ARM TrustZone as a Virtualization Technique in Embedded Systems

Torsten Frenzel, Adam Lackorzynski, Alexander Warg and Hermann Härtig
Technische Universität Dresden
Department of Computer Science
01062 Dresden, Germany
{frenzel, adam, warg, haertig}@os.inf.tu-dresden.de

Abstract

Virtualization starts to gain traction in the embedded world as methods to enforce isolation are needed to cope with the ever-growing complexity of modern systems. Originating from desktop and server systems, existing virtualization solutions have their focus on rich functionality such as migration and check-pointing while real-time functionality is of little interest. In contrast, in embedded systems this functionality is dispensable while the ability to support real-time workloads has to be retained. So far, virtualization on ARM - the arguably predominant architecture for embedded systems - was impeded by the lack of proper architectural support. In this paper, we show how TrustZone, an extension originally meant to boost security, can also be used to fill that gap and provide encapsulation in real-time conscious systems. We evaluate a minimally modified Linux on top of a real-time capable microkernel in terms of engineering and runtime costs. The results show that it is possible to improve the performance compared to existing solutions and preserve realtime properties for applications.

1 Introduction

Today, embedded systems such as smartphones handle valuable content, like sensitive data from a bank account and other personal information as timing critical resource such as network connections. They also run a growing number of applications and are able to dynamically download new software. Consequently, security-sensitive and realtime-sensitive applications and potentially malicious applications of low trustworthiness share the same platform.

An important criterion for the vulnerability of security-sensitve applications is the size of their Trusted Computing Base (TCB). It consists of all hardware and software components they have to trust for their functionality. A small TCB for such applications and services is favourable because it helps to reduce the number of vulnerabilities that can be attacked. Reducing the size of the TCB is achieved by isolating components at software level, hardware level and a combination of both.

The specification of the Trusted Mobile Platform [11] is an attempt to address the security requirements for mobile and embedded platforms. It describes a generic hardware and software architecture enriched with protocols to access secure services. The specification defines different security classes with specific isolation properties and requirements for their TCBs. For example, a highest security class device requires hardware-enforced isolation and a TCB consisting of a verified and trusted microkernel for security-sensitive applications.

ARM TrustZone [13] has been designed to enable trusted mobile platforms by providing two hardware-isolated execution domains. The *secure-world* domain supports the protected execution of security-sensitive software, while the *normal-world* domain enables the encapsulated execution of less critical software. The hardware architecture protects the security-sensitve software not only against attacks from normal-world software but also from physical attacks, such as the sniffing of bus transactions. Furthermore, TrustZone aims to provide high performance for access to physical resources for normal-world components as well as secure-world components, while reducing the costs for duplicated hard-

ware.

TrustZone imposes few restrictions on the software stack of either world. This enables third-party developers to integrate their own security solution. While there exists a reference implementation for a TrustZone software architecture [7], no results are publicly available that evaluate the TrustZone architecture with regards to the performance overhead. This paper intends to close this gap as it reports results of implementing an open security architecture, called Nizza [18], on the TrustZone platform.

As with TrustZone, Nizza is designed to support minimized TCBs for security-sensitive applications. The software architecture isolates generic operating systems and their applications from security-sensitive applications using system virtualization. A small realtime kernel or hypervisor implements the virtualization layer and a multi-server OS provides access to hardware and software resources. The kernel can host isolated, budgeted realtime applications and virtual machines with fixed priorities.

This work makes the following contributions: (1) We show how TrustZone can be used to implement a complex security architecture that facilitates the execution of isolated realtime applications in embedded systems using an approach that is similar to full virtualization. (2) We compare this approach with regards to the software-development effort to implement this solution and the performance and latency with pure paravirtualization and native execution.

To this end, we provide in Section 2 a bird's eye view on the hardware and software components of the TrustZone architecture. In Section 3 we outline the Nizza security architecture applied as a TrustZone software stack and introduce two important virtualization techniques for security architectures. In Section 4, we describe in detail our design and implementation of using the TrustZone hardware architecture as a basis for the Nizza software architecture. Section 5 provides an evaluation regarding the changes to the encapsulated operating system and the performance overhead and of our TrustZone-based virtualization solution. Section 6 discusses related work and Section 7 concludes.

2 ARM TrustZone Architecture

TrustZone restricts the access to security-sensitive components by introcuding a virtual split through the hardware in the system. The approach requires modifications in the processor core as well as the platform. The architecture is accompanied by a proposed software architecture that reflects the split approach.

2.1 Hardware

A processor with TrustZone extension provides two virtual processors establishing two security domains: the *secure world* and *normal world*. The processor can switch from one world to the other using a dedicated and controlled mechanism, that requires a secure-world software component, called the *monitor*. Furthermore, all hardware components in the system, such as memory and devices, are configured as normal-world or secure-world accessible. This can be achieved by adding TrustZone-aware components that enforce the security policies in the system, such as memory controllers [14] and special protection controllers [10]. The hardware split restricts the access rights of the normal world to normal-world resources, while the secure world can access all resources.

The world in which the processor is currently running determines its security state [1]. Whenever the processor executes a read or write instruction to access a resource, the corresponding bus transaction is tagged with the security state of the processor. This way it is propagated through the system and enables peripheral components of the platform to check whether or not an access to a resource is valid. All components that are involved in satisfying this request can verify the security state. Whenever a peripheral component itself issues a bus transaction, this operation is tagged with the security state of the component which also prevents DMA attacks.

The delivery of interrupts is also controlled by the secure world, which configures if a specific interrupt is delivered to the current world or to the monitor software. TrustZone defines two models for interrupt delivery. The first model uses one interrupt controller that is programmed by the secure world only and requires the monitor to decide to which world an interrupt is routed. The second model uses a TrustZone-aware interrupt controller [12] which provides two virtual interrupt controllers similar to the two virtual processors. This enables the platform to route interrupts to the configured world without intervention of the monitor software.

A TrustZone-enabled platform consists of the TrustZone-enabled processor core with additional on-chip secure components, such as tightly coupled memory, crypto modules, RAM and boot ROMs. If

[1] The architecture refers to this as the NS-bit.

the system bus is security-aware further secure peripheral components can be added, such as SDRAM, flash and ROM. The partitioning of the platform can be hard-wired or reconfigurable using special platform-dependent mechanisms.

