Affordable Separation on Embedded Platforms

Soft Reboot Enabled Virtualization on a Dual Mode System

Oliver Schwarz, Christian Gehrmann and Viktor Do

SICS Swedish ICT,
{oliver,chrisg,viktordo}@sics.se,
http://www.sics.se/groups/security-lab-sec

Abstract. While security has become important in embedded systems, commodity operating systems often fail in effectively separating processes, mainly due to a too large trusted computing base. System virtualization can establish isolation already with a small code base, but many existing embedded CPU architectures have very limited virtualization hardware support, so that the performance impact is often non-negligible. Targeting both security and performance, we investigate an approach in which a few minor hardware additions together with virtualization offer protected execution in embedded systems while still allowing non-virtualized execution when secure services are not needed. Benchmarks of a prototype implementation on an emulated ARM Cortex A8 platform confirm that switching between those two execution forms can be done efficiently.

Keywords: Dual Mode · Separation · Soft Reboot · Virtualization · Hypervisor · Embedded Systems · Security

1 Introduction

Embedded systems are becoming more powerful, distributed and globally connected. We see a transition from classical single function embedded systems to powerful collaborative special purpose computing devices often controlling sensitive or critical infrastructure functions, so called *cyber-physical systems*. In the past, software attacks were mainly targeting high performance computers such as desktop computers, laptops, and recently also mobile devices. This is about to change rapidly. Security threats against cyber-physical systems have become

a severe issue, requiring strong platform security protection techniques such as separation [26] without overly increasing performance or system costs.

The need for separation of security critical data and code on mobile devices motivated ARM to introduce the TrustZone technology [4], available for some (but not all) ARM systems. TrustZone is a System-on-Chip (SoC) isolation technique that establishes a high degree of separation between *trusted* and *non-trusted* execution, while keeping context switches fast. To distinguish between trusted and non-trusted address space, TrustZone adds an additional address bit to the bus system. In order to not break isolation, careful SoC adaptations at the design level of application specific integrated circuits (ASIC) are necessary to make memory interfaces, interrupt controllers, Direct Memory Access (DMA) devices etc. aware of that bit.

System virtualization is an alternative way to protect security critical assets [15,16]. However, in tiny embedded systems with limited hardware virtualization support, system virtualization implies a non-negligible performance overhead [12]. On the other hand, security services typically do not run on the system all the time. They can be scheduled on a regular basis to perform monitoring or be called upon demand (e.g., for secret key operations).

In this paper we propose an alternative system virtualization enabled approach for separation, based on dual mode execution, i.e., the ability of choosing between virtualized and non-virtualized execution mode, and switching between the modes through soft reboots. The goal of the solution is to provide separation while keeping both performance overhead and required SoC adaptations to a minimum. Only a few hardware adaptations to an existing architecture are required. In one of the typical use cases, a service for proving the device's identity to its environment wants to keep the authentication key secret from the rest of the system. The system would run non-virtualized in the majority of the time, but activate the trusted service domain only for the actual authentication process. The exchange of required challenge-response-messages throughout that process will happen via remote procedure calls (RPCs).

Different from general purpose hypervisors (also called virtual machine monitors (VMM)) such as Xen [19] and KVM for ARM [12], a hypervisor with the purpose of separation or monitoring has a more focused scope and several optimizations can be made. We have developed a tiny hypervisor for ARM Cortex A8 with focus on separation. It was recently released as open source, and isolation properties of one version of this hypervisor have been formally verified on binary level. Based on this hypervisor, FreeRTOS as main guest, and emulated ARM Cortex A8 hardware enriched by our hardware extensions, we have implemented the suggested approach for dual mode protected execution. Benchmark figures show the feasibility of the concept. The main costs for enabling isolated services consists of their decryption and the integrity check of those services and of the lightweight hypervisor. Returning to non-virtualized execution does not take much longer than the erasure of newly produced confidential data.

Contrary to other approaches, that are for example based on TrustZone or trusted computing enabled late launch [17], the solution presented in this paper

does not require any particular CPU architecture or extensions to the CPU, which keeps costs low and makes the concept applicable to a large set of embedded systems. Summarized, our solution offers the following benefits:

1. Trusted domains can be executed with guaranteed separation without causing performance overhead in phases where their services are not required.
2. If desired and the use case allows the resulting latency, the commodity OS can be paused throughout the protected phase, so that trusted domains can execute without the need of paravirtualization[1] of the commodity OS.[2]
3. The proposed protocol includes a secure boot scheme, so that confidentiality and integrity of hypervisor and trusted domains are maintained even in the presence of external accesses to their non-volatile storage.

2 Hardware and Protocol

We consider a concept that relies on minor adaptations on SoC design level to make it possible to run the system in two modes, *protected mode* and *normal mode*. In protected mode a dedicated hypervisor runs in the most privileged level on each CPU in the system and trusted guests (such as secret key services) can run separated by the commodity OS, while in normal mode no hypervisor needs to be present in the system, as depicted in Figure 1 for a single CPU system.[3] Priviliged software can cause transitions between modes by requesting a *soft reboot* (also referred to as *soft reset* or *warm reboot*), which is initiated by the system's reset signal.

