Towards Modular Security-conscious Virtual Machines

Steffen Liebergeld, Michael Peter
Technische Universität Berlin
Deutsche Telekom Laboratories
Security in Telecommunications
{steffen, peter}@sec.t-labs.tu-berlin.de

Adam Lackorzynski
Technische Universität Dresden
Department of Computer Science
Operating Systems Group
adam@os.inf.tu-dresden.de

Abstract

Many system-level concerns such as security and support for real-time workloads are hard to address in existing systems, especially if one of the main platform assets is backward compatibility. Apart from many other applications, virtualization has proven capable in running legacy software. If done right, it may relieve systems developers from the need to stay backward compatible and allows them to introduce more intrusive architectural changes.

Aside from their merits, virtual machines (VMs) also introduce additional complexity into the software stack, which enlarges the attack surface and is detrimental to security goals. In this paper, we argue that security can indeed be improved by using virtual machines if their construction follows the principle of incremental complexity growth. That is, functionality should only be included in the trusted computing base of a component if the benefits due to its utility outgrow the drawbacks due to the larger risk for introduced bugs and errors. Specifically, we apply that principle to the virtual machine monitor (VMM), a critical component needed to run VMs.

The contribution of this paper is the demonstration that such a security-oriented architecture can be implemented efficiently on top of a real-time capable microkernel. We demonstrate that high throughput and good real-time performance are achievable on one system.

1 Introduction

The size of contemporary commodity operating systems such as Linux, makes it hard to evolve their system architecture and to add new features because changes in core components affect too many other parts. For example, the replacement of the access-control list based security schemes, which are entangled with the file API, with capabilities has often been fancied, yet has never materialized. Backward-compatible extensions are plagued with substantial complexity. For example, much effort has been expended to improve the real-time characteristics of the Linux kernel, still these changes have not found their way into the mainline kernel yet.

Microkernel-based systems instead open up the opportunity to evolve a system architecture gradually: In monolithic systems, core components reside inside the operating system kernel, and often have complex interdependencies. In a microkernel-based system, these components run at user-level, and all communication is done explicitly via inter-process communication (IPC). We believe that this leads to well defined interfaces, and less complex component interdependencies. Obviously, a well-structured system is more amendable to system evolution.

Architecturally, user-level placement is also preferable because isolation is enforced in hardware with the use of address spaces. This allows a system of small components to be composed in such a way that for each component the amount of code that has to be trusted is minimized [7]. But even if fine-grained decomposition is not outright possible, the overall security properties of a system can be improved even if legacy components are not modified but certain pattern of interactions are imposed [10].

1.1 Backward Compatibility

A realistic assessment of the dynamics that govern the development of software yields the conclusion that no new system can afford not to use commodity software. Porting each application to a new operating system is laborious and many application developers will not keep pace. For innovative systems, the lack of applications is a serious drawback. An alternative approach is to port whole operating systems. Although neither porting an operating system to, say, a microkernel is a non-trivial undertaking nor is the development of a virtual machine, it comes with the advantage that no modifications to a possibly large number of applications are necessary.

In the past, many commodity architectures did not meet the requirements for efficient system virtualization [19] with IA32 being the most prominent example [20]. As a result, the encapsulation of operating systems did either require guest adaptions [15][1][4] or introduce the substantial complexity of binary rewriting [2]. The advent of virtualization extensions in commodity processors facilitates VMs that neither require guest changes nor are stricken with overly complex implementations.

1.2 Complexity

One of the often cited merits of VMs is a new level of isolation, which is guaranteed by the virtualization infrastructure. Obviously, as more workload has been migrated into virtual machines, the virtualization infrastructure became a new attack vector. We observe, that current virtual machine monitors (VMMs) such as Xen are often prone to attacks [13][25]. We attribute this to two shortcomings in their monolithic system architecture. First, whenever new features are introduced, the complexity of the virtual VMM increases. Given that there is no means of isolation between VMM components, these new features also increase the likelyhood of critical vulnerabilities. Secondly, in these systems one VMM drives many VMs. If the VMM is compromised, all of its VMs have fallen prey to the attacker.

We instead lift the VMM on a microkernel with its reliable isolation mechanisms. By making the relation between VMM and VM configurable, we make the risks introduced by running a virtualized workload manageable. Each user can settle on a solution on a per use case basis that best meets his specific requirement regarding functionality and security. If a machine is to host multiple workloads, then these decisions can be made independently.

In this paper, we make the case for specialized VMMs suiting a particular use case. A security-concerned guest is likely to accept kernel modifications in return for a smaller attack surface of its supporting infrastructure. That requirement is met by a small VMM called *Karma*[1], which favors small size over extensive functionality. To that end it makes systematic use of device para-virtualization. The price for these feats is a non-backward-compatible machine interface.

