VOSYSmonitor, a Low Latency Monitor Layer for Mixed-Criticality Systems on ARMv8-A*

Pierre Lucas[1], Kevin Chappuis[2], Michele Paolino[3], Nicolas Dagieu[4], and Daniel Raho[5]

1 **Virtual Open Systems, Grenoble, France**
`p.lucas@virtualopensystems.com`

2 **Virtual Open Systems, Grenoble, France**
`k.chappuis@virtualopensystems.com`

3 **Virtual Open Systems, Grenoble, France**
`m.paolino@virtualopensystems.com`

4 **Virtual Open Systems, Grenoble, France**
`n.dagieu@virtualopensystems.com`

5 **Virtual Open Systems, Grenoble, France**
`s.raho@virtualopensystems.com`

Abstract

With the emergence of multicore embedded System on Chip (SoC), the integration of several applications with different levels of criticality on the same platform is becoming increasingly popular. These platforms, known as mixed-criticality systems, need to meet numerous requirements such as real-time constraints, Operating System (OS) scheduling, memory and OSes isolation.

To construct mixed-criticality systems, various solutions, based on virtualization extensions, have been presented where OSes are contained in a Virtual Machine (VM) through the use of a hypervisor. However, such implementations usually lack hardware features to ensure a full isolation of other bus masters (e.g., Direct Memory Access (DMA) peripherals, Graphics Processing Unit (GPU)) between OSes. Furthermore on multicore implementation, one core is usually dedicated to one OS, causing CPU underutilization.

To address these issues, this paper presents VOSYSmonitor, a multi-core software layer, which allows the co-execution of a safety-critical Real-Time Operating System (RTOS) and a non-critical General Purpose Operating System (GPOS) on the same hardware ARMv8-A platform. VOSYSmonitor main differentiation factors with the known solutions is the possibility for a processor to switch between secure and non-secure code execution at runtime. The partitioning is ensured by the ARM TrustZone technology, thus allowing to preserve the usage of virtualization features for the GPOS.

VOSYSmonitor architecture will be detailed in this paper, while benchmarking its performance versus other known solutions.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases VOSYSmonitor, ARM TrustZone, mixed criticality, real time

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.6

* This research work has been supported by the FP7 DREAMS project under the grant agreement number 610640.

29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna; Article No. 6; pp. 6:1–6:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

An important challenge in the design of embedded systems is the consolidation of software applications with different levels of criticality on a common hardware platform. In the automotive domain, a common practice to isolate safety critical applications is through the proliferation of multiple hardware Engine Control Units (ECUs) [8], which are dedicated to basic operations, such as lowering the windows, to critical tasks as Electronic Braking System (EBS), engine control and digital dashboard applications. This is a highly inefficient way of using the available processing power since many of these ECUs are typically not used at their full potential. However, recent multi-core architectures with new hardware extensions (e.g., virtualization, TrustZone) enable the execution of multiple applications on the same platform safely and securely, thus reducing costs and vehicle weight, helping to increase efficiency.

The consolidation of different OSes on the same platform implies the concurrent execution of a critical OS with stringent real-time requirements [7] with non-critical applications. As a matter of fact, connected cars are required to support safety-critical control functions, such as EBS and Electric Power Assist Steering (EPAS) that have to be securely isolated from the In-Vehicle Infotainment (IVI) system. Similarly in avionics, functions are usually classified either as *flight-critical* (necessary for ensuring a safe flight) or *mission-critical* (essential for business execution) [25]. In this context, the main challenges are to integrate real-time tasks execution with software applications of a GPOS.

In the past, virtualization has been presented as a solution to isolate OSes in a VM. This approach offers the advantages of reducing implementation costs by abstracting the host platform. Additionally, features provided by hypervisors, such as memory partitioning, CPU and interrupts abstraction help the OSes isolation. However, the use of a hypervisor may cause performance overheads, and therefore the critical execution path of real-time operations may not be ensured.

In this context, Virtual Open Systems has developed VOSYSmonitor, a software monitor layer, which enables the native concurrent execution of a safety critical RTOS (or another type of OS) along with a GPOS with the option to use virtualization extensions, such as Linux/KVM, in order to instantiate a variety of different VMs. The monitor layer is the highest secure operating mode available on ARM processors, designed with the hardware security extension ARM TrustZone [16], which manages the interaction between two execution worlds (see Section 2). In this context, VOSYSmonitor has been designed for the ARMv8-A architecture by guaranteeing peripherals and memory isolation between both OSes with ARM TrustZone. The main advantage of such a solution is to allow dynamically cores sharing between both applications, thus offering a close to native performance. To achieve this, VOSYSmonitor supports a context switch mechanism with a minimal overhead (see Section 3.3).

The rest of this paper is organized as follows. In Section 2, there is a brief introduction of the ARM TrustZone technology. Section 3 describes VOSYSmonitor architecture as well as its main features. Section 4 outlines the works related to this paper and emphasizes the advantages/drawbacks compared to VOSYSmonitor. Methods and benchmarks are explained and detailed in Section 5. Finally, Section 6 summarizes this work findings and directions for future works.

2 TrustZone

ARM TrustZone is a hardware security extension, which provides a system-wide security approach by integrating protective measures into ARM processors, bus and system peripherals.

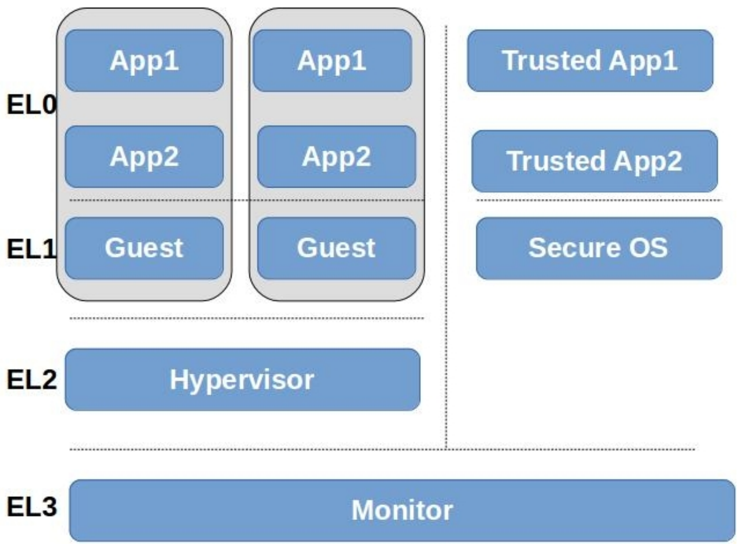

Figure 1 ARMv8-A Execution Level overview.

The security of the system is achieved by partitioning the hardware and software resources in two compartments: the Secure and the Normal worlds. The Secure world is usually used during the boot process in order to enforce a chain of trust. Indeed, starting with an implicitly trusted component, every other runtime binaries can be authenticated before their execution. In this context, some security specific configuration as well as sensitive data and peripherals can be only accessible from the Secure world. On the other hand, the Non Secure World is intended to host a rich operating system (e.g., Android or Linux). Security sensitive operations, such as the access to a private key or the interaction with a real time task, are provided to the non-secure application running in this compartment by the services run in the Secure World. Moreover, TrustZone enables a single core to safely and efficiently execute code from both worlds, allowing to save silicon area and power since a dedicated security processor is not needed.