The SoC contains two special purpose *volatile* memory registers: a *mode state register* and a *transition register*. The mode state register states whether the system is currently in protected or normal mode. The transition register is used to state the intention of commodity OS or hypervisor about which mode to enter. The mode state register can *only* be changed in early booting phases. Thereafter it will be locked through a sticky bit so that it can not be modified anymore until a chip reset (and consequently a soft reboot) occurs. The boot code responsible for the hypervisor and OS kernel launch determines which mode to boot into - and consequently the value to set in the mode state register. In a *cold boot* (full hardware reset) the default mode value is given by a boot configuration. In a warm/soft reboot the value is determined by the transition register, as set by the higher level software.

When running in protected mode, the hypervisor controls sensitive applications, I/O devices and data and can protect the system from illegal access

[1] Paravirtualization [28, p. 422] describes any modification of guest operating systems, in order to enable their execution on a virtualized environment instead of bare metal, e.g. by making them use software interrupts (*hypercalls*) to perform privileged operations, according to the hypervisor's API.

[2] Depending on the scenario, interrupts would be recorded by the hypervisor or just masked during the pause.

[3] Here, we illustrate a single CPU architecture, but the principle can easily be extended to a multicore architecture, see Section 2.1.

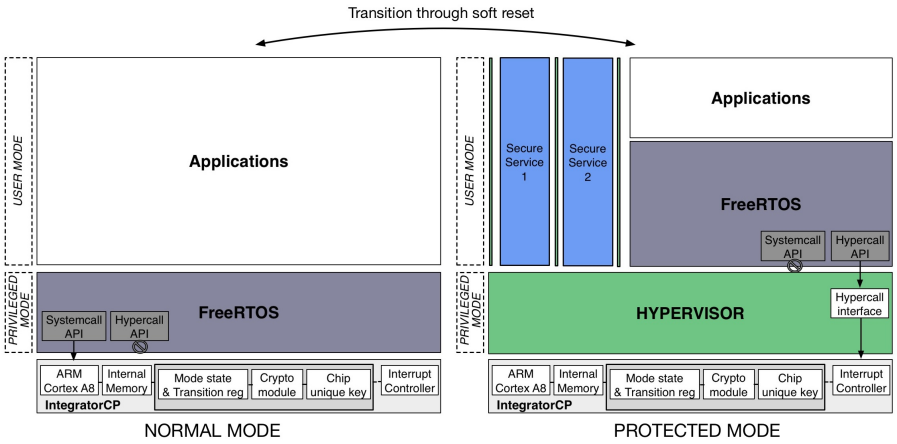

Fig. 1. Dual mode operation.

to these units. This can be achieved using the normal Memory Management Unit (MMU) or Memory Protection Unit (MPU) present in most systems. If applicable, additional hardware protection support can be utilized, such as an Input/Output MMU (IOMMU). The memory protection mechanisms are also used to make sure that, when running in protected mode, a soft reboot to normal mode can *only* be initiated by the hypervisor or hypervisor protected units, such as a watch dog timer reset function (placed in a protected address space). [4]

Figure 2 shows a SoC design according to the approach and the proof of concept implementation we have done using emulated hardware (see Section 5). In addition to the two special purpose registers, the SoC design includes one or several chip unique secret key(s), stored in non-volatile registers. They are used to decrypt and check the integrity of security critical code/data that is loaded into the chip internal or external RAM. To prevent any usage of the chip unique secrets in normal mode, they are tied to the mode state register and locked to protected mode. In our proof of concept implementation we have optimized performance with a fully functional cryptographic module, the *transition crypto module*. However, cryptographic operations can be performed in software as well, reducing the number of changes to integrated circuits, but at the prize of an increased performance overhead. If not mentioned otherwise, we assume the presence of a transition crypto module in the remainder of the paper.

In order to show how these SoC components are used in the suggested approach, below we describe the details of the cold boot, the transition from protected to normal mode and the transition from normal to protected mode.

[4] As discussed in Section 5.2, unprivileged software can at most achieve a soft reboot to protected mode or a cold reboot.

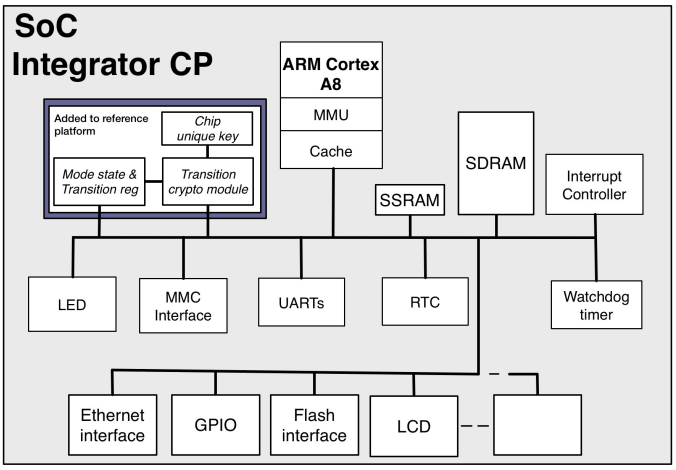

Fig. 2. SoC system view.

