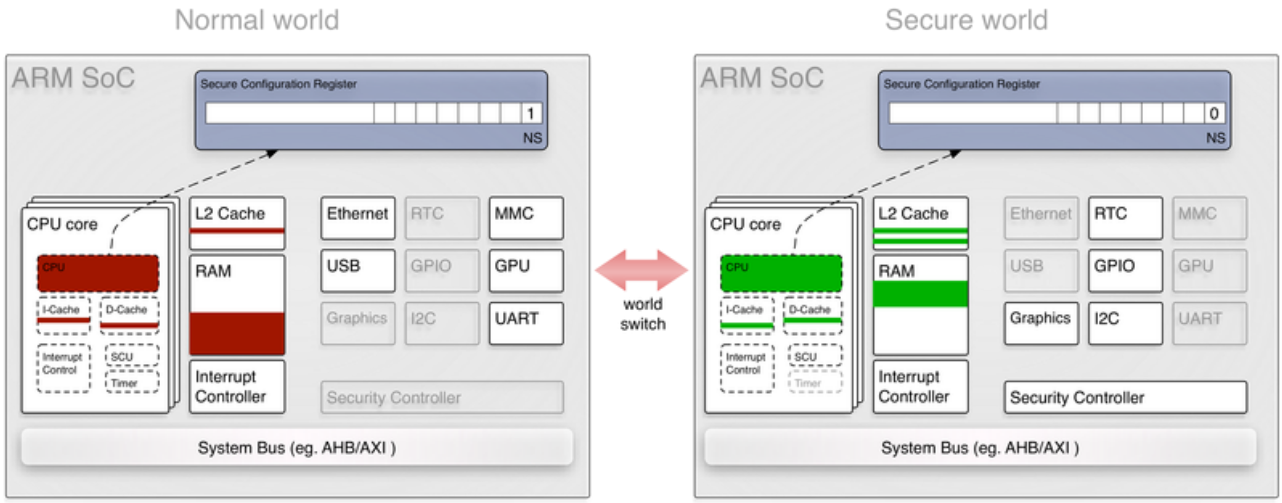
An Exploration of ARM TrustZone Technology

ARM TrustZone technology has been around for almost a decade. It was introduced at a time when the controversial discussion about trusted platform-modules (TPM) on x86 platforms was in full swing (TCPA, Palladium). Similar to how TPM chips were meant to magically make PCs "trustworthy", TrustZone aimed at establishing trust in ARM-based platforms. In contrast to TPMs, which were designed as fixed-function devices with a predefined feature set, TrustZone represented a much more flexible approach by leveraging the CPU as a freely programmable trusted platform module. To do that, ARM introduced a special CPU mode called "secure mode" in addition to the regular normal mode, thereby establishing the notions of a "secure world" and a "normal world". The distinction between both worlds is completely orthogonal to the normal ring protection between user-level and kernel-level code and hidden from the operating system running in the normal world. Furthermore, it is not limited to the CPU but propagated over the system bus to peripheral devices and memory controllers. This way, such an ARM-based platform effectively becomes a kind of split personality. When secure mode is active, the software running on the CPU has a different view on the whole system than software running in non-secure mode. This way, system functions, in particular security functions and cryptographic credentials, can be hidden from the normal world. It goes without saying that this concept is vastly more flexible than TPM chips because the functionality of the secure world is defined by system software instead of being hard-wired.



The figure above illustrates the concept of the two worlds. The normal world is active (non-secure bit is set), the OS running on the platform can only access a subset of the physical resources. When a world switch takes place, the secure world comes into effect. The system software running in the secure world can access the devices hidden from the normal world.

As former university researchers in the field of OS security, ARM TrustZone was deemed as relevant to us. We still have fond memories of a joint research project called ROBIN with STMicroelectronics where we could see the use of TrustZone on an ARM1176-based prototype platform for co-hosting Linux with an L4-based secure OS. However, since the project ended in 2008, we haven't heard much about this technology for several years. TrustZone seemed like one of so many obscure processor features that, once introduced into products, nobody could get really excited about. Well, in the case of TrustZone this is actually not true. In 2012, we were approached from different angles with questions about ARM TrustZone. It looked like, by then, TrustZone had become a kind of marketing term used by SoC vendors to sell confidence in the security of their chips. The term "TrustZone" appeared to be quite cloudy and it carried possibly wrong expectations and misconceptions with it. A lot of questions popped up: Does TrustZone provide mechanisms for secure booting and secure storage? Isn't it a kind of virtualization technology? If yes, isn't it superseded by ARM's virtualization extensions? How does it work? Is it important to consider it when developing an operating system?

These questions prompted us to dive right into the world of TrustZone. We spent two and half years on and off with experimentation on several ARM platforms, each with a quite different interpretation of TrustZone-based security. This article summarizes our findings.

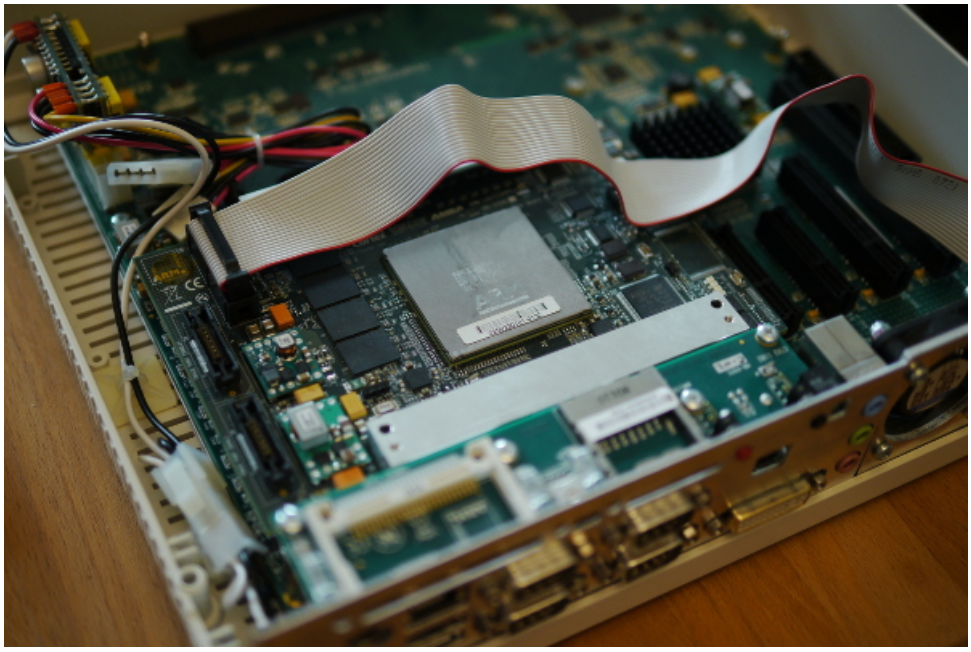
Section Starting point describes the starting point of our journey and lays out the possible routes we considered to take. It is followed by Section The design and genesis of our custom base-hw kernel platform, which gives the rationale behind creating a custom kernel platform to support our undertaking. Section Hypervisor managing the non-secure world goes into detail about the TrustZone-specific challenges we faced while building a prototype. The actual prototype is covered by Section TrustZone demonstrated in color. Finally, the article summarizes our findings in a Q&A style in Section Common questions, answered.

Starting point

The first question on the table when starting a series of experiments was the one for a suitable development platform.