While TrustZone is present since ARMv6 architecture [11], ARMv8-A provides a new architecture related to the TrustZone management. Indeed, a new highest Exception Level (EL) called EL3 (i.e., secure monitor mode) manages the interaction between these two compartments. This layer is always considered secure regardless of the current CPU state. Moreover, it manages the context switch between both worlds by triggering hardware exceptions, such as interrupts, synchronous and asynchronous events. The kernel level (EL1) and user level (EL0) are available in both worlds. This allows the execution of Trusted Execution Environment (TEE) in the Secure world, while the Normal world is expected to run a rich OS with the option to use virtualization extension, such as Linux-KVM, since the hypervisor mode (EL2) is only available in the Non-secure side.

The isolation provided by TrustZone is stronger than virtualization technology since this latter is restricted to the processor through the implementation of a hypervisor. Any other bus masters in the system, such as DMA peripherals or GPUs, can bypass the protections provided by the virtualization layer. In fact, ARM processors supporting security extensions along with TrustZone compliant Memory Management Unit (MMU) ensure the isolation of CPU execution mode, interrupts, hardware peripherals, memory and caches. However, the hypervisor can manage these peripherals for security reasons at the cost of introducing overhead.

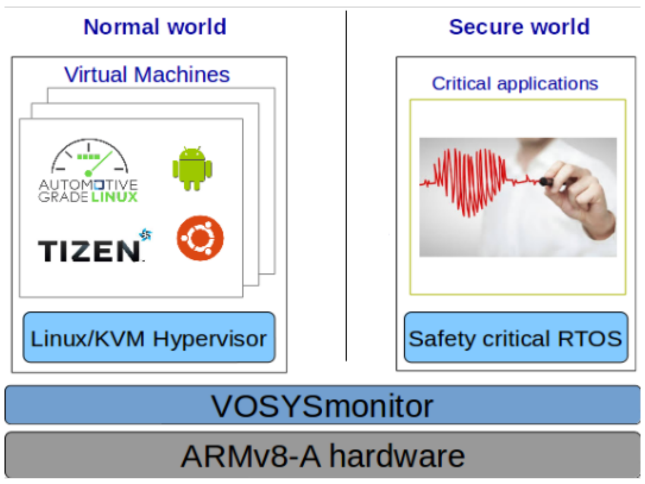

Figure 2 VOSYSmonitor overview.

3 VOSYSMonitor

In this section, VOSYSmonitor architecture as well as its interaction with the Secure/Normal worlds are described.

3.1 Architecture overview

VOSYSmonitor is a firmware that runs in the Secure Monitor mode (EL3) of ARMv8-A processors. As shown in Figure 2, it enables the native concurrent execution of two operating systems, such as for example a safety critical RTOS and a GPOS with the option to use virtualization extensions, such as Linux-KVM, in order to instantiate a variety of different VMs. VOSYSmonitor is a 64-bit monitor layer, which allows the execution of both 32-bit and 64-bit applications. Moreover, it ensures the isolation of each world by using the hardware security extension called TrustZone and provides, at the same time, functions to enable a safe and secure communication between them. Therefore the RTOS, running in Secure world, is totally isolated from applications executing in the Normal world. Finally, VOSYSmonitor manages the context switching between the two worlds by triggering a Secure Monitor Call (SMC) instruction or by hardware exception mechanisms, such as interrupts. VOSYSmonitor oversees these exceptions in order to ensure a correct operation for each world. The implementation of VOSYSmonitor is based on TrustZone to run another secure instance of an RTOS, while virtualization can be used as an additional option in the Normal world, such as XEN or Linux-KVM, giving the possibility to implement an additional layer of isolation between normal world applications.

It is important to note that VOSYSmonitor is not a boot loader. In this type of architecture, the boot process is usually split in different boot stages. Usually, the first two stages are platform specific and often provided by the board maker. As a matter of fact, the boot flow seen on Figure 3 is defined as follows:

- Primary Boot Loader(1) is the first stone of the chain of trust. It is generally implemented in the Read Only Memory (ROM), which is the only component that can not be modified or replaced by simple reprogramming attacks. It is responsible for initializing critical peripherals and authenticating the Secondary Boot loader located in external non-volatile storage.

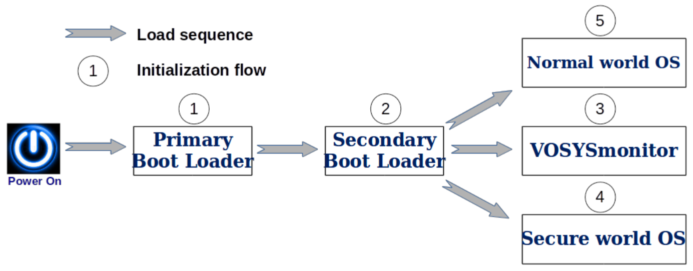

Figure 3 Trusted boot and load sequence on an ARMv8 platform.

- Secondary Boot Loader(2) is in charge to load and authenticate the different runtime binaries (e.g., Normal/Secure world applications and VOSYSmonitor), then it switches to VOSYSmonitor.
- VOSYSmonitor(3) performs basic initialization operations, such as ARM EL3 configuration, platform peripherals initialization and secure services setup, then, it gives the control to the Secure world(4) application. Once the latter has finished its own initialization, it requests a context switching through VOSYSmonitor in order to execute the Normal world(5) firmware/software (e.g., u-boot, UEFI, etc.).

VOSYSmonitor is designed to ease the support of new hardware platforms, as well as the integration of Trusted Operating Systems in the Secure world. Depending on the hardware requirements, VOSYSmonitor can reuse software components, such as common peripheral drivers, in order to minimize the integration effort. The top level architecture of VOSYSmonitor is seen in Figure 4, where different sub-systems of the firmware are depicted, including:

- EL3 Monitor Layer, mostly coded in ARM assembly, is specifically implemented targeting the ARMv8 architecture. It handles the world context switch operations triggered by SMC instruction or hardware exception mechanisms.
- Secure OS service layer is the interface between the Trusted OS, running in the Secure world (S-EL1), and VOSYSmonitor. The Secure OS service layer is in charge of the secure context initialization before jumping on the Trusted OS entry-point. Moreover, it is responsible for dispatching SMC services as well as to route interrupts reserved for the Trusted OS during its runtime.
- The Platform API is designed to abstract driver function calls from the core part of the monitor layer. Indeed, VOSYSmonitor requires access to peripherals, which can vary according to the hardware platform. Therefore, for modularity reasons the EL3 monitor layer uses these generic API functions, which, in turn, call specific driver functions.
- Drivers include all necessary drivers related to peripherals (e.g., interrupt controller, UART, power controller, etc.), which are used by the VOSYSmonitor runtime.