2.2 Software

The designers of TrustZone had a clear perspective of the software architecture in mind, which extends the hardware architecture's philosophy into the software layers on top. This software architecture consists of normal-world components, including a generic operating system and its applications accessing only normal-world hardware, and secure-world components. The secure-world software can range from a small service or encryption layer fitting into on-chip memory to a complex operating system with its applications.

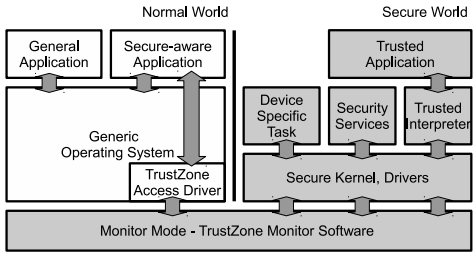

FIGURE 1: *TrustZone complex software architecture*

Figure 1 shows a complex software architecture as outlined by the ARM engineers [15, 25]. It has a generic OS and applications running in the normal world. The generic OS can access the secure-world services using TrustZone access drivers which in turn call the TrustZone monitor software to switch to the secure world. The secure software stack consists of a secure kernel, secure drivers, trusted applications and the monitor. The secure kernel contains drivers for secure devices and hosts secure services together with further device specific tasks. Trusted applications can run either direct on top of the secure service layer or are isolated by a trusted interpreter, such as a Java virtual machine. In addition to these components a secure boot loader is required to bootstrap and measure the software stack.

The monitor performs the switches between the secure world and normal world. Its provides functionality that is similar to a context switch in operating systems, ensuring that the state of the world that the processor is leaving is saved, and the state of the world the processor is switching to is restored. Normal-world entry to the monitor is only possible via interrupts, external aborts or an explicgit call, referred to as *monitor call* [2]. The secure world can enter the monitor without restrictions when running in privileged mode, in addition to the available exception mechanisms.

As TrustZone is a pure security architecture it makes no statement and poses no restrictions on wich side realtime components should run.

For the communication between both worlds TrustZone defines several APIs: a Generic API with a simple message-passing interface, an extensible Security Channel API for well-known services and APIs for specific security modules. The APIs define a remote-procedure-call standard to open connections from the normal world to the secure world.

3 Nizza Architecture

The Nizza security architecture shown in Figure 2 minimizes the TCB for security-sensitive applications using a small multi-server operating system with unprivileged components and the isolation of non-secure components from security-sensitive components. A microkernel enforces isolation between components in the system and provides fast and controlled communication between them. On top, deprivileged servers, such as a file system, a secure GUI and resource managers, as well as drivers provide a small service layer. Non-secure and security-sensitive applications can access these services, which mediate and control the access to physical resources.

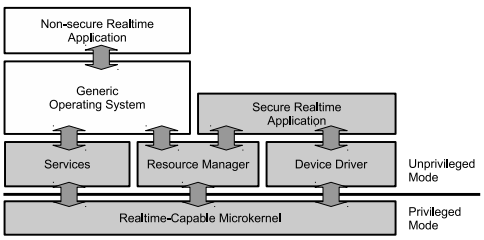

FIGURE 2: *Nizza security and realtime architcture*

The Nizza architecture closely resembles the complex software architecture as described in the previous section with a split in two worlds separated by the microkernel. In the secure world the service layer manages the platform and multiplexes resources among both worlds. The normal world consists of the generic OS and its applications.

[2] The privileged instruction is called SMC.

Although the Nizza architecture is very similar to the TrustZone's complex software architecture there are differences. The trusted interpreter as shown in Figure 1 is not required because isolation between application is provided by separate address spaces. Furthermore the Nizza architecture does allow to encapsulate many generic OSes on the same platform, which is not intended by the TrustZone architecture. Nizza has built-in support for realtime applications, which includes the low-latency preemptible kernel with a static priority scheduling scheme and periodic execution and budgeting of realtime applications.

Security architectures such as Nizza use virtualization to encapsulate and isolate OSes and their applications, excluding them from the TCB of security-sensitive and realtime applications. We first describe a paravirtualization architecture, an approach that is commonly used in embedded system, and afterwards we describe full virtualization architecture with hardware assistance.

Paravirtualization Paravirtualization modifies critical parts of the deprivileged kernel of the virtualized OS. Platform-specific components are adapted to the interface of the virtualizing kernel using source-code modifications.

A paravirtualized kernel needs modifications in the system-call interface and all platform-specific parts that interact with the hardware, like memory management, and interrupt handling. For example, page-table modifications are translated into resource-delegation requests and interrupts are signalled to the paravirtualized kernel as messages. All system calls from the application to the paravirtualized kernel need to be mediated by the hypervisor, which can cause serious performance degradation.

Paravirtualization has low hardware requirements; any processor that supports isolation by address spaces can be used to implement this virtualization solution. However, the effort required to adapt a kernel can be high.

XenLinux[16] and L4Linux [20] are examples of paravirtualized kernels. In this work L4Linux serves as a reference to compare our new approach with regarding the performance and software-development effort.

Full Virtualization Full virtualization runs the deprivileged kernel of the virtualized OS without any modifications. The hardware traps sensitive and privileged instructions to the virtualizing OS. This enables the virtualizing OS to emulate these instructions and virtualize the hardware resources. Full virtualization architectures require a *virtual machine monitor* (VMM) that controls the virtualized OS and provides the virtual hardware environment.

This approach relies on hardware assistance to reduce the number of traps and avoid serious performance degradation. Since the early seventies mainframe vendors [19] and in recent years, server and desktop processor vendors [22, 1] have implemented virtualization extensions. Such extensions split the physical processor into two virtual processors, one for the virtualized OS and one for the virtualizing OS. The execution of the virtualized OS is controlled by the virtualizing OS. It initiates a switch to the virtualized OS and enforces a switch back on critical events, such as the occurrence of interrupts and page faults.