Cold Boot The following steps are performed in a cold boot:

1. After the machine is powered on, a first stage boot code is executed. To prevent security from being compromised, this code needs to be protected from modifications by storing it in write-protected memory such as on-chip-ROM.
2. The first stage boot code loads the integrity protected second stage boot code and boot configurations into on-chip-RAM. The second stage boot code and its configurations are protected with signatures verifiable with a public key stored in write-protected memory, such as ROM, or hardware registers, such as e-fuse registers.
3. The first stage boot code reads the verified boot configurations and writes the default boot mode (normal or protected) into the mode state register, which is then locked.
4. The first stage boot code launches the second stage boot code. Depending on the system and use case, one or several intermediate boot stages are processed until the boot code responsible for hypervisor or operating system launch is reached. We call this boot stage *transition boot stage*.
5. The transition boot stage reads the current value of the mode state register. If the register indicates normal mode, the operating system indicated in the boot configurations is launched. If the register indicates protected mode, the following steps are performed:
   (a) The transition boot code loads hypervisor, trusted guest(s) and data from external memory and verifies the integrity (e.g., by using a transition crypto module). The confidentiality of trusted guests is protected through fast symmetric encryption with a chip unique secret key. If required, confidentiality protection can also be applied to the hypervisor or parts thereof.

(b) If decryption and integrity verification in the previous step were successful, the transition boot code hands over the execution to the hypervisor. Otherwise, the transition boot stage code clears all security sensitive data on the system, writes "normal mode" into the transition register and issues a soft reset, so that the system reboots into normal mode. This allows the system to recover even if it could not be started into protected mode.

Transition from Protected to Normal Mode When the system is in protected mode and secure services are no longer needed on the system, the hypervisor switches the system back to normal mode, as follows:

1. All trusted guests currently running are halted by the hypervisor. If required, persistent data is stored, integrity and confidentiality protected. Subsequently, the memory of trusted guests is cleared.
2. All confidential hypervisor data is cleared from memory.
3. The hypervisor can choose to maintain non-confidential code/data in memory to avoid reloading and reinitializing when returning to protected mode. In that case, *Message Authentication Codes* (MACs) protecting the integrity are recomputed, given that the concerned memory regions have changed.
4. The hypervisor sets the transition register to "normal" and issues a SoC-wide (soft) reset signal. This can be done via the component containing the two special purpose registers. The resulting soft reboot of the system will keep the content of most volatile memories, which allows a rather quick booting process without the need to reload all code and data from non-volatile memories.

At reset, the system will be booted into normal mode (analogous to the previous paragraph) running the OS kernel in the most privileged CPU mode as "usual", i.e., as in a non-virtualized system (see Figure 1). Before handing over execution to the commodity OS, the boot code clears all registers to avoid that confidential data from a protected mode phase is leaked into normal mode.

Transition from Normal to Protected Mode When the system is in normal mode and one or several security critical services are required, the commodity operating system writes "protected" into the transition register and issues a soft reset signal. It can inform the hypervisor about requested services and their parameters by writing service request values into dedicated transition memory before the reset. Subsequently, the boot is performed in analogy to the cold boot into protected mode, retrieving mode information from the transition register. However, the commodity OS is not loaded again and, if chosen so, the non-confidential parts of the hypervisor (such as code, page tables, constants) are not either. In contrast to that, integrity verification is always performed, possibly even for new memory regions used by, for example, page tables created in the previous hypervisor session. If hypervisor memory has been compromised in normal mode or protected mode has not been active before, a fallback option will (re-)load the entire hypervisor from the storage as done in cold boot. Once the

system is rebooted, the hypervisor will check the requested secure service(s) by reading the transition memory and launch them with the given parameters after checking that both services and parameters are valid and sound. Alternatively, this information can be passed via a hypercall from the commodity OS, once it is invoked by the hypervisor.

On a mode transition in either direction the commodity OS is usually aware of the upcoming soft reboot and will pause active processes as well as store their contexts before releasing control. Those processes (kept in memory) can then easily be resumed in the new mode. Before the hypervisor or OS reconfigures the peripherals, it needs to check whether interrupts (masked throughout the soft reboot) have occurred. Depending on the use case, the boot code can also be used to record events in a queue. In typical scenarios, the user will be aware of the inherent latency.

2.1 Implementation Alternatives

Enforced Protected Mode through Watch Dog Timer An alternative realization of the presented approach connects the watch dog timer of the SoC to the mode state register, so that the timer can *only* be reset if the system is in protected mode. If not kept alive, the watch dog issues a soft reset. At soft reset, the transition boot stage code checks the status of the watch dog timer and if it has reached zero, the transition boot code will boot the system into protected mode, *independently* of the transition register. This *forces* the system into protected mode in some pre-defined time intervals, which can be useful for monitoring or to counteract denial of other trusted services.

Soft Reboot Enabled by TrustZone The ARM TrustZone technology for ARM11 and ARM Cortex embedded processors [4] offers support for creating two securely isolated virtual cores (or *worlds* as they are termed) on a single real core. Both *secure world* and *normal world* manage an own virtual MMU, as well as an own vector table and thus own exception handlers [13]. System hardware, including memory and peripherals, can be allotted to each world. This is realized by an additional address bit. However, that separation requires that peripheral devices are adapted to the setting. A transition between the worlds is initiated by a hardware interrupt or a *Secure Mode Call* (SMC), both invoking the so called *monitor mode*, which is responsible for context switches. The concept of turning a hypervisor on and off on demand, as described in this paper, can also be implemented based on TrustZone instead of the discussed hardware extensions. Bootloader, hypervisor, trusted guest and the current mode would then be kept stored in the memory of the secure world, which only executes code to realize the soft reboot transitions. The execution of all other software (including the hypervisor and the trusted guest) happens in the normal world. Soft resets would be realized through SMCs. One of the advantages of this variant is that no soft reboot specific hardware extension in form of, e.g., a mode state register is required, something which is especially useful when TrustZone is already present anyway. Furthermore, keeping assets in the secure world reduces the need

for crypto operations considerably. However, a secure boot scheme would still be needed to ensure that the hypervisor and the trusted guest(s) are loaded into the secure world memory confidentially and integrity protected. Hardware protected keys are therefore still required. Moreover, peripherals have to be adapted in order to maintain separation between the two worlds. This limitation together with the costs of the TrustZone extension makes a TrustZone driven implementation variant only preferable to the standard one if the soft reboot is to be enabled on an already existing system that (including its peripherals) supports TrustZone from the beginning.