VOSYSmonitor is a low latency software monitor layer developed for the 64-bit ARMv8-A architecture and it already supports the last generation of ARMv8 platforms such as:

- ARM Juno Development board (2 Cortex-A57 + 4 Cortex A-53) [18],
- Renesas R-CarH3 board (4 x A57 + 4 x A53) – ISO-26262 (ASIL-B) compliant [29],
- Renesas R-CarM3 board (2 x A57 + 4 x A53) – ISO-26262 (ASIL-B) compliant [29],
- NVIDIA Jetson TX1 board (4 x Cortex-A57 in big.LITTLE configuration) [27].

Moreover, VOSYSmonitor has been designed to be compliant with the stringent ISO-26262 certification as well as to meet the following requirements:

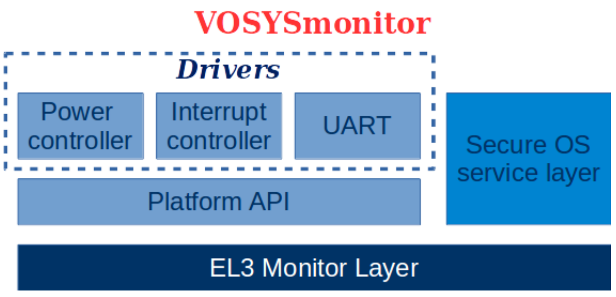

Figure 4 VOSYSmonitor top level architecture.

- Functional requirement. Safety critical OS (e.g., RTOS)/GPOS (e.g.,Linux-KVM) co-execution on the same platform.
- Functional requirement. Isolation of safety critical OS resources (e.g., Memory, Peripherals, etc.) from GPOS illegal access.
- Functional requirement. Preserve the context of each OS during a switching operation.
- Performance goal. A strong requirement for the RTOS in automotive is related to the boot time, which must be limited even in a worst-case scenario. Since VOSYSmonitor adds a software layer before the execution of RTOS, it directly impacts the RTOS boot time. In this context, VOSYSmonitor setup must be achieved in less than 1% of the full RTOS boot time. For instance, a RTOS boot time of 60 ms implies a VOSYSmonitor setup which has to be performed in less than 600us regardless of the platform.
- Performance goal. The overhead added by the co-execution of software applications must be optimized to meet real-time constraints. In this context, the overhead due to the context switching in order to forward an interrupt to the RTOS, running in the Secure world, has to be lower than 1us. This requirement is self-imposed and concerns a standard context switch where only the general-purpose registers and ARM systems registers are saved (see Section 3.3 for more details). This overhead can also be used for estimating the Worst Case Execution Time (WCET) of a task in the RTOS.

3.2 Secure/Normal world scheduling

On multi-core architectures, VOSYSmonitor is able to dynamically share a core between both worlds by operating under the assumption that the Secure world tasks should be prioritized over the Normal world execution. This means that once a core is assigned to the secure RTOS tasks, the normal world applications can use it only if the RTOS, running in the secure world, has decided to release the core resource; something that happens when there is no real-time task to schedule. For this implementation, a minor update of the RTOS is needed in order to call VOSYSmonitor service, to schedule the Normal world execution, when the RTOS workload is null. Generally, this could be achieved through the creation of an “idle task” with the lowest priority, which will be scheduled only if no other tasks need to be executed.

While giving the full priority to the Secure world may seem simple and have too much of an impact on the Normal world execution, VOSYSmonitor has been implemented under the assumption that tasks performing in the Secure world take precedence over the Normal world execution. If a task has no real-time requirement but requires to be isolated from others tasks (e.g. data encryption), it should be executed in the Normal world which can use the Virtualization Extensions to provide isolation.

For instance, tasks whose failure to meet a real-time requirement could arise to a life-critical situation (e.g. brakes control), should be executed in the Secure world. On the other hand, tasks whose failure to meet a real-time requirement does not lead to a life-critical situation (e.g. video decoding) should be performed in the Normal world.

Although multiple scheduling methods have been proposed for mixed-criticality systems [25], such a design has several benefits related to the execution of the safety critical RTOS. First of all, critical interrupts dedicated to the RTOS systematically preempt the Normal world execution in order to be handled by the RTOS with a minimum overhead (see Section 5), thus ensuring to meet real-time constraints. Moreover, there is no risk to preempt the RTOS execution during a critical operation since the RTOS releases the core only when it wants to. Finally, the core usage is optimized since VOSYSmonitor enables the scheduling of the Normal world application when the RTOS has no operations to execute. Indeed, real-time tasks are usually used to perform a brief action bounded in terms of time that could imply a low workload of the RTOS depending on the use case. However, this solution could lead the normal world to the impossibility to run its applications if the RTOS execution monopolizes the core. This is mitigated through the assignment of other cores to the Normal world GPOS.

3.3 Context switch

As ARMv8-A CPU execution is split in two parts (i.e., Secure/Normal world) and some registers are not banked, VOSYSmonitor has to preserve the world context during the switching operation. This part is the most performance related function of the system, as execution and RTOS interrupt latency are directly affected by the time consumed for the context save/restore operations. For this reason, most of code executed during the context switching is written in ARM assembly in order to reach the best performance. In the current implementation, VOSYSmonitor only saves the vital registers, such as general-purpose registers and some ARM system registers, needed for the correct RTOS/GPOS co-execution in each CPU operating mode. Indeed, for the test purpose, RTOS does not use advanced CPU features as the floating point context. However, a memory segment in the backup context structure is reserved to save additional registers during the context switching in order to extend VOSYSmonitor functionalities.

VOSYSmonitor periodically transfers the execution from one world to the other. As the RTOS/GPOS shares only a specific core, the context switch is currently tied to this processor and may take place in two main cases: an interrupt assigned to the Secure world is triggered during the execution of the Normal world application; one world requests a context switch (e.g., secure services, no RTOS workload, etc.) by calling VOSYSmonitor service through an SMC. In this context, it first saves the current state of the world suspended, then restores the state of the other world.

3.4 Interrupts management

ARMv8-A architecture including the hardware security extension TrustZone has been designed to support two interrupt types: the Fast Interrupt Request (FIQ) for low latency interrupt handling, and the more general Interrupt Request (IRQ), which is commonly available also in other architectures. The former has higher priority (IRQs are automatically masked by the CPU core when an FIQ occurs) and can directly use some banked registers without the overhead of saving/restoring them.