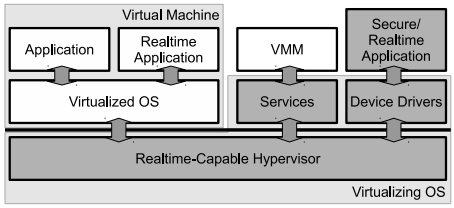

FIGURE 3: *Microkernel-based hypervisor architecture*

A microkernel-based virtualization architecture with full virtualization support as shown in Figure 3 splits the VMM into two parts. A small privileged *hypervisor* contains the mechanism to control and schedule virtualized OSes inside a *virtual machine* (VM). An unprivileged *VMM* provides the virtual hardware environment for the virtualized OS.

This approach minimizes size of the privileged component, the hypervisor, that contributes to the TCB of security-sensitive applications. The VMM does not add to the size of the TCB of security-sensitive applications because it runs unprivileged in its own address space. However, this separation requires additional communication between the hypervisor and the VMM, which can decrease the performance of the virtualized OS.

4 Full Virtualization with TrustZone

As stated in the previous section, current techniques for mobile and embedded devices to encapsulate OSes in security architectures, like paravirtualization, require a high development effort. The TrustZone hardware architecture directs the focus towards

an approach similar to full virtualization solutions. TrustZone allows to run an operating system encapsulated in the normal world with only minor modifications and under control of a software layer in the secure world.

4.1 Architecture

Figure 4 shows the Nizza security architecture applied to TrustZone as outlined in Section 3 combined with the full virtualization approach. The normal world contains the normal-world OS and its applications. The normal-world OS uses access drivers to send requests to the secure world and has drivers to access normal-world devices directly. The secure world contains the monitor, the secure kernel and the secure-world OS with secure applications. The secure-world OS consists of unprivileged components, such as secure device drivers and secure services.

This architecture contains the realtime-capable hypervisor and the VMM as new components derived from the full virtualization scenario. The hypervisor consists of the TrustZone monitor software and the secure kernel, which are tightly coupled inside the secure privileged component. This integration into one component is justified by the fact that there is no isolation between the secure-privileged processor modes and the monitor processor mode. The hypervisor is responsible for the separation of the normal world and the secure world and to switch between both worlds on request. The VMM provides a virtual platform for the execution of the normal-world OS inside the virtual machine. It receives requests from the normal-world OS and mediates them to security services or secure drivers.

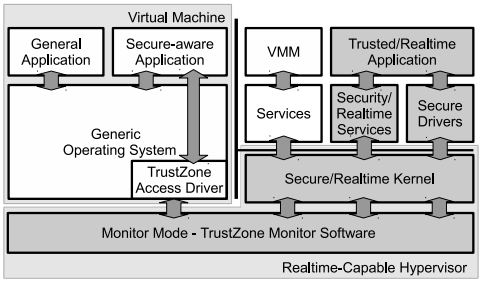

FIGURE 4: *Nizza software architecture applied to TrustZone*

Our architecture requires a core with TrustZone extension and the static or dynamic partitioning of memory into a secure and a normal area. We use the simple interrupt model with one secure interrupt controller. All other devices are secure by default, but the generic OS can get direct access to a specific device if the platform can be configured that way.

We describe the hypervisor and the VMM which drive the normal-world OS and the interface for the normal-world OS to access normal-world and secure-world resources.

4.2 Hypervisor Component

The hypervisor as the only component of the secure-world OS running in privileged mode enforces isolation and security policies. It is based on a microkernel design and offers abstractions and mechanisms to run a multi-server operating system next to a normal-world OS. It provides the following functionality:

- *Tasks and Threads.* Tasks are address spaces providing spatial isolation for services, secure drivers and applications. Threads are entities of execution running inside a task and are scheduled preemptively.

- *Virtual machine* (VM). A VM consists of a virtual processor, a memory partition and a set of accessible devices. The creation of a new VM establishes a shared-memory region, called *VM state*, that stores the normal-world processor state and is accessible by the creating task.

- *Communication* channels. Threads can send messages to and receive messages from other threads and the hypervisor.

To facilitate the execution of the normal-world OS, the hypervisor contains the monitor software that performs the switch to the normal world, called *normal-world entry*, and the switch back to the secure world on events such as monitor calls and interrupts, called *normal-world exit*.

To trigger a normal-world entry, a thread sends a message to the VM. The hypervisor loads the current processor state from the VM state, enables the memory partition and switches to normal-world processor mode. On normal-world exit the hypervisor saves the normal-world processor state in the VM state, disables the memory partition, switches to secure-world processor mode and passes execution to the calling thread, which in turn can examine the new processor state.

4.3 Unprivileged VMM Component

According to the architecture the VMM is an application consisting of a task with a thread running inside. The purpose of the VMM is to provide the virtual platform for the normal-world OS shown in Figure 5. It consists of a set of virtual device models. We only summarize the functionality of the virtual devices because a detailed description of every device interface is beyond the scope of this work.

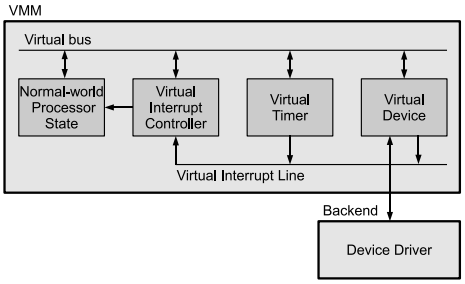

FIGURE 5: *VMM device architecture.*

The virtual interrupt controller receives virtual interrupt requests triggered by virtual devices. The normal-world OS can selectively enable or disable interrupts. A virtual interrupt that is enabled and asserted is injected into the normal-world OS as soon as possible. The virtual timer device is able to periodically generate virtual interrupts. Serial devices, block devices, network devices and framebuffer devices implement hardware-like interfaces to exchange data between the normal-world OS and the secure-world OS. A virtual device can have a communication channel to a backend in the service layer, such as a device driver or a secure service.

The VMM handles requests from the normal-world OS and events from the backends. Currently, a normal-world OS can request two services using monitor calls:

- *Read or write to a virtual device.* The VMM selects the corresponding virtual device of the virtual bus using the provided address. The virtual device modifies its state according to the passed parameters and returns a result.
- *Go to idle mode.* The VMM suspends the execution of the normal-world OS until a virtual interrupt is injected.

