that govern execution of a VCPU of that VM. This data is read-only, so sharing it is allowed and poses no synchronization problems.
- The VCPU state save structure is invariably tied to the hypervisor instance the VCPU has been placed on by the static compile-time allocation algorithm (or by explicit configuration), as we disallow migration.
- Global variables used by the hypervisor itself (e. g. scheduler queue, the list of active and queued timer events, pointer to the currently executing VCPU) are local to each instance as well, as scheduling decisions and timer events are always handled locally.

Even though the scheduling decisions happen without any communication between the hypervisor instances, we note that we could still implement gang scheduling[76] for multicore guests in order to battle lockholder preemption (as described by e. g. Uhlig et al.[90]), as all scheduling parameters are decided during static configuration, and a global platform clock as common timing reference is usually available.

We end up with only one class of data that does not fit the multikernel model: emulation device state shared across VCPUs of a VM, as we have mentioned in Subsection 3.2.5. The situation is less adverse than it may seem though, as the semantics of real peripheral devices do not guarantee any particular order of accesses by different CPUs. For our purposes it is therefore sufficient to place the data structures for such emulated shared peripherals in a shared writable location that is accessible to the hypervisor instances which host VCPUs of the same guest and leave it to the guest operating kernel to maintain consistency.

The remaining puzzle piece for our multikernel design is the concrete realization of the message passing facility. The architectural device for communication between CPUs is the inter-processor interrupt (IPI), which carry by their nature virtually no information[8]; in order to transmit messages among CPUs, we thus have to complement the signalling mechanism once more with messaging buffers. We allocate these in a matrix fashion such that each pair and direction of communicating partners (CPU n to CPU m) is given its private buffer. This facility can then be used to both implement virtual IPIs, i. e. signals among VCPUs of the same VM, and to handle communication across VMs if the communication partners run on disjoint sets of hypervisor instances.

3.3.2. Execution Model

One design choice with direct influence on the implementation of low-level operations pertains to the execution model. Operating system design terminology distinguishes *process-model* from *interrupt-model* kernels [9]. The former class devotes an individual kernel stack to each userspace application (i. e. to each guest in a hypervisor setting), which is used to capture the saved state in case of an interruption (or voluntary control transfer as for a system call). If the interrupted activity is scheduled out, the kernel stack is switched along with the address space, so when the activity is later

[8] The ARM architecture provides 16 distinct software generated interrupts (SGIs); x86 allows an arbitrary 8-bit vector number for IPIs, but this is shared with other peripheral interrupts

[9] For a discussion of the concept and classification of a few popular kernels, see Ford et al.[35].

scheduled back in, its kernel stack is reactivated and the return from the interruption can unwind the stack just as if no scheduling had ever taken place.

Implementations of the latter class only reserve a single kernel stack for each CPU, regardless of the number of applications. As the stack cannot be used to hold state information in this configuration, the process state has to be saved into an appropriate "state save area"—this data structure is commonly termed a *continuation*. After the entry reason has been handled and the scheduler has decided which continuation to activate next, the kernel performs an *upcall*: it restores nonprivileged state from the chosen continuation, resets the kernel stack pointer to the top, throwing the recorded call chain away instead of unwinding it, and resumes the interrupted activity.[10]

The advantage of the interrupt model for formal modelling purposes is that the state of interrupted VCPUs is always kept in the designated state save area instead of being interspersed on its kernel stack. It is additionally much easier to reason about kernel code paths if we can model execution as a linear path from entry to upcall instead of performing entry and exit through different call chains due to a stack switch.

Finally, given that we do not consider nested exceptions, each upcall rewinds the stack pointer to the top of the stack. This provides a clean slate to the next hypervisor entry and makes it very simple for us to specify a stack pointer invariant between kernel entries: it always resides at a known fixed location. We can further easily verify that the stack will never be overflown by determining the code path through our hypervisor with the maximum stack usage. We discuss nested exceptions and preemptibility in the following subsection.

3.3.3. Preemptibility

Another design choice with far-reaching implications is the question of preemptibility. Contemporary operating systems with substantial amounts of device driver code usually provide at least voluntary preemption: long-running operations are dotted with "preemption points" where the operation can safely be interrupted by an activity with higher priority (either by briefly enabling interrupt delivery, or by polling the interrupt controller for pending requests). Kernels which aim to provide low-latency semantics to its applications even support involuntary preemption, which allows high-priority activities to interrupt the kernel everywhere at the cost of more complex synchronization primitives.

Adding support for preemptibility would undoubtedly complicate our model, as it disturbs the simple linear execution we assume. We pay for this by impacting our latency behaviour with the longest hypervisor code path, which we measure in Section 5.3. While we are analyzing the potential to add selective preemption in Subsection 6.5.2, we leave the extension of our proof to a (partially) preemptive design as future work. However, we demonstrate a different way to reduce the time spent in the hypervisor in Chapter 7 that has negligible impact on our proof methodology, but whose applicability in turn depends on circumstances of the scenario.

Concerning the provability of preemptible system software, we refer to the work by Xu et al.[96], who have developed a verification framework for preemptible OS kernels with several layers of nested interrupts. It remains a promising future research objective to attempt to extend our work to incorporate preemptibility.

[10] If nested exceptions are supported, continuation selection and stack pointer repositioning are more complex. We omit these details here for clarity.

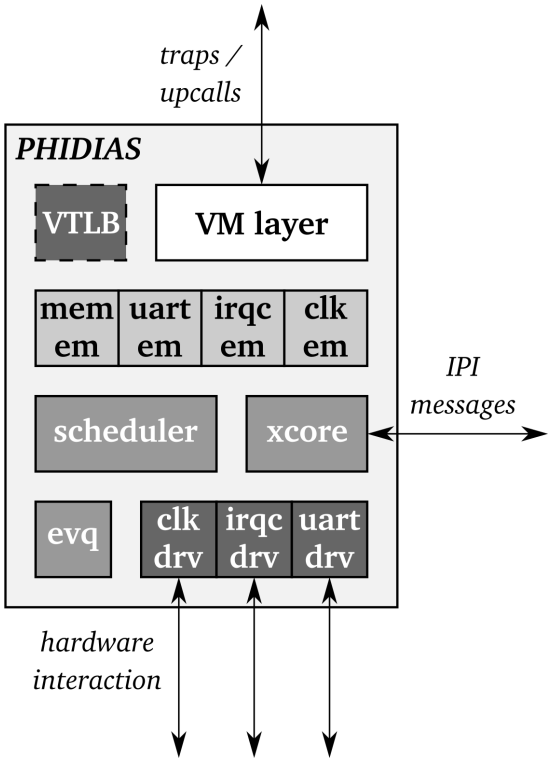

Figure 3.2.: Structural overview of Phidias.

3.4. Phidias

Phidias, the Provable Hypervisor with Integrated Deployment Information and Allocated Structures, is the second[11] implementation of our Principle of Staticity and its design consequences. Our choice of C as implementation language is straightforward: object-oriented languages like C++ and Objective-C would only add dynamic elements, which we are specifically trying to avoid—and if we restricted ourselves to use the parts of those languages we consider acceptable, we effectively arrive back at C. Additionally, it is convenient to stay close to assembler due to the frequent need of inline assembler in an operating system kernel in order to perform low-level operations.

Beyond the feature set we have described above, we have only added one additional emulation driver: a generic memory emulation that responds to read and write operations targeted at its address range with a selected fixed behaviour (e. g. reads always return zero, and writes are discarded). This is a very convenient feature to allow the execution of unmodified platform-specific Linux kernels without heavily truncating their accompanying device tree: Using this "memory emulation" feature, Linux will still try to access the device at the expected address and probe its feature set, but likely fail, as the values read do not match what the initialization function of the device driver expects.

A structural overview of Phidias is shown in Figure 3.2 with a few details omitted, such as the denotation of which components contain architecture-specific elements. The "VM layer" at the top is responsible for performing the world switch during

[11] Our first implementation, Perikles, is undergoing integration in industry products in the automotive sector at OpenSynergy[74].

upcall and entry and for dispatching events to the appropriate subsystem. If a VCPU triggers a fault, the corresponding emulation driver or the VTLB (if present) is tasked with performing the necessary actions to resolve the fault. At the heart of the implementation we have the scheduler, the event queue, which keeps track of programmed timer events, and the "xcore" module, which engages in communication with other instances of our hypervisor in order to relay vIPIs and triggered interrupt capabilities. Below all this we finally have our minimal set of required drivers.

What the figure cannot show are all the components we have removed to arrive at this design:

- Our VM layer does not contain any custom hypercall interface; it is merely a translation unit that decodes the architecturally defined entry reasons and dispatches appropriately. When paravirtualizing, the layer does expose a hypercall interface, but this is only modelled to handle the sensitive instructions executed by the VCPU.
- There is no custom map/unmap module as in L4. If a VTLB is required for virtualization, it is still not directly controllable, and its resources are coupled to the VCPU.
- Finally, there is no memory allocator in Phidias.

The figures for the implementation size of Phidias are listed in Table 3.1. We have reached an overall source code size of 7.5 KLOC, which is comparable to the non-proprietary x86-only version of *XtratuM*[33]. Of our codebase however, only approximately 5 KLOC are actually compiled at a time, as only one architecture and platform can be selected and usually only a subset of the hardware device drivers and emulation modules are included as well. The numbers for *Xen*, which has recently been ported to ARMv8 as well, merely serve as a point of comparison.

In terms of minimality, we have barely left room for improvement, as all remaining subsystems are vital for the hypervisor to operate and all dynamicity has been removed. Cutting out those optional modules that only serve debugging purposes, such as the internal trace buffer and the UART driver, does not result in a significant size reduction (around 100 lines each). However, we have bought this new level of runtime code minimality with an increase in complexity and size of the configuration framework, as we will see in the following chapter.

Component / LOC	**Xen**	**XtratuM**	**Phidias**
architecture-independent core	43,022 + 0	1,459 + 0	823 + 0
hardware device drivers	35,134 + 0	301 + 0	443 + 0
emulation drivers		904 + 0	1,173 + 0
ARMv8 architecture support	1,765 + 1,074		257 + 523
x86 architecture support		1,514 + 444	316 + 211
TOTAL	263,523 + 7,111	7,106 + 444	6,134 + 1,305

Table 3.1.: Lines of code (C + Assembler): *Xen*, *XtratuM* (v2.6, GPL), and Phidias. *Xen* codebase does not separate emulation and hardware drivers. Total sum includes all supported architectures and platforms for each hypervisor.

4. Static Configuration

Building a static hypervisor as described in Chapter 3 is only worthwhile if the scenario-specific configuration is easily modifiable. Our hypervisor implementation therefore requires an accompanying configuration framework, which is integrated into the hypervisor build process and generates the data structures for a particular deployment setting.

In order to make the build process more transparent and at least partially modular, we impose the additional requirement to aggregate all configuration settings which do not have an undeniable relation to the compilation of the hypervisor itself into a separate build artefact. We are going to refer to this artefact as the "addendum". This split removes the burden of recompiling the hypervisor each time the configuration is modified as long as the basic settings (those defined as affecting the compilation in Subsection 4.1.1) remain the same.

As the input to our framework is going to be purely declarative and fits well into a tree-like structure, we consider XML an optimal source language. Its widespread adoption and excellent tool and library support make it a perfect selection for our purpose.

We begin our development of this framework by enumerating all pieces of information we have to capture with our configuration. Our source of reference for this task will be the structural requirements we have set out in Chapter 3. Afterwards we discuss the steps our framework has to perform in order to transform an input configuration into data structures and additional metadata that can be used by our hypervisor.

4.1. Configurable Elements

The elements of our configuration system fall into two categories: those which modify the behaviour of the hypervisor itself, and those which determine the virtual machines that shall be able to execute and their environment. We will discuss both categories in turn. A complete example specification is available in Appendix A.2.

4.1.1. Hypervisor Core

One of the benefits of designing a build-time configuration framework is that we can use it to directly influence the compilation of the hypervisor core binary, similar to the well-known `make menuconfig` which is provided by the Linux kernel developers to customize its compilation. We define the following settings as influential on the hypervisor build process:

- The *selection of CPU architecture and target platform SoC* defines which port of the hypervisor shall be compiled. This includes the architecture-dependent low-level assembly code, entry and exit functions specific to the type of virtualization, and platform-dependent bootup and application processor activation code. We

can even use these settings to select an appropriate bare-metal cross-compilation toolchain.

- The *hypervisor base load address* has to be specified in both physical and virtual form. The physical address may be required during the initial bootup phase to perform the activation of paging, while the virtual address is directly inserted into the linker script to control the address the resulting hypervisor ELF binary is configured to execute from.

- The *selection of drivers* for hardware devices and for the required emulation devices is also part of the configuration. This mechanism is closest to the Linux `menuconfig` system. It allows us to avoid compiling unused code into the hypervisor and to remove runtime checks for platform drivers (as usually only one of each type is compiled in).

Beyond these items, the hypervisor also requires a few data structures for its operation:

- As each hypervisor instance boots up into its own address space with its private variables as discussed in Subsection 3.3.1, we need a global structure that contains pointers to the initial bootup page table[1] as well as to each instance's runtime page table.

- We also need the list of memory ranges accessible to the hypervisor core. This is required by the hardware drivers to locate their devices' memory-mapped input/output (MMIO) ranges as well as by the trace buffer module to find the base address and size of its allotted memory region.

- The number of desired hypervisor instances is required by the platform bootup code in order to activate the corresponding number of physical CPUs and bootstrap an instance on each.

4.1.2. Virtual Machines

The central part of our configuration is without a doubt the definition of the virtual machines which shall be executed under control of our hypervisor. Their configurability touches each of the subsections of Section 3.2, however the necessary data structures have differences with respect to their runtime usage: most of them are going to be used as read-only lookup tables, but some parts (especially those mentioned in Subsection 3.2.5) are writable or even possibly shared writable across different hypervisor instances.

Let us visit each detail of the guest configuration in turn and sort it into the appropriate access category.

- The *number of virtual CPUs* per guest is unchangeable at runtime and therefore belongs into the read-only section. The scheduling parameters of these VCPUs (cf. Subsection 3.2.3) could theoretically differ across a VM, however we see little use in offering this freedom. Thus we deem it sufficient to offer configurability of the scheduling class and parameters at the VM level. While all these settings

[1] The switch from non-paged to paged mode execution usually requires going through an intermediate stage with a special page table. The mechanisms involved are well-documented in the architecture specifications[6, 7, 49, 3], so we omit the details here.

belong in the read-only section as well, certain scheduling classes may require additional bookkeeping variables per VCPU at runtime, which would have to be placed in a writable location. For our prototype, we have not included any special scheduling classes, so no writable variables are required.

- The *memory configuration* (cf. Subsection 3.2.1) of a VM is by design static, i. e. read-only, and naturally applies to all of its VCPUs. As discussed in our structural design, we compile this list of accessible memory locations both into a flat list and into a binary lookup tree for faster access.
- The list of *capabilities* granted to a VM as defined in Subsection 3.2.7 is another read-only member of the VM control block.
- The assignments of *pass-through interrupts* to VMs (cf. Subsection 3.2.4) is compiled into a globally visible read-only table.
- The types, parameters, and corresponding *emulated memory ranges* (cf. Subsection 3.2.5) make up the remainder of the VM configuration. While these particular values can go into the read-only section as well—the memory ranges again compiled into a flat list and into a tree—, each of the selected emulation devices also requires a data structure that embodies its (mutable) device state. These have to be placed into writable or even shared writable memory, as we have already indicated in our structural discussion; in the case of the emulated interrupt controller, the state structure must even be split into a shared and a non-shared portion.

Finally, there are a few elements which we do not include in the list of pregenerated data structures, but which are directly allocated as raw memory resources. These items have in common that their size is at least one page and that their initial state consists of all zeroes:

- The *VCPU state page*, which contains the VCPU's privileged and unprivileged register state, is kept separate from the VCPU control block for several reasons. Firstly, whereas the control block is mostly immutable except for the optional scheduling parameters, the state page has to be writable, so splitting the two allows us to set different protection settings. Secondly, it is extremely advantageous for paravirtualization settings if the executing VCPU is given access to its own state page. As the VCPU state page does not contain any sensitive or security-relevant data, this does not violate any guarantees. On the other hand, the VCPU is given the opportunity to easily modify registers of its own privileged state, e. g. the virtual interrupt bit, without causing a trap into the hypervisor.
- The *backing memory pools* for the individual page table levels of a VCPU's shadow paging unit sum up to roughly 512 KB–1 MB for a VM running a usual workload like a small driver-only Linux. This memory would require initialization to zero as well, and its inclusion into the data structures would needlessly bloat the resulting final boot image. Furthermore, the individual levels may require different forms of alignment, which might introduce additional padding if all this was included into the addendum. We therefore decide to allocate these from raw memory as well and perform the initial `memset()` at runtime.

- The *page tables* play a special role and are handled separately. We turn to their creation in Section 4.5.

4.2. Completing and Sanitizing

Before the list of data structures we have accumulated above can be generated, the configuration framework is expected to perform two additional tasks: resolving underspecified attributes and checking permissibility.

Allowing and resolving underspecification is a necessary transformation of common runtime selection algorithms to our compile-time framework, the prime example being the allocator. When faced with a request for a given amount of memory, a runtime allocator might simply hand out the first appropriately-sized contiguous block of free memory it finds, as the subsystem issuing the request does not care about the exact (physical address) location of the returned memory. In fact, during subsequent runs of the same platform, maybe due to timing differences, interaction with other subsystems, etc., the location returned may differ.

In our scenario, we still do not care which exact part of memory is assigned to which VM, which part is going to hold the hypervisor trace buffer, and which page will be used as the hypervisor stack. In fact, explicitly specifying these locations would make the configuration less portable, as the physical addresses chosen will only be applicable to a subset of the supported platforms. Instead, we want to enable configuration writers to only make the important choices (in the above cases, how much memory to assign), and leave the determination of the uninteresting ones to the framework.

The same reasoning holds for the scheduling parameters: rather than having to define exactly how each quantum of the scheduling period is going to be used, specification designers should rather set the crucial parameters (e. g. assigning the required budget and period to real-time VMs) and let the framework fill in the parameters of the remaining VMs.

Permissibility is closely related and can be verified alongside supplementing the underspecified elements with their missing parameters, as the latter will automatically fail if the former is violated. Instead of denying an allocation request at runtime or refusing to create another real-time VM if its scheduling parameters would make other scheduling constraints unsatisfiable, we can detect such overcommit situations at compile time. Usually these are detected if no memory can be found to satisfy a particular underspecified element attribute, but permissibility has to be checked even if everything is explicitly specified and no underspecification resolution is taking place.

4.3. Resolving Recursive Dependencies

While trying to fill in the underspecified attributes, we are confronted with several layers of recursive dependencies. These stem from the fact that the addendum, the generated page tables and optional additional VM bootup blobs (e. g. Linux kernel images, initial ramdisks, bare metal VM binaries) are all concatenated to the compiled hypervisor to form the final binary image; however the base addresses and sizes of the individual components mutually depend on each other:

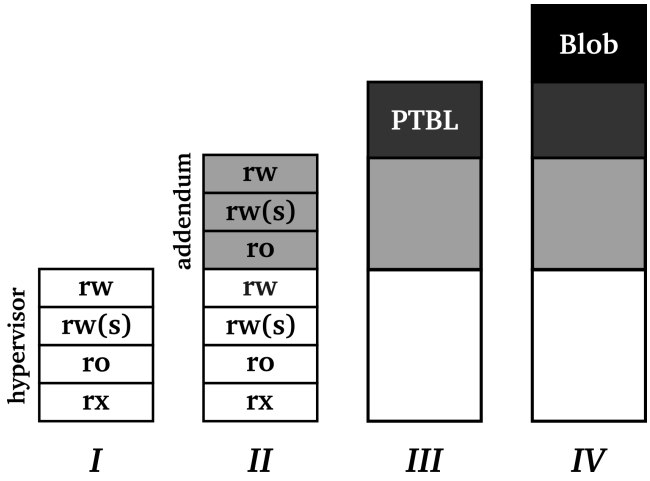

Figure 4.1.: Creation of bootable hypervisor image (steps I–IV). Components from bottom to top: hypervisor, addendum, page tables, and bootup blobs.

- The location of the hypervisor instances' and the VMs' page tables have to be included into the corresponding control blocks in the addendum.
- As the addendum has to be accessible to the hypervisor, its extents have to be mapped in the hypervisor page tables, which might theoretically change its size as the addendum grows.
- The VM bootup blobs must be similarly accessible and mapped, either to the hypervisor or to a service VM which sets up and boots other VMs. In both cases the corresponding page table might grow.

We solve these dependencies with two simple tricks. First, we note that the size of the addendum only depends on the type and number of data structures we are about to generate, not their contents. We therefore decide to place the addendum first, directly following the hypervisor binary, and generate it twice: once to determine its size, so we can compute the base addresses of the following components, and later again filled with the correct values.

Second, we solve the page table size dependency by introducing size estimates for average page tables of each paging format we support. As the page table for the hypervisor and the second-stage page tables for the guests tend to be sparsely populated and/or only contain superpage mappings at higher levels of the table hierarchy, these estimates tend to overshoot, and we leave it to the user to choose a more appropriate value. However, we thus obtain a fixed value for the size of the page table area, and are then able to calculate the base address of the bootup blob area.

The resulting series of concatenation steps is shown in Figure 4.1.

4.4. Generating the Data Structures

We now have a tree of XML entities with their corresponding attributes, some specified manually, other values chosen and added automatically—an example in both an initial human-created form and the final transformation result can be found in Appendix A.2. This tree is ready for conversion into a format that can be easily consumed by the

Addendum Component	write?	share?	size (bytes)
hypervisor master control		**X**	88
hypervisor per-CPU control			56
list of pass-through IRQs		**X**	2,048
VM control block		**X**	56
VM capability list		**X**	$16n$
VCPU control block			128
(HV and VM) memory range list & tree		**X**	$(40 + 24)n$
emulation device list & tree		**X**	$(40 + 24)n$
UART emulated device state	**X**	**X**	$[88, 112]$
timer emulated device state	**X**	(**X**)	$[96, 152]$
IRQ controller em. device state (local)	**X**		80
IRQ controller em. device state (shared)	**X**	**X**	504
VTLB emulated device state	**X**		$48 + 40l + 48c$

Table 4.1.: List of configuration data structures. Sizes of lists are given per element (n). VTLB size depends on page table depth (l) and number of cached contexts (c). Sizes for emulated devices with multiple available drivers are given as ranges.

hypervisor implementation. In order to avoid introducing a parser into the hypervisor, we have chosen to convert the XML entities into C source code, which is then compiled with the same toolchain as the hypervisor itself. The addendum is therefore directly binary compatible with the hypervisor, and careful linking ensures that pointers between data structures are directly usable as well.