VOSYSmonitor takes advantage of the ARM architecture [20], which offers the ability to trap IRQ and FIQ directly in its exception layer (EL3) without intervention of code in

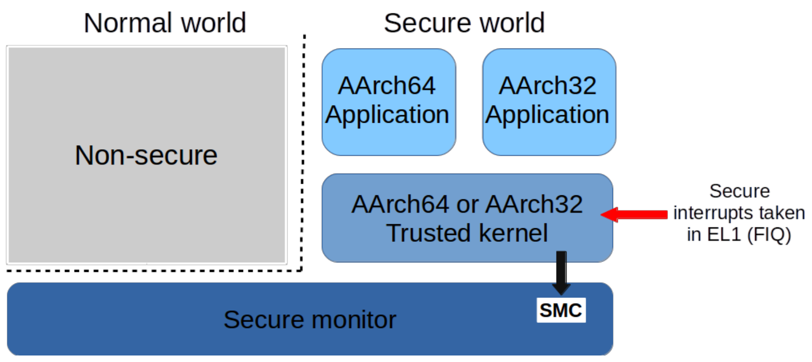

Figure 5 Secure world execution.

either world, thus allowing for the creation of a flexible interrupt management for safety critical RTOS tasks. Indeed, FIQs are considered as secure interrupts when TrustZone is supported, meaning that the configuration of FIQ cannot be altered by normal world accesses. Therefore, VOSYSmonitor sets the secure world (RTOS) to respond only to FIQs and the normal world (GPOS) to handle IRQs. This design allows critical applications to benefit from fast and high priority interrupts, while isolating them from the normal world.

3.4.1 Secure world interaction

During the Secure world execution, only FIQs are enabled in order to prevent the preemption of critical RTOS tasks by normal interrupts (IRQ) dedicated to the non-secure world. Indeed, on multi-core architecture, cores assigned only for Normal world applications can attempt to send normal interrupts for synchronization purpose, whereas the shared core executes RTOS tasks. This could generate untimely preemption of the RTOS during the critical path execution. Therefore, VOSYSmonitor prevents this side effect by masking all normal interrupts during the secure world execution.

FIQs are directly handled in the S-EL1 mode of the RTOS (see Figure 5. This minimizes the interrupts latency during the secure world execution since these are directly caught in the handler of the critical RTOS application, avoiding any context switch overhead. The Normal world only executes upon an explicit request by the Secure World that is achieved through an SMC instruction.

Although the execution of real-time tasks will not be impacted by non-critical applications, the main drawback of this implementation concerns the fact that VOSYSmonitor relies on the secure world application to induce a context switching. Indeed, VOSYSmonitor has no means to take back the control in case of the RTOS does not generate an SMC to release the core. With the current available ARMv8-A platforms equipped with the Generic Interrupt Controller (GIC) version 2 (e.g., Renesas R-Car H3), scheduling policies can be implemented (see Section 3.5) to allow the monitoring of the RTOS execution but such solutions either weaken the isolation of the safety critical RTOS or add an overhead during the context switch.

For these reasons, the current implementation depends on the safety critical RTOS to decide when the context switching should be performed. Moreover, safety critical RTOSes are generally compliant with the stringent requirements of an ISO standard (e.g., ISO-26262 Road vehicles – Functional safety [1, 2, 3]) in order to mitigate potential failures. However, last GIC version [15] (e.g., GICv3) enables the system to isolate interrupts addressed to the secure world from interrupts assigned to the monitor layer. In this context, VOSYSmonitor

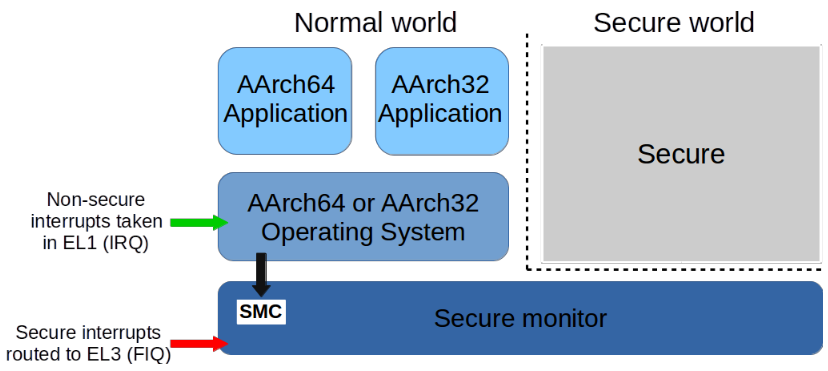
Normal world
Secure world
AArch64 Application
AArch32 Application
Secure
Non-secure interrupts taken in EL1 (IRQ)
AArch64 or AArch32 Operating System
SMC
Secure monitor
Secure interrupts routed to EL3 (FIQ)

Figure 6 Normal world execution.

will be adapted and extended when platforms equipped with GICv3 will be available in order to implement a monitoring feature without compromising isolation, security and performance.

3.4.2 Normal world interaction

By contrast with the secure world, both FIQs and IRQs are enabled during the execution of the normal world: IRQs are directly handled in the NS-EL1 (see Figure 6) while FIQs are routed to VOSYSmonitor, which operates in EL3. When an FIQ occurs, the non secure OS is immediately preempted by VOSYSmonitor, which is responsible for an operating mode switch to save the normal world context and to restore the secure world one as fast as possible. This lowers latencies and helps the critical application to meet real time constraints. During the FIQ propagation into VOSYSmonitor, all interrupts (i.e., FIQ and IRQ) are masked in order to prevent any preemption by another FIQs having higher priority. Then, once the execution control is given to the Secure world, the FIQ management is handled by the trusted application. Therefore, the Secure world could decide to disable the FIQ mask when the critical part is over.

Finally, the normal world can initiate a context switch by generating an SMC instruction either to give back control to the secure world or to request a secure service.

3.5 Secure world failure handling

As mentioned in Section 3.2, the Secure world execution is always prioritized over the Normal world. However in some scenarios, the Secure RTOS might crash because of an internal failure and never giving the control back to the Normal world. To prevent such case, VOSYSmonitor proposes a Secure world failure handler, based on the ARM Physical Secure Timer. The handler execution is similar to a watchdog: if after a specific timeout the Secure world has not given back the control, VOSYSmonitor preempts the core and restart the Secure world. The handler timeout is reset at each context switch. However this feature is not present in the upstream version of VOSYSmonitor, for two reasons:

- Huge overhead of the context switch: when performing a context switch from the Secure world to the Normal world, the Secure Timer must be disabled. Likewise when returning to the Secure world, the Timer must be enabled and the timeout value calculated. It has been measured that adding the Secure Timer doubles the context switch latency.
- All interrupts in the Secure world are FIQs. Only the interrupt corresponding to the Secure Timer is configured as an IRQ, so it can be trapped by VOSYSmonitor. However this means IRQs are no longer masked during the Secure world execution.

When the system is mono-core, there are no issues as others IRQs can be disabled. Nevertheless, if the Normal world is running on a multi-core system, secondaries cores might forward IRQs to the primary core which is in the Secure world, causing undefined behaviour and possibly a crash of the RTOS. This could be avoided by updating the GPOS source code however this is against the policy of VOSYSmonitor which should be able to run a GPOS with minimal modifications. Furthermore, this would create a vulnerability exploitable from the Normal world.
In platforms supporting GICv3, interrupts are classified in three types, where one can be reserved to the VOSYSmonitor execution. In this configuration, it is possible to use the Secure Timer while retaining multi-core support for the Normal world. However all platforms currently supported are using GICv2. Thus for the time being the Secure Timer is only proposed as an optional feature.