At the time we started our investigations, we were most interested in the ARM Cortex-A9-based SoCs. All Cortex-A9 cores are equipped with ARM's security extensions (often referred to as TrustZone capabilities). In principle, any Cortex-A9 based platform would be suitable for us. However, even though there are many low-cost ARM development boards on the market

(e.g., by Samsung, TI, ST-Ericsson, NVIDIA, ZiiLABS), we discovered that none of those options provided access to the secure mode of TrustZone. In almost all cases, bootstrap code stored in the ROM switches to non-secure mode prior starting the boot loader, possibly to prevent access to certain parts of the SoC that are not intended for public use. This narrowed the potential base platforms to FreeScale's i.MX development boards and the ARM Versatile Express platform. The FreeScale i.MX platform places no restrictions on the use of the secure mode. But at the time we started, no Cortex-A9-based board was available. The most recent board i.MX53 featured a single-core Cortex-A8 processor. The ARM Versatile Express Cortex-A9 board is the official reference board by ARM, which supports ARM TrustZone. As expected, the official reference board comes with a price tag far north of your usual low-cost development board. It's not that it is an order of magnitude more expensive. It's actually two orders! Still, we were happy to get our hands on one of those and could kick off our line of experiments on the official reference platform.



We conducted two lines of experimentation: Prototyping using an existing microkernel and the creation of a custom kernel platform. The primary motivation for the former line of work was to have a low-risk path to enable the reference hardware and to get acquainted with the principle use of ARM Cortex-A9. By taking and modifying an existing kernel platform that is known to work on this CPU core, we had a good starting point. However, to get a thorough understanding of TrustZone's mechanisms, we needed to tackle problems that lay in the scope of the kernel. Furthermore, we desired to be in complete control over the bootstrapping procedure, in particular to address questions about secure booting. Rather than patching an existing kernel and inheriting its design choices, we decided to create a custom kernel platform that gives us all the freedom needed to explore the platform without having to consider the complexities of an existing kernel implementation.

For both lines of work, we used the Genode OS Framework (Genode) as foundation. Genode is a construction kit for building special-purpose operating systems. It is a collection of small building blocks, out of which complex systems can be composed. Those building blocks include not only applications but all classical OS functionalities like kernels, device drivers, and protocol stacks. Currently, Genode supports 8 different kernels and provides over 100 reusable components for both x86-based and ARM-based platforms. We figured that its low trusted computing base (TCB) complexity yet high flexibility would make it an attractive foundation for an OS designated to run on the secure side of TrustZone.

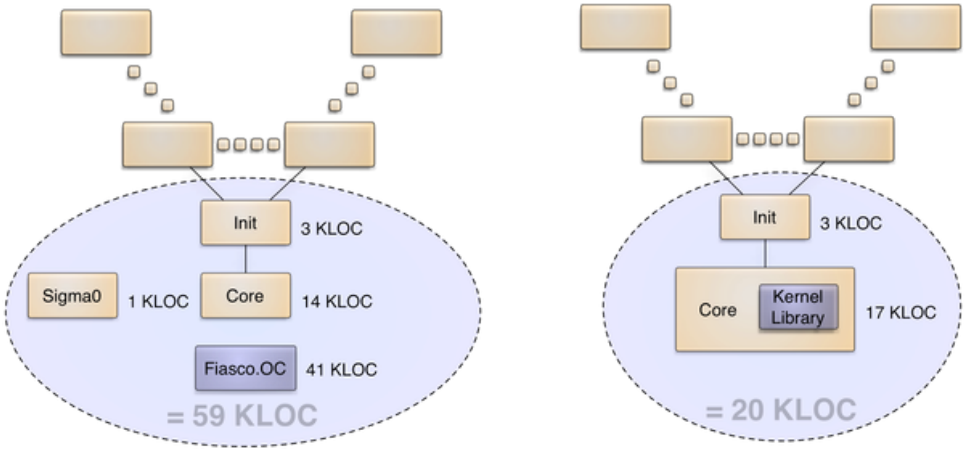
For the first line of work, the enablement of the Versatile Express platform, one of the kernels L4/Fiasco, Fiasco.OC, or Codezero were principally suitable. Of these candidates, Fiasco.OC provided the broadest support for the most recent ARM SoCs. Hence, we decided to use the combination of Genode with Fiasco.OC as starting point to enable the principal use of Genode on the platform. This required us to implement low-level driver support for basic peripherals such as the PL110 display, the PL011 UART, timer, and PS/2. Even though the driver-related work was conducted using the Fiasco.OC kernel, the outcome was beneficial for both lines of work Fiasco.OC and our custom kernel as Genode's driver components are independent from the underlying kernel.

In our second line of work, we supplemented Genode with a new base platform that we call "base-hw" ("hw" representing Genode running on bare hardware). In contrast to the already supported kernels, this new platform is tightly integrated with the core of Genode. Thereby, we aimed to dramatically reduce the TCB complexity of the base system compared to the use of a discrete kernel. The most significant benefit of the custom kernel platform in the context of our TrustZone experimentation are the insights into kernel-level problems when using the Cortex-A9 and TrustZone.

The design and genesis of our custom base-hw kernel platform

In contrast to classical L4 microkernels where Genode's core process runs as user-level roottask on top of the kernel, base-hw executes Genode's core directly on the hardware with no distinct kernel underneath. Core and kernel are melted into one novel kind of hybrid kernel/userland program. Only a few code paths are executed in privileged mode but most code runs in user mode. This design has several benefits. First, the kernel part becomes much simpler. For example, there are no allocators

needed in the kernel part because allocators are managed by the user-level part of core. Second, base-hw side-steps long-standing hard kernel-level problems, in particular the management of kernel resources. For the allocation of kernel objects, we can simply employ Genode's user-level resource trading concepts. Finally and most importantly, however, merging the kernel with roottask removes a lot of redundancies between both programs. Traditionally, both kernel and roottask performed the book keeping of physical-resource allocations and the existence of kernel objects such as address spaces and threads. In base-hw, those data structures exist only once. The complexity of the combined kernel/core is significantly lower than the sum of the complexities of a traditional self-sufficient kernel and a distinct roottask on top. This way, base-hw helps to make Genode's TCB less complex.



The figure illustrates the effect of the new design on the TCB of the root of Genode's process tree. On the left, a traditional member of the L4 family of kernels is depicted. The bluish marked TCB comprises the kernel, the sigma0 root memory manager, roottask (Genode's core), and Genode's init process. These components accumulate to circa 60 thousand lines of code. On the right, the hybrid core/kernel approach of base-hw is illustrated. By merging the kernel with roottask, systems running on top of base-hw need to trust less code to be void of bugs.

For realizing the base-hw kernel platform, we undertook the following steps. First, we designed and implemented the concept for bootstrapping a Genode-based system consisting of potentially many modules as a single boot image. This is the first point where using a custom platform over a stock Fiasco.OC kernel turned out to be beneficial because our boot concept is extremely simple compared to the procedure known from Fiasco.OC.

The kernel entry and exit code paths had been implemented and tested on both Qemu and the reference hardware. The execution model of the kernel can be roughly characterized as a single-stack kernel. In contrast to traditional L4 kernels that maintain one kernel thread per user thread, the base-hw kernel is a mere state machine that never blocks in the kernel. This is similar to how the seL4 kernel and the NOVA microhypervisor operate.