The full list of items generated into the addendum are show in Table 4.1. This table also lists the size of each structure when compiling for our main architecture ARMv8[2]. A graphical representation of the references between the structures is shown in Figure 4.2.

4.5. Generating the Page tables

After the underspecification has been resolved and the size of the addendum has been measured, the framework is also able to generate the page tables which contain the effective address translations for the hypervisor itself and the second-stage translations (from guest-physical to host-physical addresses) for the VMs. Our analysis of different page table formats has shown that it is possible to build a single generic generator which is capable of emitting all currently supported formats (x86 legacy, x86 PAE, amd64, ARMv7 short, ARMv7 LPAE, ARMv8 with different granule sizes and different input and output address widths). The differences between the formats lie in the descriptor size (four or eight bytes), directory sizes and thus translated address bits per level, directory alignment requirements, and the location of the various permission control bits.

All these characteristics can easily be described using an additional XML file. We have provided parts of the paging specification files for ARMv8 and x86 for reference in Appendix A.1.

[2]Due to the architecture-dependent size of a pointer and due to padding inserted by the compiler, sizes differ between architectures.

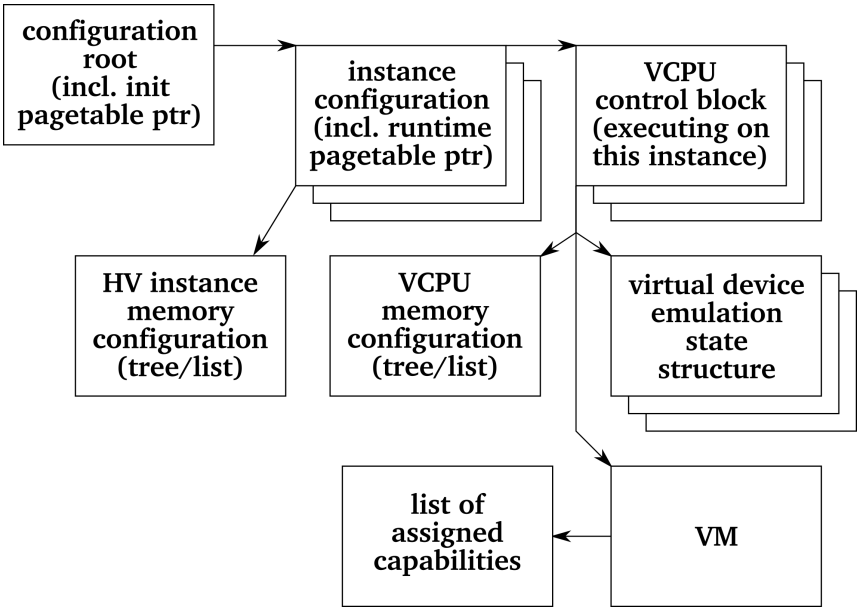

Figure 4.2.: Tree of configuration data structures.

4.6. Schism

We have implemented the series of steps described above as individual tools under the collective name Schism, the "Static Configurator for Hypervisor-Integrated Scenario Metadata". Each of those tools reads the current state of the XML specification and either performs the next set of transformations on it, thereby producing an updated XML specification document, or generates supplementary output files, such as the generated page tables or addendum source code. This procedure has the additional benefit that each intermediate step of the XML is accessible to the developer and can be independently validated[3].

An overview of the different build stages and their tasks is given in Table 4.2. The *precompilation* tool extracts the settings from the XML document that influence the compilation of the hypervisor as we have enumerated in Subsection 4.1.1 and provides them to the `make`-based build system of Phidias. *expand* inserts the desired supplemental XML definition files for the page table format and targeted SoC platform into the document. The *reparent* stage assigns memory allocations to memory resources provided by the SoC[4] and adds necessary default mapping requests, e. g. mapping the addendum into the hypervisor's address space, if those are not already present.

With the number of mapped areas per address space now fixed and the sizes of the area lists and lookup trees thus known, *generate* is now able to create the first version of the addendum, which is already correctly sized, but does not contain the final values, as most of them are not decided yet. Afterwards, *measure* inspects the compiled ELF files of the hypervisor and the addendum, thus determining the base address of the page table and the bootup blob areas. The next two stages,

[3] We have created a document type definition (DTD) for our specification hierarchy that enables formal validation at the XML level throughout the series of transformational steps in addition to semantic validation of the contained attributes.

[4] On x86 platforms the available memory may not be contiguous, as certain ranges may be reserved by the firmware (EFI, ACPI). ARM platforms on the other hand may provide different types of memory, e. g. SRAM and DRAM.

no.	stage name	XML updates	generated files	LOC
1	precompilation	–	hypervisor Makefile	244
2	expand	import SoC & arch XML	–	120
3	reparent	assign memory requests	–	204
4	generate	–	stub addendum	1,650
5	measure	ELF object sizes	–	248
6	layout_memory	physical addresses	–	174
7	layout_maps	virtual addresses	–	226
8	pagetables	–	page table data	338
9	generate	–	final addendum	(1,650)
10	combine	–	bootable image	333
TOTAL (including header files and helper tools)				4,067

Table 4.2.: Build process stages and LOC statistics (individual stages: C code only). The *generate* stage is used twice, but counted only once towards the total.

layout_memory and *layout_maps*, then assign addresses to all memory entities. The former lays out entities in physical memory and checks permissibility, i. e. validates that each memory resource is large enough to accommodate all its assigned requests. The latter then lays out the address spaces of the hypervisor and the second-stage translation regimes for the VMs, again checking permissibility while doing so[5].

Lastly, the *pagetables* tool generates the page tables for the hypervisor and the second-stage translations, checking if the overall size estimate is met (otherwise the build has to be reattempted with an increased estimate). Now the final pieces of information, the base addresses of all page tables, are known and *generate* can be invoked again to generate the final version of the addendum, and the *combine* tool is able to aggregate all parts of the build process (cf. Figure 4.1) into one bootable image in a format suitable for the target platform's bootloader.

4.7. Discussion

By comparing the source code statistics with those presented in Section 3.4 we are now able to present a complete picture of the effects of moving all dynamicity out of the runtime kernel into preparatory stages of the build process.

4.7.1. Code Deduplication and Extensibility

We immediately notice that our page table generator compares favourably with in-kernel paging implementations[6]: *Xen*'s ARM paging subsystem amounts to roughly 1,800 lines (some of it also related to cache flushing, so the number is slightly too high), and Linux ARM paging totals 1,400 lines, which is both significantly larger than our build-time stage of 900 lines including the XML paging specification.

The transfer from runtime code to a static framework also enabled us to build one generic architecture-independent page table generator. Even though ARMv7, ARMv8, x86, and amd64 all use the same basic concept of hierarchical page tables, operating systems that have been ported to each of these architectures such as Linux

[5] On 32-bit platforms with more than 4 GB physical memory, allocation of physical resources may succeed, but mapping everything into one address space would fail due to limited addressability.
[6] Measurements are taken from *Xen* 4.8 and Linux 4.10.

or BSD contain separate paging implementations. The framework we have created avoids such code duplication and offers an interface that invites more structurally similar page table formats to be added: each format specification takes up 20–30 lines of XML and requires no changes to the codebase of the framework (cf. Appendix A.1).

Spurred by this, we investigated the extensibility of Schism's memory layout tools and page table generator to an entirely different concept. As a preparation for future experiments on ARM platforms that implement a real-time profile, we added support for the memory protection unit (MPU) contained in those processors. Compared to their more powerful siblings, the memory management units, these MPUs contain similar logic to check the permissibility of an attempted memory access and to raise an access fault in case of a violation; however they do not offer any form of address translation.

The central challenge of this endeavour turned out to be the limited configurability of the ARMv7-R MPU. Instead of consulting a memory-resident structure of allocated memory ranges and corresponding permission bits, this MPU implementation contained a fixed number of registers which could be used to store enabled ranges and their accessibility (so-called "regions"), and these ranges were further constrained to powers of two. We therefore had to change the placement strategy for physical memory ranges to group entities with equal permission bits together so they could be covered by one MPU region.

Overall the extensibility study resulted in the addition of 130 lines to the physical layout stage and roughly 500 lines to the page table generator, the latter implementing a best-fit strategy to select the optimal region configuration and emitting the corresponding set of MPU register tuples instead of native page tables. We could then integrate this into the hypervisor bootup code for ARMv7-R, loading the register values into the MPU before activating it in much the same way as the existing initialization path read in and used the configured page table base addresses into the MMU before activating it. In both cases, the hypervisor does not have to contain any special memory management code beyond the early initialization phase.

4.7.2. Complexity Shift

Finally we have to discuss one characteristic of our approach we have neglected so far: the effect of our complexity shift on developers and integrators using our solution.

Taking the step back from a probing, autoconfiguring, dynamic hypervisor such as *Xen* to a static design places responsibility back into the designer's hands and the toolkit that assists her or him in creating a working specification. On the other hand, our transformation also increases confidence in the final product, as we have removed the amount and the complexity of the code we have to place trust in. While the configuration framework is not formally proven to any degree, it has the undeniable advantage that it is performing its duties offline: its resulting artefacts can therefore be scrutinized and validated by an arbitrary number of external tools long before actual devices are commissioned and built.

From a structural viewpoint, we consider Phidias and Schism a successful demonstration of the feasibility of our Principle of Staticity and an illustrative example of reliable software design for embedded systems. We are going to show in the next chapters how this design impacts overall efficiency and its provability. Furthermore, we imagine further quantification and analysis of the described complexity shift to be fruitful, e. g. searching for and devising the necessary amount and form of additional tooling to replace the forsaken simplicity of runtime autoconfiguration.

5. Evaluation

Before we set out deriving formal properties of our system, we have to demonstrate that the implementation based on our design decisions laid out in Chapters 3 and 4 actually exhibits competitive performance. This endeavour is however set back by two issues.

Firstly, the field of Type I hypervisors for the ARMv8 architecture is relatively new, so there are very few other implementations to compare to. As many implementations with similar design principles are proprietary despite having been published academically like *XtratuM*[64, 24, 89, 21], it is impossible to draw comparisons or even learn which architectures and platforms are actually supported.[1] Others like Proteus[39, 40], a paravirtualizing hypervisor for the PowerPC architecture, were published with solid performance measurements, but have vanished from the research community without a trace. On the other hand, *Xen*[10, 47] and *KVM*[26] have been ported to ARMv8 and have also very recently been evaluated by Dall et al.[25]. While these two designs follow a different philosophy and are thus not directly comparable, both projects are well tested and aggressively optimized, so it should be possible to reach a verdict on competitiveness nonetheless.

Secondly, software benchmarks typically include both micro- and macrobenchmarks, each group serving a different purpose. Microbenchmarks are used to spotlight the key functional elements of a class of software artefacts. This is usually the core of the "hot path", e. g. the context switch operation in an OS kernel or the world and/or VM switch in a hypervisor. Macrobenchmarks on the other hand attempt to capture the big picture by executing mundane realistic jobs and evaluating the overall performance. Only the combination of both allows developers and researches to gain meaningful insights, as a seemingly benign optimization to the hot path could easily cause secondary effects that cripple overall performance, thus leading to a vastly decreased macrobenchmark result despite seeing a constant improvement in a microbenchmark.

Yet our hypervisor is more or less a bare metal construction: macrobenchmarks which intend to take the full spectrum of functionality into account are impossible to carry out on our hypervisor, as we do not provide any features beyond the bare minimum. If we wanted to give virtual machines access to peripheral devices, in particular to those with DMA capabilities, we have two options: If the platform contains an Input/Output MMU (IOMMU), which can be configured to limit the physical memory access of peripheral devices, we could integrate a driver for the IOMMU into Phidias and safely allow a virtual machine to drive DMA-capable peripheral devices. Alternatively, we would have to build emulation drivers into Phidias or into a dedicated trusted VM that mediate access to the memory-mapped registers of such devices, ensuring that DMA operations issued by them match the memory restrictions of the requesting virtual machine. Both options constitute valid solutions,

[1] *XtratuM* is a rather interesting case, as there have been attempts to prove it[83]. Sadly, the proof was carried out on version 3, which is no longer publicly available. The current holder of the intellectual property only offers an old x86-only version 2.6 under the terms of the GPL.

and the resulting setup could then be used to generate meaningful macrobenchmark results.

However, at present our hypervisor supports neither. While we could just configure an "unsafe pass-through" by allowing a VM to drive a peripheral device even though we are unable to confine its DMA operations, there is little knowledge to gain from performing measurements on such a setup, as then the only difference to native execution lies in the delivery of interrupts, which can be better measured separately. We thus limit our macrobenchmark setups to virtual machines booting from an initial ramdisk, which Phidias can provide through its bootup blob package, and the communication channel primitive (shared buffer plus notification software interrupt) for interaction between compartments.

In the following two sections, we present a selection of micro- and macrobenchmarks which are applicable to our feature set and compare the measurements obtained from our hypervisor against those reported by Dall et al. on *Xen* and *KVM*. Finally, we also analyze the worst-case latency of Phidias.

5.1. Microbenchmarks

Of the seven microbenchmarks presented in the *Xen*/*KVM* paper, two measure virtual I/O, which we do not support, and one measures the performance of a hardware feature (acknowledgement of an interrupt at the virtual interrupt controller). We have replicated the remaining four test cases and combined their values with our own measurements—labelled "Phidias (plain)"—in Table 5.1.

As thoroughly explained by Dall et al. in their publication, *KVM* as a Type II hypervisor is badly hampered by the virtualization architecture chosen by the ARM designers: while the switch into the hypervisor privilege level itself is cheap, *KVM* has to swap out the full privileged register state in order to transition to the host kernel to handle a fault or intercept. As this state swap is fully software-controlled (and not performed as a hardware-accelerated atomic state swap like it is on x86), *KVM* falls behind in these microbenchmarks by an order of magnitude.

Xen on the other hand exhibits performance characteristics which are close to our numbers. For the NULL ("no operation") hypercall, *Xen* is even faster than our original design. We were able to trace this to a generalization in our upcall logic: Upon returning to a guest, our implementation always scans the VCPU's emulated interrupt controller for pending interrupts which should be injected before executing the return. We decided to perform this check for all upcalls, regardless of whether it was actually possible for a new interrupt to have become pending since the last upcall—and noting that our design is currently non-preemptible, we could rule out a significant share of entry reasons, among them the NULL hypercall.

We have therefore created a variant of our design which omits the interrupt controller scan during the upcall if it is expendable. Doing so results in a considerable reduction by approx. 330 cycles, as Table 5.1 shows in the extra column "Phidias (optimized virtual GIC)". We attribute the remaining difference to the higher overall complexity of the *Xen* hypervisor, which was already indicated by our implementation size comparison between Phidias and *Xen* in Section 3.4. *Xen* owes this huge size to its vast support library for many different peripheral devices, emulation types, and platforms, all of which results in additional bookkeeping structures and more conditional branches at runtime.

The difference in CPU cycles gets even higher the more complex the microbench-

Benchmark	KVM	Xen	PHIDIAS	
			plain	opt. VGIC
NULL Hypercall	6,500	376	475	140
IRQ Controller Trap	7,370	1,356	661	(293)
Virtual IPI	11,557	5,978	1,913	
VM Switch	10,387	8,799	2,704	

Table 5.1.: Microbenchmark results (CPU cycles)

mark test cases become. For an emulated memory access to the virtual IRQ controller, *Xen* spends twice as many cycles as PHIDIAS. We cannot reconstruct from Dall et al. which emulated register was accessed, but we note that *Xen* contains a much more elaborate IRQ controller model, including support for routing of interrupt lines to individual or subsets of VCPUs, which we do not yet allow. Additionally, the optimization we introduced previously is not fully applicable here, as many register accesses to the emulated interrupt controller do modify the VCPU's set of pending interrupts, which makes the rescan mandatory we have been striving to avoid.

The remaining two benchmarks continue the trend. For the "virtual IPI" test we created a miniature multicore-capable guest OS kernel, booted it on two hypervisor instances and measured the time for a virtual IPI round-trip, including the code executed in the VMs. This form of measurement allowed us to use the local processor-internal performance counter facility of one of the participating CPUs instead of having to resort to coarse global timers. Our experiment yielded an average round-trip time of 3,826 processor cycles and thus a one-way delivery time of 1,913 cycles. For the VM switch, we measured the number of processor cycles spent inside the hypervisor for a reschedule operation, including entry and upcall paths and the full VCPU state save and restore.

For now we cannot explain satisfactorily why our experimental results are so much lower than those measured on *Xen*. Besides the differences in implementation complexity and chosen level of abstraction we have identified, there may be other factors such as the possible discrepancy in how exactly our measurements and those by Dall et al. have been conducted. In a future, more thorough performance analysis these and other benchmarks should be repeated under a uniform measurement methodology on *Xen*, PHIDIAS and other Type I hypervisors for ARMv8 that exist by then. We are however satisfied that the overall performance of our design is comparable to existing hypervisors.

5.2. Macrobenchmarks

Most of the macrobenchmarks present in the *Xen*/*KVM* performance analysis are infeasible to run in the tight limitations of our hypervisor's functional range. The two that were possible to set up and measure are `hackbench`[97] and `netperf`[50].

We ran `hackbench` with the same parameters (100 process groups, 500 loops) as had been used by the *Xen* performance analysis, and compared our measurements on PHIDIAS with those obtained on a native Linux installation on the same platform. As this benchmark rapidly transmits tiny messages between a large number of processes, it performs a stress test of the operating system's scheduler and its message passing facilities. With the chosen parameters, the working set additionally exceeds the size of the TLB, so we expected to witness a slowdown due to the additional second

stage of address translation. However, we found that our configuration framework had aligned the one gigabyte of memory we had assigned to our Linux VM so nicely that it could be represented by a single TLB entry[2]; thus the overhead of Phidias is with 0.4 % practically negligible, whereas *Xen* exhibits a performance overhead of roughly 6 % and *KVM* of 12 %.

Due to the lack of an integrated ethernet peripheral on our HiKey development platform, it was impossible to replicate the `netperf` setup with an external agent transmitting data into (or out of) the virtualized platform, passing the data through a driver VM to the final destination VM. In order to still get a rough idea of the bulk performance of our hypervisor, we instead ran `netperf` between two VMs on top of Phidias and measured the maximum attainable bandwidth. As virtual network device we used two unidirectional ringbuffers of 1 MB each with corresponding signalling capabilities (see Subsection 3.2.7 for the introduction of those primitives) and a simple VM kernel driver which triggers the assigned interrupt capability for every packet sent.

Even with this unoptimized implementation, we achieve a throughput of 20.5 MB/s if we run both communicating VMs on the same hypervisor instance. In this case, the inter-VM interrupt can be delivered locally, but the two communication partners have to be time-multiplexed on one CPU. If we move one partner to a different instance, throughput rises to 32.5 MB/s, as the independent execution of the VMs easily outweighs the additional latency caused by the physical IPI that is necessary to perform the inter-instance VM signalling.

As our goal was only to demonstrate that our hypervisor exhibits acceptable performance even in scenarios with bulk data transfer, we are satisfied with these results. Furthermore we believe that the available bandwidth is able to match the inter-VM communication demands on the embedded platforms we target. If additional optimization was intended, work should be spent in improving the communication protocol above the hypervisor, e. g. by reducing the number of capability invocations through the introduction of appropriate buffer threshold values. As our existing ringbuffer-based solution is conceptually very similar to the established VirtIO[73] specification for virtual devices, adapting our VM drivers to conform to this model would likely result in better performance as well as in improved comparability and more community resources to draw from.

One theoretical improvement that could be implemented inside the hypervisor is "page flipping", i. e. the exchange of a physical memory pages between VMs, trading a page full of data against an unused one, as supported by *Xen*[65]. This mechanism would allow zero-copy data transfer between VMs, which would presumably resolve the main bottleneck of the current setup. However, we consider this approach detrimental for several reasons. Most prominently, migrating pages between VMs runs counter to our Principle of Staticity and would imply removing or deactivating large parts of our configuration framework. In addition, the flexibility of transferring individual pages between VMs would require maintaining fine-grained second-stage page tables instead of our efficient statically generated page table that maps megabytes or even gigabytes with each single entry. Giving up this advantage would greatly inflate the page tables and put heavy pressure on the TLB, causing the cost of virtualized execution to rise significantly, as the `hackbench` measurements have shown.

[2]The ARMv8 paging format we had configured for this setup supports page table entries that map 4 KB, 2 MB or 1 GB.

5.3. Worst-Case Latency

As we have designed our hypervisor to be non-preemptible (see Subsection 3.3.3), there is a third quality we have to measure besides raw performance on the microscopic and macroscopic level. The delivery of a pass-through interrupts into its designated VM or the scheduling switch to the next VM as a reaction to an incoming timer interrupt has a lower bound that depends on the characteristics and the communication between interrupt controller and CPU. Its upper bound however is determined by the longest instruction sequence that is executed with interrupt signalling disabled—and as our hypervisor only enables interrupts as part of an upcall or when entering the idle loop, the longest sequence is in our case the longest complete path throughout our hypervisor from any entry point.

To this end we monitored a full bootup of a Linux VM running with second-stage page tables and measured the retired CPU instructions and elapsed CPU cycles for each event, excluding the last part of the VCPU upcall, which amounts to 28 instructions (or roughly 80 cycles). The resulting histograms are shown in Figure 5.1. We conclude that in terms of retired instructions our paths can be divided in two groups. Upon closer inspection we could identify the second group as those paths that have to run through the interrupt controller rescan we have already found to be a source of overhead during our microbenchmark analysis in Section 5.1. All others complete in less than 200 instructions, regardless of their type (hypercall, emulated fault, interrupt).