Multicore Systems The presented solution is also applicable to multicore systems. Since the mode state is a global property to control access to the chip unique keys, all CPUs have to agree on the mode. Consequently, when in protected mode, all CPUs need to be protected by a hypervisor, irrespectively if they are running secure services or not. There would be some *master* hypervisor on the system, which has the responsibility to coordinate, to execute trusted services and to issue soft reboots. In order to switch from protected to normal mode, the master hypervisor would inform its neighbors and wait until it has received acknowledgments from all of them before issuing the actual reset signal. Likewise, when booting into protected mode, the master hypervisor will be booted first on the main CPU and then launch all other hypervisors.

3 Hypervisor

A prototype implementation for the described solution has been established on the basis of a type-1 hypervisor[5], available as open source from [27]. Its focus lies on providing security by MMU-supported separation and its isolation properties have been formally verified on binary level [11]. Following the system virtualization principle, it allows the parallel execution of multiple paravirtualized guests in user mode. Both Linux and FreeRTOS have been ported to the hypervisor. Isolation between guests can intentionally be relaxed by the possibility to communicate with well-defined and parameterizable RPCs via the hypervisor. In addition to inter-guest-separation, the hypervisor offers introspection features such as virtual guest modes that enable intra-guest-separation as well, for example in order to maintain the guest OS' kernel separation even when executing in the processor's non-privileged operation mode. The implementation of the hypervisor comprises 2717 lines of C code and 942 lines of assembly, resulting in a compiled binary of 31 KB. The hypervisor was developed for single-core ARMv5 and ARMv7 architectures and deployed on Beaglebone [9], Beagleboard [7], Beagleboard-xM [8], NovaThor [29] and the Integrator development board [3], as well as on emulated platforms within the OVP framework [24].

[5] Hypervisors of type 1, also called native hypervisors, are not running on any host OS, but on bare metal.

4 Software Adaptions

We have implemented a single-core prototype of the solution, based on FreeRTOS as commodity OS and the inhouse hypervisor for ARMv7 described in Section 3. Both FreeRTOS and the hypervisor had to be modified to support the soft reboot functionality, as described in this section. The trusted domain was easily implemented since it only needs to offer an entry point for receiving RPCs and the awareness about the RPC parameter passing protocol. Three different interrupt vector tables were configured and are mapped according to the mode; while the vector of the boot code is only referred to on reset, the hypervisor vector is active in protected mode and FreeRTOS' vector is either referred to directly (in normal mode) or used to receive control from the hypervisor. Otherwise, memory mapping is static and access rights only change in dependency to the current mode. Binaries are linked/built separately for each entity and, where required, encrypted and/or integrity protected before deployment.

Adaptations to the Commodity OS The core adaptation in the commodity OS consists of changes that enables it to run both as guest on top of the hypervisor and natively on bare metal with control over the privileged operation ring of the CPU. While in the latter setting, privileged operations are performed directly by the corresponding privileged instructions, hypercalls have to be used in the first setting. We added a dual API layer that selects the required implementation for each functionality in dependency of a mode indicating configuration bit set by the bootloader. Similarly, FreeRTOS was made able to switch between its own kernel separation enforcement and the kernel protection provided by the hypervisor. On startup, the commodity OS either performs its own hardware configurations or it registers itself to the hypervisor, before creating or resuming processes. Finally, we inserted code that makes use of the RPC functionality to communicate with a trusted domain and that actually initiates soft reboots for demonstration and benchmark purposes.

Adaptations to the Hypervisor The adaptations to the hypervisor were quite limited. Essentially, besides providing configuration information about commodity OS and trusted guest, only a hypercall needed to be added, that realizes the initiation of a soft reboot into normal mode, including the optional write back of the trusted domain and the erasure of all confidential data. The hypervisor makes use of the possibility to be partly kept in memory on soft reboot. In particular, this applies to the sections for code and constants, that both do not change throughout the system's uptime, and the page tables, for which a new MAC is computed after they are generated. Data section, BSS section, heap and stack are treated confidential and cleared before soft reboot. The data section is the only part that needs to be reloaded when coming back to protected mode, given that no memory corruptions have occurred in normal mode. Whether the hypervisor memory is still uncorrupted or had to be reloaded by the bootloader is indicated as argument to the hypervisor, so that page tables can be recomputed if necessary. Note that we migrated the responsibility of loading guests

from the hypervisor to the bootloader. Similarly, we decided to invoke trusted services via RPCs by the commodity OS after a soft reboot to protected mode instead of passing parameters about the desired service to the hypervisor. In that way, no decisions are required by the hypervisor upon boot, but control can simply be transfered to the guest's entry point directly.

Bootloader The implementation of the bootloader was carried out in a straight forward manner according to the protocol in Section 2, using a transition crypto module for cryptographic operations. In our implementation the bootloader is divided into two stages. The first stage boot code checks the transition register, loads and verifies the second stage boot code and is placed into ROM along with its vector table. The second stage boot code loads the commodity OS, the hypervisor and the trusted guest (depending on the mode), carries out needed verification steps and finally calls the commodity OS or hypervisor.

