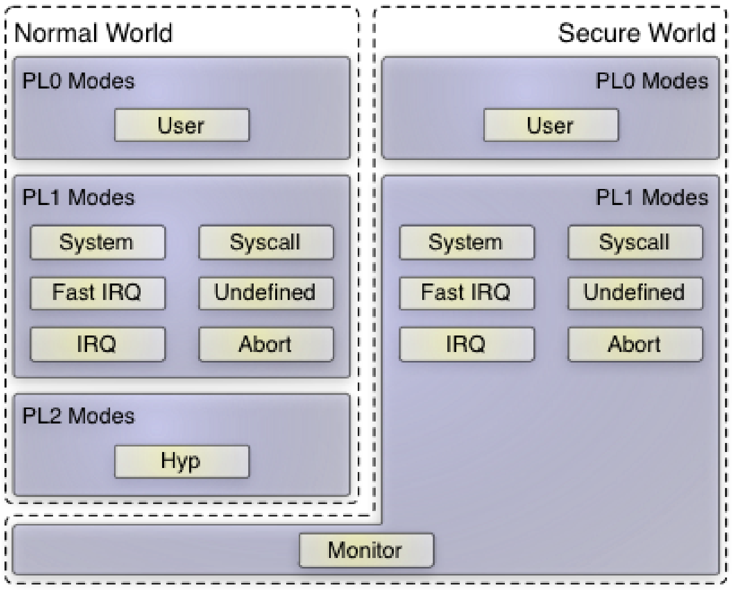
An in-depth look into the ARM virtualization extensions

Recent high end ARM CPUs include support for hardware virtualization. Due to limitations of former ARM architectures, virtualizing the hardware tended to be slow and expensive. Some privileged instructions did not necessarily trap when executed in non-privileged mode. This effectively prevented a hypervisor to run the kernel code of the guest operating system unmodified in non-privileged mode and to handle privileged instructions using a trap-and-emulate approach. Instead of modifying the architecture in a way to close the virtualization holes, meaning to always trap when privileged parts of the CPU state shall be modified, ARM decided to stay backwards compatible and extended the ARM v7 architecture to support virtualization. ARM went a similar path like Intel when VT was introduced for the x86 architecture.

Although ARM's virtualization extensions seem to be very similar to Intel's approach, the devil is in the details. We were looking forward to explore them for quite some time and are happy to share the insights of our research with you. When starting with this line of work on ARM virtualization, one of the most interesting questions for us was: How can we integrate virtualization into Genode without increasing the trusted computing base (TCB) of applications that run beside virtual machines? Moreover, how can we make the virtual machine monitor (VMM) of one virtual machine (VM) independent of others, similar to the approach taken by the NOVA OS virtualization architecture?

Overall Architecture

The ARM virtualization extensions are based on the security extensions, commonly known as TrustZone. For more information about TrustZone, refer to our previously published article. To realize the switching between different virtual machines, a new privilege level was introduced within the normal world of the processor, including one new processor execution mode - the "hyp" mode.

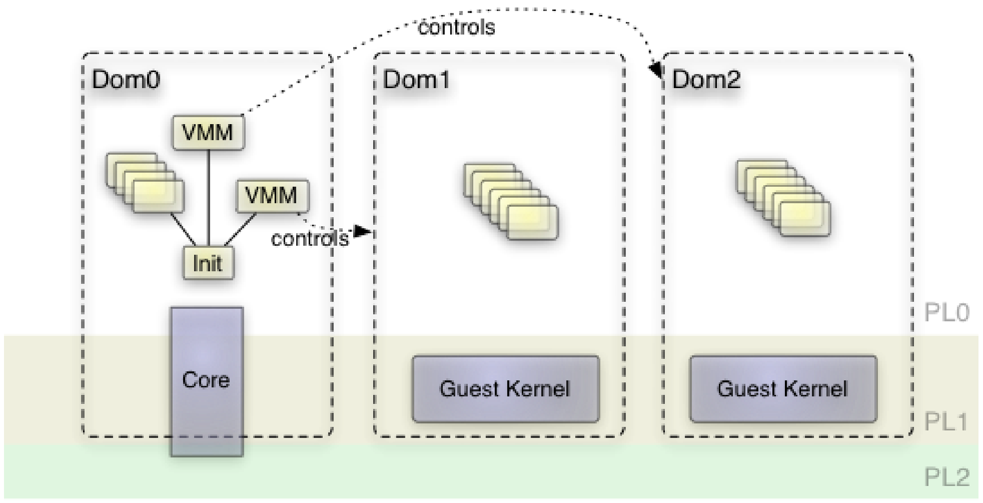


ARM privilege levels (PL) and processor modes

The figure above presents an overview of all privilege levels and modes an ARM processor may execute code in. As depicted, virtual machines can be executed in TrustZone's "normal world" only. Processor instructions that support VM separation and world switches can be used in either hyp or monitor mode only. Thereby, the first design question was whether the hypervisor-related code shall be executed in the normal world solely, or if at least parts of the hypervisor should run in the "secure world"? Which of both TrustZone worlds should host normal user-land applications, not related to virtualization?

Until now, Genode used to be executed in either the normal world or the secure world, depending on the platform. For instance, when using the Pandaboard, the secure world is already locked by OEM firmware during the boot process, and Genode is restricted to the normal world. In contrast, on the ARNDALE board, the Genode system is free to use both the secure and normal world. We decided to ignore the secure world here and to execute everything in the normal world. Thereby, Genode is able to accommodate all possible platforms including those where the TrustZone features are already locked by the vendor.

Starting point of the current work was Genode's integrated ARM kernel that we call "base-hw" ("hw" representing Genode running on bare hardware as opposed to running it on a third-party kernel). On this platform, Genode's core process runs partly in kernel mode (PL1) but most code is running in user mode (PL0). The rationale behind this design is explained in our TrustZone article. We broadened this approach by extending the core process with hypervisor-specific code running in hyp mode (PL2), thereby gaining a binary that has a global view on the hardware resources yet is executed in three different privilege levels. The concept of the base-hw platform hosting different virtual machines from a bird's perspective is depicted in the following figure.



Genode's ARM kernel (core) runs across all privilege levels

Ideally, the part of the core process, which runs in hyp mode comprises hypervisor-specific code only. This includes the code to switch between different virtual machines as well as the Genode world denoted as Dom0 in the picture. To keep its complexity as low as possible, the hypervisor should stay free from any device emulation. If possible its functionality should come down to reloading general purpose and system registers, and managing guest-physical to host-physical memory translations.

In contrast to the low-complexity hypervisor, the user-level VMM can be complex without putting the system's security at risk. It contains potentially complex device-emulation code and assigns hardware resources such as memory and interrupts to the VM. The VMM is an ordinary application running unprivileged and can be re-instantiated per VM. By instantiating one VMM per VM, different VMs are well separated from each other. Even in the event that one VMM breaks, the other VMs stay unaffected. Of course, a plain user-land application is not able to directly use the hardware virtualization extensions. These extensions are available in hyp mode only, which is exclusive to the kernel. Hence an interface between VMM and the kernel is needed to share the state of a virtual machine. We faced a similar problem when building a VMM for our former TrustZone experiments. It was natural to build upon the available solution, extending it wherever necessary. Core provides a so called VM service. Each VM corresponds to a session of this service. The session provides the following interface:

CPU state

> The CPU-state function returns a dataspace containing the virtual machine's state. It is initially filled by the VMM before starting the VM, gets updated by the hypervisor whenever it switches away from the VM, and can be used by the VMM to interpret the behavior of the guest OS. Moreover, it can be updated after the virtual machine monitor emulated instructions for the VM. This mechanism can be compared to the VMCS structure in the x86 architecture.