If the VMM receives an event from a backend component it executes the handler of the connected virtual device. Furthermore, the hypervisor passes hardware interrupts from normal-world devices to the virtual interrupt controller.

In both cases the execution of the normal-world OS is suspended by the hypervisor and the normal-world processor state is saved into the VM state. The VMM has access to the VM state and memory partition of the normal-world OS. It can inspect and modify the state to read parameters, to pass return values, and to inject interrupts. At the end of the request handling, the VMM sends a message to the hypervisor to initiate a normal-world entry with the new normal-world processor state.

4.4 Normal-world OS

The normal-world OS, called TZ-Linux, runs inside the normal-world of the TrustZone platform. It has access to the configured normal-world memory partition and to normal-world devices. It contains access drivers that send requests to the virtual device models in the VMM. The minimal configuration of access drivers for the virtual platform that the VMM provides consists of an interrupt driver, a driver for the virtual timer and a serial driver to enable console input and output for normal-world OS.

4.4.1 Access to Normal-world Devices

The normal-world OS can access a device directly if the hardware platform or secure-world OS configure the device as normal-world accessible. In this case, all read and write operations to and from the device are allowed and require no intervention of the secure-world OS except for the interrupt delivery. Interrupts from the corresponding device are routed through the secure-world OS because the normal-world OS cannot access the secure interrupt controller. Figure 6 shows the interrupt routing from the non-secure device to the normal-world OS.

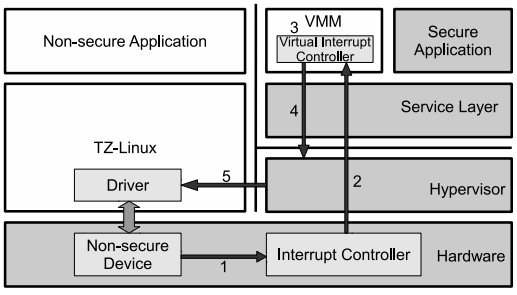

FIGURE 6: *Access path and interrupt routing for direct device access by the normal-world OS.*

The routing of interrupts to the normal-world OS is performed in five steps: First, the interrupt from the non-secure device is signalled to the secure interrupt controller. Second, the interrupt controller triggers an interrupt and the processor traps into the hypervisor. If the normal world is currently active, the monitor code in the hypervisor performs a normal-world exit and then delivers the interrupt as a message to the VMM. Third, the VMM injects a virtual interrupt into the virtual interrupt controller and sets the virtual interrupt-pending signal inside the VM state if the virtual interrupt is unmasked. Fourth, the VMM sends a message to the hypervisor. Fifth, the hypervisor restores VM state and performs the normal-world entry which delivers the virtual interrupt.

After injection of the virtual interrupt the normal-world OS accesses the virtual interrupt controller to read the pending interrupt number, to mask and to unmask the interrupt.

4.4.2 Access to Secure-world Devices

The normal-world OS cannot access secure resources directly. Instead, it uses an access driver that sends requests to the VMM in the secure world. The VMM interprets the requests and executes the corresponding actions or mediates them to secure drivers and secure services accordingly. The VMM has complete access to the state of normal-world OS and can verify that the request is valid according to a specified security policy. Furthermore the normal-world OS cannot access secure resources that are not present in the VMM as virtual devices. That means that the configuration of the virtual device environment with the communication channels to secure services and drivers implicitly restricts the environment for the normal-world OS.

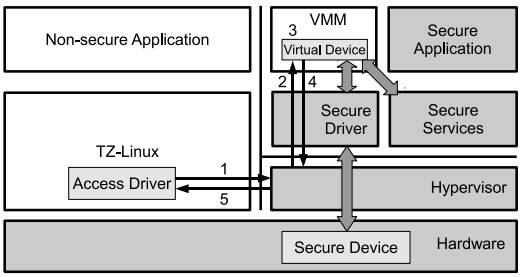

FIGURE 7: *Access path for indirect device access.*

Access to a secure device is not as fast as access to non-secure devices because it involves communcation between the normal-world OS and the secure-world OS. Figure 7 shows the control flow from TZ-Linux to a secure device. which generally involves the following five steps:

1. The access driver in the normal-world OS issues a request to the VMM in the secure-world OS.

2. The hypervisor saves the normal-world processor state into the VM state and sends a message to the VMM.

3. The VMM selects the virtual device and handles the read or write request. For example, it can send another message to a secure driver in the service layer or it passes data to an encryption service.

4. After finishing the request the VMM sends a reply message to the hypervisor.

5. The hypervisor restores the normal-world processor state from the VM state and initiates the normal-world entry.

Similar to this sequence, interrupts are routed through the VMM as explained for the normal-world device access.

As already explained, virtual devices have a communication channel to the secure component that provides the service. As an example, we describe how TZ-Linux accesses a secure network device using a virtual network device inside the VMM. The TZ-Linux kernel contains a network access driver that communicates with with a virtual network device inside the VMM to access a secure network device. The virtual network device is connected to the secure network driver. To send a packet the network access driver sends a request to the virtual network device, which in turn passes the packet the secure network driver. Receive operations are efficiently handled using a ring buffer, shared between TZ-Linux and the VMM. If the secure network driver receives a packet it stores it in the next empty slot of the ring buffer. The network access driver is notified after a successful receive operation by a virtual interrupt. It removes full slots from the ring buffer and allocates new empty slots. For the performance benchmarks in this paper we placed the network driver inside the VMM avoiding additional message passing between the VMM and the network driver and reducing the copy overhead for network packets.

4.5 Implementation

We built a prototype to verify and evaluate our design. During the implementation we investigated the capabilities and limitations of TrustZone to control the normal-world OS. As remarked earlier, TrustZone has features that are comparable with already existing virtualization extensions of other hardware architectures. The secure-world OS is able to control the execution of the normal-world OS. The philosophy of TrustZone strives to minimize the interactions between normal world and secure world. This approach has performance benefits, however it also limits the virtualization capabilities as illustrated by the following examples.