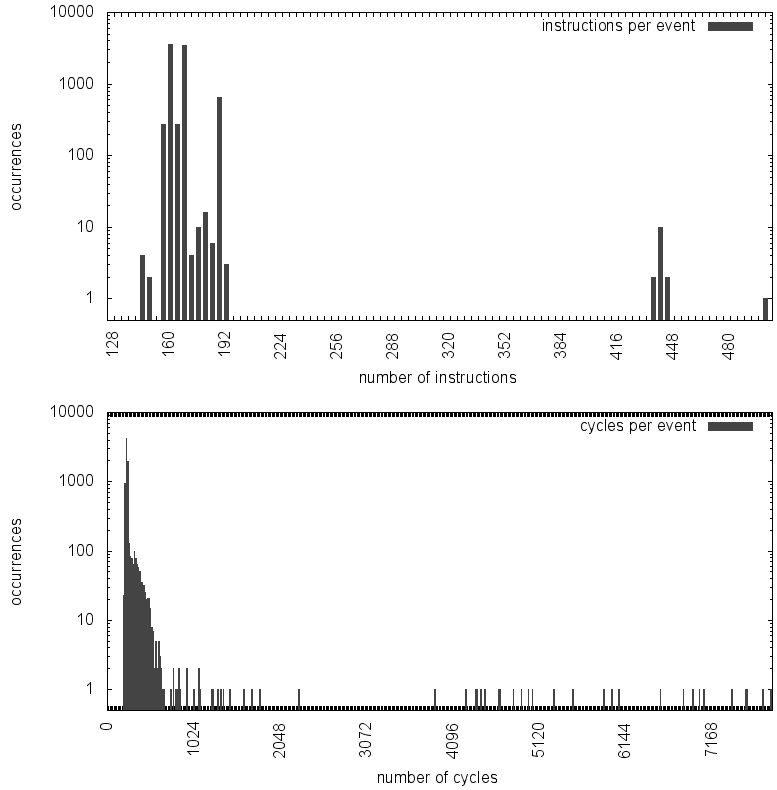

Figure 5.1.: Histogram of elapsed instructions (top, rounded to multiples of 4) and CPU cycles (bottom, rounded to multiples of 16) for all events handled during bootup of a Linux VM.

The number of elapsed cycles paints a different picture though. Despite its pronounced density maximum at 210 cycles, the distribution also shows a very long tail, the longest path taking more than 8,000 cycles. We determined that these outstandingly long paths always occur after a VM has been executing for a considerable amount of time without interruption. In cases where several hypervisor entries happen in quick succession after a long stretch of VM execution, we see each event completing faster than the previous one, until we are finally back at the density maximum. Thus we conclude that these long path durations are the result of our hypervisor having been evicted from all cache levels, such that several memory accesses on our handler path have to stall and wait for data from main memory. This problem is easy to solve, e. g. by locking crucial hypervisor components into the cache or by placing the hypervisor in much faster SRAM and disabling caching for it altogether.

If we perform a similar analysis on our VTLB, we observe a different order of magnitude in path lengths. The raw duration numbers for VTLB pager operations, i. e. without the entry code leading up to it, guest page table walk, and the upcall following it, are listed in Table 5.2. For each directory that has to be allocated and zeroed on the way to adding a new leaf entry to the shadow page table, we observe roughly 1,600 additional instructions[3], with the actual processor cycles varying wildly due to the unpredictability of the number of accessed memory locations that are present in the cache hierarchy.

We can draw several insights from these measurements. Firstly, the paths through the core of our hypervisor appear sufficiently short, and the negative effects of a cold cache can be mitigated by relocating the hypervisor to SRAM. Secondly, the VTLB operations are by far the most time-consuming and would therefore benefit from voluntary preemption points. We discuss in Subsection 6.5.2 whether adding them would be compatible with our proof strategy. Finally, it seems worthwhile to conduct experiments to offload the lengthy zeroing operation into a dedicated VM, which could be safely interrupted without impacting the provability of the hypervisor itself. This benefit comes at the cost of an additional VM switch though, so the trade-off has to be analyzed carefully.

Operation	**Instructions**	**Cycles**
AddLeaf	282	552–1,128
Alloc(1), AddLeaf	1,894	2,065–2,879
Alloc(2), AddLeaf	3,552	3,261–4,072
Alloc(3), AddLeaf	5,061	5,107–7,654

Table 5.2.: Instructions and CPU cycles spent per VTLB map operation, tabulated for the number of required page directory allocations.

[3]This checks out nicely, as our `memset()` loop needs $3 \cdot 4096/8$ instructions to clear a 4 kB memory page.

6. Provability

There have been numerous attempts, past and present, to obtain formal proofs for low-level software components. The 2009 survey by Klein[57] provides a thorough classification of the verification projects that were undertaken in the preceding forty years, and analyzes their differences in scope and methodology. Of the efforts discussed in the survey, the two we consider most notable due to their scope and amount of completion are the Verisoft project carried out at Saarland University[5] and the development and subsequent verification of the *seL4* microkernel at NICTA[58].

The former project set out with the overwhelming goal of verifying a complete compute platform, from the instruction set itself all the way up to and including library and application code in userspace. Its scope was matched by its financial volume and the amount of invested work; in his survey, Klein reports the initial funding with 14.8 million euros and the total amount of work with 30 person years. However, the results achieved in this project were groundbreaking, as the team could demonstrate that proofs of this scale have come within reach and that even layers which are commonly disregarded as trustworthy such as the instruction set architecture can be covered in practice.

Sadly, Verisoft chose the VAMP microcontroller[17] as the basis of their platform, which is of little practical relevance. They developed their own source code language C0, a dialect of C which disallows arbitrary pointer arithmetic and unsafe casts. This decision simplified the reasoning of the proof, but limited the portability in both directions: common software written in C cannot be readily integrated into the proof system, and the proof methodology does not extend to differently implemented systems without additional obligations. They also subordinated all other aspects of their implementation to ease of verification, including performance, which further hampered the adoption of their VAMP-based stack. Still, the resulting abstraction proof formally links each layer to the next one through a series of refinements, and it remains a pioneering work for the field.

The group that worked on the *seL4* microkernel and its verification on the other hand made no compromises with respect to performance and targeted ARM11, the then-prevalent generation of the ARM architecture and a predecessor of ARMv7 and ARMv8, which we are targeting now. Their initial modest goal was to verify the microkernel itself, neither the processor below nor the applications on top of it, and to only show the equivalence between their implementation at the source code level and the formal specification. The source language used was again a subset of C, which denied taking the address of stack variables and the use of function pointers and unions altogether, but allowed casts and pointer arithmetic. Yet even this project, despite its smaller scale, required roughly 16 person years to complete.

Both projects base the reasoning over their respective C dialects on the verification framework for sequential imperative languages developed by Schirmer[84], which allows to lift source code statements into the interactive theorem prover Isabelle/HOL. The *seL4* proof was later extended from C to assembly level by Sewell et al.[86] using a flow graph approach, but encountered several obstacles in doing so, e. g. relating to padded structures and higher compiler optimization levels. Both efforts show that

present day processing power and prover software have made it feasible to derive formal properties of system software. However, the amount of development work required to reach these results was daunting.

We argue that our specific design decisions provide ideal circumstances to approach the generation of a proof from a completely different angle. Both *seL4* and the Verisoft kernel analyzed by Saarland University are dynamic kernels, i. e. they allow creation and destruction of objects and resources at runtime. The preferred (and arguably, the only useful) way to formally approach these systems is through a refinement proof. In the top-level specification (TLS), dynamic objects are grouped in abstract collections, and operations on the system state are expressed in terms of these collections, such as "pick any VCPU from the collection of runnable VCPUs and switch to it" for the scheduling operation. Through a series of refinement steps, these abstract types are more and more fleshed out, e. g. as arrays or linked lists, and the accompanying proof for such steps has to demonstrate that concrete operations like the insertion of an item into a linked list or the search for an item in an array are conforming specializations of the given abstract operation. Directly reasoning on memory contents would be futile though, as any memory word could possibly contain a variety of objects based on the current state of the system and the address choices made by the allocator.

On the other hand, every memory word in our design has one single statically assigned purpose, so the possible values of each word (as long as the specific location is not directly controllable by VM execution) are directly determinable from the compiled hypervisor binary and the accompanying XML specification. In this regard, our prototype has some similarity to the *PROSPER* kernel developed by Dam et al.[27, 28]. However, their work differs from our approach in several key aspects. The most obvious difference is the verification of *PROSPER* starting out with the creation of a formal specification of the kernel in HOL4, which was then combined with the model of the ARMv7 architecture developed at the University of Cambridge[36] to create a direct refinement relation between the TLS and the compiled executable. In order to prove this relation, they went on by generating weakest-precondition contracts for individual hypervisor code paths, which were then shown to be satisfied by a mixture of semi-automatic conversion steps, including a translation from ARMv7 assembly into an architecture-independent intermediate language, and solving satisfiability problems expressed in a first-order background theory (commonly known as satisfiability modulo theories problems or SMT problems). The required preparatory steps, including analyzing the compiled artefacts with the GNU debugger in order to learn the size of objects stored in memory, amounted to a substantial part of their work.

Further distinguishing elements between *PROSPER* and our system are the former's reliance on the relatively outdated use case of paravirtualization on ARMv7, whereas we are directly targeting ARMv8 and in particular also settings that leverage hardware virtualization capabilities. *PROSPER* is also at present only designed as a single-core system, whereas we are able to target multi-core systems as well with our multikernel design (cf. Section 3.3.1). We note, however, that efforts to extend *PROSPER* to different architectures and multi-core systems are apparently underway[16].

Other proof avenues explored by other research groups include translating an intermediate output stage of the compiler into domains where verification is more feasible, as e. g. has been demonstrated in the course of the VATES project[13, 59]. The

BOSS picokernel[66] at the heart of the project was compiled into LLVM Intermediate Representation (IR) and then transformed into a Timed Communicating Sequential Processes (Timed CSP) model. The resulting model could then conveniently be analyzed with existing toolchains for CSP problems.

As our system lacks the concurrency and "communicating processes" aspects of VATES/BOSS, we deemed CSP a bad fit. Instead we opted to pursue an approach that fully commits to symbolic execution and SMT problem solving, even more than taken by the group at KTH with *PROSPER*. Before we motivate our decision and explain our methodology in more detail, we briefly touch on symbolic execution in general.

Symbolic execution has been introduced in the mid-1970s by King[55] as a powerful approach to facilitate and speed up debugging and program testing[1]. It combines the strengths of runtime analysis techniques like fuzzing with those of static analysis: similar to fuzzing, symbolic execution is able to find anomalous program behaviour and produce the corresponding input vector that elicited it, but like static analysis the program under test is not actually executed. Symbolic execution achieves this by evaluating each line of code symbolically, i. e. by inserting variables for unknown values instead of applying each possible set of concrete input values as fuzzing would. By keeping track of branch conditions and restricting the value ranges of the affected variables appropriately, symbolic execution is able to easily produce input data for every possible branch sequence—or even determine that certain branches can never be taken, if the set of conditions leading to them is found to be unsatisfiable.

The major problem of symbolic execution is the explosion of the state space: for every branch encountered in the executable under test, the engine is effectively forked into two independent branches, which means that the search space grows exponentially with the number of branch instructions. Indirect branches that use a register as target (as dereferencing a function pointer would) pose an even harder problem, unless the set of possible values for that register can be shown to be restricted to a handful of locations. Advanced techniques like Under-Constrained Symbolic Execution[79], which breaks down analysis of a program into its individual functions, or State Joining[42], which merges forked exploration branches back under certain circumstances[2], can reduce the growth, but do not offer a solution to the underlying problem.

We will not propose a general solution to the state space problem either; instead we argue that our specific design is immune to it. In the following section we provide our reasoning for this claim and show how to leverage the special properties of our design to efficiently derive formal statements.

6.1. Methodology

The advantage of our design lies in the fact that all entities which are handled at runtime already exist in enumerable form at build time. It therefore makes sense not to choose the hypervisor implementation alone as our proof target, but instead target the desired combination of compiled hypervisor and XML configuration, i. e. a complete bootable image including the addendum and the generated page tables.

[1] A recent and practical summary of symbolic execution can be found in [85]. It also considers the problem of taint tracking, which we introduce at the end of Section 6.2.

[2] This strategy has been later adopted and combined with other optimizations to build a symbolic execution engine with enhanced bug detection rate and speed by Avgerinos et al.[9].

This choice might seem detrimental at first. A proof of the hypervisor alone would certainly be more rewarding, as the formal properties shown would then hold for every imaginable configuration that is applied to it. However, attempting to do so would take away the very strength of our design: the compile-time knowledge of all data structures. We remedy this disadvantage by creating a completely non-interactive proof engine, so that the validation of arbitrary combinations of hypervisor binary and addendum is possible without any manual work. In addition, we also provide performance results to demonstrate that execution of the proof engine not only completes without human intervention, but also in a reasonable time frame.

The circumstances of our proof are similar to those Dam et al. describe in their publications on *PROSPER*. Both setups include a number of compartments that might be allowed to communicate through a designated channel, but which should otherwise be perfectly isolated from each other. Contrary to *PROSPER*, where passing messages between the two compartments involves the kernel, our VMs can exchange data without invoking hypervisor functionality. The correctness of the page tables that confine each VM to its own memory and its configured communication channels can be verified separately by comparing them with the XML specification. What is left to prove is therefore a modified form of non-interference[80]: For all code paths traversing our hypervisor, we have to show that VMs cannot influence each other by reading from or writing into each other's VCPU state save area, and that they cannot indirectly influence the hypervisor's behaviour towards other VMs and their VCPUs by modifying its internal state.

For a complete non-interference proof, we would also have to demonstrate that no side channels exist which would allow a VM to infer knowledge about the execution of other VMs. This requires a model that includes all components of the platform which might be a conduit for a side channel, such as the TLB, the data and instruction caches, and observable time. The model we present in this chapter does not account for these components yet, but we point out how to extend our engine with a TLB and cache model in Chapter 9.

The core principle of our proof engine in pursuit of this goal is to exhaustively explore the state space spanned by the hypervisor and the chosen addendum, taking advantage of the componentization into separate execution units per physical CPU. In order to cover the whole state space with our simulation, we have to identify all classes of execution paths that traverse our hypervisor. As Phidias has been constructed according to the interrupt-based execution model (see Section 3.3.2) with a single stack and no preemptibility, we only have two of these classes:

C1 the initialization path, which configures all processor registers, initializes and configures hardware devices, prepares the VMs for execution, and finally makes the first scheduling decision and performs the first upcall

C2 the runtime path, extending from the event that caused the entry into the hypervisor, usually entering at a cause-specific entry point, to the upcall to the same or a different VM or—if no VM is ready to execute, possibly as a result of the very event that was just handled—to the idle loop

The first case is difficult to handle, as parts of the initialization path run with a different view of the platform (e. g. without paging enabled) or touch components which are not part of the model (e. g. external devices). This is a problem we share

with other verification projects and which is independent of the actual proof strategy. In order to solve this problem we would have to include additional model components for all external devices and also add complexity to our memory access model. We argue that the cost far outweighs the benefits for our very early initialization code, as its only purpose is to enable paging and configure proper memory access attributes. While the remainder of this path would be feasible, we leave its exploration as future work and concentrate on the overall tractability of our concept.

Upon closer inspection, the path class C2 can be split up further into subclasses, which are distinguishable by their starting location, the presence of an executing VCPU, and the scheduling decision made by the hypervisor. The resulting classes are:

C2a The hypervisor leaves idle mode due to a hardware interrupt and reenters idle mode afterwards.

C2b The hypervisor leaves idle mode due to a hardware interrupt, which causes it to make a scheduling decision.

C2c The hypervisor gains control while a VCPU is executing; it handles the event, then resumes the interrupted VCPU by performing an upcall.

C2d The hypervisor gains control while a VCPU is executing; it handles the event, deschedules the interrupted VCPU, and saves its state, and makes a new scheduling decision.

C2e After having made a scheduling decision, the state of the selected VCPU is loaded and an upcall is performed.

For these classes we can now postulate the class-specific invariants we have to prove:

IC1 For path class **C2c**, the state of each unprivileged and privileged VCPU register at upcall is identical to the state it was in when the hypervisor was entered.

IC2 For path classes **C2b** and **C2d**, the registers of the VCPU being scheduled out are correctly saved into the corresponding state save area.

IC3 Conversely for path class **C2e**, the registers of the VCPU that is about to be resumed are loaded correctly from its state save area.

In addition, the following invariants have to hold for all paths, regardless of their specific entry and exit points. The first five items maintain hypervisor integrity, and the remaining two ensure that VMs cannot influence each other:

IG1 The linked list of runnable VCPUs (the runqueue) is cycle-free, and each element included in the list has its scheduling state set to “ready”.

IG2 Conversely, a VCPU can only be “ready” if it is either on the runqueue (i. e. waiting to be scheduled) or currently running.

IG3 The global variable that points to the currently executing VCPU (if any) identifies the one VCPU that is ready, but is not on the runqueue.

IG4 The linked list of queued timer events is cycle-free, the deadlines of elements in the list is monotonically increasing (earliest deadline at the head of the list), and the "armed" member of a timer event structure is set to 1 if and only if it is an element in the list.

IG5 The hypervisor control registers which govern the execution environment of the hypervisor itself (paging, cacheability, exception handling etc.) are never touched outside the initialization path.

IG6 The state of dormant VCPUs (which are neither currently executing nor being scheduled in or out) is neither read nor written.

IG7 The hypervisor control registers which control the execution of VCPUs (nested paging, cacheability attribute overlay for guest memory accesses, etc.) always match the read-only data set in the addendum configured for the currently scheduled VCPU.

Due to the small finite number of all these structures and their locality to one hypervisor instance, we are able to prove that these invariants are preserved by simply presenting all possible state configurations to the symbolic execution engine and testing the required assertions against each resulting symbolic output state.

6.2. Symbolic Execution Engine Design

Given an input image consisting of a hypervisor binary and a specific system configuration, our symbolic execution engine is able to create an execution trace for each supplied, partially symbolic input state and to translate the native assembler instructions into their equivalent algebraic form. We use the SMT theory of fixed-width bit vectors as domain for our equations, and we assign generation indices to the CPU registers and memory words we track in order to identify different-valued instances of the same location. Our approach is thus similar to the machine code decompilation by Myreen et al.[70, 71], but we directly translate to first-order equations suitable for SMT solving instead of Hoare triples.

Let us consider the simple ARMv8 instruction `add x0, x0, #6` as an example. We assume that the current generation counter for the architectural register $x0$ is i. Processing the instruction then leads to the creation of the new generation $i+1$ of the 64-bit bit vector variable $x0$ and to the addition of the new interdependency equation $x0_{i+1} = (x0_i + 6)\ mod\ 2^{64}$ to our set of assertions Φ. As all 31 general purpose registers can be accessed at their full 64-bit width ($x0$ – $x30$) as well as their low 32-bit half ($w0$ – $w30$), special care has to be taken to update both when either of them is written to; we would therefore also create a new generation of the 32-bit bit vector variable $w0$ and set its value to the truncation of the new value of $x0$: $w0_{i+1} = x0_{i+1}[0:31]$.[3]

Conditional branch instructions on the other hand lead to inequalities being added to our set of assertions, as the following example shows. The instruction sequence `cmp x1, x0; b.gt <label>` first causes a new generation of the hidden prover variable *CMP* to be set to the difference between the current generations of $x1$ and

[3]The example given in [70] for the even simpler-looking `inc eax` instruction is much more complex, as that instruction updates the condition flags as a side effect, which our ARMv8 `add` instruction does not. Myreen et al. do not elaborate on the half-word and byte register viewports `al`, `ah` and `ax`, though.

$x0$. The following conditional then has to evaluate whether the value of *CMP* is greater than 0 in order to decide where to continue execution. If both outcomes are permissible under the set of assertions accumulated so far, the symbolic execution engine has to split its state and continue simulating both arcs separately, adding the positive form of the condition for one arc and the negative for the other. If only one outcome is permissible, the engine can simply add the only possible branch constraint and continue down the corresponding arc.

After the simulation of a certain path with a chosen input state has reached that path's exit instruction, e. g. the `eret` instruction that transfers control back to a VCPU, or the `wfi` instruction that causes the processor to enter idle mode and wait for interrupts, we have to verify that the desired path-specific and global invariants (**IC1**–**IC3** and **IG1**–**IG7**) are preserved. We accomplish this by passing the accumulated set of assertions Φ and the negation of the property we want to test $\neg\psi$ to an SMT solver. Φ is by construction satisfiable, as it captures the effect of the series of assembly instructions we have traversed. If the SMT solver therefore reports $\Phi \cup \{\neg\psi\}$ as unsatisfiable, it follows directly that $\Phi \Rightarrow \psi$, i. e. execution of the path captured by Φ with the given input state guarantees that ψ holds.

Of the invariants specified above, **IC1**–**IC3** are trivially expressible as SMT equations, because they directly relate instances of simulated processor registers and simulated memory words with each other. The invariants **IG1**–**IG4** can be checked by directly inspecting the resulting output state: as we start each path simulation with concrete configurations for the runqueue and the list of timer events, so is the resulting state of these queues going to be concrete, i. e. non-symbolic. Thus we are able to walk the linked lists and the data structures referenced by it and verify that the desired properties hold. Invariants **IG5** and **IG6** are verifiable by asserting that no generation was ever allocated for those registers and memory words. This is particularly important for the hypervisor registers, as mere equality between entry and exit state is not sufficient: allowing a change in address translation or exception handling at a single point in time would immediately void all other assertions. Invariant **IG7** can finally be verified by inspecting the output state in the same way as we do for **IG1**–**IG4**.

Before we turn to describing our actual implementation, we briefly discuss two tweaks to the established concept of a symbolic execution engine which we have applied. The first is a very straightforward optimization that reduces the complexity of the SMT model by building on the fact that a high percentage of memory loads performed by the hypervisor are accessing the addendum, i. e. mostly read-only data. Therefore for each new equation φ that describes variable v_i as an arithmetic or bit-logical computation of other variables, we first try to determine whether the resulting value is symbolic or constant. Besides loads from ROM this may also be due to immediate constants in assembler instructions or computations of variables which themselves have already been determined to be constant. If the value is indeed found to be constant, we do not add φ to Φ, but keep the relation between v_i and its value inside our proof engine. Wherever v_i is then used in subsequent calculations, we substitute its constant value, which may in turn cause those subsequent variables to be omitted from Φ. This "constant propagation" has previously been used in related settings, e. g. by Koelbl and Pixley in their work on deriving data flow graphs from C++ programs through symbolic execution[60]. Their publication does not provide numbers on the speed gain they achieved, so we cannot compare our application of the concept against theirs, but our evaluation (see Section 6.4) shows that fixed-value

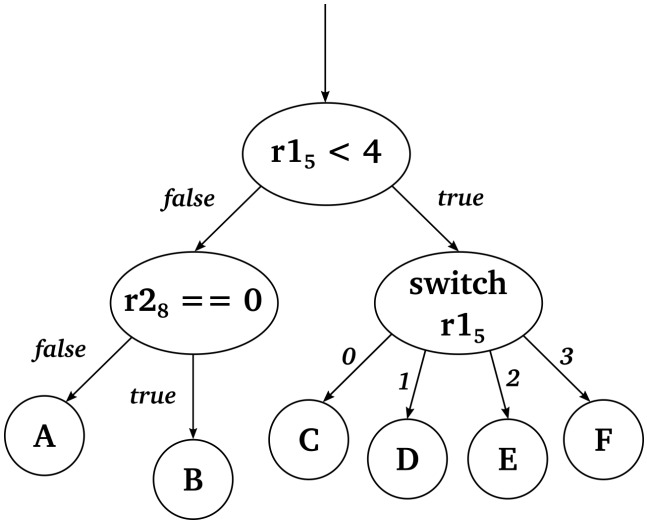

Figure 6.1.: Sample (not completely binary) decision diagram.

variables make up a significant percentage.