The next step was to add support for interrupt handling and preemptive multi-threading. So we implemented drivers for the PL390 interrupt controller and the Cortex-A9 core timer. Combined with the implemented kernel-entry code path and the interrupt controller, timer interrupts could be handled by Genode's user-level core. Genode's core component uses multiple threads of execution:

- The main thread that initializes the platform and spawns the first user-level process called init,
- A pager thread that receives and handles page faults,
- One thread per device interrupt,
- A so called RPC entrypoint that handles requests to the services provided by core, and
- The parent thread of the init process that responds to requests of the init process via the parent interface.

Because the correct functioning and scheduling of those threads is fundamental for bringing-up of the system, the next logical step was the introduction of a preemptive scheduler and core-local context switching. Core-local threads do not run independently but interact with each other via synchronous inter-process communication. For example, the main thread creates the RPC entrypoint and then acts as a client of some core services. To accommodate those use cases, the kernel interface was supplemented with system calls for synchronous inter-process communication (IPC). To keep the kernel as simple as possible, IPC is performed using so-called user-level thread-control blocks (UTCB). Each thread has a corresponding memory page that is always mapped in the kernel. This UTCB page is used to carry IPC payload. The largely simplified procedure of transferring a message is as follows. (In reality, the state space is more complex because the receiver may not be in a blocking state when the sender issues the message)

1. The user-level sender marshals its payload into its UTCB and invokes the kernel,
2. The kernel transfers the payload from the sender's UTCB to the receiver's UTCB and schedules the receiver,
3. The receiver retrieves the incoming message from its UTCB.

Because all UTCBs are always mapped in the kernel, no page faults can occur during the second step. This way, the flow of execution within the kernel becomes predictable and always returns to the user land.

In addition to IPC, threads interact via the synchronization primitives provided by the Genode API. To implement these portions of the API, the kernel was then enhanced with system calls for managing the execution control of threads.

With the entity of core running on the reference platform, it was time for spawning the first non-core user-level process, namely the init process. This process is a composition out of multiple core sessions:

- A CPU session with the main thread of the init process,
- A region-manager (RM) session describing the address space of the process,
- A protection-domain (PD) session representing the encapsulation boundary of the process,
- A RAM session containing the physical memory resources available to the init process.

To accommodate the implementation of these services, the kernel was enhanced with system calls that handle protection domains and page faults at user level.

At the startup of the init process, its first life sign is a page fault produced by the main thread on the attempt to fetch its first instruction. The kernel translates this page fault into an IPC message to the pager thread. In contrast to core-internal IPC, this cross-PD-IPC requires the kernel to not only switch thread contexts but also address spaces.

All further functionalities needed to bring up the init process such as the ELF loading were readily provided by the generic code of the Genode OS Framework.

The last step towards executing real-world application scenarios on our custom kernel was the provisioning of mechanisms required by user-level device drivers. Those mechanisms are memory-mapped I/O access and IRQ delivery, which are provided via core's IO_MEM and IRQ services. To implement those services, the kernel had to be supplemented with system calls for allocating and receiving IRQs.

With those functionalities in place, the path was cleared to execute user-level device drivers that we already had enabled when using the Fiasco.OC kernel, i.e., drivers for the user-level timer, framebuffer, and PS/2 input devices. Thereby, we became able to run a basic graphical interactive demo scenario on our custom kernel.

For more sophisticated work loads that require asynchronous communication, we subsequently improved the kernel mechanisms. For example, we added kernel-level support for Genode's signalling API and thereby achieved full coverage of the Genode API on our custom kernel platform.

Hypervisor managing the non-secure world

This section describes our experiences while creating a TrustZone-aware hypervisor that is able to schedule between one non-secure virtual-machine (VM), and multiple tasks running unprivileged in secure mode. Based on our custom base-hw kernel platform, it describes the consecutive steps that were taken to turn the kernel into a TrustZone-aware hypervisor, as well as to build a virtual-machine monitor (VMM) running as user-level component on top of that kernel.

World switch between non-secure world and secure world

We started investigating TrustZone by bringing the so called monitor-mode in the Genode/ARM-kernel to life. The monitor mode is an execution mode add-on in ARM CPUs implementing the Security Extensions (SE is equivalent to TrustZone). It is an execution mode (like e.g., system, user, or supervisor mode), the CPU switches to when certain exceptions occur. For most of the exceptions, the CPU can be configured whether it raises an exception in monitor-mode, or in the corresponding exception mode of the non-secure or secure-world. A so-called Secure Configuration Register (SCR) enables the hypervisor to configure whether the CPU should trap when fast interrupts, or normal interrupts, or external data aborts occur, while the non-secure world is active.

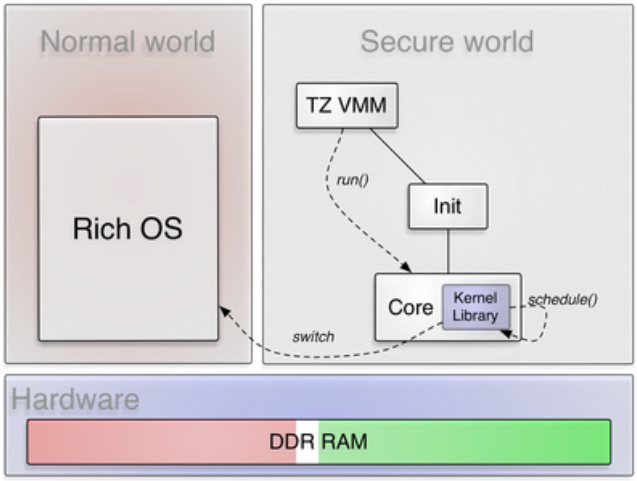
In addition to interrupt-triggered mode switches, the non-secure world is able to explicitly enter the monitor mode by the means of a software-generated exception via the `smc` instruction. So the first step was to initialize a corresponding exception vector that is used when a monitor-mode exception is raised.

Thereby, our consideration was that all exceptions that enter the monitor mode should be triggered by the non-secure world only, but never by the secure world. Given that assumption, the we designed the exception vector in a way that it always stores the current CPU state as a non-secure world's state. The CPU state is saved regardless of the type of exception. In addition to the CPU state, the exception type is stored in the same area. On the switch to the monitor mode, the kernel reloads the formerly stored secure kernel-context from a known memory area. Given this approach, the software stack running in the secure world must not execute the `smc` instruction. This drastically simplifies the assembler world-switch routine. In application scenarios where a subset of components executed in the secure world are untrusted, it is principally possible to extend the world-switch routine to check for the world that raised the fault. In general, the monitor-mode exception vector works orthogonal to the kernel-/user-mode switch. In contrast to that, the world-switch has to save/restore more register banks. To switch from the secure to the non-secure world the symmetrical operations are also performed, only that this time, the world switch is not triggered by an exception but via a newly introduced system call with a pointer to the VM's CPU state. The kernel responds to this system call by pushing the provided state to the non-secure world.

User-level Virtual Machine Monitor (VMM)

To be able to use the newly introduced system call that switches to the non-secure-world, we introduced a new VM-session interface as service in Genode's core. This interface enables a client (VMM) to affect the whole CPU state of a VM, initiate a

world switch to the non-secure world, and, after an exception-triggered return, to obtain the VM's state. Moreover, the new interface enables a client to obtain a portion of physical RAM as I/O memory, so that it can prepare it as the VM's physical RAM. Moving this physical RAM portion into the I/O memory allocator of core allows the VMM to get the appropriate RAM portion to load the VM's ELF image to its physical relocation address.