Exception handler

> The second function is used to register a signal handler that gets informed whenever the VM produces a virtualization fault.

Run

> The run function starts or resumes the execution of the VM.

Pause

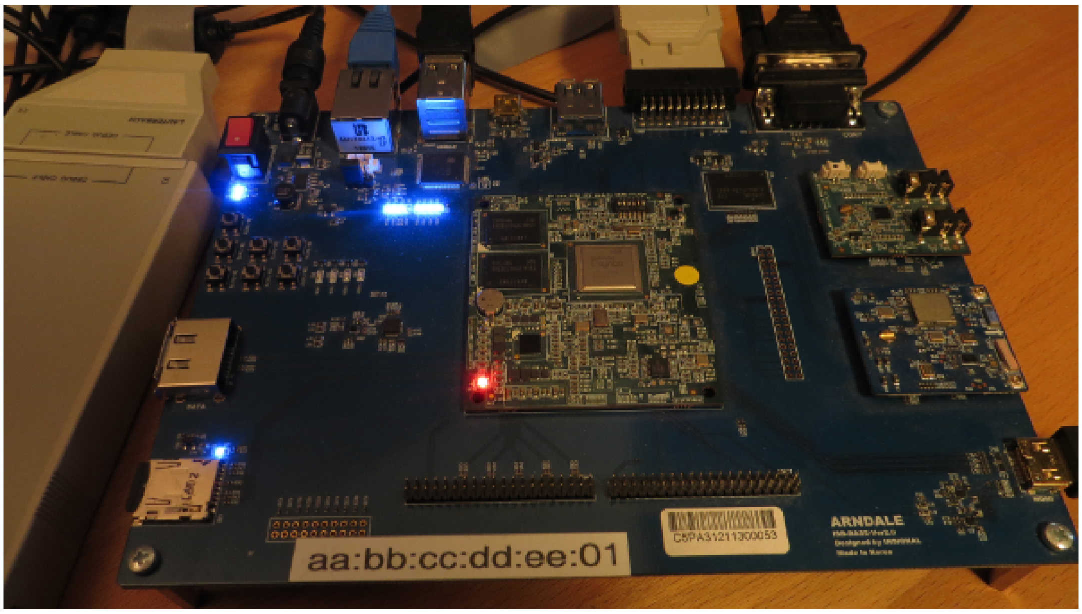
> The pause function removes the VM from the kernel's scheduler.

Given this high-level architecture the remainder of the article covers the technical challenges to realize it. We start with explaining what had to be done to initially bootstrap the platform from secure to normal world. We encountered memory as the first hardware resource we had to address. Section Memory Virtualization explains our approach to virtualizing memory. The most substantial part is the virtualization of the CPU, which is covered in section CPU Virtualization. It is followed by the explanation of the pitfalls and positive surprises while virtualizing interrupts and time in sections

Virtualizing Interrupts and Virtual Time. Finally, the article closes with a summary of the current state.

Bootstrap into Genode's "Dom0"

To practically start exploring the virtualization extensions, we used the ARNDALE development platform containing a Samsung Exynos5 SoC, which is based on two Cortex A15 CPU cores. Using this board was beneficial as Genode's base-hw platform already supported ARNDALE with most device drivers already covered. Moreover, this low-cost device offers UART and JTAG connectors, which are greatly advantageous when investigating new processor features.



ARNDALE development board with Exynos 5250

As already mentioned, the ARNDALE board boots into the secure world of TrustZone. As we decided to run the whole system within the normal world, the first step was to bootstrap into the hyp mode of the normal world. Before leaving the secure world, the following adjustments were needed to allow the normal world to access all hardware resources. The "non-secure access control register" (NSACR) had to be configured to allow access to all co-processors and to allow tweaking multi-processor related bits of the "auxiliary control register" (ACTLR). Moreover, all interrupts had to be marked as non-secure at the interrupt controller so that the normal world is able to receive them.

When doing so, we discovered a tricky detail regarding the interrupt controller that caused us some headache. ARM's generic interrupt controller (GIC) is split into a CPU interface that exists for each core and a global module called "distributor". Normally, all properties adjusted at the distributor are concerning all cores including the security classification of an interrupt that determines whether it shall be received within the secure or normal world. But this doesn't apply to the first 32 interrupts that are private to each core. During our first experiments, we let the boot CPU mark all interrupts as being non-secure. Everything went fine until the point where the second CPU core had to receive an inter-processor interrupt (IPI) from the first one. That IPI, however, is a core-private interrupt. Hence, its security classification needed to be set by each CPU core during initialization, not merely once by the first core.

After conquering the trouble with the interrupt controller, we finally configured the "secure configuration register" (SCR) to:

- Enable the hypervisor call,
- Disable the secure monitor call,
- Not trap to the secure world, thereby effectively locking the secure world,
- Switch to the normal world.

With these steps, the CPU ends up in hypervisor mode. Before continuing the regular kernel boot process, the CPU had to drop the hypervisor privilege level PL2 and enter the kernel's regular privilege level (PL1). Running the existent kernel code within the hypervisor mode without modifications would not work. There are few incompatibilities that prevent the execution of code written for PL1 within PL2. For instance, when using the multiple-load instruction (LDM) to access user-mode specific registers or to return from an exception, the result is undefined in hypervisor mode. To be able to re-enter the hypervisor mode from the kernel parts that run in lower privilege levels, some further preparations had to be done. First, an exception vector table had to be installed, which is a table of functions that are called when a hypervisor-related exception occurs. After setting up the hypervisor's exception vector table via the "hyp vector base address register" (HVBAR), the regular boot process could continue outside of the hypervisor mode.

Memory Virtualization

ARM's virtualization support extends the former MMU by an optional second stage of address translations, thereby effectively realizing nested paging. Moreover, to enable the usage of virtual memory within the hypervisor mode, ARM additionally introduced an exclusive MMU for this mode.

To overcome the limitations of a 32-bit addressable physical memory space, ARM offers a new page table format, which extends the addressable physical memory space to 40 bits. This so called "large physical address extension" (LPAE) is obligatory when using virtualization. The second stage of guest-physical to host-physical page tables as well as the hypervisor's page tables need to use the new format. When using the normal MMU that realizes virtual to physical or respectively guest-virtual to guest-physical translations, one can decide whether to use the new or the legacy page table format.

As we planned to run the core process in all different privilege levels and thereby sharing one memory view, it seemed beneficial to use the same page table set for all privilege levels. As the hypervisor mode requires the new page table format anyway, we decided to first implement that within the prior existing kernel. The benefit of this approach was that we could use the kernel as testbed for validating the new format. Furthermore, it enables comparative measurements. The old page table format has a depth of at most 2 levels whereby the new one has 3 levels. Therefore, some performance differences were expected.

After integrating the new format within Genode's base-hw platform, indeed we could measure a slight performance degradation. As runtime test, we used the compilation of Genode's core process on top of Genode itself. The compilation process spawns a lot of processes sequentially, which over and over again populate their address spaces anew. This stresses the TLB entry replacement as well as page-table walks of the MMU. Although the test makes heavy use of the MMU, it represents a real life example in contrast to an artificial micro benchmark. Therefore, it hopefully demonstrates effective costs more accurately. As illustrated by table 1, the runtime overhead when using the new 3-level page table format is less than 1% in relation to the compilation test.

To always provide a consistent memory view to core across all different privilege levels, the page table formerly shared between user-land and kernel had to be used by the hypervisor-specific MMU too. Therefore, we extended the hypervisor mode preparation described in the previous section Bootstrap into Genode's "Dom0". We set the page table address in the hypervisor's "hyp translation table base register" (HTTBR). Furthermore, we set the page attributes in the "hyp translation control register" (HTCR) and the "hyp memory attribute indirection registers" (HMAIR). Finally, we enabled the MMU and caches for the hypervisor via the "hyp system control register" (HSCTLR).