The second tweak challenges the traditional approach of starting out with a single generic symbolic state and forking the simulation engine at every conditional branch. Instead of starting with a single, maximally symbolic state and forking on demand, we have chosen the opposite, but equivalent approach of declaring the full input state space from the outset, such that for every simulated configuration of the input space all branches are determined and no fork is ever necessary. Both strategies build up decision diagrams as depicted in Figure 6.1 as they explore the state space, each leaf representing one possible control flow through the hypervisor. Our non-forking version has one major drawback: it cannot easily decide which branch conditions depend on the "pre-forked" input state and which have only one outcome due to ROM values placed in the addendum. While the latter could be omitted from the decision diagram, we have to prove full coverage of all paths through the former ones. We solve this by applying the well-known concept of taint[85]: variables representing pre-forked simulated input state are tainted, and calculations and memory references based on tainted data causes the resulting variable to become tainted itself.

Finally, both strategies share the problem of dealing with indirect branches. As we have decided not to use function pointers in our implementation (see Section 3.1), these can only be the result of switch/case statements in the source code, which are converted by the compiler into a constant table of code pointers and a table load indexed by the switch variable. The resulting part of the decision diagram is thus no longer binary, but may fork out in many more branches (cf. the bottom right part of Figure 6.1). We deal with these by tracing back to the last constraint that was added to Φ which bounds the index variable (embodying the "default" case that is executed when the variable falls outside the range of values that is accepted by the "case" statements) and thus indicates how many leaves we have to traverse to reach full coverage.

6.3. Implementation

We built our prototypical symbolic execution engine with support for the ARMv8 architecture, which includes virtualization extensions with decode assists for most faulting instructions and nested paging. The architecture has been chosen to fit in with the rest of our work. Unfortunately, there was no sufficient semantic model of the ARMv8 instruction set available yet—the SAIL ARMv8 specification developed by the Cambridge group[34], who already created a full model of the previous architecture release ARMv7[36], does not contain support for system-level instructions which make up a core part of the low-level operations of our hypervisor. For our project, we thus had to implement our own translation unit which transforms the necessary subset of ARMv8 instructions into SMT bit vector expressions.

We fixed a hypervisor binary and a configuration including two guests linked by reciprocal software interrupt capabilities and a shared buffer. The hypervisor binary was compiled without optimization (`gcc -O0`) in order to limit the diversity of opcodes emitted by the compiler. We stress that this choice was not motivated by a structural limitation—as we are about to perform unrestricted symbolic execution, we do not depend on any resemblance between higher language structures and generated assembler instructions. The heavy inlining that higher compiler optimization settings activate thus poses no problem. By either investing more manual work into our semantic translation unit or by porting our execution engine to use Cambridge's SAIL ARMv8 model, we could support arbitrary optimization levels.

At present, our symbolic execution engine fits in roughly 3,000 lines of Python code, 750 of which contain the ARMv8-specific instruction translation unit. The SMT back-end of our engine is compatible with the established SMT-LIB standard[12], so we could theoretically interface with any available compliant SMT solver that supports the bit vector theory. Our current prototype is configured to use Microsoft's Z3[29].

Upon initialization of each individual trace, we create a set of initial path constraints which capture one of the possible input states[4] and which have to be just precise enough to determine all branch outcomes encountered during the path[5]. We also flag all variables that make up the simulated input state as tainted in order to distinguish branches based on input state from those which depend on read-only data and are thus negligible in the decision diagram.

We make one further optimization to our taint propagation system in order to minimize our decision diagram: if a tainted variable becomes constant after a conditional branch that depended on it (i. e. it passed an equality test), we remove the taint, as further checks against this variable can only have one possible outcome. As cases may not always be as obvious as passing an equality test, we implement the taint removal test for variable v_i with accumulated constraint set Φ by executing the following steps:

1. present Φ to the SMT solver and trigger the creation of a model M_0, i. e. a map of variables to bit vector values such that Φ is satisfied

[4] This does not imply that we fix certain registers to a single value. Often enough it is sufficient to constrain registers to a sub-range of the full 64-bit spectrum, or only fix certain bit positions.

[5] Perhaps surprisingly this does not require knowledge of the hypervisor implementation, only knowledge of the possible architectural events: e. g. memory access faults require a fault address in a specific register. The "interesting" address ranges for each virtual machine can be retrieved from the accompanying XML specification.

Trace	V_{fix}	V_{float}	BC_{all}	BC_{taint}	Class	Time
NULL Hypercall	279	226	25	7	C2c	356ms
IRQ Cap. Invocation	432	220	32	13	C2d	512ms
Fault on Illegal GPA	810	279	46	10	C2d	1,457ms
Spurious IRQ	566	231	28	6	C2a	331ms
Scheduling IRQ	1,288	288	53	8	C2b	1,423ms
All Data Faults	104,268	47,634	7,488	2,148	C2c,C2d	135s
All Sync. Traps	177,936	90,966	12,594	3,328	C2c,C2d	232s
All IRQ Paths	21,922	9,463	1,146	296	C2a,C2b	20s
Core Proof	199,858	100,429	13,740	3,624		252s

Table 6.1.: Traversal statistics for some exemplary traces and selected subtotals.

2. build $\Phi' := \Phi \cup \{v_i \neq M_0(v_i)\}$
3. present Φ' to the SMT solver and test for satisfiability
4. if Φ' is unsatisfiable, v_i is constant and taint can be removed

As a demonstration of how our proof engine operates, we have included the output of one trace simulation in Appendix A.3.

6.4. Evaluation

In order to demonstrate the effectiveness of our approach and to get an idea of the orders of magnitude involved, we have augmented our engine with a substantial amount of instrumentation. The numbers for a few representative paths are shown in Table 6.1. For each trace during our input state space traversal we tracked the number of generated variable instances (registers and memory words), maintaining separate counts for those which could be determined as having a constant value (labelled V_{fix}) and the symbolic ones (labelled V_{float}). We also counted the total number (BC_{all}) and the number of tainted (BC_{taint}) branch constraints in each trace's Φ. Each line also denotes the path class (**C1**–**C2e**) the trace belongs to.

We observe that even the more intricate paths only encounter roughly ten tainted constraints. This confirms our initial claim that the search space of our confined hypervisor environment is indeed small enough to allow us to exhaustively cover it. We also have to attribute this low count partly to the fact that we do not apply taint to the structures involved in the two global linked lists, as our current implementation of the decision diagram coverage verifier is too weak to support the required reasoning. Instead, we draw our confidence from the fact that the input space generator of our proof engine creates all possible permutations of those lists that are accessed by a simulation trace. Thus soundness of our proof system is not impacted—just the number of tainted constraints is lower than expected.

Handling of the scheduler runqueue in particular is easy, as we have split the execution up to the scheduler switch and the upcall after it into different path classes. That it was permissible to do so, i. e. to analyze the upcall path on its own, is a result of our proof engine as well: if there was an unintended dependency of the upcall path on some register or memory location that has been touched during entry, we would have detected this as an undecidable branch. With the split in place, we can

easily assert that the descheduled VCPU either is no longer runnable or has been correctly appended to the runqueue at the end of path **C2d**, and that the loaded one is correctly taken off the runqueue; therefore our global scheduling queue invariant **IG2** (cf. Section 6.1) is maintained.

Running our proof engine single-threaded on the core set of operations for our example scenario (cf. Appendix A.2) takes less than five minutes on average hardware[6]. Completing the proof to fulfil our initial goal requires two further steps, which are both neither a threat to our overall strategy nor an insurmountable performance problem. Firstly, the initialization path is not yet part of the simulation, so while we can prove the invariants to be preserved by all operations, we have not yet shown that they are established in the first place. This path only has to be simulated once, as it is supposed to bring the hardware into a defined state where regular system operation can commence. However, it may contain special instructions which have not been added to our ARMv8 translation unit yet, so adding it may require some work. Secondly, the current state space does not simulate memory accesses to every individual location of each possible emulated device. We have omitted the exhaustive simulation of memory-mapped device registers because emulation of some devices could be pushed out into unprivileged "service VMs", with the hypervisor merely passing on the emulation request, such as a UART or a clock source. Doing so would remove these devices from the hypervisor code base and thus also from our proof obligations. Nevertheless exploring these devices as part of the hypervisor proof is certainly feasible, although it might increase the number of required data fault traces by another order of magnitude.

Furthermore, the number of paths increases with the number of emulated devices and associated memory ranges, as each of these areas has to be covered by the input state space. Even in our limited scenario, the simulation of data fault conditions already accounts for the majority of synchronous traps. Thus we envision that a more realistic scenario, which may involve several times the number of emulated memory ranges, might complete in 20-30 minutes, or, if emulation devices are fully explored as well, several hours. However, there is one key advantage of our implementation that we have not exploited so far. Due to our deliberate choice of favouring explicit state space enumeration over forking on demand, we have created a proof engine which is (except for the final decision diagram check that tests for full trace coverage) fully parallelizable. It is therefore safe to assume that even the realistic and fully exploring proof could be completed in the order of minutes on more recent computing hardware and with an improved, parallelized implementation.

Finally, we have to compare the effort we invested in our proof system against the projects we introduced at the beginning of this chapter. The development of Phidias, Schism, and the proof engine took about 18 months in total, which is less than 10 % of *seL4*'s time budget. The formal properties we were able to establish so far are weaker, but still manage to rule out many attack vectors; we have shown that "nothing bad happens, no matter what", so the only remaining possible attacks are those that prevent good things from happening either, i. e. attacks that prevent the system from making progress without violating any of the invariants we have imposed. Depending on the intended use case and threat model, this might well be sufficient, and our design has made attaining these properties amazingly cheap.

[6] Times have been measured on a Thinkpad T520 with an i7-2620M dual-core processor, by now already six years old and therefore at best "average".

6.5. Corner Cases

With our core proof complete, we can now turn our attention to a few aspects that we have left out until now for the sake of clarity. We now discuss each of these, provide reasons for their omission and put them into perspective.

6.5.1. Multicore Interaction

As our hypervisor implements the "shared nothing" model, there are no global variables that would require a synchronization protocol. Interaction between hypervisor instances occurs only in the following cases:

- If the debug facilities are compiled in, all hypervisor instances use one central UART peripheral to print diagnostic messages. In order to prevent messages from multiple instances from interfering with each other, the hypervisor guards access to the UART registers with a spinlock.
- Virtual machines that span several VCPUs across different hypervisor instances may require virtual peripheral devices that are shared among those VCPUs (cf. Subsection 3.2.5). In order to guarantee atomicity of write operations to certain emulated registers, the hypervisor uses the ISA-specific "write-exclusive" instruction, which may require some hypervisor instances to retry their operation if a race occurs.
- Delivery of signals between VCPUs that execute on different hypervisor instances is implemented through a shared message buffer and the architecture-specific form of inter-processor interrupts.

Of these cases, we consider the first one insignificant due to several reasons. Firstly, the spinlock merely exists to improve readability of the debug output, and could easily be removed. Secondly, the whole debug facility is not a required target of the proof, as it is likely to be disabled in production builds of the hypervisor anyway. Thirdly, it would also be possible to push the actual handling of the UART peripheral out into a "debug VM" and have the hypervisor instances push their diagnostic messages into the VM through individual shared buffers, thus removing the need for a lock altogether.

The second case is innocuous due to the fact that the critical section is a single bitwise operation, but the hypervisor code path leading to a modification of such an atomically emulated register takes hundreds of cycles, as we have measured in Section 5.1. It would thus require more than 100 VCPUs with access to the same shared emulated device to open up the possibility of one of them continuously failing to succeed with its write-exclusive instruction. Such configurations are beyond the range of what we target with our hypervisor. The ARMv8 ISA interpretation unit of our proof engine therefore implements the instructions with exclusivity suffix (`stxr`, `ldaxr`) as always successful.

The third case did not occur in the example configuration we have used to test the proof engine, as both VMs were scheduled to run on the same physical CPU. Future work could solve this situation by defining a sliding window in the inter-processor messaging buffer which contains the messages already sent, but not yet processed by the receiving side. By adding additional invariants to our global set we could show that the sending side only adds new messages, thereby extending the zone, but never

overwrites messages in this zone. This makes it possible to simulate the receiving side as having an atomic view of the messaging buffer.

6.5.2. Preemptibility

When we developed our design, we have deliberately decided against any form of preemptibility (see Subsection 3.3.3). With the feasibility of the proof demonstrated, we can now look back and consider where the admittance of preemptibility would have caused additional complexity in our proof system.

In an event kernel, resumption of a preempted kernel activity is impossible if the higher priority activity caused an upcall, as the kernel stack that contained the preempted execution state is invalidated at that point. On the other hand, this "clean slate" policy also ensures that no VCPU state is lost, as VCPU registers are directly stored in the state save area on entry, and no native kernel threads exist in our design. With a few simple steps we could therefore easily introduce a limited form of preemptibility which does not interfere with our proof engine at all.

As a first measure, interrupts could safely be enabled during the execution of the upcall procedure, if the interrupt handler entry code is slightly altered to rewind the stack pointer to the top on its own and to skip saving VCPU state if the entry is determined to be a preemption. These changes allow interrupts to cancel an ongoing upcall operation early instead of causing hypervisor reentry directly after the upcall has been completed. In terms of our proof, this means that traces of path classes **C2c** and **C2e** may terminate early—however, in these cases the class-specific invariant **IC1** (or **IC3**, respectively) is not required to hold either, as no upcall is taking place. Preemption interrupt traces would create the new path class **C2f**, which differs from **C2c/C2d** only insofar as the VCPU state has already been correctly saved by the previous entry. An updated proof would merely check that the global invariants **IG1**–**IG7** hold at every single instruction of the interruptible section of the trace, and define and validate a slightly adapted class-specific invariant for **C2f**.

As a second step, voluntary preemption points could be inserted into the long-running event handling operations, e. g. into the VTLB's "allocate and clear new directory" operation (see Section 5.3). If the operation is indeed interrupted, this causes the faulting VCPU to be resumed later on even though its fault condition has not been remedied, as we did not introduce additional logic to flag pending operations. This is however inconsequential, as the fault is simply rethrown and the VTLB gets another chance at handling the condition. The affected path class **C2c** would require further checking for global invariants during execution inside the VTLB, but the overall proof strategy is still applicable.

6.6. Verifying the VTLB

With the core of our hypervisor conveniently verified with respect to integrity, we now turn our attention to the VTLB, the single dynamic component we have left in our design to accommodate those use cases that still require it, such as virtualization of operating systems in the secure world ("TrustZone") of ARM CPUs.

For the VTLB, obtaining an integrity proof is our foremost goal, as an integrity-violating VTLB has the same catastrophic consequences as a runtime change to the hypervisor MMU controls: it jeopardizes basic assumptions that make up the foundations of our proof system. If on the other hand we are able to show that the

translation tables created by the VTLB conform to the XML specification, we also gain non-interference between VMs for the VTLB.

In the same way as our core proof above differs from those for *seL4* and *PROSPER*, our efforts concerning the VTLB will not produce a fully-fledged formal specification of VTLB operation as established for the *Anaxagoros*[18] and *BabyVMM*[93] hypervisors; our point is again rather to achieve a lesser, but still very satisfying result at a tiny fraction of invested effort.

By its very nature, a shadow page table as created by the VTLB occupies a variable amount of memory. VCPU accesses to hereto unmapped virtual memory addresses cause the creation of a new entry in the shadow page table, but may also require the allocation of a new page directory and its insertion into the tree structure. Additionally, the VTLB might choose to maintain several shadow page tables for a single VCPU, if the architecture or paravirtualization modifications to the OS inside the VM are able to ensure that all these stay synchronized to the guest page tables they mirror[7].

Our hypervisor satisfies these functional demands using a semi-static approach. Instead of allowing shadow page tables to grow indefinitely by making all unallocated physical memory pages available to the VTLB, our configuration system requires the integrator to explicitly reserve memory pools according to our VTLB definition in Subsection 3.2.8. Eviction of directories in case of memory scarcity has to be handled, but we do not pay attention to any specific eviction policy.

Conceptually, the VTLB consists of two separate components: the walker, which inspects guest state and returns the desired guest-physical address, and the pager, which updates the shadow page table and interacts with the allocator. For this proof we only consider the pager. The walker is performing lookups in VM memory which we can make no assumptions about, so we have to consider its results as untrusted input anyway. As far as our integrity proof is concerned, we have to show the following invariants:

IV1 Each entry in a shadow page table maintained by the VTLB is either a) invalid, b) a valid mapping into host-physical memory, or c) a reference pointing to a next-level directory. Certain types of entries may be architecturally forbidden at some levels of a page table. As the MMU would treat such entries as invalid, this limitation only reduces the set of effectively mapped addresses, so we can safely disregard this.

IV2 Valid mappings according to IV1.b always point into memory areas specifically assigned to the VM in the XML configuration.

IV3 Directory references according to IV1.c always point to a memory location in the next level's assigned memory pool.

We could add additional invariants to prove that the pool allocator behaves as desired, i. e. that allocations are only given out exactly once and that they are properly reclaimed when a directory is destroyed. However, accidental reuse of a directory in different places does not have any negative consequences, as the point of use of a directory merely determines the virtual address where the host-physical memory

[7] Maintaining synchronization is only a matter of VTLB correctness though, not of integrity. We therefore do not investigate this problem any further.

Trace	V_{fix}	V_{float}	BC_{all}	BC_{taint}	Class	Time
Rejected Addition	274	107	33	2	C3a	144ms
Mapping Addition (leaf)	1,547	107	66	7	C3a	1,041ms
Mapping Addition (1 DA)	3,300	107	582	7	C3a	9.1s
Mapping Addition (2 DA)	5,052	107	1,098	7	C3a	24.1s
Mapping Addition (3 DA)	6,804	107	1,614	7	C3a	47.1s
Mapping Removal	984	111	31	9	C3b	439ms
Shadow Pagetable Flush	—	—	—	—	C3c	—

Table 6.2.: Traversal statistics for some exemplary traces of the VTLB module, especially those requiring directory allocations (DA).

resource is made available; and as we enforce usage of the correct memory pool, mappings cannot be made larger by reusing a directory at a higher level than it was created for.

We identify the following additional path classes for which the above invariants have to hold:

C3a Adding a new mapping: This involves validating the guest-physical address reported by the VTLB walker, translating it into a host-physical address, and finally creating the appropriate entries in the shadow page table, which may involve allocating intermediate level directories.

C3b Removing a mapping: As a reaction to a trapped architectural single-address TLB invalidation (e. g. x86 `INVLPG` or ARM `tlbimva`), the corresponding entry is removed from the shadow page table.

C3c Destroying (flushing) a full shadow page table: If a VCPU uses more page tables than its VTLB's maximum number of shadow page tables, cache slots have to be recycled and the previous contents deleted.

Our traces now only cover the VTLB module itself, as our invariants can only be influenced by this module—more strictly speaking, all core path classes are assumed to have already been proven not to access the memory pools which are set aside for VTLB instances, so we can concentrate now on this specific component in isolation.

When trying to generate the first trace samples for the VTLB, we immediately noticed that some were completely intractable both in terms of runtime and memory consumption. We identified the culprit to be the clearing of a page table directory before it is inserted into the page table. This operation, represented in the source code by a call to `memset(addr, 0, 0x1000)`, caused a state explosion in our prover, as our standard implementation of `memset()`—combined with the `-O0` compiler flag (for an explanation why we use this flag see Section 6.3)—resulted in a loop with eleven assembler instructions and a four-byte increment. After we had replaced this with a handcrafted loop that still only used our limited subset of the ARMv8 ISA, but had only three instructions and an eight-byte increment, we could successfully complete our sample set. The results for those exemplary traces through the VTLB module are listed in Table 6.2.

The results show that the allocation of a new VTLB directory is still very costly to verify. Even in its optimized form, each call to `memset()` adds roughly 512 path constraints and three times as many variables to the traversal. Due to the lacking

optimization of our symbolic execution engine, the accretion of constraints and variables causes an almost quadratic increase in runtime. Still even the worst case for the four levels deep ARMv8 page table is tractable.

Flushing a shadow page table on the other hand requires a recursive descent into all intermediate directories in order to scan them for links to lower directories, as all of those have to be given back to their respective pool allocators. The current implementation of this descent involves calling a function for each entry in every directory visited, which exceeds the capacity of our engine. This is merely an efficiency problem, though.

Unfortunately, the major roadblock for an integrity proof of the VTLB is that these exemplary traces are not easily extensible to cover the full input state space, as we could before. For the core proof, we did not encounter a single case that required us to handle a symbolic memory accesses, i. e. a load or store operation with a base address that depends on a symbolic variable. Therefore our present state of the proof engine does not support such operations. The VTLB pager however does little else than performing symbolic memory accesses: if we wanted to solve this in the same way we did before, we would have to simulate all possible fault addresses and all possible states of the directory descriptors that are walked in order to create the new page mapping. This state space is completely infeasible.

Completing the VTLB proof would thus require a more advanced execution engine which is capable of reasoning about memory operations with a symbolic address; we consider this a feasible extension, but leave the actual implementation and its evaluation to future research efforts. We note that proofs of shadow paging systems have already been performed, such as those of *Xen* and *ShadowVisor* by Franklin et al.[37], that of *PikeOS* by Baumann et al.[15], and a generic formulation by Alkassar et al.[4], albeit each with different assumptions and only to the source code level. We leave it up to further investigations to combine the ideas presented in those publications with our proof methodology.

7. Lightweight Virtual Machines

As the integrity proof of the core hypervisor relies on the adherence to our Principle of Staticity, the question remains whether it is possible to apply performance optimizations to our core while retaining the proof structure. In this chapter we present and implement a new optimization to the ARM VM switch operation, measure its net effect on a sample hypervisor setup, and discuss its implications for the proof we have presented in Chapter 6.