5 Evaluation

The approach can be implemented on many current embedded architectures with minor hardware changes (a few special purpose registers and hardware protected keys), as most of the functionality relies on existing hardware features and functions implemented in software, mainly the boot code and the hypervisor. To demonstrate our solution and in order to obtain benchmarks on its performance, we have implemented the described hardware extensions within the emulation framework OVP [24]. It allows to implement and simulate the behavior of new SoC hardware components with reasonable effort. The additional registers are realized as memory mapped device connected to a SoC (emulation) with an integratorCP platform that includes a single ARM Cortex-A8 CPU. The register extension has been wired to perform system resets when required. Furthermore, a dedicated transition crypto module has been modelled in OVP, allowing us to verify the required encryption/decryption and integrity check tasks. As OVP can not provide exact simulation times, especially with respect to peripherals, MMU and caches, the main purpose has been to test the concept as such and get a good picture of the performance one can expect. Hence, the transition crypto module is simplified with respect to its hardware interface and we allow direct memory read from the transition crypto module over the bus. This allows us to test the different boot cases and the concept, but not to simulate real transition crypto module data transfers from the CPU or via DMA. We believe those simplifications are reasonable since exact time estimates for these access forms can not be obtained in an emulation environment such as OVP anyway.

5.1 Performance

The suggested approach allows running secure services isolated by a hypervisor layer only when needed instead of permanently. Consequently, the secure services can be implemented with a very small performance impact. This comes at the

price of soft resets when the secure services are needed. The objective of our benchmarks is thus to estimate the overall costs for a soft reboot.

The evaluation includes three factors:

- the number of bytes copied (or erased) between/from storage devices (NAND flash, RAM),
- the number of bytes fed into the transition crypto module for en-/decryption or integrity value calculation and integrity checks,
- the number of remaining CPU instructions not involved in any such feeding, copying or clearing operations.

These figures together allow us to estimate the overall time for all steps of the suggested approach, making the following assumptions about the platform:

- The CPU is clocked with 720 MHz and nominally executes 200 MIPS, as typical on many Cortex A8 development boards such as BeagleBoard [7].
- We assume a rather conservative RAM copy speed of 150 MB/second, which is a lower estimate from [5].
- The copy speed from NAND flash to RAM is estimated by 6 MB/second [21].
- We assume a transition crypto module supporting SHA-256 HMAC generation and AES-128 en-/decryption with a fair trade off between size and speed, clocked at 174 MHz and with the ability to perform parallel hashing/encryption or hashing/decryption with the speed of 171 MB/second. Since hashing is the dominating work load in such parallelized operations, the feasibility of such a speed can be concluded from, e.g., [10].

Table 1 provides an overview of the results for the single steps required, depending on which transition is been considered. We distinguish between a cold boot into protected mode (cp), a cold boot into normal mode (cn), a (warm/soft) reboot into protected mode (wp) and a (warm/soft) reboot into normal mode (wn). Crosses (X) indicate which step is involved in which transition. A dash (-) indicates that the step in question is optional or does only occur in the first of typically many soft reboots.

The benchmarks are based on a second stage boot code of 2.9 KB, FreeRTOS as commodity operating system with a binary blob of 1 MB, the hypervisor sections for code and constants, together 30 KB, hypervisor data of 1 KB and a trusted domain of 380 KB. Those specifications refer to the initial volumes. However, we allow the trusted domain to grow up to 1 MB for the usage of stacks, data structures etc. The space reserved for the hypervisor's heap, stack and BSS section is 900 KB, while page table memory can be up to 64 KB.

A complete soft reboot cycle including two mode switches is with 19 milliseconds estimated considerably faster than any cold boot, irrespectively of the targeted mode. Avoiding slow accesses to external storage is responsible for the main share of those performance benefits. However, also the number of boot instructions is reduced in warm reboot, in respect to both the hypervisor and the commodity OS. In both cases this optimization is mainly due to the dispensed page table reconfiguration. Preconfigured page tables could reduce the

Table 1. Execution Costs per Step

step	cp	cn	wp	wn	Bytes accessed in storage	crypto module load in B	other instr.	estimated time in ms
configure registers and mode	X	X	X	X			29	0.0001
clean registers				X			41	0.0002
load + verify 2nd stage code	X	X	X	X	2,987	2,987	26	0.4916
load FreeRTOS	X	X			1,043,288		19	165.8263
load + verify hypervisor code	X				30,500	30,500	24	5.0181
verify hypervisor code			X			30,500	22	0.1940
load + verify hypervisor data	X		X		960	960	22	0.1581
verify hypervisor page tables			X			65,536	21	0.4168
copy encrypted trusted guest	X				389,732		0	61.9562
decrypt + verify trusted guest	X		X			389,732	24	4.9558
boot hypervisor	X						247,940	1.2397
*re*boot hypervisor			X				27,707	0.1385
boot FreeRTOS, normal mode		X					9,146	0.0457
*re*boot FreeRTOS, normal mode				X			140	0.0007
(re-)boot FreeRTOS, protected	X		X				305	0.0015
compute page table MAC				-		65,536	129	0.4173
(write back trusted guest)				-		1,048,576	216	13.3341
erase confidential memory				X	1,964,252		98	12.4889
initiate reset to protected mode			X				41	0.0002
cold boot, protected mode	X						248,389	239.6374
cold boot, normal mode		X					9,220	166.3637
warm reboot, protected mode			X				28,197	6.3566
warm reboot, normal mode				X			334	12.9815