3.6 ARM convention compliance

VOSYSmonitor is compliant with several ARM standards in order to ease the integration of this software component in a full system.

SMC Calling Convention (SMCCC) [19] specifies the calling procedure (e.g., registers used as parameters and return arguments, etc.) of the SMC instruction, used in the ARMv7 and ARMv8 architectures. This convention simplifies the integration between different software layers, such as operating systems, hypervisor and secure monitor. Moreover, it categorizes SMC service providers to allow the coexistence of services in the secure monitor firmware (e.g., ARM, OEM, Trusted OS, etc.)

Power State Coordination Interface (PSCI) [17] defines a standard interface for power management that can be used by software working at different ARM privilege levels. During a power management operation, rich OS, hypervisor, VOSYSmonitor and Trusted OS must not conflict each other. In this context, PSCI aims to standardize the communication between supervisory software to arbitrate power management requests. As a matter of fact, Linux kernel AArch64 relies on PSCI calls for powering up/down secondary cores avoiding the need of platform dependent drivers.

VOSYSmonitor is compliant with the PSCI convention v1.0, which requests the support of mandatory functions related to the power management such as power up/down a core, suspend core execution, etc.

4 Related work

VOSYSmonitor is a TrustZone based monitor, which enables the consolidation of a non-critical GPOS and a safety critical RTOS. It guarantees the isolation of critical real-time tasks, while minimizing the overhead on the global execution. Moreover, this software layer has been designed to meet certification requirements [6] induced by mixed-criticality systems.

Among existing solutions, TrustZone-assisted hypervisor [28] is able to run an arbitrary number of RTOSes in virtual machines. In this design, only one RTOS is executed at a time in the Normal world, while the context of the other guests is preserved in the Secure world. Therefore, if another RTOS needs to be scheduled, the current RTOS execution is stopped and its context is saved in the Secure world, then, the TrustZone-assisted hypervisor restores the second RTOS context in the Normal world. Real-time requirements of inactive RTOSes are met by setting their interrupts as secure, thus allowing to preempt the running RTOS, which uses normal interrupts. Such a implementation does not allow an efficient cache management since all RTOSes are scheduled in the Normal world, which requires to clean/invalidate cache memory during context switches, thus increasing the overhead. Moreover, at the time of writing, the TrustZone-assisted hypervisor only supports single-core configuration. VOSYSmonitor is able to run on multi-core heterogeneous platform and take benefit of cache memory to reduce the context switch overhead. Indeed, since OSes are assigned to a specific world, VOSYSmonitor can rely on the isolation of caches lines provided by ARM TrustZone, thus avoiding the need of cache operations during a context switch.

Other solutions enable the co-execution of two or more OSes using ARM virtualization extensions. A design based on Erika OS [5] is executed along with Linux on top of XEN hypervisor [9]. This solution has been implemented on a cubieboard2 by assigning each OS on a different core. While it ensures the isolation, there is a risk of an inefficient use of computing power if Erika OS has a low workload. To prevent such a case, VOSYSmonitor is able to reallocate core resources to the GPOS if the RTOS has no real-time tasks to schedule. Others solutions based on virtualization extensions are Xtratum [24] and NOVA [31].

Furthermore, the isolation of Virtual Machines (VMs) against the host is not guaranteed. Indeed, some breach in the hypervisors can allow VMs to cause a denial service or access data from the whole system. As a matter of fact, a security vulnerability named VENOM [10], allows an attacker to escape from the isolation of a VM and get the access to the host and the others VMs.

ARM Trusted Firmware [22] hereafter referenced as ATF is a software layer able to host a Trusted Execution Environment alongside a Non-Trusted software. At the best of our knowledge, the current implementation of ATF only supports a dispatcher to execute OP-TEE OS [12] in the Secure world in order to provide trusted services to the Normal world application. VOSYSmonitor overcomes this limitation by giving the possibility to concurrently execute an RTOS and a rich OS. A further comparison between VOSYSmonitor and ATF is highlighted on Section 5.

A different approach, such as FLexPRET [32], introduces a new multi-threaded processor intended for mixed-criticality systems where threads are classified in two categories: *hard real-time thread (HRTT)* where deadlines must be met and *soft real-time thread (SRTT)* where a time constraint non-respected is acceptable. Another solution is IDAMC [26], which proposes a platform where multiples nodes are interconnected by a network-on-chip. These nodes includes processors based on the LEON3 technology connected through routers monitoring access to shared resources. While these solutions have been created for mixed-criticality systems, the main disadvantage is related to the new processor or cluster architecture, which implies, for instance, a higher workload for developers and increases the risk of unknown hardware/software bugs since the architecture is new and not widely used. On the other hand, VOSYSmonitor is based on ARMv8-A, one of the most popular embedded architectures for mobile, automotive, drone markets.

5 Evaluation

In this section, the performance of VOSYSmonitor is benchmarked with the ARM Trusted Firmware. The evaluation uses the ARMv8 Performance Monitoring Unit (PMU) [21] The tests have been performed on the ARM Juno board R1 and on the Renesas R-CarH3. Both boards have a Cortex A-57 and a Cortex A-53 cluster. As results may vary depending on the core where VOSYSmonitor is operating, all tests are performed on both A-53 and on A-57.

On top of VOSYSmonitor a bare-metal application is executed in the Normal world, which constantly loops a NOP instruction in order to have a minimal impact on the test execution. On the other hand, the Secure world is hosting a FreeRTOS version 8.2.3 modified by Virtual Open Systems in order to use FIQ for interrupt processing. While it is not safety compliant, FreeRTOS is a widely used real-time kernel and is the base for an ISO 26262 certified RTOS called SAFERTOS [23].

Since VOSYSmonitor targets the automotive market, the compiler used to create the binary file should be qualified according to the ISO 26262 standard. For this reason, the ARM Compiler [13] with compilation optimisation (-O2) is used since functional safety support package [4, 14] will be provided by Q2 2017, thus avoiding further qualification activities when the recommendations and conditions are respected. Similarly, the upstream version of the ATF has been pulled from GitHub (v1.3) and also compiled with -O2 optimizations.

5.1 Context switch latency using PMU

VOSYSmonitor aims at having small impact on the co-execution of the Normal and the Secure worlds. The context switch is the most critical aspect in the VOSYSmonitor execution, therefore it is of utmost importance to implement it in a performant way. To assess the time induced by a context switch from the Normal world to the Secure, a timer in FreeRTOS is configured to generate an FIQ every 1us and then gives the control to the Non-Secure world. On Juno, the timer used is the SP804 Dual-Timer, on R-CarH3 the Compare Match Timer.

The timer interrupt is triggered while the processor execution is in the Normal world and thus it is trapped in VOSYSmonitor vector table. The PMU counter is started there and stopped before giving the control back to FreeRTOS, then the number of clocks cycles is displayed. The same test is performed with the ATF in order to compare the results. Adding the PMU breaks the code execution, therefore, the board has to be manually powered on/off after each measurement inducing a small sample size (< 10).