In a second step we use the para-virtualized system to leverage an existing and widely used VMM that excels with faithful virtualization. We call this approach *staged virtualization*. With this technology we support a much wider range of guests while delegating the effort of implementing and maintaining faithful virtualization. Encapsulating the VMM has the advantage that it can be kept out of the TCB of secure applications to a large extend. This compares favorably to the original setup were sizeable parts were implemented as kernel module for performance reasons.

1.3 Outline

The paper is structured as follows: Section 2 starts with briefly revisiting the challenges that have to be overcome to run an operating system encapsulated and how virtualization aids that goal. Before presenting the VMM's design in Section 3, we derive our design goals from general microkernel principles. Afterwards, we shortly discuss the implementation of our VMM (Section 4). In Section 5 we discuss how staged virtualization is implemented. Our solution is evaluated in Section 6. Related work is discussed in Section 7.

2 Background

Multiserver systems built on top of microkernels have long been acknowledged as the architecture of choice for building secure systems because they foster the principle of least authority at system level. Briefly, the underlying idea is to place functionality that has accumulated in monolithic kernels into dedicated protection domains. A crash in the network stack, for example, which would have been fatal before, is

[1]Karma is the concept of "action" or "deed", understood as that which causes the entire cycle of cause and effect.

isolated and becomes survivable [9][24].

Although such a multiserver system seems to be the ideal solution to a range of problems, we admit that implementing such a system from scratch and bringing it to a state that matches the functionality of today's commodity systems is not feasible. We argue that we can leverage the functionality of commodity operating systems while retaining the security and safety properties of the microkernel system, by running them in an isolated container.

2.1 Operating System Rehosting

As illustrated in Figure 1 (a), today's commodity operating systems employ two CPU modes[2], the most privileged mode for the kernel (kernel mode), and the least privileged mode for its applications (user mode). Porting such an operating system on a microkernel requires an additional privilege level that allows the microkernel to control the commodity kernel while retaining its structure: kernel and user mode (Figure 1 (c)).

As common-off-the-shelf hardware like x86 does not support such an additional privilege level, the port of a commodity kernel requires intrusive modifications: Its applications have to be mapped to microkernel tasks, and the kernel itself has to be deprivileged and run in its own address space [8].

Figure 1 (b) illustrates that architecture. The cost for invoking a kernel service grow from one state transition to two state transitions and a context switch (see Figure 1 (a)). While the increased number of state transitions is a performance impediment by itself, the induced secondary costs such as the increased TLB footprint carry even more weight. The repercussions depend on the workload and range from negligible to severe.

2.2 Virtualization

The rather intrusive operating system rehosting laid out above was only necessary because many widespread instruction set architectures are not efficiently virtualizable. For x86 this changed with the recent introduction of virtualization extensions (AMD-V [3] for AMD and VT-x [11] for Intel). A virtual machine comes with the advantage that the guest kernel does not need modifications. It is provided with a virtual CPU, which behaves like a physical one[3]. The virtualization extensions allow the host to control the execution of the guest, and thereby enforce its own scheduling decisions and memory management.

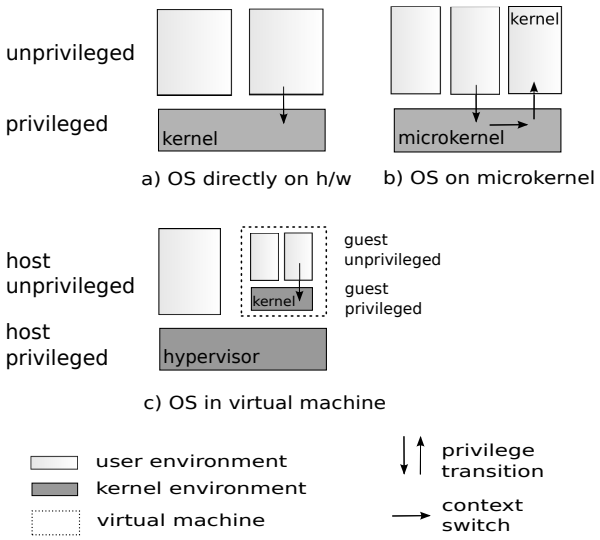

FIGURE 1: *A system running directly on hardware, on top of a microkernel and inside a virtual machine.*

In the microkernel scenario, a VM not only elliminates the need to adapt commodity operating systems to a completely different execution environment, it also yields better performance compared to rehosting [18]. When a guest executes in a VM (Figure 1 (c)), its system activities such as state transitions and page table updates can run directly on hardware without involving a microkernel[4]. Contrary to operating system rehosting, this avoids additional state transitions and their induced overhead (TLB misses).

2.3 Terminology

After years of inconsistent use, the meaning of some terms has become blurry. To avoid further confusion, we will now define our notion.