7.1. Concept

System configurations consisting of a hypervisor and virtual machines depend on the efficiency of the world switch, i. e. the transition between hypervisor and virtual machine, to a similar extent that classical microkernel-based solutions depend on a highly optimized IPC path. The burden of hypervisor entry and exit could be alleviated by exploiting the fact that the virtual machines which are time-multiplexed on any given CPU do not necessarily require the same breadth of processor features. Special-purpose VMs tasked with monitoring duties and the occasional over-the-air update might benefit from access to a cryptographic coprocessor, but could do well without access to the floating point unit. Other VMs which provide access to emulated or multiplexed devices might even not require different privilege levels altogether, if the desired functionality is embedded in the monolithic VM kernel; one obvious example is using the TCP/IP stack of the Linux kernel to implement a gateway VM to restrict the network connectivity of other more complex VMs.

Historically, the hardware virtualization extensions for the x86 architecture have left researchers and developers with little design freedom. The memory formats for VM control block and state-save area[1] are precisely specified, and the actual load/store operations for VM entry and exit are implemented in microcode: developers only issue one specific machine instruction, `VMENTER`, and a Rube Goldberg-esque machinery is set in motion whose description quite famously spans a whole 26-page chapter of the Intel Architecture manual[49]. This accretion of complexity is largely due to the full backwards compatibility of x86 to the early days, retaining outdated features like 16-bit real mode and segmentation. AMD's corresponding `VMRUN` instruction, while different in technicalities such as the control block layout and handling of the physical interrupt flag, is basically an equivalent design[3].

Both Intel and AMD recognized that this entry/exit mechanism might become the central bottleneck in many workloads and tried to ameliorate the problem by adding a cache layer to their VM control block management. If the hypervisor declared the VM

[1] The terminology used by Intel and AMD is different from ours: they call their structures "Virtual Machine Control Structure (VMCS, Intel)" and "Virtual Machine Control Block (VMCB, AMD)", although these contain the state of a single VCPU, which is in turn in our parlance part of a uni- or multi-processor VM. While we are discussing the concepts and optimizations introduced by the architecture designers in this and the following section, we stick to the official terms for easier comparison with the cited manuals.

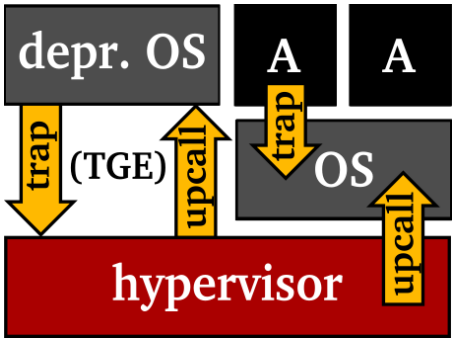

Figure 7.1.: Traps and upcalls with and without TGE bit set among hypervisor, OS (deprivileged if running under TGE) and userspace applications (A).

state structure to be unchanged from its state at VM exit, the CPU would not restore guest state by performing expensive memory accesses, but instead read that state back from internal shadow registers which are inaccessible to regular instructions. Nevertheless all registers have to be swapped atomically, as the distinction between hypervisor and VM is orthogonal to the privilege level concept, i. e. both inhabit ring 0 and thus share the same set of registers.

ARM chose the opposite approach for their virtualization extensions[6, 7]. Instead of relieving developers from micro-managing the world switch by forcing everybody to use the same elephantine switch instruction, ARM placed control (and thus also the potential for optimization) into their hands. ARM also did not make the new hypervisor mode an orthogonal dimension of privilege that would have required the introduction of a dozen new instructions to handle interaction with it, but integrated HYP mode into the existing privilege level concept and thus only had to make slight adjustments to the instruction set.

There is one additional feature in ARM's virtualization extensions that sets it apart from the rigid x86 system. When a special hypervisor control register bit `TGE` ("Trap General Exceptions") is set, the current virtual machine is limited to unprivileged mode, i. e. entering EL1 is prohibited and all traps and faults that originate at EL0 directly target the hypervisor in EL2 instead, as shown in Figure 7.1. Additionally, all architectural features that are usually controllable from EL1 are disabled or operate with default settings.

This offers a different solution to the problem of heavyweight entry and exit: where the nature of the VM permits, the bank of privileged registers can be completely disregarded. These *lightweight VMs* are neither passively affected by the values of these privileged registers nor are they able to actively inspect or change them. We can use this feature to implement a cheaper variant of the VM switch if at least one of the two VMs being switched is such a lightweight VM. Before we describe our prototype implementation on top of Phidias and compare the world switch costs of lightweight and full VMs under typical usage scenarios, we first provide a thorough analysis of the different approaches of hardware-assisted virtualization.

7.2. Hardware Virtualization Analysis

In order to understand the development of virtualization extensions on present-day architectures, we now look at the design decisions Intel, AMD and ARM have made

and discuss how they affect the costs of a *world switch* (a switch between a VM and its hosting hypervisor) and a *VM switch* (a switch from one VM context to another, the virtualization equivalent of a context switch). We discuss in particular whether each decision reduced the frequency of such events (which we call "quantitative reduction") or the cost of each individual event ("qualitative reduction").

7.2.1. VT-x and AMD-V

The search for the most efficient virtualization method on x86 has been going on for over a decade. With the advent of hardware extensions that handled basic instruction set virtualization in 2006, Adams and Agesen[2] did a first comparison of the new technology against the paravirtualization solutions that had been around for at least another decade. Unsurprisingly, they found that the world switch between hypervisor and guest was orders of magnitude more expensive than the privilege level switch between ring 0 and ring 3, which paravirtualization solutions relied upon. Together with the fact that other key features like memory virtualization still had to be performed in software, this led to the paravirtualizing hypervisor easily outperforming new hardware extension-based ones.

Additional extensions have since then removed most of the software emulation tasks that plagued the performance of early x86 hypervisors. The introduction of memory virtualization, which has been coined "Nested Paging" on this architecture, obviated the need for shadow page tables, interception of page faults and the tracing of guest page table changes. This brought a substantial quantitative reduction and also lessened the complexity of the hypervisor implementation. Its price, the two-dimensional page table walk, is rarely felt in practice, although we have seen it in action in Section 5.2 and articles by Wang et al.[95], Gandhi et al.[38] and others suggest that using Nested Paging exclusively may not be the best strategy if speed efficiency is paramount.

Another frequent cause for round-trips between VM and hypervisor is the arrival of a physical interrupt and its subsequent delivery to the corresponding VM. If one VM is the only possible recipient of a certain interrupt line, as is the usual case if a peripheral device has been exclusively assigned to it, then the delivery could be optimized by configuring the interrupt controller to directly signal this line to the virtual interrupt controller of the target VM, if the VM is currently executing. The corresponding configuration registers and the enhanced handling of virtual interrupt state were added with the recent `APIC-v` (Intel) and `AVIC` (AMD) extensions, which again meant a quantitative reduction.

Finally, peripheral devices capable of first-party DMA as well as third-party DMA engines have been a major problem in creating secure virtualized systems, as we have briefly discussed in the introduction of Chapter 5. DMA-capable peripheral devices have direct access to physical memory and therefore bypass the MMU and the restrictions imposed by the page tables, whether shadowed or nested. A VM which has been granted unmediated access to a DMA-capable device could therefore easily break out of its confinement by instructing the device to write anywhere in physical memory. Classically, hypervisors had no choice but to trap and emulate VM access to each DMA-capable device, even if only a single VM was allowed to use it.

The response to this problem was the introduction of IOMMUs: dedicated MMUs which complement system security by translating and limiting peripheral device access to the memory bus (as opposed to accesses caused by the CPU), consulting their own set of page tables. The integration of IOMMUs in the embedded platform

sector is slow though, and we consider it unlikely to reach low-end platforms soon, as the need for virtualization—and thus the benefit of adding the separation capabilities that IOMMUs provide—on these platforms has not been recognized yet.

All these enhancements have lessened the overall weight of x86 virtualization by removing more and more causes for an expensive world switch to happen. This allows VMs to run with very few interruptions, which also puts less stress on the cache hierarchy and the TLB, as repeated interventions by the hypervisor would invariably cause thrashing.

Ultimately, the cost of world and VM switches can only be partially sidestepped on x86 though. This is a direct and inevitable consequence of the realisation of virtualization mode as an orthogonal concept to the established four-tier privilege level hierarchy: even though the hypervisor is colloquially said to reside in "ring -1", in fact x86 processors have two incarnations of their original privilege levels ring 0 to ring 3, one in so-called "VM root mode", where VM control registers can be configured, and "VM non-root mode", where these controls (e. g. instruction traps and nested paging settings) apply. The hypervisor and a guest OS therefore both reside in ring 0, which inevitably requires an atomic switch of the register set.

7.2.2. ARM VE

ARM entered the hardware virtualization business five years later. Their Virtualization Extensions for the ARMv7 architecture[6] encompassed several features at once which had been introduced step by step for the x86 architecture: besides the basic instruction set virtualization, ARM VE also contain memory virtualization (dubbed "Stage-2 Translation") and one half of interrupt virtualization, enabling guests to perform acknowledgment and end-of-interrupt signalling without exiting to the hypervisor while still requiring VM exits for interrupt delivery. These extensions were later carried over more or less unchanged into the next generation architecture ARMv8[7].

The obvious fundamental difference to the x86 approach lies in the way the new execution mode for the hypervisor is integrated with the existing model: ARM's HYP mode is a linearly more privileged level of execution, occupying the new "Exception Level 2" (EL2) above the two existing ones for kernel (EL1) and userspace (EL0). The new mode uses a different set of control registers, which makes it impossible to run the same OS kernel in EL2 as in EL1. This may seem like a disadvantage. However, the upside of this is that a world switch between EL1 and EL2 does not require saving and restoring as many architectural registers as a world switch between guest ring 0 and hypervisor ring 0 does. The two privilege levels on ARM only share the general purpose registers and the FPU (and the hypervisor most likely does not use the latter itself). Not having to save and load all other control registers naturally also implies that no revalidation of the registers' contents has to happen, which is just what the x86 VM state shadowing concept accomplishes.

Nevertheless, multiple VMs which run on the same hypervisor do share all regular privileged (EL1) registers, so performing a VM switch involves saving and restoring the full EL1 set without any hardware acceleration. This is where the `TGE` control bit unleashes its potential: for VMs which do not require two privilege levels, access to EL1 can simply be temporarily deactivated and the corresponding parts of the VM switch save/restore sequence skipped.

full VCPU	FPU and EL1 required
non-FPU VCPU	only EL1 required, no FPU
lightweight VCPU	neither FPU nor EL1 required

Table 7.1.: VM classification.

7.3. Implementation

We now discuss the technical details of our optimized VM switch and the necessary changes to the basic VCPU model in Phidias, before we then motivate and describe our experimental setup.

7.3.1. Technical Design

The initial design of Phidias assumes that all VCPUs run under identical virtualization settings, and thus only a single architecture-dependent code path for world switch (and VM switch, respectively) exists. We now expand this by introducing the notion of *VM weight*: VMs which need more complexity (FPU and/or privilege separation) are considered "heavier" and require more expensive VM switch operations.

In order to measure the overhead of the individual components of a fully-fledged VM switch, we categorize VCPUs into three classes of decreasing weight according to Table 7.1. We enforce that code executed by a VCPU adheres to its configured class by preventing access to the prohibited components using the architectural EL2 registers `HCPTR` (for denying access to coprocessors 10 and 11, which provide FPU functionality) and the aforementioned `HCR` (for disabling EL1 by setting the `TGE` bit). With these safeguards in place, we can then disable the appropriate parts of the VM switch code:

I. If the destination VCPU of a switch does not use a particular component which the source VCPU did use, we note the source VCPU as owner of that component, but do not save that component's state.

II. If a different VCPU capable of accessing that component is later scheduled, the component switch is performed at that time (*lazy switch*) and ownership is transferred.

III. If the original VCPU is scheduled again and did not lose ownership of the component in the meantime, we have saved two save/restore cycles for that component.

The goal of our experiment is to quantify the relative ratio of event (III.) and to measure the costs of the VM switch operation for each component: core, FPU and EL1.

One particularly inconvenient drawback of disabling EL1 is the fact that architectural interrupt delivery to VCPUs relies on the presence of EL1 and its features, among them the `VBAR` register, which is used by OS code to specify the interrupt entry point, and the EL1 #IRQ processor mode itself. Under regular circumstances, an ARM hypervisor would flag interrupts as pending in the virtual interface of the interrupt controller, and those would result in an EL1 interrupt entry as soon as the VCPU enabled its virtual interrupt flag.

Without EL1, we have to resort to a gentle amount of paravirtualization to the guest operating system in order to work around the unavailability of the #IRQ execution mode and the `VBAR`-based delivery. Common OS implementations like Linux immediately switch out of #IRQ anyway, and the injection is trivial to build in software, so the changes are less invasive than we initially imagined. For our experiments, we used a different codebase for the implementation of our lightweight VCPU, which required even less adaptation.

7.3.2. Experimental Setup

We have analyzed two settings which might benefit from the introduction of lightweight VCPUs as defined above. The first scenario, which we are going to refer to as *throughput scenario*, involves a very high VM switch frequency and vastly asymmetric workload complexity. The following real-world examples exhibit these characteristics:

- Dual-persona smartphones have become an attractive design: they allow users to install apps at will in the "open compartment", while employment-related confidential data (company mail, documents, address book) is protected from malicious apps in a separate "secure compartment". Confidential data can further be prevented from leaking by forcing all communication links of the secure compartment through a mandatory virtual private network (VPN). In order to implement the required hardware multiplexing of the peripheral device that is providing connectivity (modem and wireless), a third "driver compartment" is set up, which is put in charge of driving the hardware and routing packets between the outside world and each of the two VMs. As this driver VM now imposes an additional scheduling step for each sent and received packet, its impact should be minimal. However, all packet forwarding and filtering is presumably done inside the kernel network stack, so a lightweight VCPU would be perfectly suited.

- Suppliers of automotive IT solutions are looking into integrating more and more functionality into a single SoC, placing high-throughput VMs like an Android-based car multimedia system next to much less complex AUTOSAR VMs that are tasked with gathering sensor readings or driving convenience subsystems like climate control. An optimal scheduling solution would give just enough processor time to the AUTOSAR VMs to meet their scheduling requirements, but yield all remaining time to Android to provide a fluent and comfortable user experience. Making the switch operations to and among the AUTOSAR VMs less expensive improves the overall system utilization and responsiveness.

- Embedded systems with high availability requirements can use virtualization as a means to add a watchdog VM in order to monitor the primary functionality of a device. The watchdog could analyze the SoC configuration, inspect the memory of the primary VM, or even run tests against its exposed interface. In order to interfere as little as possible with the intended functionality of the device, the impact of the watchdog should again be as small as possible, thus again motivating the use of a lightweight VCPU.

All these use cases have in common that there is one foreground VM, which consists of full VCPUs, and one or more lightweight VCPUs, which complement system functionality. This provides ideal circumstances for our design, as in this case

all VM switches are of type (I.) or (III.)—there is only a single possible owner for the VM switch components we have singled out (EL1, FPU).

The second scenario, the *latency scenario*, is concerned about the cost of a single VM switch instead of the cumulative switch cost per period. The example use case here is a multi-tenant system consisting of full VCPUs without any reaction latency guarantees and low scheduling priority and one or several lightweight VCPUs (again possibly AUTOSAR or FreeRTOS) with low-latency requirements and highest scheduling priority. Interrupts destined for the latter group of VCPUs thus have to lead to an immediate rescheduling operation with minimal delivery time.

7.4. Evaluation

We now evaluate our optimization under both scenarios and determine its net effect on throughput performance and hypervisor-induced latency. As for our previous analyses, we have chosen the ARMv8-A HiKey development board as the hosting platform for our experiments. We have augmented Phidias to support the different VCPU classes as described at the beginning of Section 7.3 and made the corresponding parts to the VM switch code appropriately conditional.

7.4.1. Microbenchmarks

Our experimental implementation contains two single-VCPU VMs. The first VM is running Linux, and its VCPU is granted access to both FPU and EL1. The other VM is running a port of *lightweightIP (lwIP)*[31] and is run either as a full or a lightweight VCPU for the following comparison measurements. We added a network driver on top of the shared memory buffers and software interrupts provided by Phidias to both *lwIP* and Linux, and we adapted the platform-specific code of *lwIP*—particularly the startup phase—to support running *lwIP* in either EL0 or EL1.

We then created traffic on the network link between the two VMs and ensured that Phidias would reschedule VMs after each packet, thus creating the worst case in terms of overhead. The total amount of time spent in the hypervisor was accumulated by manually instrumenting a few code locations (hypervisor entry (T1), at VM state save (T2) and load (T3), and upcall (T4)) and reading the performance counters built into the Cortex-A53 cores.

The resulting numbers for a single VM switch, averaged over several thousand samples, are listed in Table 7.2. For both directions of the VM switch, the rows of the table show the cycles spent before the actual VM switch operation (T1—T2), the cycles spent performing the low-level switch (T2—T3) and the cycles until the next VM is resumed (T3—T4). As expected we only see significant differences in the T2—T3 category. Unconditionally switching the FPU register file as shown in italics in the first column is, also unsurprisingly, a tremendous slowdown and has just been included for illustrative purposes, because the FPU—as opposed to EL1—can be switched lazily.

When we compare the second column with the third, we find that not switching EL1 resources reduces the core VM switch by another factor of two, even though this requires additional reconfiguration of hypervisor control registers (e. g. `HCR`), which the above numbers already contain. For a full VM switch round-trip we measure a time reduction of 14 % (2,799 vs. 3,248 cycles).

interval	*full*	no FPU	no FPU&EL1	VGIC opt.
Linux to lwIP				
T1—T2	*824*	825	818	578
T2—T3	*935*	437	215	213
T3—T4	*126*	143	157	156
Σ	*1,885*	1,405	1,190	947
lwIP to Linux				
T1—T2	*412*	401	404	394
T2—T3	*909*	457	216	221
T3—T4	*973*	985	989	903
Σ	*2,294*	1,843	1,609	1,518
Σ_{RTT}	*4,197*	**3,248**	**2,799**	**2,465**

Table 7.2.: Comparison of full and lightweight VM switch. All numbers given in CPU cycles. Measurement points are: hypervisor entry (T1), start (T2) and end (T3) of low-level VM switch operation, and upcall (T4).

7.4.2. VGIC Optimization

After eliminating the EL1 register set from the VM switch operation, we noticed that the specification of the `TGE` mechanism also mandates that the virtual interface of the interrupt controller (VGIC) is inactive while a virtual machine with disabled EL1 is running, regardless of the VGIC's register contents. This allows us to also omit the VGIC from the switch, as no interrupt injection will take place even if there are pending entries in the VGIC. The resulting cycle measurements are listed in the fourth column of Table 7.2.

As the interrupt controller is not handled during the core VM switch operation, but considered "peripheral state", the corresponding save and restore functionality is part of the outer measurements (E1—E2 when switching away from Linux, E3—E4 when switching back). The large asymmetry in the resulting cycle counts is again due to our conservative implementation of the pending interrupt check in the upcall logic, which we have explained during the evaluation of our core implementation in Section 5.1. Even with this high upcall cost, we note that this simple additional optimization improves our total VM switch round-trip time reduction to 25 % (2,465 vs. 3,248 cycles); with a similar upcall optimization as illustrated before, this percentage could be increased further.

7.4.3. Discussion

In order to put these numbers into perspective, we calculate the cumulative overhead per second for the "mandatory VPN VM" *throughput scenario*, assuming realistic VPN bandwidth conditions. We continue to assume the worst case of one VM round-trip per packet transferred. Then the total benefit per second of passing a data stream of 1.5 Mbits/s through a lightweight VCPU instead of a full VCPU is[2]:

$$\begin{aligned} 1.5 \times 10^6 \text{ bits/s} &\sim 1100 \text{ pkt/s} \\ &= 1100 \text{ VMRTT/s} \\ &= 1100 \cdot (3248 - 2465) \text{ cyc/s} \end{aligned}$$

[2] assuming an MTU of 1,400 bytes and a processor running at 500 Mhz

$$= 1100 \cdot 783 \text{ cyc/s} = 861300 \text{ cyc/s}$$
$$\sim 1.72 \text{ ms/s}$$

A speedup of this order of magnitude is meaningless, as the jitter on the external (physical) network connection is likely far higher. Throughput-oriented scenarios like the mandatory VPN VM can therefore be discarded as uninteresting for our optimization. It remains an open question whether there are real-world settings with an even higher VM switch ratio, where this difference might be meaningful.

We also calculate the improvement in reaction time for our *latency scenario*. If we assume the same processor speed as above, we determine that switching into a lightweight VM instead of a full VM saves almost one microsecond. While this is in itself a satisfying improvement, it is overshadowed by potential secondary costs like TLB and cache misses, which incur delays in the same order of magnitude.

7.5. Proof Implications

While the optimization we have presented implies invasive changes to core operations of our hypervisor, the necessary changes to our proof system as presented in Section 6.1 are quite manageable. The implementation requires the introduction of two new CPU-local global variables to store the current owner of each component (FPU and EL1 state), as it might now differ from the currently executing VCPU. We do not have to introduce any additional path classes, because the possible trajectories through our hypervisor remain unchanged—we merely update our lists of class-specific and global invariants (cf. page 44). The following list only contains the items which have to be modified; for improved readability we have split invariant **IC3** into three separate properties due to the higher complexity of component ownership.

IC2'. For path classes **C2b** and **C2d**, which lead to descheduling of the current VCPU, each component state of this VCPU at entry is correctly saved into the corresponding part of the VCPU state save area, or the VCPU is still the owner of that component as reflected by the corresponding global ownership variable.

IC3'a. For path class **C2e** and each component that the scheduled VCPU is the returning owner of, the component state is not altered by the path.

IC3'b. For **C2e** and each component that is owned by another VCPU, but required by the scheduled VCPU, its state at path entry is correctly saved into the other VCPU's state save area, the scheduled VCPU's component state is correctly loaded, and the corresponding global ownership pointer is updated.

IC3'c. For **C2e** and each component not required by the scheduled VCPU, the component state is not altered by the path.

IG6'. The state of dormant VCPUs (which are neither currently executing nor being scheduled in or out) is neither read nor written, except for the late component state switch as mandated by **IC3'b**.

Validating the class-specific invariants **IC1**–**IC3'c** is no more difficult than the original invariants, although inspecting the concrete output state is necessary to obtain the final owner of each component in order to generate the required SMT

equations. The modification to **IG6** only grants a specific exception, so that the lazy component switch is actually allowed, and is otherwise unchanged.