The detection of events occurring inside the normal-world OS is limited by the hardware because most control registers of the processor are banked for both worlds. This restricts the ability of the secure-world OS to virtualize processor features for the normal-world OS, such as the register containing the number of physical processors or registers of debugging features. As another example, recent processor versions provide an instruction that is executed in order to wait for an interrupt and indicates that the OS is in an idle state [3]. The detection of the execution of such an instruction by the normal-world OS enables the secure-world OS to schedule other work. However, the ability to trap this instruction is not supported. In TZ-Linux we replaced the critical instruction with a special request to the VMM to solve this problem.

The partitioning of physical memory as provided by the hardware is sufficient to encapsulate the OS inside the normal world, however it is not transparent. The normal-world OS has to know the physical memory partition it can use in the system. Changes in the configuration of the memory partitioning require a relocation of the normal-world OS. Furthermore this partitioning scheme does not allow fine grained resource control as provided by the MMU because the access policy has to be enforced by external memory controllers using coarse-grained regions.

Device virtualization using trap-and-emulate is possible but is limited to external device memory. The normal-world OS can trigger external aborts [4] by accessing secure or invalid device memory. Such aborts trap directly in the monitor if the secure-world OS has this feature enabled. This mechanism could be used to emulate access to devices. However, such aborts can be imprecise with respect to the program flow [5], which means that the processor does not wait for the completion of the instruction. Therefore, the secure-world OS has no chance to reconstruct the written value because the normal-world processor state has changed irreversibly since the instruction was executed.

Multiplexing two or more OSes inside the normal world is supported by our design. It requires careful handling of shared hardware resources, such as processor registers, memory, caches and TLBs. We encountered the maintenance of the TLB as an unsolvable problem. The TLB distinguishes normal-world and secure-world entries, but cannot separate entries from different normal-world OSes. That means one normal-world OS can use entries from another normal-world OS and break the encapsulation. To avoid this case a TLB flush is required before the activation of another normal-world OS. According to the TrustZone specification the monitor should be able to perform this task. However during our tests we were not able to achieve the desired behaviour, which remains as an open question.

Finally, recent processor cores add support for interrupt virtualization. The processor shadows the interrupt flag when it executes in the normal world and enters the monitor even if the normal world has interrupts disabled. Furthermore a virtual interrupt register is available to aid the injection of virtual interrupts from the secure world.

5 Evaluation

To evaluate the TZ-Linux approach we compare the required software-engineering effort of this work with L4Linux. We measured the performance in comparison with L4Linux and a non-virtualized Linux using micro-benchmarks and artificial workloads and analyzed the virtualization overhead for TZ-Linux.

Hardware Currently the following ARM cores include a TrustZone extension: the ARM1176JZ(F)-S, and all processors from the Cortex-A series, such as the Cortex-A5, Cortex-A8 and Cortex-A9.

Looking for a suitable hardware platform for the evaluation was a troubling experience. For example, some platforms do not allow the execution of own code inside the secure world. Rather, these platforms effectively lock the secure world after the boot process, such as the OMAP3EVM[5] and the

[3] The instruction is called wfi.

[4] An external abort is an exception generated by accessing external device memory.

[5] Whether an abort is precise or imprecise is determined by the attributes of the memory mapping of the normal-world OS.

Beagleboard[2]. All other platforms with TrustZone support known to us lack the ability to configure memory and devices into secure-world and normal-world accessible. Instead all resources of the platform are normal-world accessible.

Therefore we made the assumption that additional TrustZone-aware components, such as a memory protection controller or protection controller for peripherals can be added to a platform to enforce the security policy. This assumptions does not affect the soundness of the design and still allows us to evaluate the prototype implementation.

For the following evaluation, we chose the RealView Platform Baseboard with two Cortex-A9 CPUs (RealView/PBX) and 100 Mhz clock frequency. At a later stage we had the possibility to use an NVIDIA Tegra2 evaluation system, equipped with a dual-core Cortex-A9 CPU. The benchmarks were run with enabled caches.

Software The TZ-Linux and L4Linux are based on a Linux kernel of version 2.6.31. For the benchmarks Linux was configured with a 128MB memory partition and a periodic timer interrupt of 10 milliseconds. The performance measurements were conducted with the cycle-accurate performance counter of the processor. As a base line we used an unmodified Linux, called non-virtualized Linux, that runs with the same configuration but has direct access to all hardware resources.

5.1 Software-Engineering Effort

We count the number of changes that are required to run TZ-Linux on a virtual platform compared to the number of changes for a completely paravirtualized L4Linux kernel. We use source lines of code (SLOC) as a metric because it can be easily calculated and is used in the software development community to quantify the amount of development.

	TZ-Linux	L4Linux
Interrupt controller	67	250
Timer device	84	70
Memory Management	0	1100
Processor Management	0	200
System calls	0	100
Total	**151**	**1720**

TABLE 1: *Comparison of modifications in SLOC required in TZ-Linux and in L4Linux to virtualize the platform.*

First, we examine the required changes to modify the Linux kernel in order to run as a normal-world OS and a paravirtualized OS. Table 1 shows the results for TZ-Linux and L4Linux in terms of SLOC. In comparison to L4Linux, TZ-Linux requires approximately one tenth of modifications because the memory management and processor management need no adaption.

Second, we compare the size of the device models of the TZ-Linux VMM with the size of similar device models of the Qemu emulator [17] for the ARM Realview platform. Table 2 shows that the size of the device models to support TZ-Linux is about one tenth the size compared to the Qemu device models. For reference the figure shows also the size of the access drivers in TZ-Linux that are co-developed for the virtual devices in the VMM.

	Qemu device	TZ-Linux device	TZ-Linux driver
Interrupt controller	584	103	67
Timer device	228	72	84
Serial device	214	130	180
Block device	3500	67	210
Network device	500	210	320
Total	**5026**	**582**	**861**

TABLE 2: *Comparison of similar device models in SLOC in the TZ-Linux VMM and the Qemu VMM.*

According to our metric, the software-development effort to implement a virtual platform is considerably smaller than paravirtualizing the OS or providing support for a real hardware platform. There are mainly two reasons for the small code size of virtual drivers and devices. First, the interface between both is optimized for efficient parameter passing and simplified state transition. Second, the functionality is tailored without considering any legacy support.

The hypervisor contains the monitor software and the function to send a message to the VMM which contribute to the TCB of security-sensitive applications. The amount of this code is approximately 200 SLOC and much smaller than the size of the VMM.