The image above illustrates the relationship between the user-level VMM (TZ VMM) and Genode's core/kernel. TZ VMM initiates the switch to the normal world by invoking the run function of the VM session interface. The normal world is scheduled as a (low-priority) thread by the base-hw kernel.

Given the world-switch routine in the kernel and the VM-session interface of core, all prerequisites were in place to build a first simple version of a VMM. This version solely acquires some portion of RAM, takes a ROM module (the VM's image to load), and loads the ELF binary to physical RAM. During ELF-loading, it identifies the correct entry point, and sets the program counter of the VM's CPU state accordingly. Finally, the VMM enters an endless loop where it executes the VM and, after each exceptional return, dumps the VM's CPU state to the debugging console.

Simple test kernel for the non-secure world

To test the newly implemented world-switch routine, we created a simple test kernel was. It is built almost entirely out of assembler code, which performs the following steps:

- Setup exception vectors for data-aborts, pre-fetch aborts, interrupts, fast-interrupts, supervisor-calls,
- Initialize interrupt controller to receive UART0 interrupts,
- Initialize UART0 to be able to print simple messages, and detect interrupt reception (RX),
- Switch to user-mode and idle,
- Whenever an interrupt occurs print some short message.

TrustZone Platform Controller (TZPC) and Address-Space Controller (TZASC)

After successfully testing the VMM setup and the world-switch routine, it was time to investigate TrustZone mechanisms, which actually protect the secure world from the non-secure world. A set of different IP cores exists beside the CPU core, which helps to confine the non-secure software stack. With respect to the Versatile Express platform, these are the following:

- TrustZone Protection Controller (TZPC)
- TrustZone Address Space Controller (TZASC)
- The Cortex A9 MPCore internal Interrupt Controller

The first and easiest part seemed to be the configuration of the physical memory areas that should be preserved for the secure world but invisible to the non-secure world, i.e., the RAM where the secure software stack is located in. The bus and memory hierarchy in modern embedded architectures, like in the Versatile Express Motherboard/Daughterboard conglomerate, is highly complex. It consists of different bus systems (ABP, AXI), memory controllers (SMC, DMC), and caches. Most of the controllers and devices in this jungle are not TrustZone aware themselves but are protected by the TZPC or TZASC. Given the sparse public information available in technical reference manuals, it was hard to identify, which components can be protected and how the TZPC/TZASC are related to them.

Due to experimentation, we were able to deduct the following insights. The TZASC controller on the platform is used to secure physical address ranges that are addressable through the Static Memory Controller (SMC). In principal, it should be possible to secure another memory controller by a TZASC too, but on the platform, it is restricted to the SMC. These physical address regions correspond to the I/O resources of peripheral devices, some SRAM, and flash memory. Most of these components are

placed on the motherboard. The DDR RAM on the other hand is not subjected to the SMC, but sits behind the Dynamic Memory Controller (DMC), which is not protected by any TZASC. With the help of the TZASC, it is possible to define multiple regions as secure or non-secure. However, these definitions do not apply to DDR RAM.

The TZPC is used to protect on-chip peripherals (e.g., the TZPC and TZASC themselves) as well as bus accesses to external subsystems. Thereby, one can configure whether access to a corresponding on-chip device is allowed. For example, one bit in the TZPC is reserved to enable or disable access to the DMC from the non-secure world. The TZPC doesn't allow a more fine-grained confinement with respect to a single device, like for instance the DMC.

Given those findings, the Versatile Express platform apparently does not allow the partitioning of DDR RAM into secure and non-secure address ranges. One can only decide to assign it to either of both worlds. SRAM and flash-memory on the other hand, which are under control of the TZASC, can be configured more fine-grained. Furthermore, when running an unmodified Linux software stack in the non-secure world, we need to assign the entire DDR RAM to the non-secure world. Fortunately, the platform hosts 32 MiB of SRAM, which we can preserve for the exclusive use by the secure world.

Interrupts

In the previous section, we identified how to partition the physical resources of the reference platform into two worlds by assigning memory-mapped I/O (MMIO) resources of peripherals to either of both worlds. However, access to MMIO is only the half way towards the practical use of these peripherals. We also need a way to route the corresponding device interrupts to the respective worlds.

Unfortunately, the assignment of device interrupts to either the secure or non-secure world cannot be configured directly. Instead, it is merely possible to distinct the handling of normal interrupts (IRQ) and fast interrupts (FIQ). The ARM architecture defines these two types, which differ in the following ways:

- Each type has a unique exception vector so that FIQs enter the kernel using a different code path than IRQs.
- On the occurrence of an IRQ, the CPU automatically saves a more comprehensive register state compared to an FIQ.
- The most distinctive characteristic is that an FIQ can preempt any exception handler except for the FIQ handler. So exceptions can nest.

For a device interrupt, it is possible to define whether to take the IRQ or FIQ exception entry in the PIC configuration. This mechanism indirectly enables the partitioning of interrupts between both worlds by configuring the PIC to use FIQs for secure and IRQs for the non-secure world exclusively. Using the Secure Configuration Register, it is possible to revoke the use of FIQs from the non-secure world and force a trap into the monitor for each FIQ that occurs while the non-secure mode is active. To implement this idea, we modified our custom kernel to handle FIQs instead of IRQs, which is slightly more complicated because of the possible nesting of exceptions. We solved the problem of nested exceptions by checking for the originating CPU mode that was active when the FIQ occurred. If the CPU was already in kernel mode, we can conclude that the exception refers to a nested exception. In this case, we ignore the FIQ and fall back to resuming the previous exception.

To summarize the above, FIQs are exclusively used by the device drivers of the secure world whereas IRQs are exclusively used by the non-secure world (as expected by the Linux kernel). Because for each device, we can define whether its interrupts are delivered as FIQs or IRQs, we are thereby able to assign individual device interrupts to either of both worlds.

Booting Linux in the non-secure world

Having the principal separation between the secure and non-secure world in place, we can now replace our simple custom non-secure kernel with the Linux kernel. This work comprises the following steps:

- Loading the kernel. The kernel has to be complemented with additional boot-time information called ATAGs. The ATAGs data structure contains the location of the RAM, the address of the RAM disk as well as the kernel command line.
- In addition to the kernel, a RAM disk must be loaded into the memory of the non-secure world. This RAM disk contains the initial boot image.

To test the loading procedure, we started Linux with all access rights to the peripherals granted and observed the successful boot-up of the kernel. At this point, we disabled the access to all peripherals and entered the iterative process of booting the kernel, looking where it hangs because of a missing permission, and then taking one of the following decisions: For devices that we deemed as uncritical for the secure world, we would grant direct device access. For devices that must be preserved for the secure world, we slightly changed the Linux kernel code to issue a hypercall instead of accessing the device resource directly. When the non-secure OS issues a hypercall using the `smc` instruction, the CPU enters the monitor mode and passes control to the hypervisor. The hypervisor reflects that hypercall to the user-level VMM, which is able to respond to the individual hypercalls. Those hypercalls are:

SP810_ENABLE

: Enable the timer.

CPU_ID:

: Request the CPU ID, which is the hard-coded value `0x0c000191` for the reference platform.

SYS_COUNTER

: Returns the value of the `Sys_24MHz` system register.

MISC_FLAGS

: Returns the value of the `Sys_misc` system register.

SYS_CTRL

: Perform one of the system configuration control operations OSC1, DVI_SRC, and DVI_MODE.

MCI_STATUS

: Returns the value of the `Sys_mci` system register.