At last, we tested the nested paging by creating a 1:1 guest-physical to host-physical translation table and enabled the second stage translation for the normal Genode system. The "virtualization translation table base register" (VTTBR), the "virtualization translation control register" (VTCR), and the "hyp configuration register" (HCR) were used to set the address of the second-stage table, to set page attributes, and to enable nested paging. They naturally can exclusively be accessed in hypervisor mode. Hence, setting up those registers is performed during the formerly described bootstrap process. After enabling the second-stage translation, we measured the compilation test again and discovered a run-time overhead of 3.7% compared to the version using the old page table format without nested paging. The table 1 comprises our measurements.

page table format	duration in ms	overhead
2 level, 1 stage	214	
3 level, 1 stage	216	0.9 %
3 level, 2 stages	222	3.7 %

Table 1: Compilation test measurements using different translation schemes.

Given the measured performance degradation while using nested paging, we decided to enable the virtualization MMU when switching to a guest OS only, leaving it disabled as long the Genode software stack is running. Using this approach, we observed an interesting side effect at a later point of development.

To reduce the necessity of TLB maintenance operations when switching between different processes, ARM introduced "address space identifiers" (ASIDs) in the past, which were used to tag entries within the TLB. Thereby, only those TLB entries tagged with the currently active ASID are regarded as valid ones. Now that the MMU is shared not only between address spaces but also between virtual machines, an extra "virtual machine identifier" (VMID) was added. It can be set in the VTTBR system register, which is responsible for setting the second-stage page table. The important learning effect for us was that the VMID is used to tag the TLB regardless of whether the nested paging is enabled or not. Therefore, we reserved the VMID 0 to be used by the host system and use the remaining 255 VMIDs to identify virtual machines.

CPU Virtualization

After experimenting successfully with memory virtualization, the next consequential step was to realize the world-switch regarding the CPU state. We started with a very simple VMM that used the VM session interface described in section Overall Architecture to provide the VM's initial register set to the hypervisor. The register set, denoted as VM state in the following, at first comprised merely the general purpose registers (r0-r15), the "current program status register" (CPSR), and "banked" copies of some of these registers related to the different execution modes.

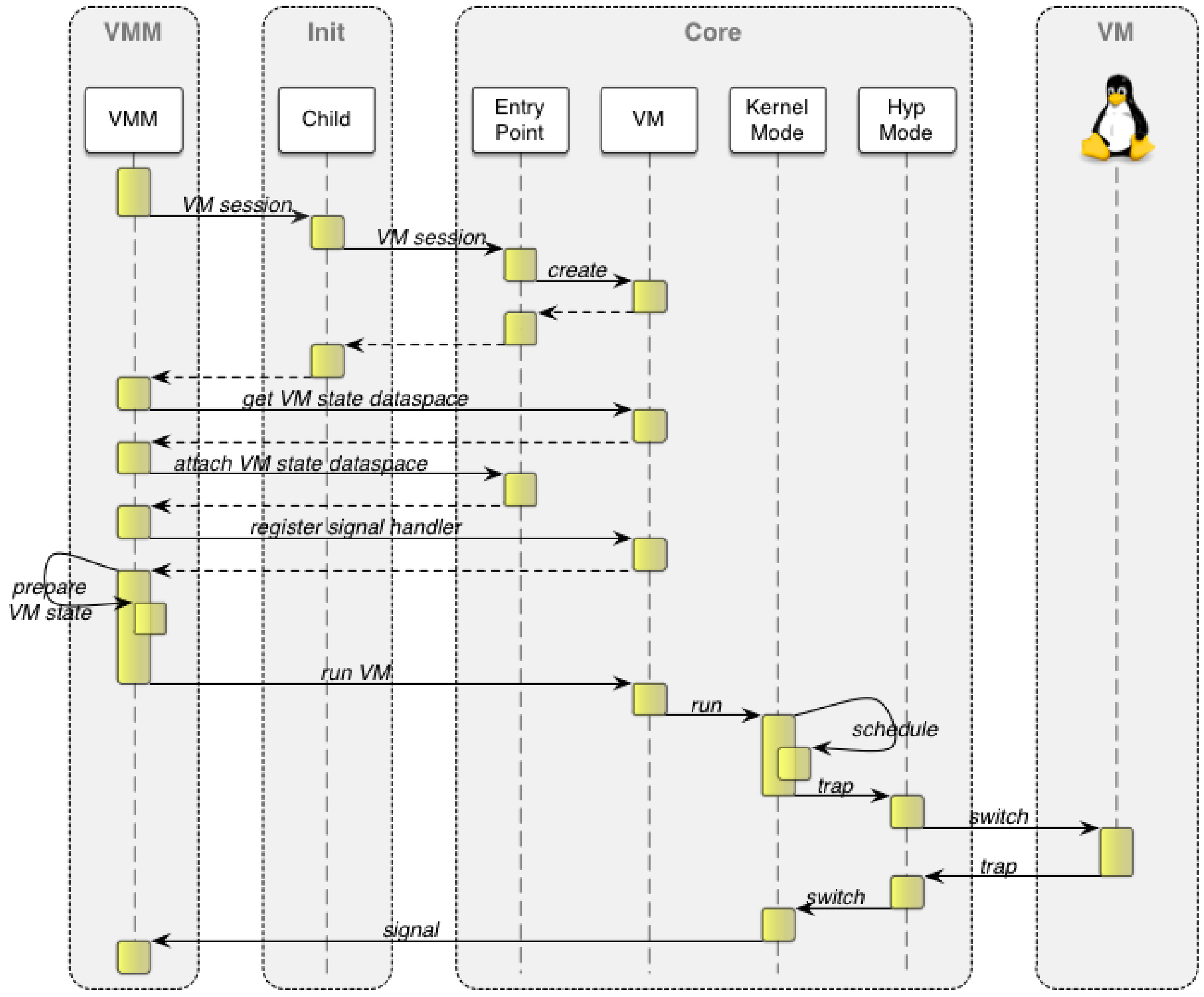
At first, the VMM requests the dataspace containing the VM state via its VM session from core. It prepares the state by providing corresponding reset values. Using the VM session, it registers a signal handler and starts the execution of the VM. After that, the VMM waits for virtualization events that are delivered to its signal handler.

When a VM session is opened, the core process creates a new VM object. This object comprises the dataspace for the VM state, the VMID, and the initially empty second-stage page-table used for guest-physical to host-physical translations. When the session's client issues a run call, core adds the VM object to the scheduler. If the scheduler selects the VM to run, all mode-specific banked registers are reloaded and a hypervisor call is triggered using the "hvc" instruction. The hypervisor responds to this call by first checking whether the virtualization MMU is enabled. If disabled it infers that the call was triggered by the Genode host system, not by a VM. In that case, the hypervisor assumes a host-to-VM switch.

For the actual switch, the virtualization MMU is enabled, the VM's second stage table and VMID are loaded into the VTTBR, and the trap behavior is configured. Whenever a co-processor is accessed, including access to the system co-processor (CP15) that contains all system registers, the VM enters the hypervisor. This trap control is managed via the "hyp coprocessor trap register" (HCPTR), the "hyp system trap register" (HSTR), and the HCR. Finally, the hypervisor loads all the VM's general-purpose registers and the CPSR, and drops its privilege level to actually execute the VM. The detailed process of VM creation and execution is depicted in the figure below.