5.2 Performance

In this section, we compare the performance of TZ-Linux with a non-virtualized Linux and L4Linux using the LMbench [23] micro-benchmark tool and artificial workloads. Furthermore, we analyze the virtu-

alization overhead for our architecture. The benchmarks are conducted using the RealView PBX board.

5.2.1 Benchmarks

Figure 8 shows the performance of arithmetic operations normalized to the execution time of a non-virtualized Linux. As expected, there is only insignificant overhead visible in TZ-Linux as well as L4Linux.

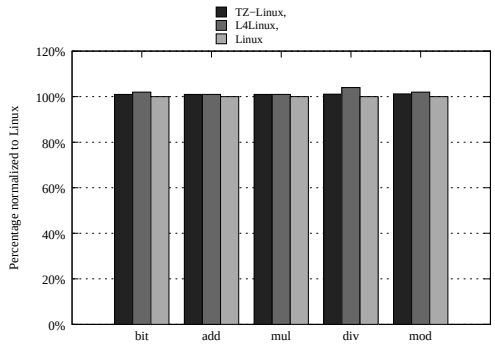

FIGURE 8: *LMbench3: Comparison of arithmetic operations with the execution time normalized to Linux.*

Figure 9 shows the performance of selected system calls normalized to the non-virtualized Linux. As expected, TZ-Linux has a insignificant overhead around 2 percent. L4Linux has large overheads (from 200 to 1750 %) because system calls are interposed by the virtualizing OS.

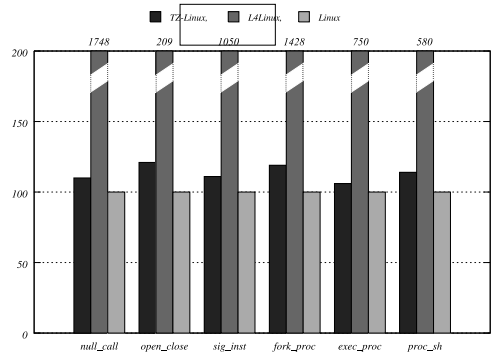

FIGURE 9: *LMbench3: Comparison of system call performance with the execution time normalized to Linux.*

To measure the behaviour of TZ-Linux under high pressure we use three workloads with different characteristics: a compute intensive workload that decodes an audio file, an IO-intensive workload that downloads a file from the local network using the *wget* command, and a mixed workload that compiles the jpeg library. In all scenarios the Linux has direct access to a network device and uses a network-based filesystem to launch the applications and access the data. Furthermore, we measure the configuration as described in Section 4.4.2, called TZ-Linux/secure. The network driver inside the VMM accesses the network device and TZ-Linux uses an access driver to send and receive network packets. Figure 10 shows all benchmarks with the performance normalized to the non-virtualized Linux. We measured the time spent in kernel mode and in user mode to point out their different characteristics. A high kernel-mode ratio indicates a high number of user-kernel interactions. While user-mode activity dominates the audio-decoding benchmark, kernel-mode activity dominates the file-download benchmark.

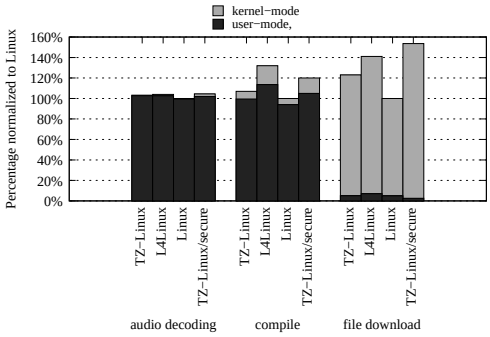

FIGURE 10: *Comparison of workloads with the execution time normalized to Linux.*

The performance of the audio-decoding benchmark is nearly the same for all versions of Linux because few user-kernel interactions are required. In the compile benchmark, L4Linux has 30% more overhead than TZ-Linux because it involves a high number of system calls. In the third benchmark TZ-Linux has an overhead about 23% and L4Linux has an overhead of roughly 40% compared to the non-virtualized Linux. This indicates that the routing of network interrupts through as secure-world OS dominates the observed performance degradation.

For TZ-Linux/secure the performance degrades compared to TZ-Linux with direct network access depending on the generated network traffic by the benchmark. There are two reasons for this performance degradation. First, TZ-Linux/secure has to make a request to the VMM for every send and receive operation. Second, the secure network driver inside the VMM requires additional context switches from the hypervisor to the VMM.

5.2.2 Virtualization Overhead

We examine the measured overhead for TZ-Linux in more detail. The total overhead depends on the virtualization overhead and concurrent access to shared resources, such as caches and TLB, by the normal world and the secure world. The virtualization overhead is determined by the time to handle one normal-world exit and the number of normal-world exits for the specific workload during the measurement period.

Phase	Description	RealView	Tegra2
1	Normal-world exit	8	0.9
2	Msg from HV to VMM	4.25	1.2
3	Request handling	14 - 26	4 - 8
4	Msg from VMM to HV	4.25	1.2
5	Normal-world entry	8	0.9
	Total	38.5 - 50.5	8.2 - 12.2

TABLE 3: *Execution time in microseconds for the phases of a request from the normal-world OS (Msg: Message, HV: Hypervisor)*

As discussed in Section 4.4.2 the handling of a request from the normal-world OS can be separated into five phases. Table 3 contains the average execution time of every phase measured with micro benchmarks. The time spent in the VMM to handle the request varies depending on the selected virtual device.

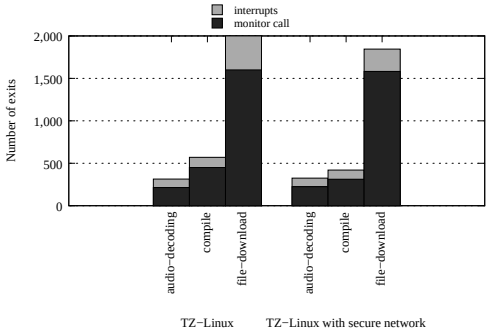

FIGURE 11: *Average number of normal-world exits during a period of 1 second*

Figure 11 shows the average number of normal-world exits for period of 1 second. There are two different types of normal-world exits that can occur in our architecture: interrupts and monitor calls from the normal-world OS.