A *hypervisor* enforces isolation, that is it implements protection domains, VMs being one flavor of them. For doing so, it needs to run in the most privileged CPU execution mode. A

[2]Even though some processors offer more privilege levels, these are often quite specialized and of little value for general purpose operating systems. For example, two of the four rings of the IA32 ISA are only meaningful when used with the arcane segmentation which renders them useless for many modern operating systems.

[3]The behavioral equivalence does not cover the virtualization extensions themselves. Any attempt to use them will be intercepted.

[4]The processor needs to feature *nested paging*.

microkernel applies the principle of minimality to the portion of software running in the most privileged execution mode. We understand microkernel and hypervisor complementary. *Hypervisor* implies a function whereas *microkernel* denotes a particular design principle. As such, a microkernel can assume the role of a hypervisor, or conversely, a hypervisor can be implemented as microkernel.

A *virtual machine monitor* provides VMs with functionality beyond CPU and memory virtualization, which is the duty of the hypervisor. Typically, the VMM supplies a VM with (virtual) devices and coordinates its execution. Contrary to a hypervisor, which always requires maximal CPU privileges, a VMM can be either part of the kernel or implemented as a user task.

We distinguish between two forms of virtualization: In *full* or *faithful* virtualization, a guest is presented with a complete duplicate of a physical machine. In *para-virtualization* instead, the VM presents a modified machine interface. Under faithful virtualization, a VM can run unmodified guest operating systems, whereas para-virtualization requires modifications to the guest operating systems.

In para-virtualization, a guest may invoke services on the VMM. Such service invokations are called *hypercalls* and can be initiated via a special *vmmcall* instruction. They work similar to system calls in traditional operating systems.

3 Design

Before we present our design, we state which implications of hosting a VM are tolerable and which are not. First, hosting VMs should not put unrelated secure applications that run on the same machine at risk. Second, a guest that can withstand attacks aimed at itself shall not become vulnerable to an attacker who instead redirects his efforts on the VM infrastructure. Accordingly, the hypervisor and VMM should exhibit the same invulnerability against external attacks as physical hardware. Yet, we do not strive to counteract attacks that the guest was not able to thwart itself when running on the physical machine.

Figure 2 provides an overview of our design.

3.1 Host System

With *host system* we mean all parts of the system except for the VM and its supporting VMM. Invariably being part of the *trusted computing base* (TCB) of any application, the host kernel has special requirements in terms of security. It should be small so that inspection of its implementation can achieve a high degree of thoroughness. Therefore, we opted for a microkernel. Virtual machines have to be supported as a type of protection domain, which lets the microkernel assume the role of a hypervisor. As shown in previous research [18], VM hosting capabilities can be added in an non-intrusive way, so that security properties are not compromised. To implement the principle of least authority at system level, it is useful to have object-capabilities at the kernel level at hand [16]. Fiasco.OC [14], the microkernel we chose, features capability-based access control.

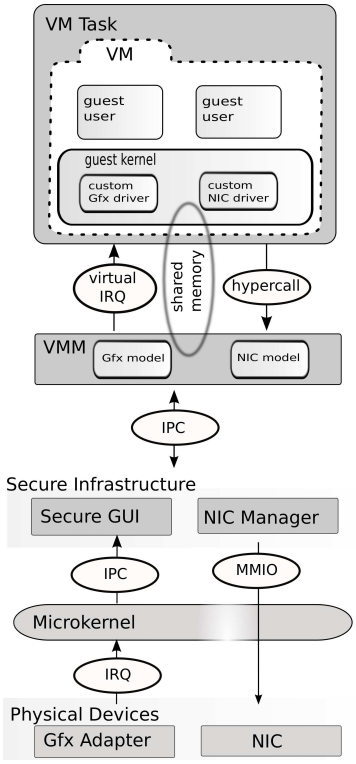

FIGURE 2: *Architecture of the Karma VMM.*

Contrary to monolithic kernels, device drivers run on top of the microkernel at user level. As any other task, they are encapsulated in address spaces. Endowed with the hardware access needed for device communication, they offer services of higher abstractions to the layers above. The exact nature of these services is device specific and may include

advanced features, e.g. bandwidth reservation. As long as IO-MMUs are not in widespread use, DMA operations are not contained by address space boundaries. As such, the device manager of a DMA-capable device has to be counted to the system-wide TCB.

We make two assumptions regarding the security properties of our hosting system: First, the microkernel enforces protection boundaries, in particular those of virtual machines. If there are agents that are able to circumvent these boundaries[5], then they are assumed to be trustworthy. Second, the system is resilient against peer attacks. A peer attack is an attempt to trick a peer, which provides a service, into an activity that violates the service contract. Usually such attacks draw on implementation flaws, e.g. missing argument checks. Such a rather strong assumption is warranted by the scope of this paper, which is on the security impact of VMs on their accommodating host system.