When the virtual machine is trapped, the "hyp syndrome register" (HSR) is updated by the hardware. The HSR contains the type of exception and additional information specific to the exception. Among such syndromes are an attempt to access the system's co-processor, a page fault related to the second stage page tables, and a hypervisor call. Additional exception-specific information encoded into the HSR are for instance the source and destination registers when accessing a system register. In addition to the HSR, there are other system registers containing information about the guest-virtual address that caused a page fault (HDFAR and HIFAR) as well as the page-aligned guest-physical address that produced a fault (HPFAR).

The information provided via the HSR, HDFAR, HIFAR, HPFAR, and the general-purpose registers principally suffices for the emulation of virtual devices. There is no need to walk the page tables of the guest OS, access its memory, and decode instructions to gain the relevant information. It was a smart decision by ARM to encode the exception related information already known by the hardware into these hypervisor-specific trap registers. It does not only simplify the VMM software but also alleviates the need of the VMM to access the VM's dedicated memory. Such accesses may suffer from ARM's weak cache coherency.



Example sequence of creating and running a VM

Given the above insights, we decided to extend the VM state by HSR, HDFAR, HIFAR, and HPFAR. When switching from a VM to the host system, these fields of the VM state are updated within hypervisor mode. After core signaled that the VM is trapped, the VMM can use the updated state information to handle the exception.

In our experiments, until now the VMM just started a virtual machine with a given set of registers. No physical memory was reserved for the VM, not to mention the missing load of the guest OS binary to memory. As we don't want to trust the VMM more than any other user-land application, we can't let the VMM control the VM's memory via the second stage table directly. If a VMM had direct access to the nested pages, it might provide access to any memory area to its VM including memory containing the hypervisor itself. Therefore, we had to extend the VM session interface with the facility to populate the guest's physical memory. To enable a VMM to assign host-physical memory to the guest-physical address space, it first needs to proof that it has appropriate rights to access the host-physical memory. This proof is provided in form of a dataspace capability. The kernel adds the host-physical memory referenced by the dataspace to the second-level page table at a guest-physical address specified by the VMM.

To test the preliminary world-switch routine as well as the VM's physical memory-control functions, we implemented a very simplified guest OS kernel. This kernel does not do anything except from accessing some of its binary data and afterwards provoking a hypervisor call. The altered VMM first copies the simple kernel image into the dataspace it previously requested via its own RAM session. Then it uses the VM session interface to attach the same dataspace to the guest-physical memory at the address the test kernel is linked to. The actual incorporation into the second-stage page tables of the VM is done by the core process. After that the VMM prepares the VM state and starts the virtual machine. Whenever a virtualization event is signaled to the VMM, it dumps the whole VM state as debug output and halts the VM.

The first trap the VM went into was the very first instruction. But in contrast to our assumptions, the reason was not a misconfiguration related to the second-stage page tables. The information gained by the hypervisor-specific fault registers helped us to identify the problem. The exception got raised by the MMU that tried to resolve the first instruction's address. As we had not considered the "system control register" (SCTRL) in the world switch at that point, all of its properties applied to the VM execution, too. This includes the MMU and caches being enabled. By adding the SCTRL register to the VM state, setting the correct reset value (no MMU) in the VMM, and reloading the SCTRL register whenever switching between the VM and the Genode host system, we were able to run our simple kernel test to the point where it calls the hypervisor explicitly.

At this point, we had a solid foundation to start virtualizing a real guest OS. Naturally, we choose an open-source system as candidate to be able to easily determine what the OS is trying to do when raising virtualization events. Linux was our first candidate as it supports the target hardware best and we were most experienced with it. We picked ARM's Versatile Express Cortex A15 development board as virtual hardware platform to provide it to the Linux guest. Being the reference platform of ARM, documented predominantly well, and with QEMU having support for it, the platform seemed promising.

We used the standard configuration for Versatile Express platforms to compile the latest vanilla Linux kernel. Nowadays in the embedded world, the hardware description is separated from the Linux source code by using so-called "device tree binaries" (DTBs). To avoid the need to virtualize the entire hardware at once, we created a minimal device tree for the Versatile Express board that only contains one CPU core, the interrupt controller, timer, one UART device, and the necessary clock and bus configuration. After successful tests of the minimal DTB together with the unmodified Linux kernel on QEMU, we used our VMM to load both kernel and DTB into memory, and start its execution.

From then on, we went step by step from one virtualization event to the next. As the hypervisor prepared the machine to trap the VM whenever a system register or non-existing physical address is accessed, we could clearly see what the Linux guest is doing in what order. By incrementally emulating system registers, optionally adding them to the VM state where necessary, and disabling trap behavior where appropriate, we completed the CPU virtualization step by step. For instance, all it takes for emulating certain identification registers is returning meaningful values. But for other registers, the access must be forwarded to real hardware. For instance, a change of MMU attributes like the current page table, cannot be emulated but needs to be propagated to the hardware. Those system registers need to be saved and restored by the hypervisor. Therefore, they were added to the VM state successively. When system registers have to be reloaded anyway, in general they also can be accessed by the guest VM directly without the need to trap. Moreover, there are system registers used to maintain TLB, instruction-, and data-caches, which use the currently active VMID and ASID to e.g. flush cache lines associated with the current process respectively VM. Those operations are uncritical and should be done directly by the VM.

Fortunately, ARM enabled fine-grained control regarding the trap behavior of the system co-processor registers. For other co-processors, it is a binary decision whether the VM shall be trapped or not whenever accessing them. The trap behavior with respect to the system coprocessor is divided into 14 different sections, which are controlled via the "hyp system trap register" (HSTR). One of these sections for instance comprises all TLB-maintenance-related system registers. The principle development procedure was to first enable all sections to trap the VM as soon as it accesses one of its registers. If all system registers within a section are reloaded by the hypervisor anyway, the trap behavior of that section can be disabled and direct access can be granted to the VM. As the VMM records each register trap, we could also identify hot spots in the system register set, e.g., the SCTRL register.

Without optimizing at all, and by just giving direct access to TLB and cache maintenance registers, we identified a minimal system register set that needed to be reloaded by the hypervisor. Table 2 shows the minimal register set.

Section	Abbreviation	Name
1	SCTRL	System Control Register
1	CPACR	Coprocessor Access Control Register
2	TTBCR	Translation Table Base Control Register
2	TTBR0	Translation Table Base Register 0
2	TTBR1	Translation Table Base Register 1
3	DACR	Domain Access Control Register
5	DFSR	Data Fault Status Register
5	IFSR	Instruction Fault Status Register
5	ADFSR	Auxiliary DFSR
5	AIFSR	Auxiliary IFSR
6	DFAR	Data Fault Address Register
6	IFAR	Instruction Fault Address Register
10	PRRR	Primary Region Remap Register
10	NMRR	Normal Memory Remap Register
13	CIDR	Context ID Register
13	TPIDR 0-2	Software Thread ID Registers

Table 2: Minimal system register set to be reloaded by the hypervisor.

At least these 16 system registers in addition to the 37 core registers had to be reloaded by the hypervisor on each switch from host to guest system and vice versa.

Virtualizing Interrupts

At a very early step in development, we recognized that if the hypervisor does not configure the trap behavior accordingly, device interrupts under control of the Genode host system where received within the VM. To that effect, the hypervisor has to direct all interrupts to itself whenever switching to a VM. To minimize the overhead, this trap behavior is disabled again when switching back to the host system. Correspondingly, when interrupts occur while normal Genode applications are executed, the hypervisor code isn't involved at all.

During the completion of the CPU model described in the previous section and following the VM's execution step by step, we crossed Linux' initialization routine of the interrupt controller. In addition to the CPU, ARM introduced virtualization extensions for its interrupt controller, too. As explained in section Bootstrap into Genode's "Dom0", ARM's generic interrupt controller (GIC) is split into a core-local CPU interface and one global distributor interface.

