VOSYSmonitor, a TrustZone-based Hypervisor for ISO 26262 Mixed-critical System

Pierre Lucas, Kevin Chappuis, Benjamin Boutin, Julian Vetter, Daniel Raho
Virtual Open Systems
Grenoble, France
p.lucas, k.chappuis, b.boutin, j.vetter, and 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 (e.g. real-time constraints, multiple Operating Systems (OS) scheduling, providing temporal and spatial isolation). In this context Virtual Open Systems has developed VOSYSmonitor, a thin software layer, which allows the co-execution of a safety-critical and non-critical applications on a single ARM-based multi-core SoC. This software element has been developed according to the ISO 26262 standard. One of the key aspects of this standard is the control of *random* and *systematic* failures, including the ones induced by faulty or aging hardware. In the case of a software component, the means to detect anomalies on the hardware are limited and depend on choices of the manufacturer (i.e. implementation of Dual redundant Core Lock step (DCLS)). However, the software is able to check a part of these failures. It can be by either reading the configuration registers of a peripheral, or checking the sanity of a memory region. The purpose of this paper is to showcase how a safety-related software element (e.g. VOSYSmonitor) can detect and recover from failures, while ensuring that the safety-related goals are still reached.**

I. Introduction

In mixed-criticality domains, the term “functional safety” has become a topic of high importance. Indeed, “functional safety” generally means that malfunctions of the operating system, which contain mission-critical tasks, that lead to any kind of threat or even accident have to be avoided or mitigated. Therefore, it is fundamental in the field of functional safety to identify and understand potential risks and failure causes of a system. If ideally all potential failure causes are known and the consequences understood it is possible to define countermeasures. Thus, failures are detected before a hazardous event occurs and the safe state is initiated with the needed of functional safety reaction.

The safe states can importantly vary according to the final application as well as the injuries which might be led by the system failure without countermeasures. As every application is different and has its own particularities and thus potential failure causes and related safe states, the functional safety analysis is very interesting challenges.

In this context, many functional safety standards have been established to define the main requirements to fulfill during the development of critical systems in order to ensure a high level of reliability in the critical systems. The main functional safety standard is the IEC/EN 61508 that defines the basis for functional safety developments for E/E/EP (electronics, electronic or programmable electronic) applications. In addition, the IEC/EN 61508 is expanded by additional industry sector specific standards, such as the ISO 26262 – Road vehicles – Functional Safety which has been specially defined for the automotive domain (see section II-A).

Indeed, the automotive industry is rapidly evolving towards the connected autonomous vehicle which will considerably increase the hardware/software complexity, while functional safety will be a topic of high importance since critical features will be controlled by electronics components (e.g., autonomous driving, etc.). Thus, the ISO 26262 defines a functional safety lifecycle for each automotive product development phase, ranging from the hazard analysis and risk assessment to design, implementation, integration, verification, validation and production release.

In this context, Virtual Open Systems has developed VOSYSmonitor, a hypervisor based on ARM® TrustZone® that enables the consolidation of mixed-critical Operating Systems (e.g., Linux-KVM along with a RTOS) on a single ARM-based platform with special attention to safety and security. This software technology has been developed as a Safety Element out of Context (SEooC) in compliance with the ASIL-C requirements of the ISO 26262 standard and it ensures freedom from interferences for the safety critical partition.

As a mater of fact, VOSYSmonitor is a perfect solution to support a modern generation of car virtual cockpit where the In-Vehicle Infotainment (IVI) system and the Instrument Digital Cluster are consolidated and interact on a single platform. Indeed, traditional gauges and lamps are replaced by digital screens offering opportunities for new functions and interactivity. Vehicle information, entertainment, navigation, camera/video and device connectivity are being combined into displays. However, this different information does not have the same level of criticality and the consolidation of mixed-critical applications represent a real challenge that must respect the stringent requirements of the ISO 26262 functional safety standard.

Since VOSYSmonitor is only a software component, the paper will detail the definition of safety functionalities by applying the ISO 26262-6 *Product development at the software level*. After a summary of the ISO 26262 standard and the different technologies involved in the Section II, we present related work and emphasize the advantages and drawbacks of existing solutions compared to our design in Section III. Then, a more-detailed presentation of the safety features of

VOSYSmonitor is presented in Section IV. These features are divided into two parts: detection and recovery mechanisms. The performances of these mechanisms is evaluated in Section V, by measuring the latency between a fault detection and the entry to the mitigation state. Finally, Section VI summarizes this work findings and directions for future works.