3.2 Virtual Machine Monitor

The architectural sketch gives a hint as to the role of the VMM. The two sides it interacts with–the VM above and the secure infrastructure below–use different means of communication. It falls to the VMM to serve as a protocol translator.

In our design, VMMs are implemented as user-level protection domains (tasks). They do not hold any special privileges since sensitive operations like switching the CPU into guest mode (world switch) are provided by the hypervisor. As a task, each VMM is subject to the regular capability-based communication control and the secure resource delegation mechanisms. Accordingly, the VMM does not pose a novel threat to other protection domains in the system as these have to withstand local attackers anyway. The actual VM is the least privileged component in the stack. It is under full control of the VMM which provides it with memory and takes care of the interaction with the other parts of the system.

CPU and memory virtualization are handled by CPU extensions. However, a VM also needs a range of devices, from platform devices - such as interrupt controllers and timers - to peripheral devices like network cards (NICs) and hard disks. This raises the question how such devices are to be implemented.

The interface used for device communication is geared towards hardware implementations. A logical request is split into a sequence of simple operations such as writing an individual device register. Such an interface is unfavorable in VMs as each individual device access normally causes a costly VM exit. In faithful virtualization, the VMM must decode the semantic content of an operation, e.g. the value that was to be written and the address that it was to be written to, before it feeds these values into its device model.

To mitigate that problem, we introduced a higher-level interface between the VM and the VMM. In many cases, the device drivers implementing the custom interface have a much lower complexity than their counterparts implementing the device interface. For example, a sequence of low level operations such as register access for a particular disk controller to request a block on a logical volume is replaced with one service invocation to the VMM (hypercall).

Along that line of device para-virtualization, we further reduced the Karma VMM's size by omitting legacy functionality such as 16bit code and the BIOS emulation for early bootstrapping stages.

As para-virtualization requires adaptions to the guest operating system, the number of operating systems suited to run on the Karma VMM is limited. But even where source modifications are possible, it may be desirable to run guests unmodified, for example to avoid having to go through certification again. We will address this need with staged virtualization.

3.3 Attack Scenarios

The infrastructure needed for VMs adds new targets that an adversary can aim at. To assess the security situation it is necessary to understand what the risks for individual components are and which implications arise from successful attacks.

We would like to point out that we do not discuss the security situation of applications in a VM, which obviously also depends on the traits and qualities of the guest operating system in addition to the virtualization stack. It is far beyond the scope of this paper to examine operating systems in general. Our suggestion for security-concerned applications is to run as native application of a microkernel system, which was designed with that use case in mind.

[5] e.g. device managers that may initiate DMA

3.3.1 Attacks from within the VM

Direct attacks from within the VM to break its encapsulation are thwarted by the microkernel. By using secure mechanisms to construct protection domains including VMs, the kernel can always rely on correct machine-level data structures such as page tables. Together with specification compliant hardware, this ensures that a VM has only access to resources that were explicitly granted to it.

An attacker running with user privileges in the VM might try to gain control over the guest kernel by subverting the VMM. These efforts have little chances of success because the VMM hardly interacts with user level guest activities which minimizes the chances for a flawed implementation. In particular, a VMM does not accept device requests through its hypercall interface if they are issued with user privileges in the VM.

The situation is different for an attacker with guest kernel privileges. The para-virtualization interface is more involved which increases the odds of implementation defects in the VMM. Still, the best such an attacker can hope for is to subvert the VMM thereby turning it into a local attacker. The compromised VMM runs in a dedicated protection domain and can only access resources in other protection domains through capability controlled communication channels. This access is not problematic under the assumption that the VMM was only granted resources that cannot be used to affect secure applications in the first place.[6] As such, the risk for secure applications is not increased because, as stated above, one underlying assumption is that they can cope with such local attacks.

A malicious VM could try to gain control over its sibling VMs by taking over the shared VMM. To mitigate that risk, VMs that are to be separated do not share a VMM.

3.3.2 External Attacks

An external attacker who cannot defeat a guest directly could re-target on the VMM. Being part of the TCB of the VMs, a VMM that is compromised turns its VMs in as well. Fortunately, the attack vector on the VMM is limited because all external traffic has to pass through a device manager before it reaches the VMM. The device managers are responsible for secure device arbitration which may include low level protocol processing. For example, in the case of the NIC manager, that involves assembling fragmented packages and implementing NAT. The VMM itself is left with the rather simple task of forwarding packets to the VM without inspecting them. This can be done with a ring buffer, the simplicity of which gives little chances for implementation bugs.

As with VMMs defeated by its VMs, a VMM taken over by an external attacker does not pose a threat to secure applications running alongside as explained above.

3.3.3 Non-mitigated Attacks

As we are only interested in the risk changes due to the presence of a VMM, we are not concerned about attacks that do not involve the VMM. This class includes both internal and external attacks. The guest faces the same threats irrespective of whether it runs on a physical machine or in a VM.