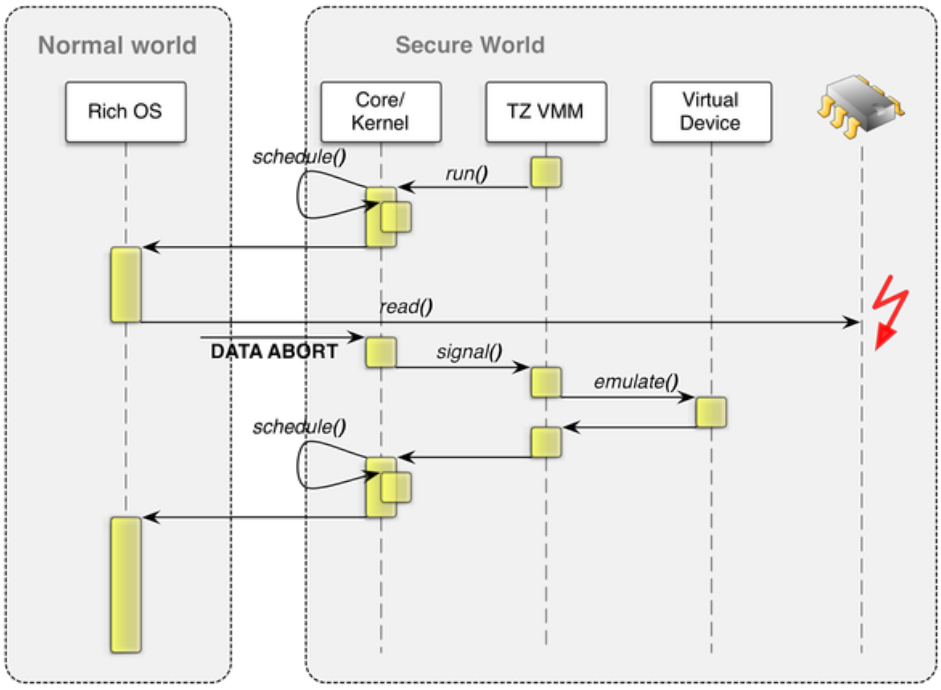
We would have preferred to employ a trap-and-execute emulation scheme for those register accesses. However, as described in the following section, this technique is not feasible with the mechanisms provided by TrustZone.

Device emulation

We considered device emulation as a less intrusive alternative to the introduction of hypercalls. In contrast to the hypercall approach, device emulation does not require us to change the Linux kernel.

The basic idea of emulating device access is to let the hypervisor pass control to the VMM as soon as the non-secure OS accesses an address outside the permitted physical address ranges. The VMM can then inspect the address in question and the program counter of the non-secure OS that raised the access violation. Given the program counter value, the VMM can fetch and decode the faulting instruction and emulate it in software. Because ARM is a RISC architecture, the instruction decoding is rather simple. The instruction in question can only be a load or a store instruction. No other instruction would raise an access fault. For read operations, the VMM would provide the result of the operation by changing the corresponding entry of the VM state structure.

That said, we found that the trap-and-execute emulation model is not possible to implement with the TrustZone protection mechanisms in general. Dependent on the concrete platform, the CPU will not immediately enter the hypervisor when the fault occurs but attempts to perform the bus transaction. This transaction will trigger an external data abort. This abort is similar to a device interrupt. It principally raises an exception (so the violation can be detected) but not always immediately. Therefore, there is no way to uniquely reconstruct what happened in between the invalid access and the reception of the external abort exception in the hypervisor. Neither can the hypervisor recover the non-secure world to a useful state.



The image above displays the flow of control when the normal-world OS performs an illegal device access. The Genode system starts in the secure world. The user-level TZ VMM component bootstraps the rich OS, and finally control to the normal world via the `run` function of the kernel, which, in turn, initiates a world switch to the normal world. If the rich OS accesses a device, which was not assigned to the normal world, an external data abort occurs, and control is passed back to the secure world. The virtual device model may try to trap-and-execute the faulting instruction.

We found that the latency of reporting access violations can be reduced by adding memory barriers to the faulting code. However, as this comes down to changing the Linux kernel, this would defeat the initial incentive for using device emulation instead of introducing hypercalls.

The general pattern of changing the original code to using a hypercall looks as follows (`ctr` refers to a critical device register):

```
+#ifdef RUNS_IN_SECURE_WORLD
  if (ctr)
```

```
    return readl(ctr);
+#else
+ if (ctr) {
+   static u32 ret;
+   asm volatile("mov r1, #3       \n"
+          "dsb              \n"
+          "dmb              \n"
+          "smc              \n"
+          "mov %[value], r0 \n" : [value] "=r" (ret) :: "r0", "r1");
+   return ret;
+ }
+#endif
```

We replaced the original call to the access function `readl(ctr)` with the snippet above.

A ready-to-use configuration for the Linux kernel accompanied with the needed modification to the kernel source code are available here:

Adaptation of the Linux kernel

```
https://github.com/skalk/linux/commit/284073c4f6c0e6a77c02ae9d296da9c46f6f5104.patch
```

We tried to keep the kernel patch as small as possible. The patch comprises the addition of only 73 lines of code (6 hypercalls) to the kernel. With these few modifications, we were able to boot Linux completely in the non-secure world.

TrustZone demonstrated in color

The goal of our ARM TrustZone experiments was to push the envelope of this technology beyond the typical scope of TPM-like functionality on a mobile tablet device. We aimed at executing a complete Genode-based operating system in the secure world while running a largely unmodified Android OS in the normal world. The secure OS does not merely sit in the background but comes with a graphical user interface that responds to user input via the touchscreen. At any time, the user is able to switch between both worlds using a button.

We identified the Freescale i.MX53 SABRE tablet as a suitable platform for this experiment. It is one of the few ARM development platforms that allows the developer to access the secure world of TrustZone and it has a tablet form factor. As we performed our initial TrustZone-related research on the ARM Versatile Express platform, we were furthermore interested in learning more about the differences of the TrustZone implementations of two different SoC vendors.

The practical work consisted of the following steps:

1. Adaptation of Genode to the i.MX53 platform,
2. Enabling TrustZone on i.MX53,
3. Implementing a TrustZone monitor that deals with the specifics of the i.MX53 with the goal of booting Linux in the normal world besides Genode,
4. Display driver running as user-level device driver on Genode
5. Touchscreen driver running as user-level device driver on Genode
6. Interfacing Genode's display and user-input drivers with the Android software stack.
7. Multiplexing the display between Android and Genode using hardware overlays.

In the following, each of those topics is briefly covered.

Genode on i.MX53 SABRE tablet

We conducted our first TrustZone experiments on the ARM Versatile Express platform based on a Cortex-A9 CPU. With Freescale i.MX53, we took the challenge to transfer our results to an entirely different platform. We considered two variants of a i.MX53-based platform. The so-called Quick-Start Board (QSB) is a low-cost development board whereas the SABRE-tablet reference platform comes in the form factor of a tablet. The principle adaptation of Genode to the new platform required us to add support for the Cortex-A8 CPU to our kernel and add device drivers for the i.MX-specific interrupt controller, UART, GPIO, and EPIT timer. With these adaptations, simple Genode scenarios could be executed on the platform.

TrustZone on i.MX53

As the i.MX53 SoC is based on a Cortex-A8 CPU instead of a Cortex-A9 as used on the Versatile Express platform, we expected the need of minor changes to the TrustZone world-switching code.