hypervisor's booting phase also in a cold boot, but that would come at the costs of an increased foot print and less flexibility. Since we allow the trusted guest to grow to a size of up to 1 MB, writing it back (including MAC computation and encryption) is comparatively expensive, and so is its deletion. In order to optimize write back and clearing, one would need to narrow down the space actually claimed by the trusted guest. However, writing the trusted guest back might not be needed in many cases and is therefore listed as optional. The share of cryptographic operations on the estimated costs of a warm reboot to protected mode is 88%. Clearing confidential data is constituting the main part of the costs when soft rebooting into normal mode. We believe that the soft reboot performance is more than reasonable in settings where a hypervisor is only needed sporadicly. Assuming the estimations from above and a hypervisor overhead of at least 2%, an execution phase of 1 second in which secure services are not required is already enough to make a temporarily deactivation of service and hypervisor through a soft reboot profitable. As the soft reset, different from a full reset, keeps all volatile memory content, soft reboots are also considerably faster than cold reboots with full resets. In order to achieve the same functionality

of enabling and disabling virtualization on demand with full resets, additional costs to the ones listed above would arise, for example for storing application data before rebooting.

5.2 Security

Attacker Model We assume that the attacker has full control over the commodity OS. However, the hypervisor is supposed to be free from vulnerabilities, which can be assured by formal verification. Furthermore, we trust CPU, MMU and BIOS. Hardware attacks are out of scope of this paper. Devices are assumed to reset whenever a reset signal is issued.[6] In particular, no previously pending DMA operations will be performed after the reset until DMA controllers are reprogrammed. We furthermore assume that the hypervisor is aware of the specification of all present DMA devices, so it can intercept accordingly, and that the devices' behavior actually follows their (non-hostile) specifications. Alternatively, an IOMMU can be used to protect against DMA attacks.

We assume that the attacker aims at obtaining confidential data about the trusted guest and/or to affect its execution outside of the controlled communication channel provided. Denial of Service (DoS) attacks are out of scope of this paper, since a malicious commodity OS has the ability of shutting down the machine or otherwise introducing delays anyway. However, making the watch dog timer aware of the mode state register as described in Section 2.1 improves the protection against DoS attacks, even though complete protection is not achieved by this enhancement either.

Protection in Different Execution Phases In the following, we discuss the different aspects of the system's security in detail.

Execution in Normal Mode When in normal mode, the trusted guest and confidential parts of the hypervisor are stored in encrypted form. Access to the corresponding chip unique key(s) is rejected.

Entering Protected Mode The system can only enter protected mode along with the execution of a trusted and unmodifiable bootcode. In order to change the mode register, it needs to be unlocked. It is guaranteed by hardware that this unlocking is performed together with a CPU reset. The reset sets the program counter to a fixed address pointing to the bootcode in ROM. On ARM processors, neither this address nor the endianess or the instruction set used after reset can be changed by the commodity OS, even when running in privileged mode, since the values for those system parameters are copied from the System Control Register (`SCTLR`) register of coprocessor 15 which in turn is set back to default values first on reset [2, pp. B1-1202, B1-1203]. In particular, the MMU is disabled

[6] For functionality, the operating system or hypervisor respectively needs to wait until devices have finished pending tasks before issuing a reset signal. However, the specific time of a reset has no effect on the security.

[2, p. B3-1308], so that the used entrance point of the exception vector table can not be translated to a different address. Standard interrupts are masked by the reset and not unmasked before control has been transferred to the hypervisor. Fast interrupts are disabled by the boot code, even though there are no devices tied to fast interrupts in our setting. The remaining bits of the Current Processor State Register (`CPSR`) are set to default values by the boot code. If the integrity verification of either hypervisor or trusted guest fails, the memory is cleared and a reset to normal mode is enforced, so that compromised software will never be executed.

Execution in Protected Mode The hypervisor is the first software invoked by the boot code. It configures the system's memory protection in such a way that the hypervisor code and data, the trusted domain, the transition crypto module, chip unique keys and register extensions are inaccessible to guests. All exceptions are mapped to handlers under the control of the hypervisor.

Leaving Protected Mode In order for the commodity OS to (re-)gain privileged rights, a reset has to be issued, since the hypervisor is maintaining control over the system in all other cases. From a functional perspective, this is ideally done through the hypervisor by sending an unlock request to the mode register. However, from a security point of view we have to assume that the attacker can establish a reset signal at any arbitrary time. In case this happens when the transition register is (still) set to protected, the system will either get back to a state where the hypervisor is in charge or (if integrity verification fails) all data will be erased and the mode changed to normal before booting the commodity OS. Even achieving one or several more reset signals during the soft reboot process will not be of any benefit to the attacker since she has no possibility to set the transition register to normal during that phase. In the other case that the transition register is set to normal before reset, the system has either been in normal mode anyway (and confidential data is not present) or the hypervisor has already erased all confidential data (as required by the protocol before setting the transition register back to normal). The MMU is preventing unprivileged access to the transition register. Multiple randomized overwriting of confidential memory regions can be used instead of single overwriting, if deleted information must to not be retrievable in hardware forensics. Before handing over execution to the commodity OS, the boot code clears all registers to avoid that confidential data from a protected mode phase is leaked to normal mode.