II. Background

In the following sections, a brief overview of ARM® TrustZone®, ARM® Virtualization Extensions (VE) as well as the different types of hypervisors is provided. Then, the ISO 26262 standard is introduced. Finally, this section is concluded with a concise introduction of VOSYSmonitor, the underlying firmware layer which ensures freedom from interferences for the safety critical system when it is consolidated with other non-critical applications.

A. *ISO 26262 standard*

ISO 26262 is a relatively recent (First version published in 2012) functional safety standard tailored for the electronic components in the automotive domain. Derived from the industry standard **IEC 61508**, similar concepts are found in both, albeit named differently, such as Automotive Safety Integrity Level (ASIL - ISO 26262) and Safety Integrity Level (SIL - IEC 61508).

The ASIL is decomposed in four levels from ASIL A (the lowest hazard risk level) to ASIL D (the highest hazard risk level). Indeed, the ASIL definition of an electronic element is tied to the risk incurred by the use of this component: an higher ASIL means a risk has an higher occurrence and/or more severe consequences, thus more stringent verifications are required. The ASIL is determined during the product development called the **safety life cycle**.

A product development respecting ISO 26262 is made following the V-Cycle model, the first half starting from the definition of safety requirements and the corresponding specifications to the implementation. The second half is the verification and validation, which ensures that all defined requirements are met, from low-to-high (see Fig. 1).

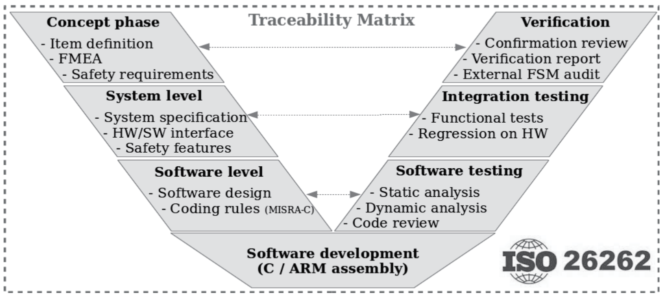

Fig. 1. V-Cycle ISO 26262 Road vehicles Functional Safety

B. *ISO 26262 Faults classification*

During the ISO 26262 safety life cycle, all kind of possibles faults that lead to a failure according to the safety concept have to be considered. The ISO 26262 standard distinguishes two groups:

- **Random fault** occurs unpredictably during the lifetime of an electronic element but follows a probability distribution. Hardware failures (e.g. short-circuit, memory corruption, bit flip) are random faults, in the sense that the risk of occurrence cannot be suppressed but failure rates can be predicted with reasonable accuracy.
- **Systematic fault** is manifested in a deterministic way. Software failures (e.g. stack overflow, non-aligned memory access) are systematic failures. Indeed, while the causes of a software failure are sometimes hard to define or recreate, it will always trigger when a set of conditions is met.

For the sake of simplification, it is assumed here that a random failure is always triggered by a single random fault, and likewise for a software failure. Latents faults are not in the scope of this paper.

In case the product developed is a software-only element (such as VOSYSmonitor), the random faults cannot be ignored. Indeed, these faults must be always detected in a reasonable delay and an appropriate mechanism must exist to handle the situation.

As said above, the software is limited to detect such faults. Furthermore, in case of a SEooC, since the software component is not developed for a specific hardware platform. In this context, these faults are difficult to control without knowledge of the hardware.

On the other hand, software failures can be mitigated with verification & validation activities by following a strict coding standard (such as MISRA-C [8]) and by performing code coverage analysis. These points are addressed in the ISO 26262-6 *Product development at the software level*, sections 5.4.7 and 9.4.5. However, these solutions are performed during the safety life cycle and as such, if a software failure still occurs during the product runtime, other mechanisms need to be implemented.

Section IV, will detail how the issue was tackled with VOSYSmonitor, by adding **self-tests** to identify the failure and **safe states** to recover and preserve the execution of safety-related tasks.

C. *ARM 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 [6]. 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.

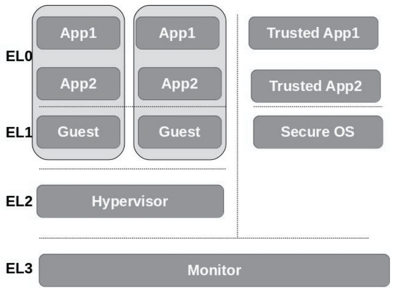

Fig. 2. ARM architecture overview

D. ARM Virtualization Extensions

ARM® added full virtualization support as an optional feature in ARMv7 [10]. Systems with these extensions have an additional execution mode called hypervisor mode (hyp). This mode is located in the new privilege level EL2, placed below EL0 and EL1. In addition software executing in EL2 is provided with additional control registers for reconfiguring execution in EL0 and EL1 by trapping certain instructions in order to have full access to all system control. ARM® VE also introduced a nested paging mechanism. This additional stage of translation gives the hypervisor full control over the address space of systems executing in EL1.