One particular i.MX-specific difference is the routing of interrupts. Because i.MX does use a custom interrupt controller, the assignment of interrupts to either the secure or the normal world works slightly differently than on the ARM Versatile Express platform. Furthermore, the i.MX53 comes with a custom TrustZone protection controller called Central Security Unit (CSU). Similar to the TrustZone protection controller used on the Versatile Express platform, the CSU allows the assignment of device groups to the secure or normal worlds using corresponding configuration bits. The CSU differentiates 64 device groups. Subsuming the individual devices to differentiable groups is up to the SoC vendor. A noteworthy advantage of the CSU

compared to the ARM TZ protection controller within the ARM Cortex A9 reference board is the way of how access violations are handled. As stated in Device emulation, the ARM TZ protection controller responds to invalid accesses with an asynchronous external abort exception, similar to a device interrupt. Upon the execution of an offending instruction, the TZ protection controller detects the violation, yet the CPU would continue the execution of further instructions until the flagged violation eventually reaches the CPU, triggering an external abort exception. This scheme effectively rules out any attempt to emulate device accesses. In contrast, the i.MX CSU responds to access violations by synchronously yielding control to the exception handler. So when such an exception occurs, the offending instruction can be determined and emulated in software. However, even though device emulation using the CSU is principally possible, we haven't investigated this opportunity further.

The second distinctive property of i.MX53 compared to the Versatile Express platform is the so-called Multi-Master Multi-Memory Interface (M4IF), which is part of the DDR memory controller. On the Versatile Express platform, we used 32 MiB SRAM for Genode and assigned the platforms DDR memory to the normal world. On i.MX53, this scheme would not work because there is no substantial amount of memory available besides the DDR RAM. This rules out a mere partitioning of memory using the CSU. However, the M4IF enables the masking of DDR RAM resources such that particular ranges can be preserved for the exclusive access by the secure world. We successfully used the M4IF to partition the DDR RAM between the secure world and the normal world.

TrustZone monitor

The user-level TrustZone monitor (we call it VMM because it acts very similar to a virtual machine monitor) for the i.MX53 had to be developed from scratch. Almost no code of the Versatile Express platform could be reused. To us, this experience is a valuable insight into the TrustZone technology. Even though the principle TrustZone mechanism for dividing control between both worlds is unified, it is up to the SoC vendor to implement their respective interpretation of security. I.e., the assignment of devices to the bits of a protection controller, the signalling mechanism for violations, the handling of the NS bit by the devices, or additional TZ-related bus components are not standardized. Hence, the mere claim that a product uses TrustZone does not automatically imply the presence of any meaningful security mechanisms. This is unfortunate if the use case of a product has not been envisioned by the SoC vendor. For example, in most application scenarios, TrustZone is used as a TPM-like component that is explicitly called by the normal world. It does not become active on its own. In our case, however, the secure world executes a complete OS including a preemptive scheduler. Therefore, access to clock and power management becomes critical in our scenario. Unfortunately, the clock and power management module does not support the protection of selected clocks and regulators from the normal world. The whole module can be either assigned to the normal world or the secure world. In the former case, the liveliness of the secure would would depend on the normal world. The latter case would require the implementation of extensive device-emulation code to use an unmodified OS kernel in the normal world.

For our prototype, we partitioned the platform where easily feasible (e.g., for DDR memory, interrupts) but we did not attempt to implement device emulators. In the case of the clock and power management module, we decided to grant the normal world access to the devices, yet disabled code paths in the Linux kernel that would interfere with the liveliness of secure world. We feel that this approach is appropriate for a demonstrator. For building a real product, the decision would come down to an even-handed judgement.

The hypercalls implemented by the i.MX53-specific VMM are related to the access of a few platform features that must not be handed to the normal world directly, in particular the virtual framebuffer and the virtual touchscreen.

Additional device drivers

In addition to the low-level drivers outlined above, the demonstration scenario required us to implement a number of peripheral device drivers, in particular a framebuffer driver for the LCD display, a touchscreen driver, and a driver for responding to the capacitive sensors of the SABRE tablet.

The task of creating the drivers, in particular the framebuffer driver, turned out to be more complex than anticipated. Even though we had documentation for the Image Processing Unit (IPU) available, the device is enormously complex. The specification of the IPU is around 1000 pages. In order to focus on parts that are relevant for us, we turned to analysing register traces of the driver that ships with the vendor's kernel. Unfortunately, we found the driver code to be extremely shaky and prone to race conditions. For example, added instrumentation code that slightly slowed down the register accesses performed by the driver would result in a defective driver. We were eventually able to pin-point the problem to a few critical register-access patterns but this had been a long-winding endeavour.

Compared to the framebuffer driver, enabling the touchscreen device was a smooth experience with no major surprises. With both framebuffer driver and the touchscreen driver in place, we could successfully run Genode's Nitpicker GUI server on the SABRE tablet. However, in the naive implementation, Nitpicker would need to copy pixels of its GUI clients to the physical frame buffer via software-blitting. Even though we have assembly-optimized blitting routines for ARM in place, we expected a significant performance overhead of running Android as a Nitpicker client compared to the native execution of Android without any indirection. For this reason, we investigated the use of hardware overlays.

Similar to most embedded graphics devices, the i.MX53 Image Processing Unit (IPU) comes with the ability to compose the final image from a number of so-called hardware overlays. Each overlay is fed by a stream of pixels fetched via DMA. The physical pixel color is the result of a compositing function that is evaluated per pixel. This function could give preference to one particular overlay, with the effect that the respective overlay is always displayed on top of all others. Alternatively, color keying can be used to decide which overlay is visible at a given screen position. The IPU can even mix colors of different overlays (alpha blending). Because the IPU fetches the pixels directly from memory using DMA and the pixel compositing function is executed in hardware, the display multiplexing can effectively be offloaded from the ARM CPU.

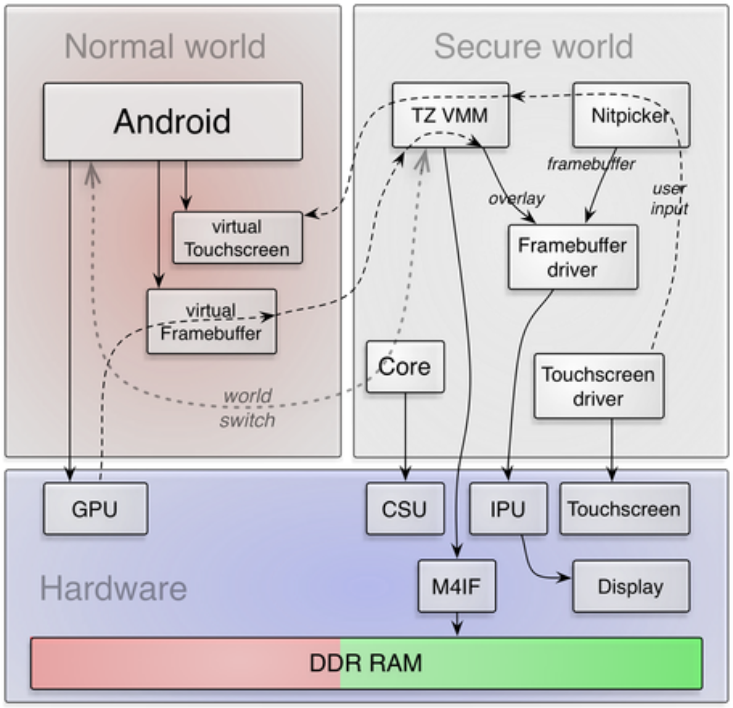
Since we wanted to run Android at almost-native performance in the normal world, we found that the use of hardware overlays was mandated. Unfortunately, the exploration of the hardware-overlaying feature of the i.MX53 IPU turned out to be highly complex so that we had to invest significant development time into enabling this feature on the Quick Start Board, just to find out that this feature behaves differently on the SABRE tablet. Because there seems to be almost no community of users of the

SABRE tablet platform, we were not able to leverage any public community knowledge while working on this. We eventually discovered that the order of seemingly loosely related register accesses was the tipping point for the driver to reach a functional state.