As a result we are able to determine that the addition of the VM switch optimization does not lead to an increased number of traces that have to be simulated; the necessary changes to the invariants just mandate a slightly extended validation mechanism in order to verify the preservation of the desired integrity property.

7.6. Conclusion

The original intent behind the introduction of the `TGE` feature we have used in this experiment most likely was to offer implementers an easy way to build ARM Type II hypervisors by lifting their existing OS kernels into EL2. Doing so allows the execution of userspace programs on top of the host OS using `TGE` as well as driving full VMs with `TGE` deactivated. ARM has further facilitated this road with the architectural extension ARMv8.1[8], which increased compatibility between EL1 and EL2, significantly lowering the bar for porting legacy EL1 kernels to run at the new privilege level.

Our experiments with `TGE` have demonstrated that it also offers performance benefits for latency-critical virtualized systems which are controlled by a Type I hypervisor. While the amortized overhead is likely negligible, the interrupt delivery latency can be reduced by one microsecond, which constitutes approximately one third of the total time spent in the hypervisor during the delivery. On top of these findings, we have also explained why the optimization does not jeopardize our automated proof generation.

Embedded systems with such demanding latency requirements as to noticeably benefit from our measurements usually do not choose an ARM application profile (ARMv7-A / ARMv8-A), as worst-case execution times would have to take TLB misses, two-dimensional page table walks and cache evictions into account. Instead, these systems are often based on the real-time or microcontroller profiles, which on the other hand do not offer any virtualization support.

ARM's upcoming ARMv8-R profile promises to fill this gap, offering full hardware-assisted virtualization while only providing memory protection, thus obviating the need for page table walks or TLBs, similar to ARMv7-R. There is still no publicly available specification, but the announced feature set includes running tasks directly under hypervisor control without an intermediary OS. Thus we are looking forward to porting our experiments to such platforms and further studying the performance implications as soon as devices become available.

8. Derived Work

We have described a new rigid design system for the creation of highly platform-targeted hypervisor binaries from a flexible and generic code base accompanied with a compile-time configuration system. Our design exists as a fully-fledged implementation, ready for academic use on the platforms that are already supported. Further platforms and even architectures can be easily added, as we have demonstrated with our porting study from MMU-based to MPU-based ARM processors.

Before we conclude our work with an outlook on further research avenues based on this evolving design, we first provide an overview of already existing research projects we have taken part in and which either directly build upon our hypervisor or at least share some of its design principles.

8.1. Direct Descendants

We begin our survey with two articles which are directly based on PHIDIAS and SCHISM. Both gently extend the basic model defined in Chapters 3 and 4 in order to provide additional security features.

8.1.1. XNPro

> Title: **XNPro: Low-Impact Hypervisor-Based Execution Prevention on ARM**[72]

By the very nature of privilege levels, attacks, and infiltrations which successfully acquire control over a certain privilege level can only be reliably detected from a higher level. The main threat for ARM-based platforms are currently kernel rootkits, as even more advanced rootkits which target the firmware are inherently non-portable and immensely complex. These kernel rootkits aim at injecting or altering code into the kernel and then redirecting execution flow to this malicious code while the processor is executing in privileged mode.

Even though ARM has recently retrofitted ARMv7 with a PXN ("Privileged Execute Never") page table bit which can be used to flag pages as ineligible for execution in kernel mode, this protection can possibly be circumvented. If a bug exists in the guest OS kernel that allows an attacker to overwrite arbitrary memory locations to redirect execution flow or to plant malicious code, that same bug might be used in a similarly well-targeted manner to overwrite page table entries and clear the PXN bit.

We have therefore devised a protection solution that operates on a higher privilege level, thus removing the attacker's capability to reconfigure the kernel page tables and make the memory page with the injected code executable again. To this end, we slightly altered the page table generation code and the VM paging control logic to use two different page tables for a single VM: one for VM execution at EL1 and one for execution at EL0. Transitions between these two guest-controlled privilege levels cause a legitimacy check and (if passed) a page table switch.

As the guest kernel announces the set of benign EL1 execution pages during bootup, before any hypothetical attack vectors are available to attackers, our hypervisor can ensure that these pages are never altered again (by clearing the 'W'ritable bit in the stage 2 page table) and prevent execution of any other code at EL1 (by clearing the e'X'ecutable bit in the complement set of the stage 2 page table). Advanced attack techniques such as *ret2usr*[54] or even *ret2dir*[53] are thus easily defeated.

The XNPro (Execute Never Protection) extension to our hypervisor benefits from the clean and rigid design. The accompanying benchmark results show that our hypervisor implementation is only approx. 5 % behind the highly optimized and Linux-tailored *Xen* hypervisor, and that the XNPro feature itself only costs additional 1.5 % of performance.

The extension can operate in two modes. One involves determining the set of EL1 executable pages and feeding this data into the compile-time page table generator, thus gaining protection from system boot. The other approach is more flexible and introduces a non-reversible hypercall which the guest kernel uses to communicate the set of executable pages to the hypervisor. This violates our Principle of Staticity and requires read access to the page tables at runtime, because the hypervisor has to create the two page table descendants on arrival of the notification hypercall. This is a necessary trade-off if the guest kernel's physical base address is not constant across reboots[1].

The full XNPro feature required an addition of around 300 lines of code, demonstrating the feasibility of using our hypervisor design for the creation of useful and low-overhead security solutions.

8.1.2. RkDetect

Title: **Uncloaking rootkits on mobile devices with a hypervisor-based detector**[94]

In addition to reducing the number of possible attack vectors for the deployment of kernel rootkits, we have also developed a detector for already present rootkits based on our hypervisor. In order to be able to detect all prevailing types of kernel rootkits, we had to implement a wide range of validity checks, some of them crossing the semantic gap between hypervisor and guest OS (for a detailed discussion of the concept, see e. g. Dolan-Gavitt[30]).

As the implementation of all these checks inside the hypervisor would have massively bloated its size, we opted for a "sibling VM" approach which is tasked with the actual detection. Our only addition to the functionality provided by the hypervisor was the capability to perform atomic snapshots of a VM and the visibility of such created snapshots to other VMs, as configured by the specification. These snapshots included both a full RAM copy and the state of the architectural registers. The detector VM could then peruse the state dump and apply its detection techniques.

While performing the snapshot as a single uninterruptible hypervisor operation was a safe implementation choice and required no further changes, we again saw the need to optimize, as taking a snapshot of an Android VM with 512 MB of RAM caused the system to freeze for several seconds. We thus reintroduced the runtime page table clone feature from XNPro and implemented a primitive copy-on-write

[1] The Linux kernel contains a RANDOMIZE_BASE option for the x86 architecture since late 2013, which causes it to choose a random physical base address during bootup. The ARMv8 port gained identical functionality in January 2016.

mechanism: the target VM is switched to a fully read-only stage 2 page table, and pages of RAM are copied into the snapshot area both on a write fault (i. e. when the VM tries to replace the contents of a page with new data, which causes a trap due to the read-only permission override of the second stage translation) and continuously in small increments.

The revocation of write permission ensures that the memory snapshot is indeed atomic, even though the actual copy operation is not. The continuous background copying of pages on the other hand ensures that the whole operation completes in finite time even if the target VM does not access every page. This combined strategy completely removes the noticeable system freeze, leaving only a slight slowdown directly after initiating a snapshot due to the brief initial surge in write page faults caused by the victim VM.

The number of added code lines roughly matches the figure of the XNPro implementation; the runtime page table derivation code is comparable, and the addition of the snapshot hypercall is of similar low complexity to the hypercall added above. The copy engine is just a few lines of code hooked into the VM page fault handler and the time event queue.

Performance figures also closely match the XNPro results. The total overhead caused by our hypervisor is approx. 3 %. Most benchmarks show no measurable slowdown during creation of a snapshot—only Antutu's "RAM Speed" is affected, presumably by cache thrashing on the platform's shared L2 cache caused by the second CPU's ongoing copy operations.

8.2. Technologically Related Articles

We continue with two more articles which illustrate individual key decisions of our design, even though these research efforts do not build on our hypervisor implementation.

8.2.1. Usurping Hypervisor Mode

Title: **The threat of virtualization: Hypervisor-based rootkits on the ARM architecture**[20]

Where the two previous articles demonstrated that hypervisor mode can be used to protect the less privileged layers from malware attacks, we were able to show in this instalment that other popular open source projects take little care while operating at this privilege level.

As ARM's hypervisor mode (EL2) is not equivalent to regular privileged mode (EL1), Linux checks the execution level at bootup and switches out of hypervisor mode if necessary[2]. In order to make it possible to reclaim EL2 privileges if a hypervisor module (e. g. *KVM*) is loaded later during bootup or even much later on demand, Linux leaves the `HVC` instruction enabled and installs minimal stub code which allows to reset the EL2 exception vector base address. If no benign hypervisor module is loaded, this is an open invitation to attackers to "bluepill" the platform by installing malicious code into the hypervisor privilege level, in the same way as Rutkowska et al. have famously demonstrated for the x86 architecture[82].

[2] The first update to the ARMv8 architecture (ARMv8.1)[8] introduces additional coprocessor registers for EL2 which finally allow to run Linux almost unmodified in hypervisor mode, which—at least from the vantage point of our publication—made the situation even worse.

Even on systems where the *KVM* module has successfully taken control of hypervisor mode, the danger of an attacker taking over EL2 is not averted. Due to the tight integration of *KVM* within Linux, regular kernel code is allowed to freely execute function calls in EL2 through a convenient elevation hypercall. Thus the wide open barn door is completely unhinged by *KVM*, making it even easier for attackers to elevate their privileges to hypervisor mode and injecting their stealthy rootkit.

We demonstrated the validity of our findings by implementing attack code for several injection vectors, among them the ones discussed above, and installed a minimal rogue hypervisor, whose implementation follows the same basic principles as our main design, Phidias. In order to minimize the impact and visibility of our attack, we stripped our rogue hypervisor of several major subsystems, among them VM switching and scheduling (as we assume that a single (host) OS is executing at the time of attack) and most device emulation code. We retained the interrupt controller emulation to hide activation of the hypervisor timer interrupt and integrated the pregenerated stage 2 page table directly into the image to further reduce our memory footprint.

Our final stripped hypervisor fits into at most four memory pages, depending on the type of malicious background activity and the chosen stealth technique. We arrived unsurprisingly at even lower overhead figures, many of them one order of magnitude below the standard deviation and thus undetectable unless a vast number of measurements is involved.

8.2.2. Covert Channels in Microkernels

Title: **Undermining Isolation through Covert Channels in the *Fiasco.OC* Microkernel**[77]

This article is not directly related to the hypervisor implementation we have presented in this thesis, but it still demonstrates the strengths of our design. Based on our stance towards dynamic management of kernel objects, we analyzed the implementation of the resource management system built into the *Fiasco.OC* microkernel[75].

Fiasco.OC is a member of the L4 family of microkernels and thus adheres to the L4 interface specification. Its suffix "OC" indicates that it has adopted an object capability model: kernel objects are represented by capabilities (opaque handles similar to file descriptors on Linux) in userspace, and these capabilities can be used to transfer rights over these objects or to invoke functionality on them. New objects are created by allocating a fresh empty capability and then calling the appropriate kernel factory to create a new object and link it to that capability.

Objects are created from a central kernel memory pool instead of individual pools per creating entity, but each entity is restricted by a memory quota. The actual allocation process is twofold: memory is taken from the main buddy allocator to form a slab for the desired element size, and then objects are created from that slab.

Effective memory consumption thus depends on the number of slabs, but quota is accounted based on the actual number and size of created kernel objects. Therefore it was possible for an agent to use up a multiple of its allotted quota size. We could determine that an agent could use the sixfold amount of memory by carefully constructing the worst-case ratio between the number of created objects and the (mostly empty) slabs they resided in.

We also found two other covert channels which relied on the dynamic nature of the underlying microkernel. The first one abuses the microkernel's `map`/`unmap` facility.

As L4 tasks are always accompanied by their corresponding page table, mapping a new page causes a page table update, which possibly requires allocation of a new page directory. This allocation can again be used to cause memory starvation which is detectable from a different task. The second covert channel uses the arbitrary limitation that any single physical page can only be mapped a finite number of times. We combined this limitation with the fact that every task contains a read-only mapping of the L4 runtime environment (L4Re), a userspace support library which provides a convenient C API for the functionality offered by the microkernel. By raising the number of mappings for each of L4Re's pages near below or to the maximum, we can again transmit data between unrelated entities.

We finally determined that two colluding compartments were able to exchange 30 kbit/s of data despite being prohibited to communicate by the system configuration. This is a tremendous blow to the isolation guarantee provided by that microkernel.

While some of the channels above can be attributed partially to bugs specific to this particular implementation, we cannot help but recognize that the main culprit is the direct manipulability of system-level resources. As L4 system calls may cause direct modifications to page tables or the creation and destruction of kernel objects, the effect of those system calls may be visible to more entities than intended.

These results vividly demonstrate the fragility of dynamic systems and the difficulties of creating interfaces for dynamic reconfiguration that cannot be abused. Open Kernel Labs created an elegant solution to this problem by moving the burden of providing the backing memory for new objects into userspace as well[58]. Still, a feature that is not present cannot be attacked. By strictly adhering to our Principle of Staticity for the core design, we avoid these pitfalls from the outset.

8.3. Ongoing Projects

With the current available set of ARM architectures covered and portability to x86 tentatively verified, we have recently extended our attention to architectures which challenge some of the underlying assumptions of our design. The MIPS architecture has been an attractive target for porting our hypervisor ever since an application-specific extension (ASE) for virtualization[48] was announced. The special characteristic of MIPS is that the presence of a hardware page table walker is completely optional, even for platforms equipped with virtualization support. On the other hand, MIPS offers excellent support for low-latency interrupt delivery by automatically switching between register sets.

Creating a port for MIPS seizes the opportunity to both test the limits of our Principle of Staticity, caused by the requirement to construct a driver for the software-filled TLB, and push the lower bound of the latencies that are incurred by Phidias. It will be especially interesting to compare the performance results of our port with other MIPS hypervisor implementations like e. g. *Hellfire*, recently renamed *prpl-Hypervisor*[68, 67], which occupy less extreme stances in the design spectrum.

9. Conclusion and Future Research

In this work, we have introduced a new design paradigm for system software for embedded devices. Reducing complexity in itself is by no means a novel idea, especially not in the context of operating systems: in fact, we directly build on Liedtke's minimality principle, which is all about reduction in kernel complexity. We argue that this work breaks new ground by a combination of features. Firstly, we shift the focus from the number of implemented components to the way they are implemented—we even allow a few elements back into the kernel which could be pushed out into a separate userspace VM, as long as they rigidly conform to our principle. Secondly, we demonstrate that there is a tangible benefit in persevering with the application of our principle: ease of automated verification. Finally, the design of our static configuration framework, our evaluation, and the optimization experiment show that our concept does neither introduce any implicit performance degradations nor impair overall versatility.

Besides our conceptual contributions to system software design, we have also provided an implementation to the new field of Type I hypervisors for the ARMv8 architecture. Concentrating on this port has limited our own options of comparison throughout our evaluation, but this choice may turn out to be beneficial to future researchers, as many manufacturers of mobile phones and embedded system-on-chips have already begun migrating their product portfolio to ARMv8. Nevertheless, our design is not restricted to a single architecture either, as we have shown on multiple occasions, such as the extension of our configuration framework to support memory protection units.

Our work has opened the door for further exploration of this newly-created kind of design. Future work could take off in many different directions from where we left off. We consider the following ideas to be very promising:

- The symbolic execution engine can be employed for many different use cases beyond the integrity proof we have developed it for. Especially interesting would be an integration of a TLB and cache model into its memory access logic, similar to the work done by Chattopadhyay et al. for their CHALICE framework[22]. This extension would allow the creation of cache impact predictions for the different path classes and thus possibly lead to assertions with respect to cache-based covert channels.

- Reacting dynamically to changes in performance demands contradicts our Principle of Staticity, but the problem cannot be simply dismissed. However, ARM's big.LITTLE multicore design offers an excellent opportunity to explore a limited form of dynamicity without compromising our core principle. The resulting system configuration might prove to be an interesting study of how well our design is able to meet desired trade-off points between performance and power consumption.

- Our experiments have already shown that our design is easily extensible to support processor architectures which contain an MPU instead of an MMU, such as ARMv7-R. Where the ARMv7 MPU was severely limited in its configurability of the protection ranges, ARMv8-R offers a new MPU configuration scheme. It also introduces the two-stage MPU, finally bringing full hardware-based virtualization to the ARMv8 realtime profile. Our hypervisor could play a leading role in the exploration of this new form of virtualization.

- While the non-secure world of recent ARM chips offers full hardware virtualization, hypervisors for the secure world still have to resort to paravirtualization in order to run multiple secure operating systems side by side. The new ARMv8 architecture does not bring any change to this dichotomy. Our hypervisor is not only a natural fit for the role as secure world virtualization host due to its small size; it might also be worthwhile to extend its current implementation to create a hybrid which simultaneously controls virtualization in both worlds.

In addition to these areas, we also envision a continuation of our research efforts into even smaller devices, building upon our nascent MIPS port. Possible future targets include devices based on an ARM Microcontroller Profile (ARMv7-M and future ARMv8-M), fault-tolerant SPARC LEON3 SoCs, and Infineon TriCore platforms.

A. Code Fragments

A.1. XML Architecture Definitions

The following two excerpts have been taken from the ARMv8 and x86 architecture definition files of our configuration system library. Each illustrates the specification of a single page table format, including the size of entries, the number of bits translated per level, and the positions of permission bits.

```
<?xml version="1.0" encoding="UTF-8"?>
<arch id="arm64">
  <!-- Assumptions:
       * The hypervisor is operating at EL2.
         EL1 will require different bit patterns, and especially a
         usable 'g'lobal bit.
       * MAIR_EL2 index 0 is programmed as Normal Memory.
       * MAIR_EL2 index 1 is programmed as Device_nGnRnE.
         (No other MAIR index is used.)
  -->
  <paging_format id="arm64:g4io40" va_width="64" pa_width="64"
                       size_estimate="0x10000">
    <level dir_base="0x8000000000000003" shift="39"
                       width="1" bpe="8" align="12">
      <flag name="r" value_set="0x0" value_clear="0x0" />
    </level>
    <level dir_base="0x8000000000000003" leaf_base="0x461" shift="30"
                       width="9" bpe="8" align="12">
      <flag name="r" value_set="0x0" value_clear="0x0" />
      <flag name="w" value_set="0x0" value_clear="0x80" />
      <flag name="x" value_set="0x0" value_clear="0x40000000000000" />
      <flag name="g" value_set="0x0" value_clear="0x0" />
      <flag name="d" value_set="0x4" value_clear="0x0" />
      <flag name="u" value_set="0x0" value_clear="0x0" />
      <flag name="s" value_set="0x200" value_clear="0x0" />
    </level>
    <level dir_base="0x8000000000000003" leaf_base="0x461" shift="21"
                       width="9" bpe="8" align="12">
      <flag name="r" value_set="0x0" value_clear="0x0" />
      <flag name="w" value_set="0x0" value_clear="0x80" />
      <flag name="x" value_set="0x0" value_clear="0x40000000000000" />
      <flag name="g" value_set="0x0" value_clear="0x0" />
      <flag name="d" value_set="0x4" value_clear="0x0" />
      <flag name="u" value_set="0x0" value_clear="0x0" />
      <flag name="s" value_set="0x200" value_clear="0x0" />
    </level>
    <level leaf_base="0x463" shift="12" width="9" bpe="8" align="12">
      <flag name="r" value_set="0x0" value_clear="0x0" />
      <flag name="w" value_set="0x0" value_clear="0x80" />
      <flag name="x" value_set="0x0" value_clear="0x40000000000000" />
      <flag name="g" value_set="0x0" value_clear="0x0" />
      <flag name="d" value_set="0x4" value_clear="0x0" />
      <flag name="u" value_set="0x0" value_clear="0x0" />
      <flag name="s" value_set="0x200" value_clear="0x0" />
```

```
    </level>
  </paging_format>
  <paging_format id="arm64:g4io40n" va_width="64" pa_width="64"
                 size_estimate="0x10000">
    <!-- Nested Translation Format ... -->
  </paging_format>
</arch>
```

Listing A.1: Paging specification for ARMv8 (excerpt)

```
<?xml version="1.0" encoding="UTF-8"?>
<arch id="x86">
  <!-- no NX for legacy x86 paging -->
  <paging_format id="x86:legacy" va_width="32" pa_width="32"
                 size_estimate="0x10000">
    <level dir_base="0x007" leaf_base="0x081" shift="22" width="10"
                 bpe="4" align="12">
      <flag name="r" value_set="0x0" value_clear="0x0" />
      <flag name="w" value_set="0x2" value_clear="0x0" />
      <flag name="x" value_set="0x0" value_clear="0x0" />
      <flag name="g" value_set="0x100" value_clear="0x0" />
      <flag name="d" value_set="0x18" value_clear="0x0" />
      <flag name="u" value_set="0x800" value_clear="0x0" />
      <flag name="s" value_set="0x4" value_clear="0x0" />
    </level>
    <level leaf_base="0x001" shift="12" width="10" bpe="4" align="12">
      <flag name="r" value_set="0x0" value_clear="0x0" />
      <flag name="w" value_set="0x2" value_clear="0x0" />
      <flag name="x" value_set="0x0" value_clear="0x0" />
      <flag name="g" value_set="0x100" value_clear="0x0" />
      <flag name="d" value_set="0x18" value_clear="0x0" />
      <flag name="u" value_set="0x800" value_clear="0x0" />
      <flag name="s" value_set="0x4" value_clear="0x0" />
    </level>
  </paging_format>
</arch>
```

Listing A.2: Paging specification for x86 (excerpt)

A.2. XML Sample Scenario Definition

The following XML configuration defines the scenario used throughout our proof efforts (see Chapter 6) and—with slight modifications—for our experiments on lightweight verification (see Chapter 7). Note the two communication channel buffers *cca0* and *cca1* and the two IPC capabilities giving each virtual machine a signalling mechanism to its communication partner.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE scenario SYSTEM "xml/dtd">
<scenario version="1.00" cbi="phidias" image="raw">
  <platform board="hikey" arch="arm64" />
  <hypervisor ncpus="8" load_base="0x10008000">
    <feature name="driver:uart" value="pl011" />
    <feature name="driver:timer" value="arm_generic" />
    <feature name="driver:clock" value="arm_generic" />
    <feature name="driver:irq" value="gic" />
    <memreq id="cca0" size="0x100000" />
```