Although VOSYSmonitor includes the interrupt handler described in Section 3, the context switch latency has also been measured without this handler in order to gain time execution and assess the optimum value possible. This implementation makes that any FIQ trapped, when the processor execution is in the Non-Secure world, causes a request for a context switch. While lacking flexibility, this implementation may be of interest for scenarios where the RTOS response must be as fast as possible.

Figure 7 shows the context latency expressed in microseconds on the Juno board and Figure 8 for the Renesas R-CarH3 board. The results for the Juno board, shows that, in the worst case (VOSYSmonitor with interrupt handler on Cortex-A57), VOSYSmonitor is 118% faster than ATF and almost twice as fast if VOSYSmonitor without interrupt handler is executed on an A-53 core. On the R-CarH3, the comparison is in favor again of VOSYSmonitor, which can be more than thrice faster than ATF. However, it should be noted than the ARM Trusted Firmware used here is a software updated by Renesas, and thus may include features not present in the upstream version. VOSYSmonitor aims at having a context switch faster than 1us. Although on Cortex-A53 the requirement is met on VOSYSmonitor with or without an interrupt handler, a context switch on Cortex-A57 barely misses the prerequisite on Juno.

In this measurement, the interrupt handler called by VOSYSmonitor is a minimal implementation, which consists in fetching the pending interrupt ID, then after comparison, jumps to the context switch macro. Therefore, a decrease in performance can be expected if a more complex interrupt management is requested. However, the same can be said for ATF and the latency difference between both ATF and VOSYSmonitor should remain as it is.

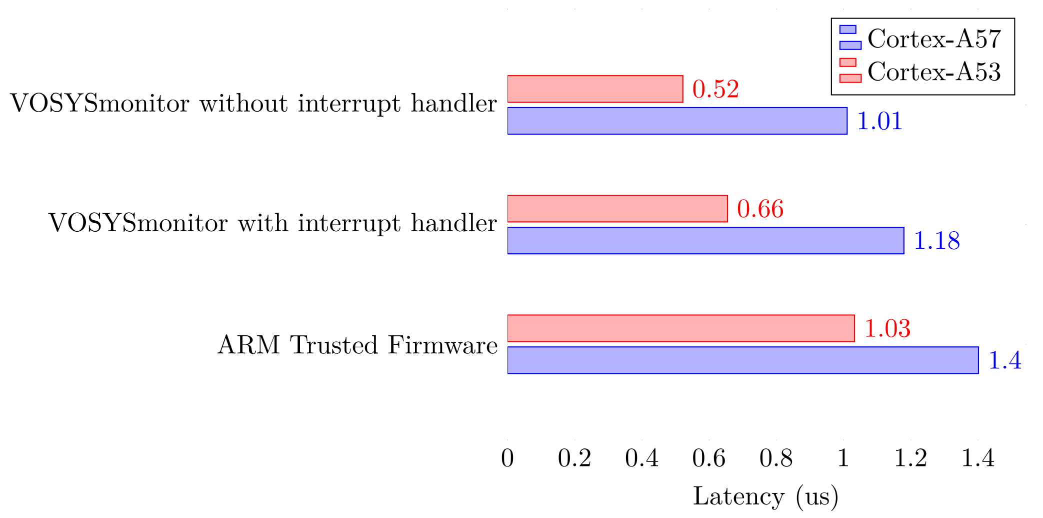

Figure 7 Juno context switch latency.

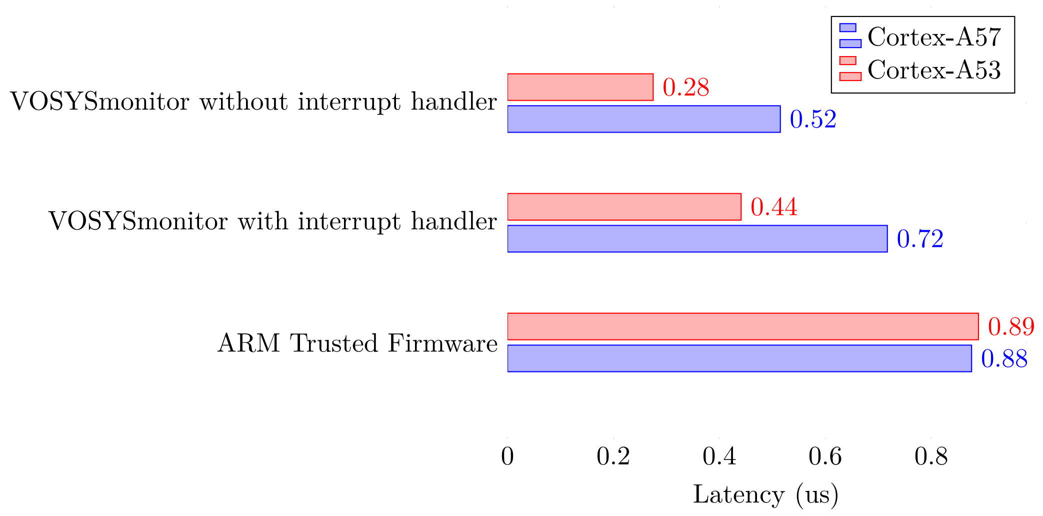

Figure 8 R-CarH3 context switch latency.

5.2 Context switch including the hardware latency

Although the previous measurement allows to measure the exact number of clock cycles consumed during a context switch, a second test has been performed to overcome some limitations of the first one. Indeed, VOSYSmonitor is running with both instructions and data caches enabled, which drastically improve performance. However, caches misses can cause a decrease in performance in worst-case scenarios, which can not be estimated unless the sample size is big enough.

In this context, a second performance test has been performed in order to take into account the FIQ latency induced by the hardware, i.e. the time between an FIQ triggering and the beginning of its processing by the software, on an important sample size. To achieve this, the ARM EL1 physical timer is used to trigger an FIQ while the core is in the Normal World as in the previous test.

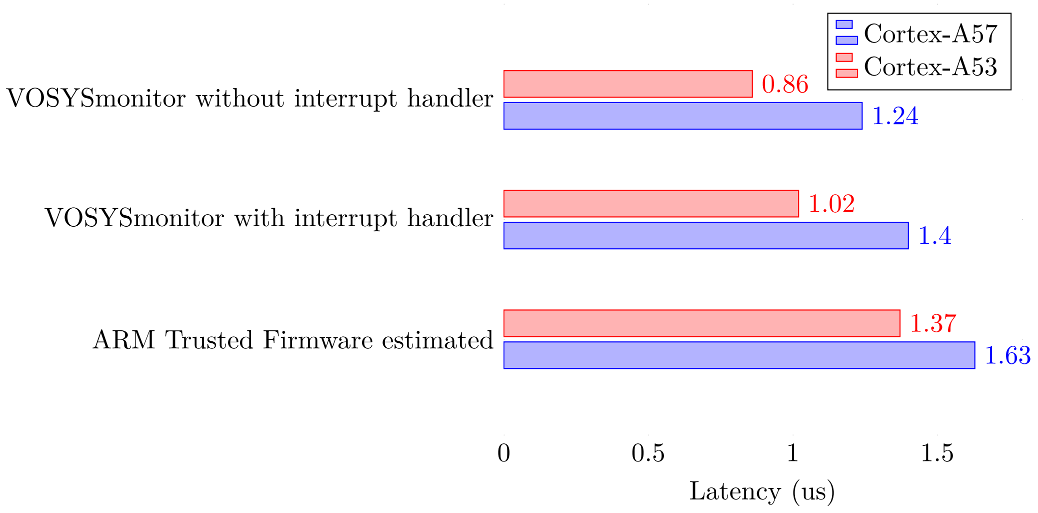

Figure 9 Juno board context switch with hardware latency.