With our custom IPU driver with hardware-overlay support in place, the seemed to be paved to securely multiplex the display between (untrusted) Android running in the normal world and Genode's Nitpicker GUI server running in the secure world. Because the access to the IPU and GPU from either of both worlds can be individually configured in the i.MX53 CSU, we could assign the IPU exclusively to the secure world while handing out the GPU to the normal world. This way, Android can leverage hardware-accelerated graphics while the secure world retains control over the display.

Unfortunately, however, the TrustZone implementation of the i.MX53 SoC has a limitation that renders the isolation between IPU and GPU ineffective. From the investigation by security researchers of SRLabs, we learned that even though the CSU could be configured to restrict the access from either of worlds to the devices individually, there is no way to individually restrict the access from each of the devices to the secure and normal worlds. Both devices IPU and GPU require direct memory access (DMA) to operate. The IPU uses DMA to fetch pixels from the frame buffer. The GPU uses DMA to access GPU buffer objects (containing textures, shaders, vertex arrays) and the target surface of rendering operations. On the i.MX53, it is not possible to configure DMA access of IPU to the memory of the secure world while restricting DMA operations of the GPU to the normal world's memory. In fact, both devices IPU and GPU share the same DMA channel ID. Consequently, we had to grant DMA access from both devices to the memory of both worlds. This way, the normal world is actually able to compromise the secure world using DMA via the GPU. Thanks to SRLabs for this valuable insight! To fix this security hole, the SoC would either need to separate the DMA policy of both devices or the secure would would need to virtualize the GPU.

Demonstration scenario



Architectural overview of the TrustZone demo on i.MX53

Figure 6 presents an overview of the key components of the demonstration scenario. Dashed lines symbolize information flow. Solid lines represent a relationship of control.

At boot time, Genode's core sets up the CSU configuration and thereby defines the permissions of the normal world to access peripherals. The CSU is configured such that security-critical peripherals such as the IPU and the touch screen controller are preserved for the secure world only. Not all relevant devices are depicted (e.g., timers and GPIO controller are omitted in the picture). After platform initialization, core spawns the Genode process tree, which contains several user-level device drivers as well as the user-level TZ VMM. The VMM is responsible for managing the normal world. In particular, it loads the image of the normal-world OS and triggers the switch to the normal world using a service provided by core. Each time, the normal world passes control to the secure world or attempts to perform an invalid operation, core reflects the resulting world switch to the VMM. So the VMM can process hypercalls or invoke an emulation function accordingly and explicitly yields control back to the normal world.

For the demonstration scenario, the VMM provides a virtual framebuffer and a virtual touchscreen device to the normal world. To keep the complexity of the VMM low, the interactions of the normal world with these virtual devices are implemented as hypercalls (rather than faithful emulations of the complex i.MX hardware devices). The CSU is configured such that the Android OS is able to access the Graphics Processing Unit (GPU) directly. However, the GPU targets all rendering operation to the virtual framebuffer rather than the physical framebuffer as used by the IPU. This way, the VMM can deliberately direct the GPU output to a dedicated buffer, which is composed into the physical screen using a hardware overlay. Because the framebuffer driver is the only software component in control of the IPU, the display remains under the sole control of the secure world, yet there exists a direct data path of pixels of the normal world to the display. Consequently, the graphics

performance of Android in the demo scenario is on par with the performance of Android natively executed on the same platform.

User input is always received by the secure world via the touchscreen driver and the Nitpicker GUI server. Hence, the routing policy of user-input events is under control of the secure world at all times. Only if the TZ VMM has the current input focus, nitpicker will route input events to the VMM, which, in turn, translates those events to input events for the virtual touchscreen used by Android.

The demonstration scenario is available at the following branch of Genode.

https://github.com/skalk/genode/tree/i.MX53_tablet_demo

> Genode as secure OS on a i.MX53 SABRE tablet.

Detailed instructions about reproducing it are provided by the README file at the *os/src/server/vmm/imx53/* directory.

Common questions, answered

In this Section, we attempt to answer common questions that we were repeatedly asked. Note that we are by no means experts in all areas covered. So if you detect errors, please report them back, preferably on our mailing list.

What are the capabilities of TrustZone?

TrustZone technology can be viewed from two angles, as virtualization solution and as mechanism to implement functionality similar to Trusted Platform Modules (TPM). When regarded as virtualization solution, TrustZone is severely lacking.

- The number of virtual machines is limited to two, one running in the secure world and one running in the non-secure world. There are no means for protecting multiple OSes sharing the non-secure world.
- There is no support to virtualize MMIO resources via the trap-and-execute model. As described in Section Device emulation, access violations of the non-secure OS can be detected but not emulated. Consequently, multiplexing a single device for both worlds is not possible in a transparent way.
- To guarantee that both worlds access distinct device resources only, certain device drivers of the non-secure OS must be modified. The use of TrustZone is not transparent. Even though those modifications are minor, this limitation effectively prevents the reuse of operating systems that are available as binaries only.
- There is no way to virtualize the physical memory as used by the non-secure OS. The guest-physical memory always corresponds to the host-physical memory.

Despite those limitations, we identified a single advantage of TrustZone compared to other virtualization technologies (such as VT-x and recent ARM virtualization extensions), which is the direct assignment of device interrupts to the non-secure world without involving the VMM as indirection.

We ultimately reached the conclusion that perceiving TrustZone as a virtualization mechanism is ill-guided. When looking at TrustZone as an alternative for TPMs, the motivation behind this technology become much more clear. In contrast to fixed-function TPMs, TrustZone is a vastly more versatile mechanism. The currently active world is represented by the so-called non-secure bit (NS bit) at the interconnect. The NS bit is routed to peripherals similar to an additional address line. This enables peripherals to respond to this condition by implementing restrictive policies for the NS mode. Thereby, security functions implemented in the secure world of TrustZone can utilize such peripherals. In contrast to TPMs, the security functions of the secure world of TrustZone are freely programmable using a powerful general-purpose CPU architecture. The port of the Genode OS Framework (as a poster child of a sophisticated component-based operating system) to the secure world is illustrative in this respect. In contrast, running Genode within a TPM would naturally be out of question.

What are the limitations of TrustZone for compartmentalization?

- There are only two "worlds".
- Access by the non-secure side to the physical address space can be restricted but physical addresses are not virtualized.
- The granularity of access restricts depends on the SoC. For example, the Versatile Express platform provides no means to partition the DDR RAM into secure and non-secure areas. In contrast, the Freescale i.MX SoC provides this principal ability.
- The use of TrustZone is not entirely transparent to the non-secure side because non-accessible physical resources appear as holes in the physical address space. Furthermore, access to central devices (such as the system control registers) cannot be transparently emulated. The OS running on the non-secure world must be slightly modified.

How to host two different compartments/zones (one trusted, one untrusted)?