The audio-decoding benchmark, which is compute intensive has the fewest number of normal-world exits. The periodic injection of timer interrupts is the only required interaction between normal-world OS and secure-world OS. The number of exits increases for the other two workloads because additionally network interrupts need to be handled more frequently.

The number of normal-world exits due to interrupts is higher for TZ-Linux with direct access to the non-secure network device compared to TZ-Linux with secure network device especially for the file-download benchmark. This fact indicates a higher service rate for network interrupts for the first configuration.

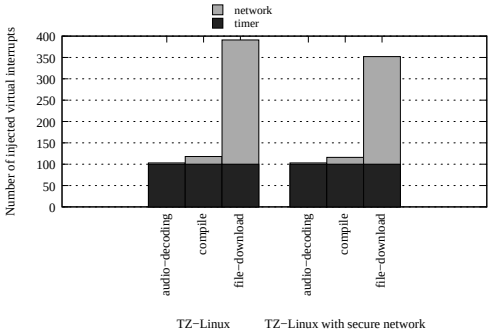

FIGURE 12: *Average number of injected interrupts during a period of 1 second*

Figure 12 shows the average number of injected virtual interrupts during a period of one second. Two types of interrupts can be injected in this setup: interrupts for the virtual timer device and interrupts routed for the network device. Network interrupts in TZ-Linux with indirect access to the secure network device are generated by a successful receive operation by the virtual network device inside the VMM. The number of timer interrupts is fixed in all three workloads because they are generated with a constant period of 10 milliseconds. However, the number of network interrupts depends on the workload of the benchmark. The audio-decoding benchmark requires no network access, while in the file-download benchmark network interrupts dominate.

TZ-Linux with a secure network device has fewer virtual network interrupts injected compared to the first configuration indicating a lower network throughput. For a detailed description of the reasons that cause the observed performance degradation additional examinations are required.

5.3 Latency

In order to study the latency of an application that has timely requirements we measured the time that passes in order to deliver an external event such as an interrupt.

The latency depends mainly on two factors: the length of the critical path to deliver the interrupt and the availability of critical resources that are shared with other components in the system. The first factor heavily depends in which world the component is running because it determines the length of the critical path. The second factor determines the number of caches and TLB misses that occur in the path of the interrupt delivery of the realtime application. (We do not consider memory as a scarce resource in our scenarios.)

First we examine the latency of realtime applications running the secure world and then look at the latency for normal-world realtime applications. All measurements are conducted with a Tegra2 board.

5.3.1 Secure-World Realtime Applications

For a secure-world realtime application only the hypervisor is involved in order to deliver the interrupt using a message. Furthermore, if the system is running in the normal world an additional world switch is required.

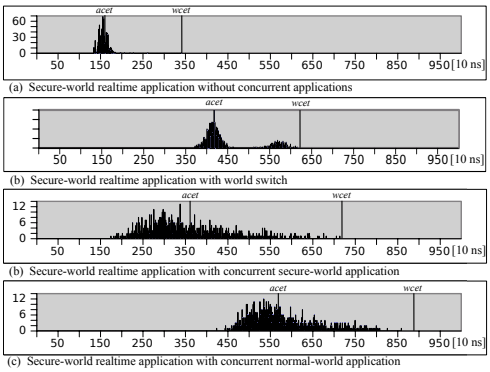

FIGURE 13: *Latency to deliver an interrupt event to a secure-world realtime application.*

Figure 13 shows the distribution of the latency in four different scenarios. Scenario (a) measures the latency without any resource pressure nor normal-world application. It servs as a baseline for the other scenarios. Scenario (b) increases the length of the interrupt path by adding a small normal-world application beside the realtime application. Scenario (c) runs a concurrent secure-world application that consumes as much cache and TLB resources as possible. Scenario (d) runs a concurrent normal-world application that is consuming resources.

The first scenario measures the latency of the interrupt path without pressure on TLB and caches. The interrupt is delivered from the hypervisor to the realtime application with a message resulting in latencies of 1.5 microseconds for the average case and 3.4 microseconds for the worst case. The second scenario enforces a world switch in order to deliver the interrupt. The latency increases by 2 microseconds, both, for the average and the worst case. In the third scenario another secure-world application contents on TLB and cache resources. In consequence the average and worst-case time increase to 3.7 microseconds and 7.2 microseconds respectively when compared to the first scenario. In the last scenario a normal-world application contents on the shared resources. This further increases the latency as an additional world switch is required in order to deliver the interrupt. The added latency is roughly the execution time of a world switch in comparison with the third scenario.

5.3.2 Normal-World Realtime Applications

If the realtime application is running in the TZ-Linux the latency to deliver an event increases compared to a secure-world application because the VMM is part of the interrupt path as shown in Figure 6. For the following discussion we assume that the priorities of all software components are configured with the highest priority. For example, that means the VMM cannot be preempted by another application with a higher priority in the system.

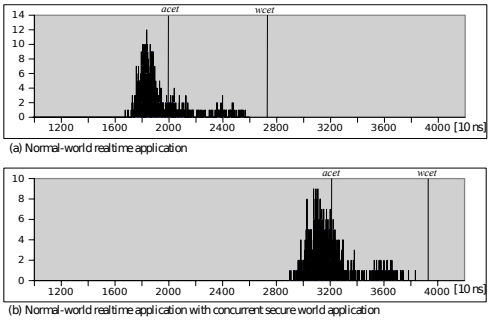

FIGURE 14: *Latency to deliver an interrupt event to a normal-world realtime application under two different load scenarios.*

Figure 14 shows the distribution of the latency in two different scenarios. Scenario (a) measures the latency of the interrupt path without resource pressure for the normal-world realtime application. Scenario (b) runs a cache and TLB flooder workload in the secure world to consume as much resources as possible.

The intention of the first scenario is to measure the 'pure' latency of the interrupt for the normal-world application without contention on resources.