Table 1 FreeRTOS FIQ latency.

Platform	Processor	Average (ns)	Min (ns)	Max (ns)
Juno	Cortex-A53	220	180	280
	Cortex-A57	220	200	260
R-CarH3	Cortex-A53	240	120	360
	Cortex-A57	240	120	360

The ARM timer is composed of three registers:

- CNTP_CTL_EL0 used to enable/disable the timer.
- CNTP_TVAL_EL0 contains the current value of the timer.
- CNTP_CVAL_EL0 holds the compare value. When equal to CNTP_TVAL_EL0, the interrupt is triggered.

When the context switch occurs, the timer interrupt trapped in VOSYSmonitor is forwarded to the FreeRTOS vector table. There, the very first instruction consists in reading the value of CNTP_TVAL_EL0, the timer is disabled and we jump to FreeRTOS handling routine. Finally, VOSYSmonitor latency between an FIQ triggering and the interrupt handler can be deduced by subtracting from CNTP_CVAL_EL0. Since VOSYSmonitor code has been untouched, we are able to run this test for as long as necessary and without any instrumentation of the code execution.

As before VOSYSmonitor is tested with and without the interrupt handler. The results with a sample size of 2048 contexts switch are presented on Figures 9 and 10. By taking into account the FIQ latency, it confirms that VOSYSmonitor is still faster than the ATF version in all scenarios. Moreover, it is better, in terms of performance, to have the RTOS executing on an A-53 core since the context switch is at least twice as fast as an A-57 for VOSYSmonitor. In this test, an issue has been the inability to run FreeRTOS and a Non-Secure OS in co-execution with ATF. However, the full context switch with hardware latency has been extrapolated by adding the previous results of ATF measured in Section 5.1 and the FIQ hardware latency shown in Table 1.

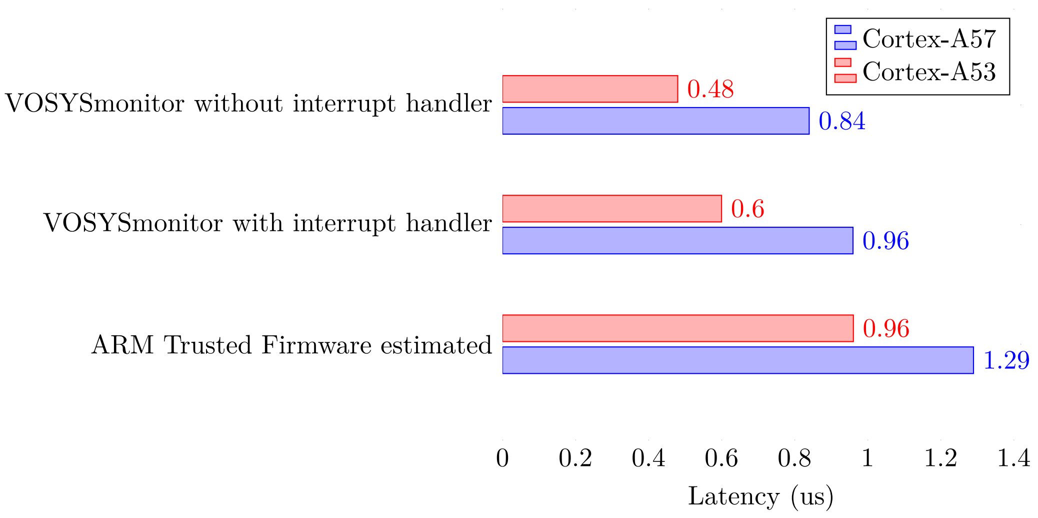

Figure 10 R-CarH3 board context switch with hardware latency.

Table 2 VOSYSmonitor and ATF setup time.

Platform	Processor	ARM Trusted Firmware (us)	VOSYSmonitor (us)
Juno	Cortex-A53	853.480	17.746
	Cortex-A57	1119.815	31.004
R-CarH3	Cortex-A53	1661.806	44.493
	Cortex-A57	1119.815	31.004

5.3 VOSYSmonitor and ATF booting time

The performance analysis also compared the setup time of VOSYSmonitor and ATF. The setup time corresponds to the time consumed between VOSYSmonitor/ATF entry point and the first jump to the Secure world. In this test, PMU has been used since the GIC and the timers are not yet configured. As for the previous tests, the measurements have been performed on both A-53 and A-57 cores. Table 2 shows that on Juno VOSYSmonitor is able to achieve its setup configuration within 18us, while ARM Trusted Firmware needs around 870us on Cortex-A53 and 920us on Cortex-A57. VOSYSmonitor is able to outperform ATF because the page tables, used by the Memory Management Unit, are defined by the developer and included statically during the compilation, whereas ATF generates the pages during the setup time.

Although it requires more effort and reduce flexibility, it is worth the trade-off since VOSYSmonitor impact is significantly reduced during the setup and meets the 600us boot time requirement presented in Section 3.1.

5.4 OSes co-execution workload and multicore support

RTOS/GPOS co-execution is ensured by sharing a core between these two OSes. The full priority is given to the RTOS, running in the Secure world, in order to meet real-time requirements. With this implementation, the GPOS execution can be impacted depending on the RTOS workload. In order to measure this impact, the *hackbench* [30] benchmark has been used in three scenarios:

Table 3 Juno board VOSYSmonitor hackbench result.

Linux one core			
Hackbench (s)	Linux standalone	Linux + FreeRTOS low workload	Linux + FreeRTOS 60% workload
	3.411	3.431	14.251
Linux multicore			
Hackbench (s)	Linux standalone	Linux + FreeRTOS low workload	Linux + FreeRTOS 60% workload
	0.500	0.498	0.596

- Linux standalone.
- Linux + FreeRTOS low workload. FreeRTOS requests a context switch every 1us and gives back the control immediately after.
- Linux + FreeRTOS 60% workload. The CPU will spend around 60% of the runtime executing FreeRTOS code.

Table 3 shows that adding FreeTOS with a low workload has no impact on the performance, whether using one or more cores. However, a deterioration can be noticed when FreeRTOS is executing 60% of the time on a core. By executing the test on a multi-core configuration (four Cortex-A53 and two Cortex-A57), we observe that the performance degradation is minimal between the Linux standalone and the 60% workload scenarios. Although a core is busy executing Secure world application, the others cores are nonetheless able to continue performing.