It is worth noting that all EL2-controllable traps and the additional address translation only pertain to execution in the non-secure world (i.e., EL2 exists only in the non-secure world and its power does not extend beyond). However, the opposite holds: the monitor mode in a processor incorporating TrustZone® is able to access all non-secure EL2 controls.

E. Hypervisors

In general, hypervisors can be classified into two types: The Type-I hypervisor, also called bare-metal hypervisor, directly runs on the hardware without relying on a host operating system. Such hypervisor has to bring its own set of device drivers and low-level system mechanisms (e.g., virtual memory management). Famous examples for such a hypervisor are Xen [2] or Hyper-V [4].

Type-II hypervisors on the other hand rely on a host operating system to run on. They leverage the operating system facilities, which are already in place and run as a normal process. However, the host operating system has to corporate with the hypervisor process and reflect specific types of exceptions back to this process. Famous examples for such a hypervisor are VirtualBox [1] or Parallels[12].

Kernel-based Virtual Machine (KVM) [13] is one of a few exception that do not allow a clear classification into one of the two types. In it's design it is a Type-I hypervisor, because it runs in a privileged mode (unlike a Type-II hypervisor), but as a Type-II hypervisor relies on a host operating system, in this case Linux.

F. VOSYSmonitor

VOSYSmonitor [9] is a low level certified software layer developed on the ARM® architecture, which runs in the Secure Monitor mode of ARM® Cortex-A processors. It enables the native concurrent execution of a safety critical Real-Time Operating System (or another type of critical application) along with a non-critical General Purpose Operating System (GPOS) with the option to use virtualization extensions, such as Linux/KVM, in order to instantiate a variety of different Virtual Machines (VM). This software layer isolates the RTOS from the virtualized instances and provides, at the same time, functions to enable a safe and secure communication between them.

This software architecture, while being applicable to a wide range of mixed-criticality use-cases, particularly targets the automotive industry. As specified in the abstract, it has thus been developed with respect to the ISO 26262 standard.

One must note, however, that in cases where VOSYSmonitor is used in other domains than automotive, the applying certification process is facilitated as the product is already certified for ISO 26262. For instance, for a medical use-case requiring an IEC 60601 certification or a railway use-case requiring an EN 50128 certification, a gap analysis can be performed by a certification company, thus avoiding a certification process from scratch.

VOSYSmonitor is based on ARM TrustZone technology, which enforces among others, memory, CPU and interrupt isolation between the RTOS and the GPOS. The design goal of VOSYSmonitor is to give the full priority to the safety critical domain in order to meet real-time constraints, while being compliant with the ASIL-C requirements of the ISO 26262 standard.

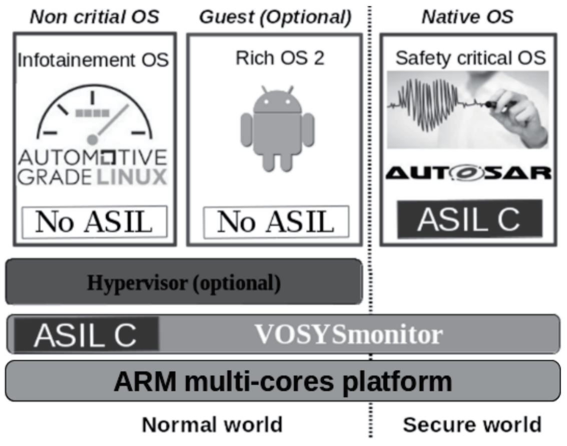

Fig. 3. VOSYSmonitor overview

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 safety critical domain, the normal world applications can use it only if the safety critical domain, isolated in the Secure world, has decided to release the core resource; something that happens when there is no real-time task to schedule. In this context, hardware exception mechanisms, such as interrupts, are used in order to ensure an efficient context switching between the two worlds. In addition, both domains can voluntarily give up their execution time by invoking the *SMC* instruction. VOSYSmonitor keeps tight control over these exceptions in order to ensure a proper operation of each domain.

VOSYSmonitor is a scalable component that can be ported on any ARM platforms with TrustZone support. This firmware enables the consolidation of a wide range of use-cases, from bare-metal applications to rich OS in the Secure/Normal worlds, assuming certain requirements are met. In order to differentiate the different features, VOSYSmonitor is divided in several functional blocks as depicted in Fig. 4. The next part of the paper will focus on the top-right block, the *Safety features*.