Further Aspects

DMA Devices In normal mode, devices do not have any more privileges than the commodity OS. In protected mode, the hypervisor is able to intercept all attempts to program DMA devices or can configure an IOMMU to protect security critical parts of the memory. On soft reboot, pending DMA tasks are canceled. In particular, the only DMA operations performed during the booting phase are those executed with respect to the (trusted) transition crypto module.

Proof of Mode A design assumption of our solution was that the fact that the system is running in protected mode will be proven to the user by functionality. For many common applications (e.g., for secret key services such as signing) it is impossible for the attacker to make the user believe the trusted application was active if it was actually not. However, alternative embodiments are possible where a secret is displayed to the user or a LED is tied to the mode register.

6 Related Work

In [18] IBM describes a method for directing the system's reset signal to a specific partition in a virtualized setting. The method is therefore another suggestion on how to make use of reset functionality in virtualized environments, but does not address virtualization overhead.

Instead of disabling virtualization completely when it is not needed, a natural first step is to reduce its costs to a minimum. For example, in specific I/O operations hypervisors can be bypassed [20]. However, this requires hardware support and applies only to a subset of all (I/O) operations. Naughton et al. [23] discuss approaches to extend the Xen hypervisor dynamically by loading additional modules on runtime. In that way, the usage of space and other resources can be optimized. Still, a basic instance of Xen would always be active, something we avoid in our solution.

How to turn off a hypervisor while keeping other software running has been demonstrated for a machine with a dedicated processor mode for virtualization [14]. However, in many embedded architectures - for example on the common ARMv7-processor - the additional requirement of lifting the operating system to the privileged ring needs to be accomplished as well. Furthermore, the soft reboot approach described in the present paper allows turning on the hypervisor (again), guaranteeing the integrity protection of both the booted hypervisor and additional guests while the hypervisor is off.

The separation facilities provided by TrustZone (see Section 2.1) can be used to execute trusted services isolated without suffering from the performance overhead introduced by virtualization and without the need of paravirtualizing the commodity OS. At the same time, other CPUs on the system stay unaffected, which can be seen as additional advantage over the soft reboot approach, which requires all CPUs to agree on the mode. However, even if considering a system with a CPU already supporting TrustZone (which is not given for many embedded processors, such as CPUs with ARMv5 architecture), using TrustZone to execute software isolated requires from the SoC that peripherals are adapted in order to respect the extended address format and thus maintain separation between the two worlds. In contrast, the solution presented in this paper requires only minor additions to the SoC. If the execution of several isolated services or a symmetric protection between service(s) and commodity OS is required in a TrustZone solution, the secure world will need to run a separation kernel, as used in the proposed soft reboot solution as well. Note that TrustZone based approaches still need to make sure that trusted services are kept confidential before

being loaded from external storage to the secure world. To achieve this, further hardware extensions are required in order to provide a secure boot scheme.

An alternative way to securely invoke a hypervisor at an arbitrary point of time is provided by *trusted computing* technology [30]. Similar to our solution, trusted guests (and hypervisor) would be kept encrypted and integrity protected until a cryptographic hardware module (in that case the *Trusted Platform Module* (TPM)) decrypts and verifies them. However, in this method called *sealed storage*, the collaboration of the decrypting module does not depend on a mode, but on binaries loaded to the system. Applying the *late launch* technology, as available for modern Intel and AMD processors, this check ignores already loaded software and instead ensures that a dedicated secure load block (SLB) is executed. Only a loaded and unmodifed SLB will enable the decryption of the sealed data [17,1,22]. This principle is comparable to the entanglement of the mode register's unlocking and the reset that enforces the execution of the first stage boot code in our approach. However, not only is the technology not available for embedded systems, it has also been demonstrated that late launch can be circumvented and hypervisors can be modified by malicious code injected to the system before the late launch [31,25]. Even if this attack cannot be applied to all architectures and the vulnerability might be fixed in the future, it gives reason to doubt that TPM-based solutions provide a holistic principle covering the entire system. Furthermore, TPM-operations are comparatively expensive, due to a slow bus connection and relatively slow asymmetric decryption algorithms. A proper (and still simple) mode aware cryptographic module (with DMA support), which we suggest for our approach, is more efficient and cost-effective and does not require any modifications to the CPU.

Making use of the same enablers (sealed storage and late launch), the Flicker environment [22] focuses on the isolated execution of single trusted applications instead of the delayed activation of a hypervisor. This decision against virtualization certainly decreases the trusted computing base even more, but comes with the drawback that the commodity operating system has to be paused while the trusted application is being executed and that only one trusted service can be active at a time. A similar functionality to the one of Flicker can be achieved with the hardware extensions that we propose. However, the feature of remote attestation is naturally reserved to platforms with trusted computing support. Furthermore, [22] admittedly provides a stronger protection against replay attacks even without further hardware extension.

SICE [6] makes use of x86's System Management Mode (SMM) to provide an asymmetric isolation between commodity OS and isolated software, based on a TCB including only the hardware, the BIOS and the SMM with a software foundation of 300 LoC (excluding cryptographic libraries). However, isolated software can not access peripherals directly and - as the authors point out themselves - since the SMM was not designed with security in mind and several attacks on it are already known, careful security reviews are necessary before deployment. While still seeming to be a promising approach for asymmetric isolation on x86 systems, SICE's principle is not applicable to embedded systems.