In such a disregarded attack, a process in the VM might try to exploit a vulnerability in the guest kernel and gain control over sibling processes. The VMM could counteract these threats, e.g. by augmenting the machine model with additional access controls for kernel data. Such techniques are complementary to our solution and are an active field of research [5][22][17].

4 Implementation

The functionality that eventually makes up the virtual machine is provided by distinct parts of the system. The Karma VMM coordinates them. Its implementation can be roughly divided into the following three parts: core virtualization, system environment and peripheral devices. We will cover each of them in the following sections.

4.1 Core Virtualization

Virtualizing the CPU and memory relies on recent CPU extensions. The microkernel supports protection domains–*tasks* in our terminology–that can be used to manage the memory of a VM. Any memory bound to a domain associated with a VM is visible inside the VM as physical memory. Unlike threads, virtual CPUs are not active entities scheduled by the kernel. Instead, a virtual CPU

[6] Granting a VMM access, e.g., to page with confidential data of a secure application is considered a fault on part of the infrastructure.

resumes execution whenever a thread switches to it. During that switch, the thread provides the CPU state, i.e. the register contents, with the memory environment taken from the protection domain.

The center piece of a VMM is an event loop as shown in Figure 3. Before it transfers control into a VM, the VM thread checks for pending events. These events may occur when device managers signal the completion of device requests. Control is returned from the VM to its controlling thread when either a host interrupt occurs, the VM requests a service from the VMM[7], or a condition arises that cannot be handled by the VM.

```
VM_state state;

while(vm_active){
   if (virtual_irq_pending){
       set_virq_for_injection(&state);
   }
   syscall_vm_run(&state);
   if (request_pending(&state)){
       handle_reqest(&state);
   }
}
```

FIGURE 3: *Main event loop*

The computing power of multiprocessors can only be fully harnessed in VMs if multiple virtual CPUs are exposed. A virtual CPU is added to a VM by letting an additional thread execute an event loop. If this takes place on an additional physical CPU, the VM enjoys actual hardware concurrency.

4.2 System Environment

As we do not to support 16bit execution mode nor a BIOS implementation, traditional boot loaders cannot be used. Instead, we implemented the EFI 32 bit boot protocol to boot Linux directly in protected mode. This requires setting up initial segments, instruction and stack pointers and a data structure providing Linux with information such as the physical memory layout and the command line.

Some devices are not managed by device managers but used by the microkernel itself. For example, the platform timer is not exposed directly. Instead, the kernel makes it accessible in the shape of IPC timeouts. Our timer implementation employs a thread that infinitely runs a loop wherein it blocks for the duration of a timer tick. After each timeout, it flags the occurrence of a timer tick. The main event thread notices that and injects a timer interrupt into the VM.

[7] A special *vmmcall* instruction is included for that purpose.

4.3 Peripheral Devices

In a microkernel system, the kernel itself does not contain device drivers. Devices are managed by user-level servers, which provide a high level message passing interface, possibly complemented by shared memory for efficient bulk data transfer.

An IPC interface poses a challenge to a VM because IPC is not readily available. Instead, device requests are forwarded to the VMM, which then sends an IPC to the device manager on behalf of the VM. Some device managers may employ an RPC-style interface where the caller blocks until the request is completed. Such a behavior is at odds with basic operating system design rules, which assume that peripheral device act concurrently to the processor. In such cases, the VMM employs dedicated helper threads, which may block without stopping the main thread.

Given our preference for efficient para-virtualized devices, the device virtualization follows a general pattern; tiny Linux stub drivers communicate via a streamlined shared-memory-based protocol with driver backends implemented in the VMM. The driver backends implement IPC based communication channels to L4 services.

Currently, the following devices are supported: graphics, network, and serial line communication.

The only exception is the hard disk, as at the time of this writing no L4 device manager for block devices is available. Instead, we implemented direct pass-through hard disk access. This is only a temporary solution as the hard disk pass-through uses DMA and thus completely evades our security architecture. It also limits hard disk access to one VM at a time.

4.4 Complexity and Size

One way of estimating the complexity of an application is to look at the amount of lines of source code. By this criterion, our VMM is of very low complexity as it comprises only about 3800 SLOC. The patch to Linux is about 2800 SLOC, with 2300 SLOC being the stub drivers.

The small source size translates into a small resource footprint at run-time. Together with some support libraries, the actual memory footprint of the VMM amounts to 284kB per instance. These demands compare very well to previous work [18]. The approach chosen there–running a VMM with its

own Linux-based operating system–is estimated to have a lower resource bound of 16MB.

5 Staged Virtualization

Our low-complexity Karma VMM can only run para-virtualized operating systems. License restrictions or lacking access to the source code make para-virtualization of some operating systems infeasible. However, with some binary-only systems being wide-spread, it would be desirable to accommodate such systems as well and thereby support their large application base.