With hardware virtualization present, for each core, a "virtual CPU interface" (GICV) is available. Each GICV is accommodated by a dedicated "virtual CPU control interface" (GICH). The GICV is meant as transparent replacement for the normal CPU interface and is used directly by the VM. The control interface is preserved to the hypervisor respectively VMM to manage interrupts appearing at the GICV.

In contrast to the CPU interface, there is no special virtualization support for the global distributor implemented in hardware. Therefore, the VMM has to emulate this device for the virtual machine. Whereas the CPU interface is frequently accessed by the guest, the distributor is supposedly rarely touched, mostly during initialization. Therefore, providing optimization via special hardware support for the CPU interface suggests itself.

At the beginning, we implemented an almost empty shape of a virtual GIC. Without backing the physical address where normally the Distributor interface resides with memory, the VM always triggers a virtualization event when trying to access it. For all GIC registers accessed by Linux, the VMM records their states but does not implement any logic. Thereby, Linux first successfully went through the initialization of the distributor.

To make use of the new virtualization features of the interrupt controller, we decided to provide the Virtual CPU interface to the VM instead of emulating the CPU interface. Therefore, we had to back the physical address range where the VM normally expects the CPU interface, with the GICV's memory-mapped I/O (MMIO) registers. Although, we already extended the VM session interface to enable the VMM to provide arbitrary dataspaces to a VM, this is not appropriate for attaching the GICV's MMIO registers. Given that more than one VMM might run in parallel to control different VMs and there is only one physical GICV, we cannot grant the VMM direct access to it. Otherwise, different VMMs would interfere. On the other hand, only the VMM knows where the CPU interface shall be placed in guest-physical memory, dependent on the platform it emulates. To tackle this problem, the VM session got further extended by a specific function to attach the interrupt controller's CPU interface.

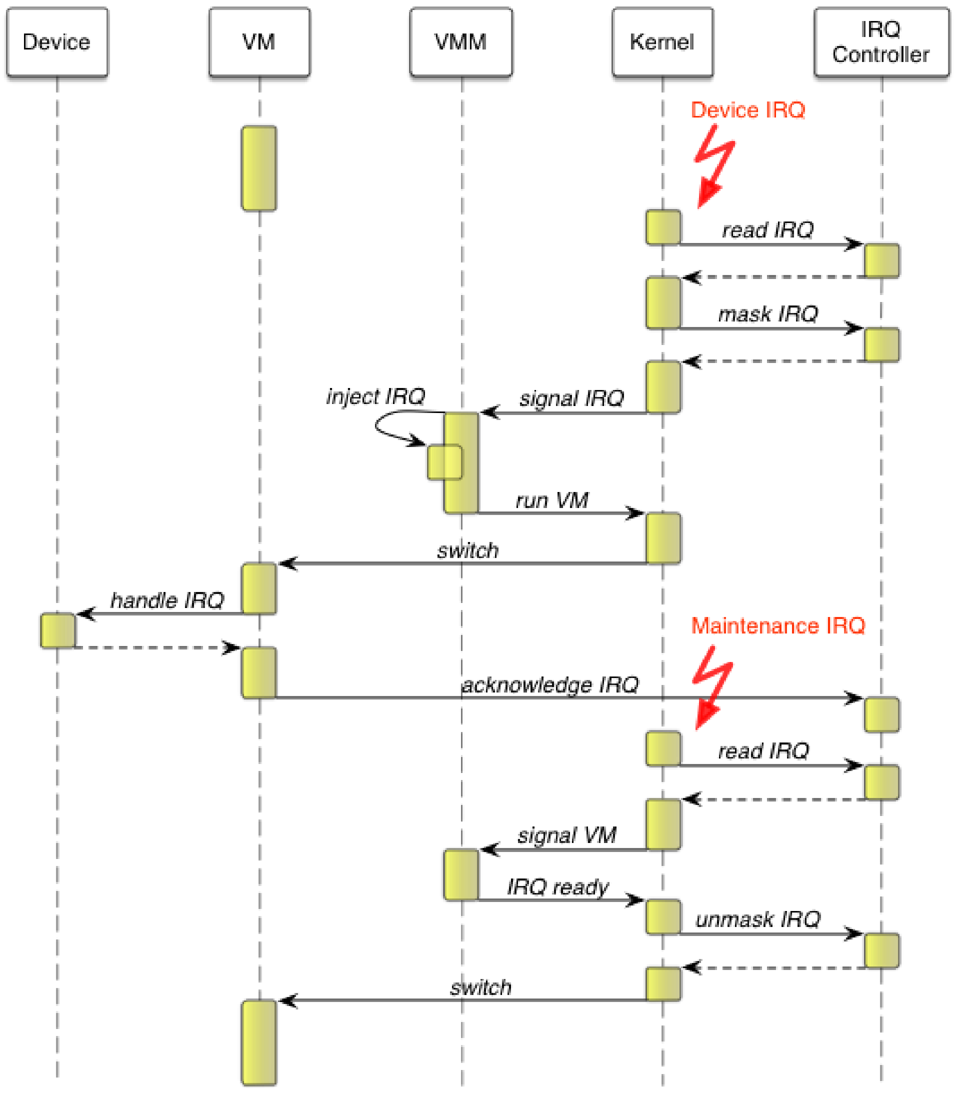
After adding the core-local interrupt controller interface to the memory layout of the VM, the Linux kernel successfully finalized the GIC initialization and continued its boot process to the point where it issued a "wait-for-interrupt" (WFI) instruction. This normally puts the CPU into a light sleep until the next interrupt occurs. When configured accordingly (via the HCR), a VM that executes a WFI instruction is trapped. The VMM recognizes that the VM is waiting for interrupts and stops its execution until the next interrupt occurs. To further follow the initialization of the Linux system, the next logical step was to fill the GIC model with life and inject interrupts to the VM.

The first result was a very simple interrupt controller model. It ignores multi-processor support, priorities, and the assignment of different security levels (TrustZone) for interrupts. Whereas the former is definitely a must-have feature when supporting more than one CPU for the VM, priorities and the security extensions are not used by our Linux guest kernel anyway. As all-encompassing priority support would make the state-machine of the GIC much more complex, it is desirable to omit it if possible. The current GIC model holds state for the individual interrupts, whether they are enabled or not, and whether the interrupt controller itself is in charge.

To signal an interrupt to the VM, the GICV's control interface can be used. It is under exclusive control by the core process (hypervisor), not the VMM. This design prevents interferences between different virtual machines and monitors. Our first intuitive approach, following the micro-kernel paradigm, was to minimize complexity within the hypervisor by just reloading the GICH registers when switching between different VMs. We introduced a shadow copy of the GICH registers into the VM state. The VMM operates on the shadow copy only. When the hypervisor performs a world-switch, it loads the shadow GICH state into the real GICH.

When dealing with the very first interrupts, limitations of this approach appeared. The first interrupt source that got implemented was the virtual timer that is used by the Linux guest OS for scheduling. Due to reasons explained in the following section, the interrupt of the virtual timer needs to be shared by different VMs and thus is handled directly within the kernel. Therefore, the timer interrupt either had to be injected by the kernel directly or its occurrence had to be signaled to the VMM, which in turn would inject the interrupt via the shadowed GICH registers. The first approach requires synchronization between kernel and VMM regarding the GICH register set of the VM, which is undesirable as the kernel must never depend on user-land behavior. On the other hand, as we show below, it is not sufficient to inject the interrupt when a device is handled by the VM directly, like in the case of the timer.