7 Conclusion

We have presented a dual mode approach to turn the system's hypervisor on and off on demand. Integrity and privacy of trusted guests are maintained at all times: while virtualization is active (in *protected mode*), while it is not (in *normal mode*), and while the machine is powered off. The solution requires only minor additions to an existing SoC design, namely two new registers and hardware protected keys. Hardware support for the cryptographic operations guarantees efficiency. No extensions to the CPU or adaption of other devices are needed. The performance measurements of a prototype implementation in emulated hardware show that soft reboots can provide benefits in several scenarios for embedded systems. In particular, the efficiency is higher than when performing a cold reboot or maintaining virtualization while not needed. The main costs for enabling isolated services consists of their decryption and the integrity check of those services and of the lightweight hypervisor. Returning to non-virtualized execution does not take much longer than the erasure of newly produced confidential data. Furthermore, paravirtualization is not necessary in settings where the commodity OS can be paused while in protected mode. We leave the formal verification of our approach as possible future work.

8 Acknowledgements

Work supported by framework grant "IT 2010" from the Swedish Foundation for Strategic Research.

References

1. AMD: AMD64 virtualization: Secure virtualization: Secure virtual machine architecture reference manual. AMD Publication number 33047, revision 3.01 (2005)
2. ARM: ARMv7-A architecture reference manual, issue C. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c`
3. ARM: Integrator baseboards. `http://infocenter.arm.com/help/topic/com.arm.doc.subset.boards.integratorbaseboards`
4. ARM: TrustZone Technology. `http://www.arm.com/products/processors/technologies/trustzone.php/`
5. ARM Technical Support Knowledge Articles: What is the fastest way to copy memory on a Cortex-A8? `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13544.html` (2011)
6. Azab, A.M., Ning, P., Zhang, X.: SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 375–388. ACM (2011)
7. BeagleBoard.org Foundation: BeagleBoard product page. `http://beagleboard.org/Products/BeagleBoard`
8. BeagleBoard.org Foundation: BeagleBoard-xM product page. `http://beagleboard.org/Products/BeagleBoard-xM`
9. BeagleBoard.org Foundation: BeagleBone product page. `http://beagleboard.org/Products/BeagleBone`

10. Chaves, R., Kuzmanov, G., Sousa, L., Vassiliadis, S.: Improving SHA-2 hardware implementations. In: In Workshop on Cryptographic Hardware and Embedded Systems, CHES 2006 (2006)
11. Dam, M., Guanciale, R., Khakpour, N., Nemati, H., Schwarz, O.: Formal verification of information flow security for a simple ARM-based separation kernel. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS'13) (2013)
12. Ding, J.H., Lin, C.J., Chang, P.H., Tsang, C.H., Hsu, W.C., Chung, Y.C.: ARMvisor: System virtualization for ARM. In: Linux Symposium (2012)
13. Douglas, H., Gehrmann, C.: Secure virtualization and multicore platforms state-of-the-art report. Tech. Report. http://soda.swedish-ict.se/3800/ (2009)
14. Gábriš, F.: Turning off hypervisor and resuming OS in 100 instructions. Presentation at FASM CON 2009, Myjava, Slovak Republic, http://fdbg.x86asm.net/Turning_off_hypervisor_and_resuming_OS_in_100_instructions.ppt
15. Goldberg, R.P.: Architectural principles of virtual machines. Ph.D. thesis, Harvard University (1972)
16. Goldberg, R.P.: Survey of virtual machine research. IEEE Comp. Magazine (1974)
17. Grawrock, D.: The Intel safer computing initiative: Building blocks for trusted computing (2006)
18. Harrington, B.R., Mehta, C., Milton, D.M.I., Perez, M.A., Randall, D.L., Willoughby, D.R.: System and method for selectively executing a reboot request after a reset to power on state for a particular partition in a logically partitioned system. US patent US 7146515 B2, http://www.google.com/patents/US7146515
19. Hwang, J.Y., Suh, S.B., Heo, S.K., Park, C.J., Ryu, J.M., Park, S.Y., Kim, C.R.: Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In: CCNC (2008)
20. Liu, J., Huang, W., Abali, B., Panda, D.K.: High performance VMM-bypass I/O in virtual machines. In: Proceedings of the annual conference on USENIX '06 Annual Technical Conference. pp. 3–3. ATEC '06, USENIX Association, Berkeley, CA, USA (2006)
21. Make Linux Software: Super fast boot of embedded Linux. http://www.makelinux.com/emb/fastboot/omap
22. McCune, J.M., Parno, B.J., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: an execution infrastructure for TCB minimization. SIGOPS Oper. Syst. Rev. 42, 315–328 (2008)
23. Naughton, T., Vallee, G., Scott, S.: Dynamic adaptation using Xen. In: System-level Virtualization for High Performance Computing (HPCVirt) (2007)
24. Open Virtual Platforms: OVP website. http://www.ovpworld.org/
25. Schellekens, D.: Design and Analysis of Trusted Computing Platforms. Ph.D. thesis, Katholieke Universiteit Leuven (2012)
26. Shafi, Q.: Cyber physical systems security: A brief survey. In: Computational Science and Its Applications (ICCSA). pp. 146–150 (2012)
27. SICS: SICS Thin Hypervisor (STH) source. https://bitbucket.org/sicssec/sth
28. Smith, J.E., Nair, R.: Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann Publishers, USA (2005)
29. Sony Mobile: NovaThor U8500 product page. http://developer.sonymobile.com/knowledge-base/technologies/novethor-u8500/
30. Trusted Computing Group: PC client specific TPM interface specification. Version 1.2, Revision 1.0 (2005)
31. Wojtczuk, R., Rutkowska, J.: Attacking Intel trusted execution technology. Black Hat DC (2009)