Since implementing faithful virtualization from scratch is a large effort that takes many man-years, we chose to re-use an existing VMM. Such a VMM usually makes use of operating system services such as file system access or a graphical user interface. Moreover, a VMM assumes direct control over the hardware which is necessary to initiate a world switch into a VM and exercise control over physical memory.

For our system, we settled on KVM [12], a project that enhances Linux with hypervisor functionality and leverages Qemu for device virtualization. It is backed by a vibrant community and already supports a number of guest operating systems. We placed KVM together with its infrastructure (Linux and Qemu) in a VM managed by our low-complexity Karma VMM. We call a VM that is controlled by the Karma VMM a *first-stage VM*, and consequently, we call a VM run by KVM a *second-stage VM*. Note, that a second-stage VM is also managed by our light-weight VMM to a certain degree. The difference between them is that the light-weight VMM provides the (para-virtualized) devices to first-stage VMs, whereas this duty is assumed by first-stage VMs for second stage ones.

Since–for security reasons–privileged instructions are the preserve of the microkernel, the VMM cannot directly initiate a world switch but has to invoke a microkernel operation. We replaced the privileged *vmrun* in the KVM kernel module with requests to the light-weight VMM, who, in turn, invokes a microkernel operation. The same goes for memory operations (map and unmap). An illustration of this setup can be seen in Figure 4.

When Qemu instructs the KVM kernel module to initiate a world switch (1), KVM issues a hypercall (2). The VMM then instructs the microkernel to actually initiate the world switch (3). Now the execution runs inside the second-stage VM. On each fault, for example due to the guest commanding a device, control falls back at the VMM (4). The VMM will then give control back to the first-stage VM kernel (5) and finally to Qemu (6).

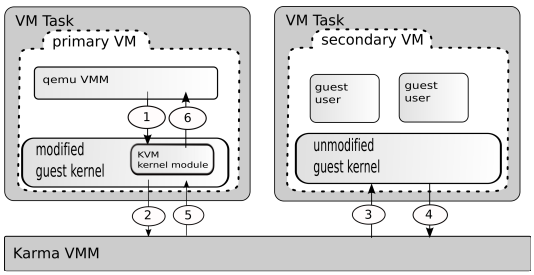

FIGURE 4: *Staged Virtualization.*

To a certain degree, this setup mimics the setup of Xen, with the first-stage VM being the Dom0 and the second-stage VM being a DomU. Similar to Xen, we also allow the first-stage VM to drive many second-stage VMs. The difference to Xen is, that we can run multiple first-stage VMs on one machine and guarantee isolation between them.

6 Evaluation

All measurements were done on an AMD Phenom 8450 triple-core processor with 2.1GHz and 512Mb RAM. In all setups, we used Linux in version 2.6.31.5, running Debian 5.0.

The benchmarks were run on the para-virtualized Linux for first-stage VMs. For second-stage VMs, we made sure that the para-virtualized Linux in the first-stage VM was idle prior to running the benchmark inside the second-stage VM. We compared these measurements to a benchmark run on a native Linux that had the same configuration as Linux in the first- and second stage VMs.

6.1 Throughput

The first benchmark consists of a shell script that creates and destroys a fair number of processes. It stresses the Linux virtual memory subsystem as well as its system call interface. These operations proved troublesome for previous rehosted operating systems. The numbers derived from that benchmark are shown in Table 1. Surprisingly, on a single CPU the benchmark ran faster on the para-virtualized Linux than on native Linux. We ascribe that to scheduling anomalies.

Next we measured a compile run of a Linux 2.6.30.4 kernel with standard configuration for the x86 platform. The kernel compilation was done with warm caches, and the measured run-times can be found in Table 2.

	Native Linux	First-stage VM
1 CPU	947s	876s
2 CPUs	459s	455s
3 CPUs	307s	357s

TABLE 1: *Synthetic system benchmark*

	Native Linux	First-stage VM
1 CPU	619s	620s
2 CPUs	307s	316s
3 CPUs	223s	234s

TABLE 2: *Run-time of kernel compilation*

The kernel compilation was also used to evaluate the performance of the second-stage VMs shown in Table 3. In comparison to an unmodified KVM, the second-stage VM performed with negligible performance degradation.

In both benchmarks, the performance gaps widens as the number of CPUs increases. In the second-stage VM this effect is even more pronounced. We believe that this is due to an increased number of cross-CPU interactions that involve APIC operations, the handling of which is expensive. The slowdown compared to native Linux is due to the KVM setup. As both the native KVM and the second-stage VM run the same setup, both would equally benefit from tuning.