Each time a device interrupt is received by Genode's host kernel, the kernel masks the interrupt at the distributor until the corresponding device driver handled it. So the interrupt does not re-occur in the mean time. In a monolithic kernel, one can omit this masking. The kernel calls the top half handler of the interrupt, which is a small routine that acknowledges the interrupt directly at the corresponding device so that the interrupt signal at the interrupt controller vanishes. In a micro-kernel-based system like Genode where device drivers run like normal applications unprivileged in user land, this is not the case. It is in general not prior known when a device driver will have handled a related interrupt at the device. Therefore, the kernel for the time being masks the interrupt at the controller until the associated device driver signals that the interrupt got handled. For the virtualization case, this means if a VM uses a device directly by itself, for instance the virtual timer, the completion of the interrupt handling needs to be recognized. Otherwise the host kernel could not unmask the corresponding interrupt again. Fortunately, ARM has made provisions for this situation with the virtualization extensions. If configured accordingly, an injected interrupt can trigger a special maintenance interrupt as soon as the guest OS marks that interrupt as handled.



Interrupt delivery of a device dedicated to the VM.

To sum up, the VMM implements the interrupt handling of the VM. Whenever an interrupt of a physical device under control of the VM occurs, the kernel signals that interrupt to the VMM. The VMM injects the interrupt via the GICH registers in the VM state and resumes the VM. The kernel, in turn, reloads the GICH registers when switching to the VM. Within the VM the interrupt occurs. The guest OS kernel handles the interrupt at the related device and acknowledges the interrupt at the interrupt controller's Virtual CPU interface. This triggers a maintenance interrupt at the hypervisor, which forwards it to the VMM. The VMM responds to the maintenance interrupt by acknowledging the original interrupt at the kernel. The kernel unmasks the interrupt again and the whole process is completed. The sequence diagram in the figure above illustrates the delivery of a device interrupt dedicated to one VM. Assuming this procedure occurs while the VM is running, it will take four world switches and at least four kernel-VMM switches until the VM is able to proceed. This indicates high interrupt latencies.

Although the overhead of handling interrupts that are directly assigned to a VM seems to be quite significant, we decided to start with it and optimize later if necessary. The advantage of this pure architecture is that the hypervisor does nothing more than saving and restoring the registers of the Virtual CPU control interface from the VM state. Thereby, we strictly follow the principle of minimizing the common TCB.

The VMM controls the GICH registers and injects interrupts whenever necessary. If an interrupt shall be associated with a VM directly, the VMM gains access to it like any other device driver would do in Genode: It opens an IRQ session at the core process. In contrast to that, virtual timer and maintenance interrupts of the interrupt controller cannot be obtained via an IRQ session. They are shared between all VMMs. When they occur, only the VMM of the currently active VM shall receive it. Such an interrupt is signaled by the hypervisor to the VMM implicitly via the VM state. When the VMM recognizes that the VM was stopped due to an interrupt, it reads the interrupt information out of the VM state.

In contrast to devices under direct control of a VM, a VMM might provide emulated devices. These devices normally use services of the host system as backend. Section Interacting with the guest OS describes an example in form of an UART device that uses Genode's terminal service. If such emulated devices need to inject interrupts into the VM, the process is less complex. The successful handling of such virtual interrupts in contrast to physical ones does not need to explicitly signaled to the VMM.

Virtual Time

As has been mentioned in the previous section, one of the first devices that are used by the Linux kernel during initialization is the timer. Virtualizing time is a critical operation. When implementing a timer by the trap-emulate approach, significant performance penalties due to the frequent access are inevitable. To circumvent these foreseeable problems, ARM added virtualization support for its core-local generic timer, which is accessed via co-processor fourteen (CP14).

When starting to enable ARM's generic timer, we encountered serious problems. When originally enabling support for the ARNDALE board, we failed to use ARM's timer but used Samsung's own multi-core timer instead. The documentation of Samsung's Exynos 5 SoC, the one used in the ARNDALE board, does not even mentioned the generic timer at all. By following an interesting discussion on the ARM Linux kernel mailing list, involving a developer from Samsung, it turned out, that Samsung's multi-core timer and ARM's generic timer in fact use the same clock on this SoC and that certain bits in the Samsung timer need to be enabled so that the ARM timer runs successfully. Another minor issue was the frequency the timer runs at. It needs to be configured in the secure world. After sailing around these cliffs, we could finally continue examining the timer's virtualization support.

ARM's generic timer is extended by a virtual timer, which is coupled with the normal physical one. It counts with the same frequency but has an offset with respect to the physical reference that can be set. When switching between different virtual machines, the register containing the actual counter value and the control register of the virtual timer need to be saved and restored. Apart from that, the VM has direct access to the counter and always gets actual values. No trapping is needed.

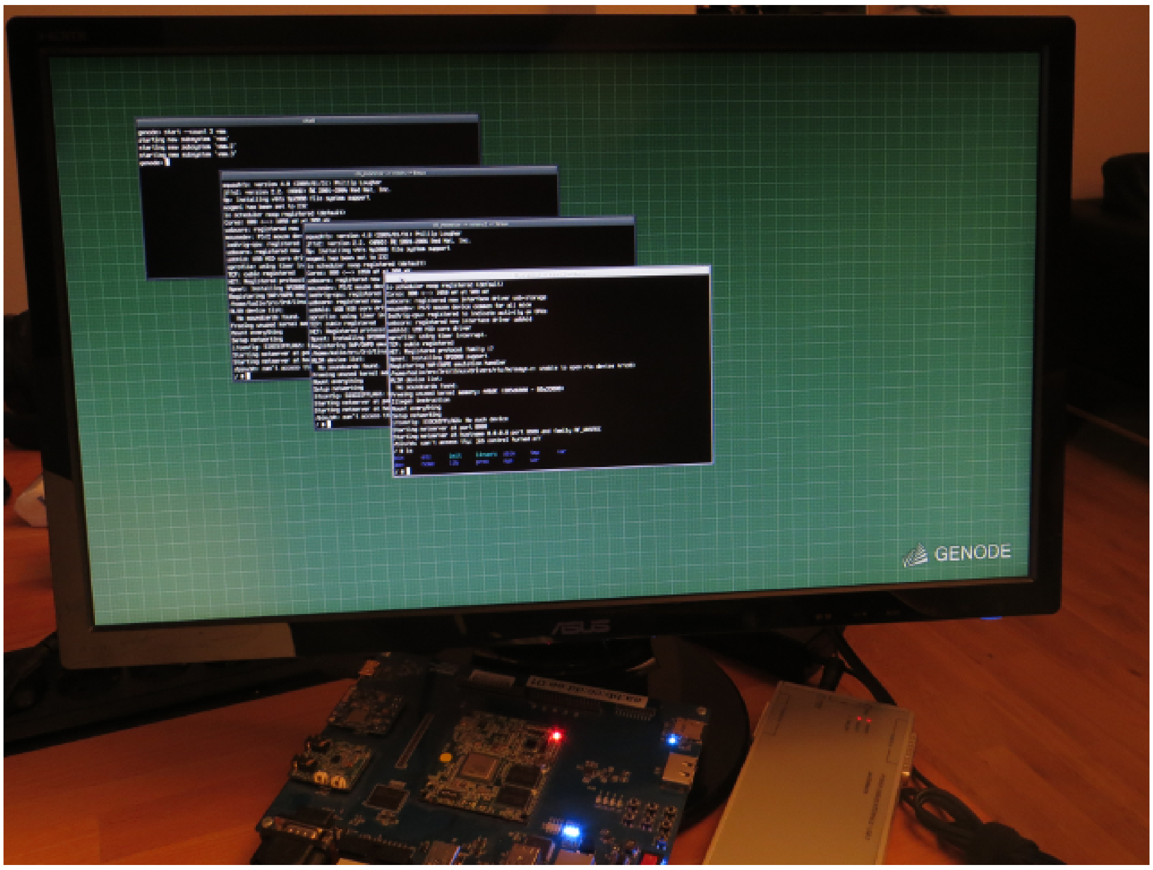
When the counter of the virtual timer reaches zero, an interrupt occurs, if this is enabled in the control register. However, in contrast to other devices that can be associated with exactly one device driver or VM, the virtual timer is potentially shared among several VMs. Therefore, the virtual timer interrupt plays a special role. It should be delivered only to the VMM with the currently active VM. This process was described in depth in the previous section.