6 Conclusion

This paper presents VOSYSmonitor, a low latency monitor layer for mixed-criticality systems on multicore heterogeneous ARMv8-A platforms, providing architecture details and a performance analysis. VOSYSmonitor offers a strong isolation based on the ARM TrustZone technology, which gives the full priority to the safety critical RTOS, thus meeting real-time constraints as well as ensuring the execution of critical tasks.

The benefits of VOSYSmonitor are its modular and scalable architecture, which allows the system evolution according to the requirements. Indeed, it is possible to run, on top of VOSYSmonitor, a hypervisor in order to instantiate a variety of different VMs for multi-OS support (e.g., Linux, Android, etc.) in the Normal world. Moreover, the isolation provided by VOSYSmonitor is stronger than virtualization technology since this latter is restricted to the processor through the implementation of a hypervisor, whereas TrustZone can be extended to other master peripherals (e.g., DMA, GPU, etc.). Finally, the small footprint of VOSYSmonitor allows to mitigate the certification effort, thus reducing costs.

As for the analysis of the overhead introduced by the proposed solution, it is possible to claim that VOSYSmonitor is a perfect solution for the consolidation of real-time applications along with a rich OS whithout compromising hard real-time requirements. While it is not possible to compare with all technologies presented in Section 4, it has been proven that VOSYSmonitor is better than ARM Trusted Firmware in terms of setup time and context switching latency.

Finally, the future work includes the ASIL-C certification of VOSYSmonitor according to the stringent automotive standard ISO 26262. This will enable the usage of VOSYSmonitor technology in automotive use cases, such as the consolidation of Infotainment In-Vehicle system along with safety vehicle cluster applications. With this in mind a full fledged software stack including VOSYSmonitor, an ASIL certified RTOS and GPU sharing support will be developed, tested and benchmarked. Related to the safety critical RTOS support, this work has given us the possibility to ensure the isolation of this software layer along with non-critical applications. However, only one safety critical RTOS can be executed with the current implementation. This problem will be investigated in future works, exploring design methods to overcome this limitation of the TrustZone hardware implementation. Regarding the world scheduling policy, the current architecture relies on the Secure world OS to initiate a context switch. Although this implementation ensures the execution of critical tasks without any preemption from another software layer, VOSYSmonitor lacks of a way to take back the control if the Secure world OS does not release the core resource. Software methods and new interrupt controller architecture, such as GICv3, will be explored to overcome this limitation due to the interrupt management.

References

1 International Standard ISO 26262-4. Road vehicles – functional safety – part 4: Product development at the system level. Standard, International Organization for Standardization, November 2011.

2 International Standard ISO 26262-6. Road vehicles – functional safety – part 6: Product development at the software level. Standard, International Organization for Standardization, November 2011.

3 International Standard ISO 26262-8. Road vehicles – functional safety – part 8: Supporting processes. Standard, International Organization for Standardization, November 2011.

4 Hopkins Andrew. The functional safety imperative in automotive design. Standard, ARM Ltd, September 2016.

5 Avanzini Arianna. Integrating Linux and the real-time ERIKA OS through the Xen hypervisor. *Industrial Embedded Systems (SIES)*, 2015. `doi:10.1109/SIES.2015.7185063`.

6 Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 13–22. IEEE, 2010.

7 Alan Burns and Rob Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, 2013.

8 Helmut Fennel, Stefan Bunzel, Harald Heinecke, Jürgen Bielefeld, Simon Fürst, Klaus-Peter Schnelle, Walter Grote, Nico Maldener, Thomas Weber, Florian Wohlgemuth, et al. Achievements and exploitation of the AUTOSAR development partnership. *Convergence*, 2006:10, 2006.

9 Linux Foundation. The Xen Project, the powerful open source industry standard for virtualization. URL: `https://www.xenproject.org/`.

10 Jason Geffner. VENOM Virtualized Environment Neglected Operations Manipulation. URL: `http://venom.crowdstrike.com/`.

11 Richard Grisenthwaite. ARMv8 Technology Preview. In *IEEE Conference*, 2011.

12 Linaro. Op-tee. URL: `https://wiki.linaro.org/WorkingGroups/Security/OP-TEE`.

13 ARM Ltd. ARM Compiler 6. URL: `https://developer.arm.com/products/software-development-tools/compilers/arm-compiler-6`.

14 ARM Ltd. ARM Compiler Safety Package. URL: `https://developer.arm.com/products/software-development-tools/compilers/arm-compiler/safety`.

15 ARM Ltd. Programmable Interrupt Controllers: A New Architecture. URL: `https://www.community.arm.com/processors/b/blog/posts/programmable-interrupt-controllers-a-new-architecture`.

16 ARM Ltd. TrustZone. URL: `https://developer.arm.com/technologies/trustzone`.

17 ARM Ltd. *Power State Coordination Interface*, August 2012.

18 ARM Ltd. *Juno ARM Development Platform SoC*, r1p0 edition, June 2013.

19 ARM Ltd. *SMC Calling Convention*, June 2013.

20 ARM Ltd. *ARM Cortex – A Series*, March 2015. Programmer's Guide for ARMv8-A.

21 ARM Ltd. *ARM Architecture Reference Manual*, January 2016. ARMv8, for ARMv8-A architecture profile.

22 ARM Ltd. Github repository. `https://github.com/ARM-software/arm-trusted-firmware`, 2016.

23 HighIntegritySystems Ltd. SAFERTOS Safety Certified RTOS. URL: `https://www.highintegritysystems.com/safertos/`.

24 Miguel Masmano, Ismael Ripoll, Alfons Crespo, and J. Metge. Xtratum: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, pages 263–272. Citeseer, 2009.

25 Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy K. Baruah, and John A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1864–1871. IEEE, 2010.

26 Boris Motruk, Jonas Diemer, Rainer Buchty, Rolf Ernst, and Mladen Berekovic. Idamc: A many-core platform with run-time monitoring for mixed-criticality. In *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*, pages 24–31. IEEE, 2012.

27 NVIDIA. *NVIDIA Tegra X1 Mobile Processor Technical Reference Manual*, November 2015.

28 Sandro Pinto, Jorge Pereira, Tiago Gomes, Mongkol Ekpanyapong, and Adriano Tavares. Towards a TrustZone-assisted Hypervisor for Real Time Embedded Systems. *IEEE Computer Architecture Letters*, 2016.

29 Renesas. *R-Car Series, 3rd Generation User's Manual: Hardware*, February 2016.

30 Russell Rusty. Ubuntu Manpage: hackbench – scheduler benchmark/stress test. URL: `http://manpages.ubuntu.com/manpages/precise/man8/hackbench.8.html`.

31 Udo Steinberg and Bernhard Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222. ACM, 2010.

32 Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. PFlexPRET: A processor platform for mixed-criticality systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 101–110. IEEE, 2014.