	Native KVM	Second-stage VM
1 CPU	734s	746s
2 CPUs	365s	382s
3 CPUs	265s	297s

TABLE 3: *Kernel compile in second-stage VM and in native KVM*

6.2 Latency

To assess the capabilities of our solution regarding event handling, we measured latencies for the same machine running stock Linux and L4. The measurements on L4 were conducted with a small test program developed for that purpose. It runs alongside a Karma VM and uses the *HPET* as external event source. We measured the system idle and with loaded with *hackbench* serving as load generator inside the VM. The results are depicted in figure 5.

The measurements confirm the good preemptibility of the Fiasco microkernel. With a running VM, higher latencies are to be expected due to *VM entries* and *VM exits*. The additional delays are in the range of 5*us* for regular operations and below 15*us* for pathological cases [21]. The measured increases in that range under load are in line with these previous findings.

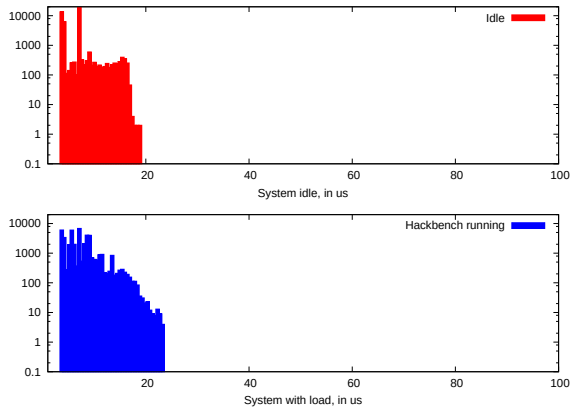

FIGURE 5: *Latencies on L4.*

For the Linux measurements we used the `cyclictest` tool [6]. The test was run on Linux 2.6.35 configured for full preemption (*CONFIG_PREEMPT*) but without the *RT_PREEMPT* patch applied. The results are shown in figure 6.

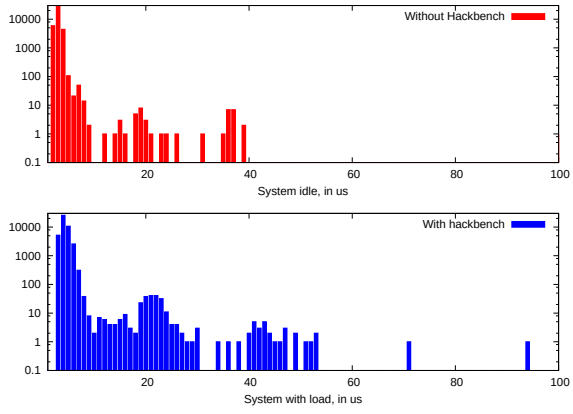

FIGURE 6: *Latencies on Linux.*

Fiasco has clearly an edge over mainline Linux regarding worst-case latencies. The simplicity of Fiasco allows for better preemptibility. Complex

workloads, which come with longer non-preemptible portions, can be encapsulated in VMs where they have control over preemptions visible *to them* but do not interfere with the system scheduler. It remains to be seen how more preemptible versions of Linux will fare. The RT_PREEMPT line of development shows promise to significantly cut down on worst-case latencies.

The larger average latencies of the L4 system under idle conditions can be attributed to the following reasons. First, while the current version of Fiasco supports message-signalled interrupts (MSIs) for PCI devices, it does currently not so for the HPET. Accordingly, its interrupts have to pass through the IO-APIC. Second, Fiasco has to mask the level-triggered HPET interrupt before it can acknowledge it. That is necessary because an unacknowledged interrupt could prevent the delivery other pending interrupts which could give rise to priority inversion. Masking the interrupt on the IO-APIC takes 2.5μs alone.

7 Related Work

Various research projects aimed at running encapsulated operating systems efficiently on top of small kernels.

L4Linux [8] proved that it is possible to slip a microkernel under an operating system while retaining acceptable performance. As the L4 microkernel interface is sufficiently different from a CPU model, the changes to the Linux kernel are significant. Apart from the implementation overhead, this approach also incurs non-negligible runtime overhead because many operations, such as page table updates, are security sensitive and have to be mediated by the microkernel. In contrast, the changes on guest OSes required by Karma affect only peripherals and leave the processor specific parts unchanged.

Microkernels can be readily extended to feature VMs as protection domains without losing their design superiority. KVM-L4 [18] showed how an existing, feature-rich VMM can be encapsulated so that security concerned applications are not put at risk. The drawback of this approach is, that the VMM itself relies on a rehosted operating system, in that case L4Linux. If used as environment to host the VMM, L4Linux has a much larger resource footprint than Karma. That advantage only holds for VMs that run (first stage) guests with Karma adaptions. Support for unmodified guests requires two-staged virtualization. The first-stage VM is itself a Linux which negates the size advantage over L4Linux. The performance edge, though, remains since Karma supports multiple processors per VM whereas KVM-L4 only runs on uniprocessors.

Xen [4] is–in our terminology–both a hypervisor and a virtual machine monitor. The system only offers VMs as protection domains, which implies some resource overhead and is detrimental to an architecture involving many small components.