The general approach consists of the following steps:

1. The system starts in secure mode and boots the hypervisor.
2. The hypervisor sets up the non-secure world (i.e., by configuring the TZPC, TZASC, and PIC).
3. The hypervisor loads the OS image to be executed in the non-secure world.
4. The hypervisor enters the non-secure world at the entry point of the non-secure OS.
5. The non-secure OS issues hypercalls for functions that are hidden from the non-secure world. Those hypercalls are handled by the hypervisor.

In our particular experiments, the hypervisor is represented by an instance of the Genode OS, which separates the handling of hypercalls from the low-level mode switching using microkernel techniques.

The individual steps of this process are covered in more detail in section Hypervisor managing the non-secure world.

Does it make sense to run commodity OSes like Android in the secure world?

Following the rationale of Section What are the capabilities of TrustZone?, the use of TrustZone is meaningful only for implementing security functions. Thereby, the complexity of implementation is crucial. Traditionally, the code executed in the secure world has library-like functionality. Running a complete OS such as Genode in the secure world yields more flexibility on the cost of added complexity. Running a monolithic OS on the secure world would increase the complexity in multiple orders of magnitude.

On platforms such as the Versatile Express platform, only small portions of memory can be preserved to the secure world when also running a non-secure OS. I.e., the particular platform hosts 32 MiB of SRAM. This limitation naturally rules out complex operating systems such as Android.

How does TrustZone compares to ARM's virtualization extensions?

The virtualization extensions introduced with Cortex-A15 are orthogonal to TrustZone. There are no significant changes between the TrustZone implementations of Cortex-A9 and Cortex-A15.

The virtualization extensions devise the implementation of the virtual machine monitor as part of the non-secure world. Those extensions are unavailable in the secure world (which is aligned with the idea of implementing mere TPM-like functionality in the secure world).

Are there any security improvements of the Cortex-A15 over the A9 architecture then?

The virtualization extensions introduced with Cortex-A15 offer an additional instrument for partitioning of the system through the "Hyp" (hypervisor) mode on the non-secure side. With this mode and the additional virtualization features, low-complexity virtual machine monitors become feasible. However, with regard to the protection of the secure world, there is no improvement over Cortex-A9.

That said, the security properties of TrustZone largely depend on the SoC rather than the revision of the ARM CPU core. Cortex-A15-based SoCs may offer more flexibility with regard to the secure world (such as more peripherals responding to the NS bit). However, we have not thoroughly examined those SoC-vendor-specific implementations at large.

What are the capabilities of ARM's virtualization extensions?

There is a new CPU privilege level ("Hyp" mode) below the existing CPU modes, which is only supported in non-secure mode.

Virtualization holes present in previous generations of the ARM architecture have been fixed for the Hyp mode. Consequently all privileged instructions trap into Hyp mode when executed in the other modes. Thereby, the architecture has become fully virtualizable by the means of the trap-and-execute model.

The trap-and-execute model principally allows for executing unmodified guest OSes on top of the VMM. However, this model has inherent performance penalties when the guest OS performs privileged operations at a high rate, for example, when clearing interrupts at the interrupt controller. To mitigate those effects, hardware support for effectively avoiding the frequent intervention of the hypervisor has been introduced.

In particular, the handling of virtual interrupts can be accelerated by using a new virtual IRQ controller (vIRQ) device. The hypervisor can present the vIRQ to the guest OS at the guest-physical location of the normal interrupt controller. Hence, the guest will always interact with the vIRQ device. When the guest OS clears an interrupt, the hypervisor does not get involved. However, the injection of interrupts into the virtual machine is exclusively done by the hypervisor. There is no way to directly assign device interrupts to virtual machines because the scheduling of (multiple) virtual machines on one CPU is implemented in the hypervisor. Hence, each device interrupt designated for a virtual machine will take the hop through the hypervisor.

The Cortex-A15 facilitates the emulation of memory-mapped device registers by providing useful information to the virtual machine monitor. In addition to the program counter and fault address, the VMM receives the decoded instruction and the type of access. This alleviates the need of the VMM to access guest memory (to fetch the instruction) and simplifies the instruction decoding.

To support multiple virtual machines, a guest-physical to host-physical address translation has been introduced, which is a mechanism similar to extended page tables (EPT) of VT-x. Thereby, guest-physical memory becomes fully virtualizable. This feature comes along with a new page-table format, which is optional for guest-virtual to guest-physical mappings and mandatory for guest-physical to host-physical mappings. Besides the support for guest-physical to host-physical mappings, the new format has the benefit of supporting more than 4 GiB of RAM in total (per VM, the amount of usable RAM is still restricted by 32-bit addressing). Thanks to tagged TLBs, there exists a fast path for the translation from guest-virtual to host-physical mappings.

There is no virtualization of DMA accesses issued by bus peripherals (i.e., display controller uses host-physical addresses). If the guest OS contains a device driver, the guest has to use host-physical addresses to interact with the device. For example, addresses of DMA buffers supplied to a device cannot be specified as guest-physical addresses. Therefore, in the general case, DMA-using devices cannot be passed directly to a guest OS. Access to such devices must be intercepted by the hypervisor to translate those addresses.

There is no DMA protection. Additional per-device MMUs would be required. The presence of such per-device MMUs depends on the SoC.

IRQs designated to a device driver running in a guest OS are always handled by the hypervisor, which responds to the IRQ by injecting a virtual interrupt to the vIRQ device. Hence, each interrupt carries the overhead of switching the context between the Guest OS and the hypervisor. However, the relevance of this overhead in practice is uncertain.

For more information on ARM virtualization extensions and the procedure of creating a Cortex-A15 based virtual machine monitor, let us refer you to the excellent document "Implementing Hardware-supported Virtualization in OKL4 on ARM" by Prashant Varanasi.

How does TrustZone help to securely store secrets?

Hiding peripherals and memory from the non-secure world is a key feature of TrustZone. TrustZone does not define, which peripherals and memory are subjected to this mechanism. This is in the hands of the SoC vendors. For example, SoCs of Freescale i.MX family come with RAM and ROM resources that are entirely located on chip. Hence, bus transactions concerning these resources are not visible at the wiring of the chip. By declaring those resources to be exclusively accessible by the secure world, the information stored and processed in these resources remain private to the secure software stack.

That said, each SoC has different characteristics with regard to assigning or partitioning storage resources for the access of either world. Hence, there is no general answer to this question.

How does TrustZone facilitate secure booting?

As a precondition for the use of TrustZone for secure booting, the code running in the secure world must be bootstrapped in a secure way, which is SoC-specific. For example, the i.MX family provides a high-assurance boot (HAB) feature that could be used to securely bootstrap the secure world. Alternatively, the secure software could be fixed in a chip-internal flash or ROM at production time. For example, the popular range of low-cost ARM development boards such as Pandaboard come with a fixed ROM boot code that implements several hypercalls and switches to non-secure mode right before starting the u-boot boot loader. Because the internal ROM code cannot be bypassed, there is no way for any system booted on such devices to access device resources that are preserved to the secure world.

Given that the secure world is booted in a secure fashion, the loader of the non-secure OS is in the position to validate the integrity of the boot image of the non-secure OS via cryptographic measures. In the case of Genode running in the secure world, such functionality could be implemented within the user-level VMM component.

If the SoC lacks a way to fix the boot code for the secure world, secure booting cannot be implemented even if the platform is equipped with TrustZone technology.

When using Genode as secure OS, the SoC should provide sufficient memory to be preserved for the secure world. Early platforms such as the Samsung S3C6410 that come with only a few KiB of secure RAM and ROM are ruled out.