```
    <memreq id="cca1" size="0x100000" />
    <address_space type="mmu" format="arm64:g4io40">
      <map xref="serial" flags="w" />
      <map xref="irqc" flags="w" />
      <map xref="gpio4j" flags="w" />
      <map xref="timers" flags="w" />
    </address_space>
  </hypervisor>
  <guest id="linux1" ncpus="1" cpumap="[0]">
    <memreq id="linux1_main" size="0x40000000" flags_demand="rw"
            flags_prevent="xdus" />
    <memreq id="linux1_arch" size="0x1000" flags_demand="rwg"
            flags_prevent="xdus" cpumap="*" />
    <vdev id="linux1_uart" type="serial" frontend="pl011"
            master="master">
      <emulate base="0xf7113000" size="0x1000" />
    </vdev>
    <vdev id="linux1_gic" type="irq_controller"
            frontend="arm_gic_virtext" master="master">
      <emulate base="0xf6801000" size="0x1000" />
    </vdev>
    <vdev id="linux1_sp804" type="timer" frontend="sp804">
      <emulate base="0xf8008000" size="0x1000" />
    </vdev>
    <vdev id="linux1_armcp14" type="timer" frontend="armcp14" />
    <vdev id="linux1_mmio_mediactrl" type="memory32">
      <emulate base="0xf4410000" size="0x1000" />
      <value type="default_mask_mem_r" value="0xffffffff" />
    </vdev>
    <vdev id="linux1_mmio1" type="memory32">
      <emulate base="0xf7020000" size="0x14000" />
      <value type="default_mask_mem_r" value="0xffffffff" />
    </vdev>
    <vdev id="linux1_mmio_aoctrl" type="memory32">
      <emulate base="0xf7800000" size="0x2000" />
      <value type="default_mask_mem_r" value="0xffffffff" />
    </vdev>
    <vdev id="linux1_mmio3" type="memory32">
      <emulate base="0xf8011000" size="0x5000" />
      <value type="default_mask_mem_r" value="0xffffffff" />
    </vdev>
    <vdev id="linux1_mmio_uart1" type="memory32">
      <emulate base="0xf7111000" size="0x1000" />
      <value type="default_mask_mem_r" value="0xffffffff" />
    </vdev>
    <vdev id="linux1_mmio_uart2" type="memory32">
      <emulate base="0xf7112000" size="0x1000" />
      <value type="default_mask_mem_r" value="0xffffffff" />
    </vdev>
    <vdev id="linux1_mmio_uart4" type="memory32">
      <emulate base="0xf7114000" size="0x1000" />
      <value type="default_mask_mem_r" value="0xffffffff" />
    </vdev>
    <vdev id="linux1_mmio4" type="memory32">
      <emulate base="0xf7100000" size="0x8000" />
      <value type="default_mask_mem_r" value="0xffffffff" />
    </vdev>
    <vdev id="linux1_mmio2" type="memory32">
      <emulate base="0xf8000000" size="0x8000" />
      <value type="default_mask_mem_r" value="0xffffffff" />
    </vdev>
```

```
    <address_space type="mmu" format="arm64:g4io40n">
      <map xref="linux1_main" base="0x00000000" flags="rwx" />
      <map xref="cca1" base="0xfee00000" flags="r" />
      <map xref="cca0" base="0xfef00000" flags="rw" />
      <map xref="irqc" base="0xf6802000" subsize="0x1000"
           offset="0x6000" flags="rw" />
    </address_space>
    <init arch_page="linux1_arch">
      <copy xref="linux_kernel" dref="linux1_main" offset="0x80000" />
      <copy xref="linux_initrd" dref="linux1_main" offset="0xa000000" />
      <copy xref="linux_dtb" dref="linux1_main" offset="0x8000" />
      <cap type="ipc" target_xref="lwip1" param="0x20" />
    </init>
    <entry bp_xref="linux1_main" bp_offset="0x80000" />
    <sched class="wfq" />
  </guest>
  <guest id="lwip1" ncpus="1" cpumap="[0]">
    <memreq id="lwip1_main" size="0x01000000" flags_demand="rw"
            flags_prevent="xdus" />
    <memreq id="lwip1_arch" size="0x1000" flags_demand="rwg"
            flags_prevent="xdus" cpumap="*" />
    <vdev id="lwip1_uart" type="serial" frontend="pl011"
            master="master">
      <emulate base="0xf7113000" size="0x1000" />
    </vdev>
    <address_space type="mmu" format="arm64:g4io40n">
      <map xref="lwip1_main" base="0x40000000" flags="rwx" />
      <map xref="cca0" base="0xfee00000" flags="r" />
      <map xref="cca1" base="0xfef00000" flags="rw" />
    </address_space>
    <init arch_page="lwip1_arch">
      <copy xref="lwip_image" dref="lwip1_main" offset="0x00000" />
      <cap type="ipc" target_xref="linux1" param="0x76" />
    </init>
    <entry bp_xref="lwip1_main" bp_offset="0x10000" />
    <sched class="wfq" />
  </guest>
  <files>
    <file id="linux_kernel" href=".../arch/arm64/boot/Image" />
    <file id="linux_initrd" href=".../ramdisk/arm64.sq" />
    <file id="linux_dtb" href=".../hi6220.dtb" />
    <file id="lwip_image" href=".../lwip/build/lwip" />
  </files>
</scenario>
```

Listing A.3: ARMv8 HiKey sample XML configuration

The following XML file is the final transformation result of the above definition after it has been processed by the configuration framework.

```
<scenario version="1.00" cbi="phidias" image="raw">
  <platform arch="arm64" board="hikey">
    <board id="hikey">
      <device id="serial" base="0xf7113000" size="0x1000"/>
      <device id="irqc" base="0xf6800000" size="0x8000"/>
      <device id="gpio03" base="0xf8011000" size="0x4000"/>
      <device id="gpio4j" base="0xf7020000" size="0x10000"/>
      <device id="timers" base="0xf8008000" size="0x9000"/>
      <device id="thermal" base="0xf7030000" size="0x2000"/>
      <device id="CATCHALL" base="0xf0000000" size="0x0fff0000"/>
```

```
<memory id="dram" base="0x00000000" size="0x80000000">
  <memreq id="trace" base="[0]=0x1200000;[1]=0x1240000;
          [2]=0x1280000;[3]=0x12c0000;[4]=0x1300000;
          [5]=0x1340000;[6]=0x1380000;[7]=0x13c0000"
          size="0x40000" cpumap="[0,1,2,3,4,5,6,7]"
          flags_demand="rwg" flags_prevent="xdus" on="dram"/>
  <memreq id="cca0" base="0x1000000" size="0x100000" on="dram"/>
  <memreq id="cca1" base="0x1100000" size="0x100000" on="dram"/>
  <memreq id="xcore" base="0x1400000" size="0x40000"
          flags_demand="rwgs" flags_prevent="xdu" on="dram"/>
  <memreq id="core_rx" base="0x10008000" size="0x8000"
          flags_demand="rxg" flags_prevent="wdus" on="dram"/>
  <memreq id="core_r" base="0x10010000" size="0x3000"
          flags_demand="rg" flags_prevent="wxdus" on="dram"/>
  <memreq id="core_rws" base="0x10013000" size="0x1000"
          flags_demand="rwgs" flags_prevent="xdu" on="dram"/>
  <memreq id="core_rwt" base="0x10014000" size="0x1000"
          flags_demand="rg" flags_prevent="wxdus" on="dram"/>
  <memreq id="core_rw" base="[0]=0x1540000;[1]=0x1541000;
          [2]=0x1542000;[3]=0x1543000;[4]=0x1544000;
          [5]=0x1545000;[6]=0x1546000;[7]=0x1547000"
          size="0x1000" cpumap="[0,1,2,3,4,5,6,7]"
          flags_demand="rwg" flags_prevent="xdus" on="dram"/>
  <memreq id="config_r" base="0x10015000" size="0x5000"
          flags_demand="rg" flags_prevent="wxdus" on="dram"/>
  <memreq id="config_rw" base="[0]=0x1548000;[1]=0x1549000;
          [2]=0x154a000;[3]=0x154b000;[4]=0x154c000;
          [5]=0x154d000;[6]=0x154e000;[7]=0x154f000"
          size="0x1000" cpumap="[0,1,2,3,4,5,6,7]"
          flags_demand="rwg" flags_prevent="xdus" on="dram"/>
  <memreq id="config_rws" base="0x1001b000" size="0x1000"
          flags_demand="rwgs" flags_prevent="xdu" on="dram"/>
  <memreq id="config_rwt" base="0x1001a000" size="0x1000"
          flags_demand="rg" flags_prevent="wxdus" on="dram"/>
  <memreq id="pagetables" base="0x1001c000" size="0xb0000"
          flags_demand="" flags_prevent="wxdus" on="dram"/>
  <memreq id="blob" base="0x100cc000" size="0xa62000"
          flags_demand="rg" flags_prevent="wxdus" on="dram"/>
  <memreq id="stack" base="[0]=0x1550000;[1]=0x1551000;
          [2]=0x1552000;[3]=0x1553000;[4]=0x1554000;
          [5]=0x1555000;[6]=0x1556000;[7]=0x1557000"
          size="0x1000" cpumap="[0,1,2,3,4,5,6,7]"
          flags_demand="rwg" flags_prevent="xdus" on="dram"/>
  <memreq id="linux1_main" base="[0]=0x40000000" size="0x40000000"
          cpumap="[0]" flags_demand="rw" flags_prevent="xdus"
          on="dram"/>
  <memreq id="linux1_arch" base="[0]=0x1558000" size="0x1000"
          cpumap="[0]" flags_demand="rwg" flags_prevent="xdus"
          on="dram"/>
  <memreq id="lwip1_main" base="[1]=0x0" size="0x01000000"
          cpumap="[1]" flags_demand="rw" flags_prevent="xdus"
          on="dram"/>
  <memreq id="lwip1_arch" base="[1]=0x1559000" size="0x1000"
          cpumap="[1]" flags_demand="rwg" flags_prevent="xdus"
          on="dram"/>
  <memreq id="lwip1_vtlbpool1" base="[1]=0x1440000" size="0x40000"
          cpumap="[1]" flags_demand="rwg" flags_prevent="xdus"
          on="dram"/>
  <memreq id="lwip1_vtlbpool2" base="[1]=0x1480000" size="0x40000"
          cpumap="[1]" flags_demand="rwg" flags_prevent="xdus"
          on="dram"/>
```

```
      <memreq id="lwip1_vtlbpool3" base="[1]=0x14c0000" size="0x40000"
              cpumap="[1]" flags_demand="rwg" flags_prevent="xdus"
              on="dram"/>
      <memreq id="lwip1_vtlbpool4" base="[1]=0x1500000" size="0x40000"
              cpumap="[1]" flags_demand="rwg" flags_prevent="xdus"
              on="dram"/>
    </memory>
    <memory id="sram" base="0xfff80000" size="0x12000">
      <memreq id="linux1_sram" base="0xfff80000" size="0x11000"
              cpumap="[0]" on="sram"/>
    </memory>
  </board>
  <arch id="arm64">
    <!-- imported architecture specification -->
  </arch>
</platform>
<hypervisor ncpus="8" load_base="0x10008000" entry="0x10008000">
  <feature name="debugger" value="yes"/>
  <feature name="tracer" value="yes"/>
  <feature name="driver:uart" value="pl011"/>
  <feature name="driver:timer" value="arm_generic"/>
  <feature name="driver:clock" value="arm_generic"/>
  <feature name="driver:irq" value="gic"/>
  <address_space type="mmu" format="arm64:g4io40"
          base="[-1]=0x10021000;[0]=0x1002d000;[1]=0x1003a000;
          [2]=0x10046000;[3]=0x10052000;[4]=0x1005e000;
          [5]=0x1006a000;[6]=0x10076000;[7]=0x10082000">
    <map xref="serial" base="0xf40be000" flags="rwgd"/>
    <map xref="irqc" base="0xf40b2000" flags="rwgd"/>
    <map xref="gpio4j" base="0xf4097000" flags="rwgd"/>
    <map xref="timers" base="0xf40a8000" flags="rwgd"/>
    <map xref="thermal" base="0xf40bb000" flags="rwgd"/>
    <map xref="CATCHALL" base="0xf4200000" flags="rwgd"/>
    <map xref="core_rx" base="0xf4000000" flags="rxg"
          is_init="is_init"/>
    <map xref="core_r" base="0xf4008000" flags="rg"/>
    <map xref="core_rws" base="0xf400b000" flags="rwgs"/>
    <map xref="core_rwt" base="0xf40c0000" flags="rg"/>
    <map xref="core_rw" base="0xf400c000" flags="rwg"/>
    <map xref="config_r" base="0xf400d000" flags="rg"
          is_init="is_init"/>
    <map xref="config_rw" base="0xf4012000" flags="rwg"/>
    <map xref="config_rws" base="0xf4013000" flags="rwgs"/>
    <map xref="config_rwt" base="0xf40c2000" flags="rg"/>
    <map xref="blob" base="0x104200000" flags="rg"/>
    <map xref="stack" base="0xf40c4000" flags="rwg"/>
    <map xref="trace" base="0xf4015000" flags="rwg"/>
    <map xref="xcore" base="0xf4056000" flags="rwgs"/>
    <map xref="linux1_main" base="[0]=0x40000000" cpumap="[0]"
          flags="rw"/>
    <map xref="linux1_arch" base="[0]=0xf40c6000" cpumap="[0]"
          flags="rwg"/>
    <map xref="lwip1_main" base="[1]=0x200000" cpumap="[1]"
          flags="rw"/>
    <map xref="lwip1_arch" base="[1]=0xf41ca000" cpumap="[1]"
          flags="rwg"/>
    <map xref="lwip1_vtlbpool1" base="[1]=0xf40c6000" cpumap="[1]"
          flags="rwg"/>
    <map xref="lwip1_vtlbpool2" base="[1]=0xf4107000" cpumap="[1]"
          flags="rwg"/>
    <map xref="lwip1_vtlbpool3" base="[1]=0xf4148000" cpumap="[1]"
```

```
            flags="rwg"/>
      <map xref="lwip1_vtlbpool4" base="[1]=0xf4189000" cpumap="[1]"
            flags="rwg"/>
    </address_space>
  </hypervisor>
  <guest id="linux1" ncpus="1" cpumap="[0]">
    <vdev id="linux1_uart" type="serial" frontend="pl011"
              master="master">
      <emulate base="0xf7113000" size="0x1000"/>
    </vdev>
    <vdev id="linux1_gic" type="irq_controller"
              frontend="arm_gic_virtext" master="master">
      <emulate base="0xf6801000" size="0x1000"/>
    </vdev>
    <vdev id="linux1_sp804" type="timer" frontend="sp804">
      <emulate base="0xf8008000" size="0x1000"/>
    </vdev>
    <vdev id="linux1_armcp14" type="timer" frontend="armcp14"/>
    <vdev id="linux1_mmio_mediactrl" type="memory32">
      <emulate base="0xf4410000" size="0x1000"/>
      <value type="default_mask_mem_r" value="0xffffffff"/>
    </vdev>
    <vdev id="linux1_mmio1" type="memory32">
      <emulate base="0xf7010000" size="0x24000"/>
      <value type="default_mask_mem_r" value="0xffffffff"/>
    </vdev>
    <vdev id="linux1_mmio_dwmmc" type="memory32">
      <emulate base="0xf723d000" size="0x3000"/>
      <value type="default_mask_mem_r" value="0xffffffff"/>
    </vdev>
    <vdev id="linux1_mmio_xrange" type="memory32">
      <emulate base="0xf7500000" size="0x300000"/>
      <value type="default_mask_mem_r" value="0xffffffff"/>
    </vdev>
    <vdev id="linux1_mmio_aoctrl" type="memory32">
      <emulate base="0xf7800000" size="0x2000"/>
      <value type="default_mask_mem_r" value="0xffffffff"/>
    </vdev>
    <vdev id="linux1_mmio3" type="memory32">
      <emulate base="0xf8011000" size="0x5000"/>
      <value type="default_mask_mem_r" value="0xffffffff"/>
    </vdev>
    <vdev id="linux1_mmio_uart1" type="memory32">
      <emulate base="0xf7111000" size="0x1000"/>
      <value type="default_mask_mem_r" value="0xffffffff"/>
    </vdev>
    <vdev id="linux1_mmio_uart2" type="memory32">
      <emulate base="0xf7112000" size="0x1000"/>
      <value type="default_mask_mem_r" value="0xffffffff"/>
    </vdev>
    <vdev id="linux1_mmio_uart4" type="memory32">
      <emulate base="0xf7114000" size="0x1000"/>
      <value type="default_mask_mem_r" value="0xffffffff"/>
    </vdev>
    <vdev id="linux1_mmio4" type="memory32">
      <emulate base="0xf7100000" size="0x8000"/>
      <value type="default_mask_mem_r" value="0xffffffff"/>
    </vdev>
    <vdev id="linux1_mmio2" type="memory32">
      <emulate base="0xf8000000" size="0x8000"/>
      <value type="default_mask_mem_r" value="0xffffffff"/>
```

```
    </vdev>
    <address_space type="mmu" format="arm64:g4io40n"
            base="[0]=0x10084000">
      <map xref="linux1_main" base="0x00000000" flags="rwx"/>
      <map xref="linux1_sram" base="0xfff80000" flags="rwx"/>
      <map xref="cca1" base="0xfee00000" flags="rs"/>
      <map xref="cca0" base="0xfef00000" flags="rws"/>
      <map xref="irqc" base="0xf6802000" offset="0x6000"
            subsize="0x1000" flags="rw"/>
    </address_space>
    <init arch_page="linux1_arch">
      <copy xref="linux_kernel" dref="linux1_main" offset="0x80000"/>
      <copy xref="linux_ramdisk" dref="linux1_main" offset="0xa000000"/>
      <copy xref="linux_dtb" dref="linux1_main" offset="0x8000"/>
      <cap type="ipc" target_xref="lwip1" param="0x20"/>
    </init>
    <entry bp_xref="linux1_main" bp_offset="0x80000"/>
    <sched class="wfq"/>
  </guest>
  <guest id="lwip1" ncpus="1" cpumap="[1]">
    <vdev id="lwip1_uart" type="serial" frontend="pl011"
              master="master">
      <emulate base="0xf7113000" size="0x1000"/>
    </vdev>
    <vdev id="lwip1_dummy_vtlb" type="vtlb" frontend="arm64:g4io40"
              master="master">
      <param type="backing" xref="lwip1_vtlbpool1" value="level1"/>
      <param type="backing" xref="lwip1_vtlbpool2" value="level2"/>
      <param type="backing" xref="lwip1_vtlbpool3" value="level3"/>
      <param type="backing" xref="lwip1_vtlbpool4" value="level4"/>
    </vdev>
    <address_space type="mmu" format="arm64:g4io40n"
              base="[0]=0x1008a000">
      <map xref="lwip1_main" base="0x40000000" flags="rwx"/>
      <map xref="cca0" base="0xfee00000" flags="rs"/>
      <map xref="cca1" base="0xfef00000" flags="rws"/>
    </address_space>
    <init arch_page="lwip1_arch">
      <copy xref="lwip_image" dref="lwip1_main" offset="0x00000"/>
      <cap type="ipc" target_xref="linux1" param="0x76"/>
    </init>
    <entry bp_xref="lwip1_main" bp_offset="0x10000"/>
    <sched class="wfq"/>
  </guest>
  <files>
    <file id="linux_kernel" href=".../Image" offset="0x0"
              size="0x895200"/>
    <file id="linux_ramdisk" href=".../ramdisk/arm64.sq"
              offset="0x895200" size="0x199000"/>
    <file id="linux_dtb" href=".../hi6220new.dtb" offset="0xa2e200"
              size="0x7ce3"/>
    <file id="lwip_image" href=".../lwip" offset="0xa35ee3"
              size="0x2be10"/>
  </files>
</scenario>
```

Listing A.4: ARMv8 HiKey sample XML configuration (completed)

A.3. Proof Example Trace

The following traces have been taken from the full exploration of the runtime path class as defined in Section 6.1. The output for each trace shows the initial definition of a combination of input space variables, the ELF function symbols encountered during the simulation, i. e. which C and assembly functions were entered, the list of accumulated path constraints Φ (those which make up the initial input state are labelled "IN", and later ones are identified as tainted or not by the annotation "T"), several statistics, and validation of the invariants. Note that one invariant for the exception link register ELR_EL2 is reported as violated; this is inevitable as we simultaneously assert the two conflicting constraints $ELR\_EL2_n == ELR\_EL2_0$ and $ELR\_EL2_n == ELR\_EL2_0 + 4$. The former applies for asynchronous events, where the interrupted instruction in the guest must be retried after handling the event, and architectural traps which have already advanced the link register before the event is delivered, such as the hypercall instruction `HVC`. The latter applies for events where the hypervisor is emulating an instruction, e. g. for device emulation. A production-grade proof system would combine the type of simulated event with the correct invariant for ELR_EL2_n. This has only implications for the faithfulness of our virtualized environment though; the integrity properties we show are not affected.