While NOVA [23] shares many of the goals regarding a small TCB and capability-based security mechanisms, it disregards support for real-time workloads. NOVA relies on virtualization extensions for legacy operating system encapsulation, whereas it is optional to make use of virtualization extensions to run legacy operating systems on Fiasco. As such, a developer may run a performance-sensitive VM with virtualization extensions along with a realtime-VM not using them. As the latter is made up of native tasks, it enjoys lower event latencies. Furthermore we believe that the requirements on VMMs are too diverse to be addressed by a single implementation. Without a scalable approach, a VMM is prone to either lack functionality or grow to a size that severely calls its trustworthiness into question.

8 Conclusion

In this paper, we presented a two-pronged approach to run legacy environments on top of a microkernel-based system. First, we created a well-performing, low-complexity VMM that establishes a VM to run a slightly modified Linux. We use these light-weight VMs to employ a feature-rich legacy VMM, which in turn provides backward-compatible VMs.

Our evaluation showed that the performance degradation incurred by our light-weight Karma VMM is small enough to be non-noticeable for most real-world workloads. The second-stage VM with faithful virtualization matches the performance of the VMM it is derived from.

Fiasco also shows promise with regard to worst-case event handling latencies. Its good preemptibility can be retained in the presence of virtual machines. It remains to be seen if system developer are willing to dispense with a developed ecosystem for a simpler system architecture.

9 Acknowledgements

Our thanks is due to Björn Döbel who diligently fixed any bug in the network server that was awoken during our measurements. Further, we are grateful to Jean Wolter who perseveringly took part in a race of him finding fault with inconsistencies and us ironing them out.

We also want to thank our colleagues Janis Danisevskis and Matthias Lange, who are contributing to the development of the Karma VMM.

We would also like to thank the European Commission for their Research Programme FP7 supporting us through the projects eMuCo[8] and TECOM[9].

References

[1] L4Linux - running Linux on top of L4. `http://os.inf.tu-dresden.de/L4/LinuxOnL4/`.

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM.

[3] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*.

[4] 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, 2003.

[5] P. Chen and B. Noble. When virtual is better than real. In *hotos*, page 0133. Published by the IEEE Computer Society, 2001.

[6] cyclictest. URL:`https://rt.wiki.kernel.org/index.php/Cyclictest`.

[7] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *In IEEE CollaborateCom 2005*, 2005.

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

[9] J. N. Herder, D. C. van Moolenbroek, R. Appuswamy, B. Wu, B. Gras, and A. S. Tanenbaum. Dealing with driver failures in the storage stack. *Dependable Computing, Latin-American Symposium on*, 0, 2009.

[10] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *EW 11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 22, New York, NY, USA, 2004. ACM.

[11] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide*.

[12] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Linux Symposium*, 2007.

[13] K. Kortchinsky. Cloudburst – Hacking 3D and Breaking out of VMware. Black Hat USA, 2009.

[14] A. Lackorzynski and A. Warg. Taming Subsystems: Capabilities as Universal Resource Access Control in L4. In *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, pages 25–30, Nuremberg, Germany, 2009. ACM.

[15] B. Leslie, C. van Schaik, and G. Heiser. Wombat: A portable user-mode Linux for embedded systems. In *Proceedings of the 6th Linux. Conf. Au, Canberra*, 2005.

[16] M. Miller, K.-P. Yee, J. Shapiro, and C. Inc. Capability Myths Demolished. Technical report, 2003.

[17] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 308–318, New York, NY, USA, 2008. ACM.

[18] M. Peter, H. Schild, A. Lackorzynski, and A. Warg. Virtual machines jailed: virtualization in systems with small trusted computing bases. In *VTDS '09: Proceedings of the 1st EuroSys Workshop on Virtualization Technology for*

[8] `http://www.emuco.eu/`

[9] `http://www.tecom-project.eu/`

Dependable Systems, pages 18–23, Nuremberg, Germany, 2009. ACM.

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

[20] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 10–10, Berkeley, CA, USA, 2000. USENIX Association.

[21] H. Schild, A. Lackorzynski, and A. Warg. Faithful Virtualization on a Real-Time Operating System. In *Proceedings of the Eleventh Real-Time Linux Workshop*, pages 237–243, Dresden, Germany, 2009.

[22] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 335–350, New York, NY, USA, 2007. ACM.

[23] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 209–222, New York, NY, USA, 2010. ACM.

[24] D. Vogt, B. Döbel, and A. Lackorzynski. Stay strong, stay safe: Enhancing reliability of a secure operating system. In *Proceedings of the Workshop on Isolation and Integration for Dependable Systems (IIDS 2010), Paris, France, April 2010*, New York, NY, USA, 2010. ACM.

[25] R. Wojtczuk. Subverting the Xen hypervisor, 2008.