The diagram shows the distribution of the execution time to deliver a timer interrupt to the TZ-Linux kernel. The average execution time is around 20 microseconds while the measured worst-case execution time is 2750 micoseconds. The high latency is caused by the interrupt routing for normal-world interrupts. Two secure-world components are involved during interrupt delivery, the hypervisor and the VMM. Interrupt injection takes place in several phases as described in Section 4.4.1 with a worst case execution time of each phase shown in Table 3. Due to the implementation of the interrupt controller model in the VMM there is an additional exit beside the exit for the host interrupt, which acknowledges the interrupt. So there are two exits required for one interrupt delivery.

In the second scenario another secure-world application consumes a much as cache and TLB as possible. The VMM (and therefore the normal-world OS) have a higher priority than the secure-world application avoiding starvation. The measured average case and worst case time are considerable longer than in the first scenario. Even we have no detailed knowledge of the execution time of each phase the reasonable assumption is that cache and TLB set an order of magnitude large than in the secure-world realtime application scenario.

In conclusion, the virtualization overhead as well as shared resource contention increase the measured latency and worst case time in reasonable ways for realtime workload running in the normal world.

6 Related Work

To the best of our knowledge this work is the first publication that examines and evaluates the capabilities and limitations of virtualization support of the TrustZone architecture in the embedded systems research community.

Trusted Logic [7] develops the Trusted Foundations Software that is deployed as reference implementation for secure world components in TrustZone-enabled systems. The goal is to offer a secure environment and a common framework for the integration secure software development. Operating systems running in the normal world can call secure services using the Trusted Foundations API.

Winter [26] instantiated a Linux systems in the normal world and an SE-Linux system in the secure world. Both Linux instances communicate using device model similar to the one presented here. However there is no performance evaluation available.

Yan-ling [27] describes a system with an SE-Linux system in the normal world providing isolation for secure components on top and accessing the secure-world OS using the provided TrustZone API. As the system is not implemented an evaluation is missing.

The work of Sangorrin et. al. [24] describes a similar architecture as presented in this paper running a small realtime OS in the secure world and a Linux-based OS in the non-secure world. However, realtime applications in the secure OS are not isolated and can compromise the complete system.

There is a large number of closed-source products available that support isolation of system-level components including operating systems. Green Hills [3] uses a security architecture that employs TrustZone technology to assist virtualization. Paravirtualization techniques are applied by solutions such as PikeOS [6], OKL4 [4], Trango [9] and VirtualLogix [8]. Until today all requests regarding information about the design or the performance or requests to evaluate the software were not successful, so that a comparison is not possible.

Xen[16] is popular virtualization solutions for desktop and server systems. The Xen hypervisor has been ported to ARMv5 architecture and uses paravirtualization to virtualize a normal-world OS [21]. The numbers for the LMbench benchmark presented indicate lower overhead than L4Linux shows but are still an order of magnitude higher than measured for TZ-Linux.

7 Conclusions

This paper presented an approach based on full virtualization using TrustZone to isolate an operating system as needed in state-of-the-art security architectures. The use of TrustZone's virtualization capabilities resulted in a much lower number of required changes to the Linux as a normal-world OS (about 153 lines of code) as compared to a paravirtualization approach. The device drivers and models needed for the implementation of the platform are in the order of 900 source lines of code. The resulting performance overhead ranges from barely measurable to up to 20% depending on the characteristics of the workloads. The realtime capabilities of the system strongly favor secure realtime applications while not exclude the execution non-secure realtime applications. We think that this work provides a baseline for upcoming virtualization capabilities of future ARM cores.

8 Acknowledgements

We would like to thank the European Commision for supporting us through the 7th Framework Program with the projects TECOM [6] and eMuCo [7].

References

[1] AMD Virtualization (AMD-V) Technology. Located at: `http://www.amd.com`.

[2] BeagleBoard. Located at: `http://beagleboard.org/`.

[3] INTEGRITY Secure Virtualization. `http://www.greenhillssoftware.com/news/20091021_ARM_TrustZone_Solution.html`.

[4] OKL4 Homepage. Located at: `http://www.ok-labs.com`.

[5] OMAP3EVM Evaluation Platform . Located at: `http://focus.ti.com/docs/toolsw/folders/print/tmdsevm3530.html`.

[6] Pikeos homepage. Located at: `http://www.sysgo.com/`.

[7] Trusted Logic Homepage. Located at: `http://www.trusted-logic.com/`.

[8] Virtual Logix Homepage. Located at: `http://www.virtuallogix.com`.

[9] VMWare Homepage. Located at: `http://www.VMWare.com`.

[10] Primecell infrastructure amba™ 3 trustzone™ protection controller, November 2004.

[11] Trusted mobile platform software architecture description, October 2004.

[12] Amba™ 3 trustzone™ interrupt controller, September 2008.

[13] ARM Security Technology Building a Secure System using TrustZone Technology, 2009.

[14] Trustzone address space controller (tzc-380) technical reference manual, March 2010.

[15] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security, 2004.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)*, pages 164–177, Bolton Landing, NY, October 2003.

[17] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. USENIX, 2005.

[18] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The nizza secure-system architecture. In *In IEEE CollaborateCom 2005*. IEEE Press, 2005.

[19] E. C. Hendricks and T. C. Hartmann. Evolution of a virtual machine subsystem. *IBM Syst. J.*, 18(1):111–142, 1979.

[20] M. Hohmuth. Linux-Emulation auf einem Mikrokern. Master's thesis, TU Dresden, August 1996. In German; with English slides. Available from URL: `http://os.inf.tu-dresden.de/~hohmuth/prj/linux-on-l4/`.

[21] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. pages 257 –261, jan. 2008.

[22] Intel Corporation. *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*, April 2005.

[23] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[24] Daniel Sangorrin, Shinya Honda, and Hiroaki Takada. Dual operating system architecture for real-time embedded systems. July 2010.

[25] P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves. Implementing embedded security on dual-virtual-cpu systems. *Design Test of Computers, IEEE*, 24(6):582 –591, nov.-dec. 2007.

[26] Johannes Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *STC '08: Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30, New York, NY, USA, 2008. ACM.

[27] Xu Yan-ling, Pan Wei, and Zhang Xin-guo. Design and implementation of secure embedded systems based on trustzone. *Embedded Software and Systems, Second International Conference on*, 0:136–141, 2008.

[6] http://tecom-project.eu/
[7] http://emuco.eu/