```
================================================================================
0xf407400c: 0x00000000
(1, 0, 0, [1])
(1, [1])
['LLarmed: (M64_f400d050__1 == 0x1)',
 'LLexpired: (R_cntpct_el0__0 == (M64_f400d048__0 + 0x3e8))',
 'LLh: (M64_f400d060__1 == 0xf400d028)',
 'LLnxt: (M64_f400d028__1 == 0x0)']
(2, 0, 0, [1, 2])
(1, [False, 1])
['LLschedstate: (M64_f40130a0__1 == 0x1)',
 'LLschedstate: (M64_f4013148__1 == 0x1)',
 'LLcurr: (M64_f400d010__1 == 0xf4013088)',
 'LLcurr: (M64_f400d008__1 == 0xf4013008)',
 'LLh: (M64_f400d018__1 == 0xf4013130)',
 'LLt: (M64_f400d020__1 == 0xf4013130)',
 'LLnxt: (M64_f40130a8__1 == 0x0)',
 'LLnxt: (M64_f4013150__1 == 0x0)']
================================================================================
Initially constrained vars: ['M32_f407400c__1', 'M64_f400d050__1',
 'R_cntpct_el0__0', 'M64_f400d048__0', 'M64_f400d060__1',
 'M64_f400d028__1', 'M64_f40130a0__1', 'M64_f4013148__1',
 'M64_f400d010__1', 'M64_f400d008__1', 'M64_f400d018__1',
 'M64_f400d020__1', 'M64_f40130a8__1', 'M64_f4013150__1']
### 0xf4008480 == vbar_irq_lel64
### 0xf4000d6c == interrupt_handler
### 0xf4000c6c == irq_get_irq_raw
### 0xf4000c20 == core_memarea
### 0xf4005564 == irq_gic_get_irq_raw
### 0xf4005378 == mmio_read32
### 0xf400534c == mmio_write32
### 0xf4000c88 == irq_get_irq
### 0xf4000cc8 == irq_disable_irq
### 0xf4000c20 == core_memarea
### 0xf40055f0 == irq_gic_disable_irq
### 0xf400534c == mmio_write32
```

```
### 0xf4000ca0 == irq_ack_irq
### 0xf4000c20 == core_memarea
### 0xf40055c0 == irq_gic_ack_irq
### 0xf400534c == mmio_write32
### 0xf4000b7c == upcall
### 0xf40005b4 == emulate_irq_activate_highest_interrupt
### 0xf40043b8 == emulate_irq_gicve_activate_highest_interrupt
### 0xf400385c == core_memarea
### 0xf400383c == mmio_read32
### 0xf4006098 == vm_cpu_upcall
Exit: upcall at f4006100 after 390 instructions
Complexity: 566vx/231vl//28c/6tc
Invariant violated: ((R_elr_el2__0 + 0x4) == R_elr_el2__1)
Inv: walking LL, HD f4013130 TL f4013130
Elem: f4013130
Inv: walking LL, HD f400d028 TL 0
Elem: f400d028
Invariants checked: 79spec/0glbl
Accumulated path constraints:
[01] INT {--------} (0x0 == 0x0)
[02]     {--------} (0x1 == 0x1)
[03]     {--------} (R_cntpct_el0__0 == (M64_f400d048__0 + 0x3e8))
[04] IN  {--------} (0xf400d028 == 0xf400d028)
[05] IN  {--------} (0x0 == 0x0)
[06] IN  {--------} (0x1 == 0x1)
[07] IN  {--------} (0x1 == 0x1)
[08] IN  {--------} (0xf4013088 == 0xf4013088)
[09] IN  {--------} (0xf4013008 == 0xf4013008)
[10] IN  {--------} (0xf4013130 == 0xf4013130)
[11] IN  {--------} (0xf4013130 == 0xf4013130)
[12] IN  {--------} (0x0 == 0x0)
[13] IN  {--------} (0x0 == 0x0)
[14]   T {f4005590} !(((0x0 - 0x3ff)[s>64]) == 0x0)
[15]   T {f4000d84} !(((0x0 + 0x1)[u>64]) == 0x0)
[16]   T {f4000da0} ((((0x0 & 0x3ff) - 0x3ff)[s>64]) s<= 0x0)
[17]   T {f4000dd0} ((0x0 - 0x0) == 0x0)
[18]   T {f4000dfc} ((0x0 - 0x0) == 0x0)
[19]     {f4000b94} !((0xf4013008 - 0x0) == 0x0)
[20]     {f4000bb0} ((((0x1[31:0]) - 0x1)[s>64]) == 0x0)
[21]     {f4000bc8} (((0x0 - 0x0)[s>64]) == 0x0)
[22]     {f40005cc} !((0xf4012300 - 0x0) == 0x0)
[23]     {f40005f0} !(((0x50012 - (0x2 | 0x50000))[s>64]) == 0x0)
[24]     {f4000600} (((0x50012 - (0x12 | 0x50000))[s>64]) == 0x0)
[25]     {f40043e8} !(((0x0 - 0x3ff)[s>64]) == 0x0)
[26]     {f400440c} !(((0x0 - 0x3ff)[s>64]) == 0x0)
[27]     {f4004430} (((0x0 - 0x0)[s>64]) == 0x0)
[28]     {f4000bf8} (((0xffffffff + 0x1)[u>64]) == 0x0)
Modified memory cells:
['f400d058', 'f4013010', 'f4013068', 'f40130a0', 'f4013164',
 'f4073180', 'f4074010', 'f4075000', 'f4076030']
================================================================
```

Listing A.5: Sample prover path exploration trace

B. Open Source Repositories

Our full code base is available under an open source license. The various subprojects that our endeavour comprises can be found at the following umbrella URL, together with additional information on the necessary steps to replicate different kinds of running setups:

http://phidias-hypervisor.de/

At the time of writing, the following repositories are part of our project, ordered by their appearance in this thesis. Each of these can be cloned freely by passing the repository URL to `git clone`:

- **http://phidias-hypervisor.de/repos/core.git**
  PHIDIAS, the hypervisor.
- **http://phidias-hypervisor.de/repos/xml.git**
  SCHISM, the configuration framework. Also contains the XML library of supported architectures and platforms.
- **http://phidias-hypervisor.de/repos/abi.git**
  Glue layer between PHIDIAS and paravirtualized VMs. Specifies the trap mechanism and hypercall numbers understood by PHIDIAS.
- **http://phidias-hypervisor.de/repos/symex.git**
  Symbolic execution engine used for the integrity proof.
- **http://phidias-hypervisor.de/repos/c10r.git**
  The "configurator", a nascent additional GUI tool for developers who want to create new or comfortably modify existing scenario specifications. Tools like this may be helpful in countering the complexity growth for developers we have discussed in Subsection 4.7.2.

C. Glossary

GIC The Generic Interrupt Controller is the standard interrupt controller on ARM platforms. It handles both distribution of peripheral interrupts and dispatch of interrupts to processors and therefore unites the functionality of LAPIC and IOAPIC, except for the presence of a timer.

IOAPIC The I/O Advanced Programmable Interrupt Controller is an interrupt controller on x86 platforms that relays and distributes interrupts from peripheral devices to configured LAPICs.

LAPIC The Local Advanced Programmable Interrupt Controller is the standard processor-local interrupt controller on x86 platforms. It handles interrupt dispatch to the processor and also contains an internal timer which can be configured as a local interrupt source.

MMIO Memory-Mapped Input/Output areas are ranges of physical memory addresses that are not backed by actual memory and thus handled by the memory controller, but which are claimed by peripheral devices instead. Reads and writes to these addresses cause an interaction with the device and can result in the device changing its state or other side effects beyond the initial memory bus transaction.

MMU The Memory Management Unit is a processor component which transparently translates virtual to physical (i. e. bus) addresses by walking the currently active page table. In order to speed up the translation process, the MMU usually contains one or several TLBs.

MPU The Memory Protection Unit is a processor component that only performs permission checks for attempted memory accesses, but does not offer any form of address translation. On systems with an MPU, virtual and physical addresses are therefore equivalent, if no other form of translation is applied.

Page table Pagetables are tree-like structures in memory that define the desired address translation from virtual to physical (i. e. bus) addresses. The control software (hypervisor, operating system kernel) programs an architecturally defined base register (e. g. `CR3`, `TTBR0`) with the address of the root of the page table tree. The MMU uses this register to walk the tree in order to obtain specific translation results.

RCU Read-copy-update is a synchronization mechanism that circumvents taking a spinlock by temporarily keeping both the old and new version of the changed data structure, thus allowing readers in the critical section to complete their operation undisturbedly. As soon as all readers are guaranteed to have left the critical section, which could be implemented by waiting for every CPU to acknowledge the next timer interrupt, the old version is discarded.

SLAT Second-Level Address Translation is an extension to the page table walk performed by the MMU. If SLAT is active, the MMU performs two translations, first from guest-virtual to guest-physical addresses, customarily using a page table under control of the guest OS, and then to host-physical addresses using a second, hypervisor-controlled page table. If SLAT (also called Stage-2 Paging or Nested Paging) is not available, a hypervisor has to implement a VTLB to perform memory virtualization.

TCB The Trusted Computing Base is the collection of code that a software component has to assume as trustworthy. In system setups with monolithic kernels the TCB amounts to millions of lines of source code, whereas in microkernel-based setups the TCB only comprises of the microkernel itself and the user-level servers whose functionality the component relies on.

TLB The Translation Lookaside Buffer is a cache inside the MMU that stores full or partial memory address translations. Some architectures fill the TLB transparently as a side effect of successful page table walks (x86, x86_64, ARM), while others like MIPS require the TLB to be filled by software.

VCPU The Virtual CPU is the interface a faithfully virtualizing hypervisor presents to its virtual machines. It resembles the physical CPU as close as possible. The key difference to a thread, the classical scheduling entity in an operating system, is that a VCPU replicates all privilege levels offered by the physical CPU, whereas a thread context only contains the unprivileged general purpose register file (and possibly additional special-purpose registers which are accessible in unprivileged mode, e. g. the floating-point unit).

vtable Virtual function tables are a standard form of how compilers for polymorphic languages implement dynamic function dispatch. The compiler generates one vtable for each polymorphic class, and instantiated objects carry a pointer to the vtable of their respective class, which allows the correct function to be called even if the object is referenced through a pointer of a more generic type.

VTLB The Virtual TLB is a software component of the hypervisor that reads guest-virtual to guest-physical translation entries from a guest page table and creates corresponding guest-virtual to host-physical addresses in a shadow page table which is under hypervisor control. This is required if the MMU does not support SLAT, as a virtual machine could otherwise easily escape its memory confinement if its page table was directly used by the MMU.

D. List of Figures

E. List of Tables

F. Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Summer Conference Proceedings 1986*, volume 4, pages 64–75. USENIX Association, 1986.

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices*, 41(11):2–13, 2006.

[3] Advanced Micro Devices. AMD64 architecture programmer's manual volume 2: system programming. `http://support.amd.com/TechDocs/24593.pdf`, 2016.

[4] E. Alkassar, E. Cohen, M. Hillebrand, M. Kovalev, and W. J. Paul. Verifying shadow page table algorithms. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, pages 267–270. FMCAD Inc, 2010.

[5] E. Alkassar, W. J. Paul, A. Starostin, and A. Tsyban. Pervasive verification of an OS microkernel. In *International Conference on Verified Software: Theories, Tools, and Experiments*, pages 71–85. Springer, 2010.

[6] ARM Holdings. Arm architecture reference manual ARMv7-A and ARMv7-R edition. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/`, 2014.

[7] ARM Holdings. ARMv8-A reference manual (Issue A.k). `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487a.k_10775/index.html`, 2016.

[8] ARM Holdings. ARMv8.1 reference manual supplement. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0557a.b/index.html`, 2016.

[9] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1083–1094. ACM, 2014.

[10] 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. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[11] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The VMware mobile virtualization platform: is that a hypervisor in your pocket? *ACM SIGOPS Operating Systems Review*, 44(4):124–135, 2010.

[12] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB standard: version 2.5. `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.5-r2015-06-28.pdf`, 2016.

[13] B. Bartels and S. Glesner. Formal modeling and verification of low-level software programs. In *2010 10th International Conference on Quality Software*, pages 200–207. IEEE, 2010.

[14] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[15] C. Baumann, T. Bormer, H. Blasum, and S. Tverdyshev. Proving memory separation in a microkernel by code level verification. In *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, pages 25–32. IEEE, 2011.

[16] C. Baumann, M. Dam, V. Do, C. Gehrmann, R. Guanciale, N. Khakpour, H. Nemati, O. Schwarz, and A. Vahidi. Verifying a security hypervisor. `http://www.vinnova.se/PageFiles/751327324/A10%20SSF%20PROSPER%20poster.pdf`, 2016.

[17] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul. Putting it all together–formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(4):411–430, 2006.

[18] A. Blanchard, N. Kosmatov, M. Lemerre, and F. Loulergue. A case study on formal verification of the Anaxagoros hypervisor paging system with Frama-C. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 15–30. Springer, 2015.

[19] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing SLA violations. In *10th IFIP/IEEE International Symposium on Integrated Network Management (IM'07)*, pages 119–128. IEEE, 2007.

[20] R. Buhren, J. Vetter, and J. Nordholz. The threat of virtualization: hypervisor-based rootkits on the ARM architecture. In *Information and Communications Security*, pages 376–391. Springer, 2016.

[21] E. Carrascosa, J. Coronel, M. Masmano, P. Balbastre, and A. Crespo. XtratuM hypervisor redesign for LEON4 multicore processor. *SIGBED Rev.*, 11(2):27–31, 2014.

[22] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller. Quantifying the information leak in cache attacks through symbolic execution. *CoRR*, abs/1611.04426, 2016.

[23] L. Cherkasova, D. Gupta, and A. Vahdat. When virtual is harder than real: resource allocation challenges in virtual machine based IT environments. *Hewlett Packard Laboratories, Tech. Rep. HPL-2007-25*, 2007.

[24] A. Crespo, I. Ripoll, and M. Masmano. Partitioned embedded architecture based on hypervisor: The XtratuM approach. In *Dependable Computing Conference (EDCC), 2010 European*, pages 67–72. IEEE, 2010.

[25] C. Dall, S.-W. Li, J. T. Lim, J. Nieh, and G. Koloventzos. ARM virtualization: performance and architectural implications. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 304–316. IEEE, 2016.

[26] C. Dall and J. Nieh. KVM/ARM: the design and implementation of the Linux ARM hypervisor. *ACM SIGARCH Computer Architecture News*, 42(1):333–348, 2014.

[27] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 223–234. ACM, 2013.

[28] M. Dam, R. Guanciale, and H. Nemati. Machine code verification of a tiny ARM hypervisor. In *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, pages 3–12. ACM, 2013.

[29] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[30] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: narrowing the semantic gap in virtual machine introspection. In *IEEE Symposium on Security and Privacy (SP)*, pages 297–312. IEEE, 2011.

[31] A. Dunkels. Design and implementation of the lwIP TCP/IP stack. *Swedish Institute of Computer Science*, 2:77, 2001.

[32] K. Elphinstone and G. Heiser. From L3 to seL4: what have we learnt in 20 years of L4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 133–150. ACM, 2013.

[33] fent Innovative Software Solutions. XtratuM GPL Edition. `http://www.fentiss.com/en/rdi/downloads.html`, 2014.

[34] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *ACM SIGPLAN Notices*, volume 51, pages 608–621. ACM, 2016.

[35] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and execution models in the Fluke kernel. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 101–115. USENIX Association, 1999.

[36] A. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *International Conference on Interactive Theorem Proving*, pages 243–258. Springer, 2010.

[37] J. Franklin, S. Chaki, A. Datta, J. M. McCune, and A. Vasudevan. Parametric verification of address space separation. In *Principles of Security and Trust*, pages 51–68. Springer, 2012.

[38] J. Gandhi, M. D. Hill, and M. M. Swift. Agile paging: exceeding the best of nested and shadow paging. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 707–718, 2016.

[39] K. Gilles, S. Groesbrink, D. Baldin, and T. Kerstan. Proteus hypervisor: full virtualization and paravirtualization for multi-core embedded systems. In *Embedded Systems: Design, Analysis and Verification*, pages 293–305. Springer, 2013.

[40] S. Groesbrink, L. Almeida, M. de Sousa, and S. M. Petters. Towards certifiable adaptive reservations for hypervisor-based virtualization. In *Real-Time and*

Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th, pages 13–24. IEEE, 2014.

[41] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. J. Magenheimer. Are virtual machine monitors microkernels done right? In *HotOS*, 2005.

[42] T. Hansen, P. Schachte, and H. Søndergaard. State joining and splitting for the symbolic execution of binaries. In *International Workshop on Runtime Verification*, pages 76–92. Springer, 2009.

[43] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. *The performance of µ-kernel-based systems*, volume 31. ACM, 1997.

[44] G. Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, pages 11–16. ACM, 2008.

[45] G. Heiser and B. Leslie. The OKL4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of the first ACM Asia-Pacific Workshop on Systems*, pages 19–24. ACM, 2010.

[46] G. Heiser, V. Uhlig, and J. LeVasseur. Are virtual-machine monitors microkernels done right? *ACM SIGOPS Operating Systems Review*, 40(1):95–99, 2006.

[47] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim. Xen on ARM: system virtualization using Xen hypervisor for ARM-based secure mobile phones. In *5th IEEE Consumer Communications and Networking Conference (CCNC)*, pages 257–261. IEEE, 2008.

[48] Imagination Technologies. MIPS virtualization. `https://www.imgtec.com/mips/architectures/virtualization/`, 2013.

[49] Intel Corporation. Intel 64 and IA-32 architectures software developer's manuals. `https://software.intel.com/en-us/articles/intel-sdm`, 2016.

[50] R. Jones et al. NetPerf: a network performance benchmark, 1996.

[51] R. Kaiser and S. Wagner. Evolution of the PikeOS microkernel. In *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems*, pages 50–57, 2007.

[52] N. A. Kamble, J. Nakajima, and A. K. Mallick. Evolution in kernel debugging using hardware virtualization with Xen. In *Linux Symposium*, page 1, 2006.

[53] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: rethinking kernel isolation. In *USENIX Security*, pages 957–972, 2014.

[54] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: lightweight kernel protection against return-to-user attacks. In *USENIX Security Symposium*, pages 459–474, 2012.

[55] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[56] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[57] G. Klein. Operating system verification—an overview. *Sadhana*, 34(1):27–69, 2009.

[58] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.

[59] M. Kleine, B. Bartels, T. Göthel, S. Helke, and D. Prenzel. LLVM2CSP: extracting CSP models from concurrent programs. In *NASA Formal Methods Symposium*, pages 500–505. Springer, 2011.

[60] A. Koelbl and C. Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming*, 33(6):645–666, 2005.

[61] J. Liedtke. Improving IPC by kernel design. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 175–188. ACM, 1994.

[62] J. Liedtke. *On micro-kernel construction*, volume 29. ACM, 1995.

[63] J. Liedtke, K. Elphinstone, S. Schonberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved IPC performance (still the foundation for extensibility). In *The 6th Workshop on Hot Topics in Operating Systems*, pages 28–31. IEEE, 1997.

[64] M. Masmano, I. Ripoll, A. Crespo, and J. Metge. XtratuM: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, pages 263–272. Citeseer, 2009.

[65] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 13–23. ACM, 2005.

[66] S. Montenegro, H.-P. Röser, and F. Huber. BOSS: software and FPGA middleware for the "flying laptop" microsatellite. In *DASIA – Data Systems in Aerospace*, volume 602, 2005.

[67] C. Moratelli, S. J. Filho, M. V. Neves, and F. Hessel. Embedded virtualization for the design of secure IoT applications. In *2016 International Symposium on Rapid System Prototyping (RSP)*. IEEE, 2016.

[68] C. Moratelli, S. Johann, and F. Hessel. Exploring embedded systems virtualization using MIPS virtualization module. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 214–221. ACM, 2016.

[69] C. Mulliner, S. Liebergeld, and M. Lange. Poster: HoneyDroid—creating a smartphone honeypot. In *IEEE Symposium on Security and Privacy*, volume 2, 2011.

[70] M. O. Myreen, M. J. Gordon, and K. Slind. Machine-code verification for multiple architectures: an application of decompilation into logic. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, page 20. IEEE, 2008.

[71] M. O. Myreen, M. J. Gordon, and K. Slind. Decompilation into logic—improved. In *Proceedings of the 2012 International Conference on Formal Methods in Computer-Aided Design*, pages 78–81. IEEE, 2012.

[72] J. Nordholz, J. Vetter, M. Peter, M. Junker-Petschick, and J. Danisevskis. XN-Pro: low-impact hypervisor-based execution prevention on ARM. In *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices*, pages 55–64. ACM, 2015.

[73] OASIS Virtual I/O Device Technical Committee. Virtual I/O Device (VIRTIO) Version 1.0. `http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.pdf`, 2016.

[74] OpenSynergy GmbH. OpenSynergy COQOS SDK. `http://www.opensynergy.com/fileadmin/user_upload/Datenblaetter/COQOS-Datasheet.pdf`, 2017.

[75] Operating Systems group, Technische Universität Dresden. Fiasco.OC microkernel. `http://os.inf.tu-dresden.de/fiasco`, 2016.

[76] J. K. Ousterhout et al. Scheduling techniques for concurrent systems. In *ICDCS*, volume 82, pages 22–30, 1982.

[77] M. Peter, M. Petschick, J. Vetter, J. Nordholz, J. Danisevskis, and J.-P. Seifert. Undermining isolation through covert channels in the Fiasco.OC microkernel. In *Information Sciences and Systems 2015*, pages 147–156. Springer, 2016.

[78] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[79] D. A. Ramos and D. Engler. Under-constrained symbolic execution: correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.

[80] J. Rushby. *Noninterference, transitivity, and channel-control security policies.* SRI International, Computer Science Laboratory, 1992.

[81] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[82] J. Rutkowska and A. Tereshkin. Bluepilling the Xen hypervisor. *Black Hat USA*, 2008.

[83] D. Sanán, A. Butterfield, and M. Hinchey. Separation kernel verification: the XtratuM case study. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 133–149. Springer, 2014.

[84] N. Schirmer. A verification environment for sequential imperative programs in isabelle/hol. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 398–414. Springer, 2005.

[85] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and privacy (SP), 2010 IEEE symposium on*, pages 317–331. IEEE, 2010.

[86] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. *ACM SIGPLAN Notices*, 48(6):471–482, 2013.

[87] J. S. Shapiro, J. M. Smith, and D. J. Farber. *EROS: a fast capability system*, volume 33. ACM, 1999.

[88] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[89] S. Trujillo, A. Crespo, and A. Alonso. Multipartes: multicore virtualization for mixed-criticality systems. In *2013 Euromicro Conference on Digital System Design (DSD)*, pages 260–265. IEEE, 2013.

[90] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Virtual Machine Research and Technology Symposium*, pages 43–56. Citeseer, 2004.

[91] Universität Karlsruhe. L4 API (X.2 rev 6). `http://l4hq.org/docs/manuals/l4-x2-r6.pdf`, 2007.

[92] S. H. VanderLeest. ARINC 653 hypervisor. In *29th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, pages 5.E.2–1–20. IEEE, 2010.

[93] A. Vaynberg and Z. Shao. Compositional verification of a baby virtual memory manager. In *International Conference on Certified Programs and Proofs*, pages 143–159. Springer, 2012.

[94] J. Vetter, M. Junker-Petschick, J. Nordholz, M. Peter, and J. Danisevskis. Uncloaking rootkits on mobile devices with a hypervisor-based detector. In *International Conference on Information Security and Cryptology*, pages 262–277. Springer, 2015.

[95] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li. Selective hardware/software memory virtualization. *ACM SIGPLAN Notices*, 46(7):217–226, 2011.

[96] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A practical verification framework for preemptive OS kernels. In *Proceedings of the 28th Conference on Computer Aided Verification*, 2016.

[97] Y. Zhang. HackBench benchmark suite. `http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c`, 2008.