If a VM gets interrupted, it's virtual time counter is stored and the virtual time stops. This is acceptable for relatively short periods of time, e.g., during the handling of a virtualization event. But if a VM gets preempted permanently, for instance after the guest OS signaled that it waits for interrupts, virtual time needs to proceed. At this point the ARM's virtual timer hardware is of little use. We cannot program the timer for an inactive VM and use it for another active VM at the same time. Instead, another time resource needs to be used to monitor time progress of an inactive VM. Therefore, whenever the VMM recognizes that the VM will stop execution for a longer time, it uses the last virtual timer counter value from the VM state to program Genode's timer service. When the timer service signals that the time period expired, the VMM resets the timer counter and injects a virtual timer interrupt.

Interacting with the guest OS

An essential device model needed during the first steps was the UART. At first, we implemented a very simple device model in the VMM without interrupt support that did not provide any characters to the VM (RX direction) but merely printed characters transmitted by the VM (TX direction). By enabling early debug messages in the Linux kernel, it was possible to again and again compare these messages with the output of the Linux kernel running within QEMU. This helped a lot to detect problems by identifying differences in the output.

After the boot process was finished, we liked to use the UART device to first communicate with the guest OS interactively. Therefore, the device model was extended and the former print backend was replaced by Genode's terminal service. By injecting an interrupt each time a character is received from the terminal and trapping each time the VM tries to read from or write a character to the UART registers, we gained a functional though slow serial console. This performance limitation, however, is not special to our virtualization approach but due to the working of UART devices, which are ungrateful for virtualization in general.


Three Linux serial consoles running in parallel on top of Genode

I/O MMU for device virtualization

When hosting virtual machines, the direct assignment of a physical device such as a USB controller, a GPU, or a dedicated network card to the guest OS running in the virtual machine can be useful in two ways. First, if the guest OS is the sole user of the device, the direct assignment of the device maximizes the I/O performance of the guest OS using the device. Second, the guest OS may be equipped with a proprietary device driver that is not present in the host OS otherwise. In this case, the guest OS may be used as a runtime executing the device driver and providing a driver interface to the host OS. In both cases the guest OS should not be considered as trustworthy. In contrary, it bears the risk to subvert the isolation between components and virtual machines. A misbehaving guest OS could issue DMA requests referring to the physical memory used by other components and even the host OS kernel and thereby break out of its virtual machine.

Accompanied with the virtualization support, ARM introduced an "I/O memory management unit" (IOMMU) fitting their architecture that is called "System MMU" (SMMU). In general, an IOMMU translates device-visible virtual addresses to physical addresses analogously to traditional MMUs used by a CPU. It helps to protect against malicious or faulty devices, as well as device drivers. Moreover, it simplifies device virtualization. When a device dedicated to a specific guest OS initiates DMA transfers, it will potentially access wrong address ranges. A DMA capable device is programmed by the guest OS' driver to use certain physical addresses therefore. But the guest-physical memory view differs from the actual host-physical one. Here the IOMMU comes into play. It translates the device memory requests from guest-physical to host-physical addresses.

While exploring the ARM virtualization support, the following question arose: How to provide direct device access to a virtual machine (VM)? From our prior experiences, we had a notion of how to separate device drivers from the rest of the system via IOMMUs of the x86 platform. But on ARM, we had no platforms equipped with such technology yet. Therefore, a further question was: How can ARM's IOMMU be integrated in a microkernel-based operating system like Genode?

As it turned out, the Samsung Exynos5 SoC of the ARNDALE board doesn't make use of ARM's SMMU but uses its own derivative SysMMU. Nevertheless, given the reference-manual description, both IOMMUs are quite similar. Although Samsung's SysMMU seems to be less complex than the ARM variant.

Guest OS DMA attack

Before exploring the SysMMU available on our development platform, we decided to implement an attack used as test vehicle. A simple driver that uses a DMA capable device to read or override Genode's kernel binary seemed to be an appropriate example. When looking for a proper device, we naturally hit on the DMA engine available on the Samsung

SoC. This DMA engine is used by some peripherals that do not have a dedicated DMA engine on their own. Moreover, it includes several DMA channels to perform memory-to-memory copy transactions. Therefore, it seemed to perfectly fit our demands. However, after studying the functional description of the DMA engine superficially, it turned out to be more complex than anticipated. As we did not want to implement overly complex test drivers, an alternative approach was chosen. Given our minimal virtualization environment, we were able to run a simple Linux guest OS within a VM. If we would extend the virtual environment with direct access to the DMA engine, the Linux kernel could perform the attack, as it already contains a driver for the corresponding device.

The first step was to add the DMA engine to the "device tree binary" (DTB) of the Linux guest. As described in section CPU Virtualization, the virtual environment delivered by the VMM differ from the underlying hardware platform. Instead of providing a virtual ARNDALE board, we use a kernel and minimal DTB built for ARM's Versatile Express Cortex A15 development board. Luckily, the original DTB hardware description of this board already contained the same DMA engine like the one on the ARNDALE board. Therefore, we only needed to add the original DMA-engine description to our minimal DTB again.

After booting the Linux guest OS, the kernel was not able to initialize the DMA engine. As it turned out, the platform driver of the Genode host OS, responsible among others for clock and power domains, disabled the device when initializing clock and power controllers. After fixing this little problem, the kernel was finally able to identify and initialize the DMA engine.

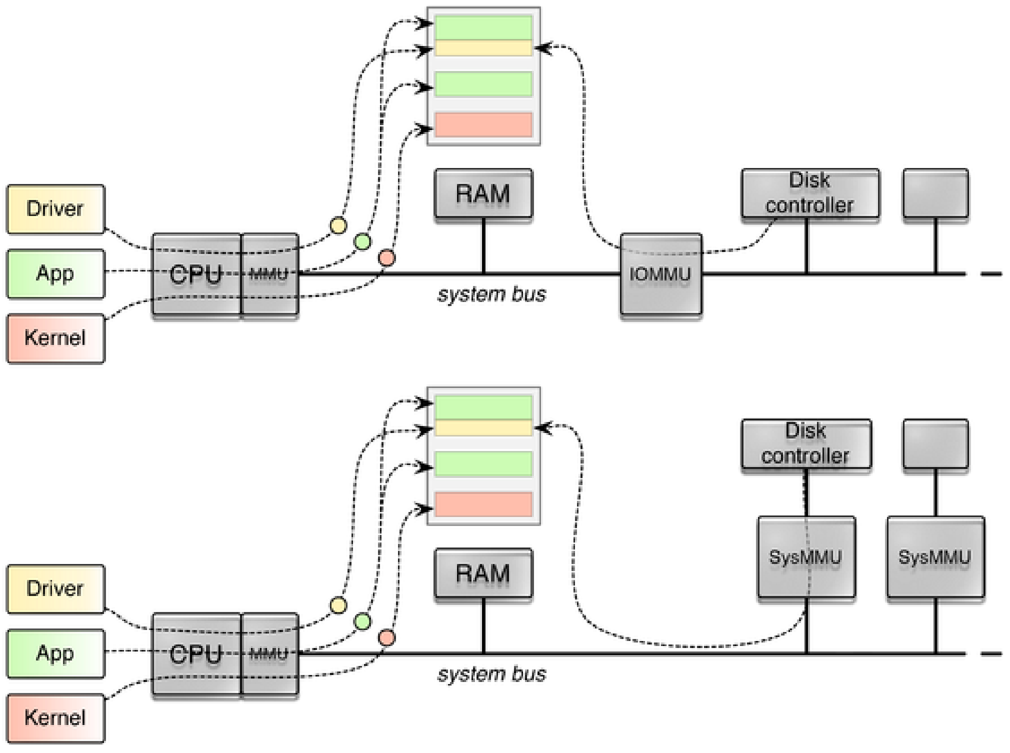
To execute the actual attack, we adapted a DMA engine debug test that is already present in the Linux kernel. It uses the generic DMA API of the kernel to run a number of tests on the corresponding engine. We altered the test module in a way that it tries to copy the Genode OS host kernel to some empty Linux guest memory. At that stage, we had no IOMMU support in place. Therefore, we calculated the offset of guest-physical to host-physical memory to the addresses handed over to the DMA API. Nevertheless, the expected effect did not occur. The target memory always remained untouched. But the DMA engine reported that it completed the transaction successfully. After further inspecting the DMA engine's device driver of the Linux kernel, it turned out that the engine fetches micro-code from memory to execute its operations. The micro-code is written by the driver, and its physical address is provided to the device by the driver too. In other words: to be able to execute a DMA transaction, the engine already initiates a DMA transaction before.

After applying the guest-to-host physical memory offset to the instruction pointer's address of the DMA engine device driver, the attack finally worked as expected. We were able to read out and overwrite the host OS kernel from the guest OS.

Limiting DMA transactions

To prevent DMA attacks driven from the guest OS, the next step was to put the SysMMU into operation. As it turned out, there is no single SysMMU in the SoC, but a bunch of different ones sitting in front of different devices. As the the SysMMU is not capable to differentiate requests from its source bus, the SoC is required to use a dedicated SysMMU in each case a device shall be separated from others. On the other hand, the use of one SysMMU per device allows for a largely simplified SysMMU compared to Intel's IOMMU.

On Intel, the IOMMU relies on the PCI device's bus-device-function triplet as unique identification for DMA transactions, and it can manage different translation tables for different devices. Without having practical experiences and by only studying the reference manuals, ARM's SMMU works similar and can distinguish different devices according to the AXI bus ID. The different approaches are depicted in the following figure.



IOMMU approaches: Intel VT-d (above) and Samsung Exynos5 (below)

The Samsung Exynos5 SoC that we used for our experiments comprises more than 30 SysMMUs in it. We identified the one situated between that part of the DMA engine, which is responsible for memory-to-memory transactions, and the memory controller itself.

To test that we target the right SysMMU, we configured it to block any request. Samsung's SysMMU has three different modes: "disable", "enable", and "block". In disable mode it just lets all transactions pass without any translation. This is the default when the system was reset. In block mode the SysMMU just blocks any request from the source bus. In enable mode it uses a translation table to look up device-virtual to physical translations, and a TLB to cache the mappings. Putting the DMA engine's SysMMU into block mode led to timeouts of all DMA transactions of the Linux guest OS, which met the expected behavior.

The next step was to put the SysMMU into enable mode. Thereby, we first did not define a valid translation table. To be informed about translation faults or other failures, each SysMMU provides a dedicated interrupt. When trying to determine the interrupt number of the DMA engine's SysMMU, a new issue arose. In contrast to most other ordinary device interrupts, the ones originating from the SysMMUs are grouped. That means they partially share the same interrupt line. To be able to distinguish the different sources within one interrupt group, a simplified interrupt controller for each group exists, the so called interrupt combiner. The interrupt combiners are connected to the global interrupt controller. Genode's integrated ARM kernel did not made use of these combiners, as they were not necessary for the device drivers already present. Unfortunately, it is not enough to merely enable the interrupt combiners. A namespace of virtual interrupt numbers have to be introduced to consider the different interrupts of one group. Instead of solving that problem by design, we chose the pragmatic path. At this point, we only enabled the interrupt for the corresponding SysMMU and ignored all other interrupts of the same group. The interrupt is primarily used for error detection and not necessarily needed for the correct functioning of the SysMMU. Therefore, it seemed feasible to use an interim solution for the time being.

Now that interrupts could principally be delivered to the SysMMU driver, the device signaled a bus error. This was expected. As we did not set an address to a translation table for the device, it used an invalid bus address to access its table. After setting up a valid portion of memory initialized with zeroes and propagating its physical address to the SysMMU as translation table, we finally received an interrupt indicating a translation-fault correctly. Finally, we had to set up a valid translation table containing the guest-physical to host-physical translations of the Linux guest OS memory. The translation table format of Samsung's SysMMU is a compatible but simplified form of ARM's two-level page-table format. In contrast to the original model, the SysMMU derivative does not include some memory attributes such as cache-policy attributes. After putting the populated translation table into effect, the Linux guest OS was finally able to do DMA memory-to-memory transactions within its own memory areas, but failed to access ranges outside of it. Moreover, all patches to the Linux kernel's DMA driver, regarding the offset calculation of guest-to-host physical memory, naturally became superfluent.

In the current example implementation, the driver for the DMA engine's SysMMU was implemented as part of the VMM. This was done for the convenience of implementation. Nevertheless, in the final design of a secure virtualization environment, we do not want to trust the VMM more than any other application or driver. Since we identified the setup of the SysMMU translation table to be critical to all programs and devices of the system, the programming of the SysMMUs has to be done by trusted components exclusively. Therefore, we aspire to incorporate IOMMUs for ARM within Genode's trusted platform driver in the future. As all device drivers in Genode depend on that driver, it is the natural place for implementing DMA protection. Note that the SysMMU handling is still outside the host kernel but implemented in user mode.

Outcome

The present work examines ARM's virtualization extensions as well as I/O protection mechanisms. It shows how it can be combined with a component-based OS architecture like Genode. The resulting implementation is a proof-of-concept and not yet incorporated in the mainline Genode OS framework. Although it is still a prototype, we can already draw some conclusions.

In our work, we followed the paradigm of minimizing the common TCB as far as possible by implementing nearly all virtualization-related functionality into the unprivileged VMM. The complexity of the privileged hypervisor is almost negligible. In fact, Genode's core/kernel had been supplemented by merely 600 lines of code (LOC) to support this virtualization architecture.

The further addition of features to the VMM such as sophisticated device models will not increase the complexity of the common TCB. The only omission of the current version of the hypervisor is the lack of on-demand switching of FPU contexts when switching between VMs and Genode. However, as the base-hw platform already comprises support for FPU switching for normal Genode applications, it is assumed that FPU virtualization won't add much complexity.

Another open issue is support for symmetric multi-processing (SMP) in VMs. In general, SMP is already supported by the base-hw platform on Cortex A15 CPUs. Having more than one CPU in a VM is mostly a question of enhancing the VMM accordingly. We envision to represent each virtual CPU by a dedicated VM session. The VM objects implemented in the kernel are almost identically to normal threads with regard to scheduling and can be assigned to different cores. For this, no additional code has to be implemented in the kernel. Apart from that, the VMM has to guarantee synchronicity with regard to commonly used device models when dealing with several VM sessions.

Most our the ARM-virtualization efforts described throughout this article (except for the I/O protection support) have been integrated into the version 15.02 of the Genode OS framework.

Acknowledgements

The study was conducted by Genode Labs GmbH during the year 2014. The work was supported by funding from the German Federal Office for Information Security (BSI). We thank the BSI for the opportunity to explore and document technologies like ARM virtualization in the level of detail as presented.