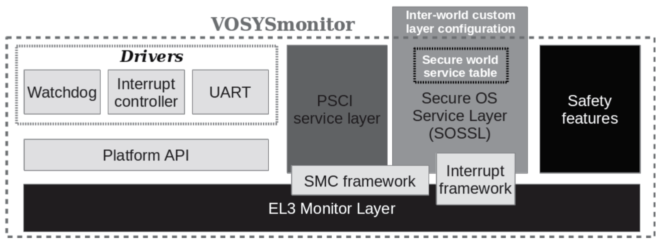

Fig. 4. VOSYSmonitor block diagram

III. State of the art

A. ARM Trusted Firmware and SafeG

ARM Trusted Firmware (ATF) is a software layer able to host a Trusted Execution Environment (TEE) alongside a Non-Trusted OS. Similar to VOSYSmonitor, this firmware relies on the isolation properties of the TrustZone technology (see section II-C) for ARMv8-A architecture.

The isolated Secure world TEE, running on top of ATF, can provide cryptographic primitives to Non-Secure system components to ensure confidentiality and integrity. However, this monitor layer does not support any safety-related features. If we consider the definition of safety as “safety is the absence of unacceptable risk” [16] , those features are mandatory in many different contexts, such as aeronautics and automotive, where any software misbehavior can have disastrous consequences. To circumvent such problems, a number of companies created automotive oriented consortiums. One of them is called Automotive Grade Linux (AGL) and tries to leverage on virtualization for enhancing the next-generation automotive vehicle architectures [3].

On the other hand, an other open source initiative called SafeG [15] (i.e., short for Safety Gate) provides a thin monitor software layer that leverages the isolation properties of ARM TrustZone to execute two OSs concurrently on the same hardware platform. Like VOSYSmonitor, the focus of SafeG is to execute one RTOS and one GPOS but SafeG is currently limited to ARMv7-A based processors. In addition, it is important to notice that this solution does not provide any safety-related features.

B. XEN hypervisor

The XEN project, originally developed at the University of Cambridge, is now a Collaborative Project under the umbrella of the Linux Foundation. Xen is a type-1 hypervisor that enables flexible virtualization by executing multiple operating systems concurrently on a single hardware platform.

Initially dedicated to servers, a revisited version of the project emerged for ARM platforms, called EmbeddedXEN [14]. The latter deals with peripherals and ARM cores heterogeneity and runs on top of ATF with the latest SMCC version and PSCI features. However, XEN does not integrate yet any safety features but the Xen team already started to develop an ISO 26262-ready version.

C. Proprietary solutions

A number of proprietary solutions exist on the market. Among them, OpenSynergy’s COQOS Hypervisor [11], Green Hills’ Integrity Multivisor [5], Mentor Graphics’ Mentor Embedded Hypervisor.

The COQOS Hypervisor allows to run several OS in separate VMs. Moreover, the COQOS hypervisor provides a pre-integrated AUTOSAR environment in a dedicated VM. As for the COQOS hypervisor, some elements of the instrument cluster are safety critical and have been developed according to Automotive SPICE and ISO 26262 ASIL-B practices. By placing the instrument cluster software and a guard mechanism in two different VMs, they ensures that this safety feature is protected from interference. In case of any software failure in the VM running the instrument cluster, the guard mechanism is still able to react and can activate near-immediate recovery of the instrument cluster.

On the other hand, the Mentor Embedded Hypervisor is a Type 1 hypervisor with a small memory footprint, which runs on top of ATF. It takes advantage of ARM TrustZone to support both GPOSs (e.g., Mentor embedded Linux) and secure operations (Secure boot, key management, etc.).

Finally, Green Hills provides a software layer called INTEGRITY multivisor. This software layer allows to run one or more guest operating systems alongside safety critical functions. As the Mentor Embedded Hypervisor, it allows devices and peripherals to be exclusively assigned or shared between guest operating systems and critical functions. The INTEGRITY multivisor is also ISO 26262 ASIL D certified and Green Hills Software is part of the AGL consortium.

IV. Safety features

In this section, the safety measures of VOSYSmonitor, which enables the fault detection as well as to preserve the correct execution of the safety critical domain, are detailed.

A. Supporting process

1) Core synchronization: In order to enable the synchronization between the different cores, where VOSYSmonitor is executing, a mechanism based on a mailbox has been implemented in order to share messages between cores. Such a solution provides the capacity to send a synchronization event to a specific core without impacting the others. In addition, a software interrupt is used to notify the receiving core of a mailbox update. By setting this interrupt as a secure interrupt (FIQ), this solution allows to preempt the core from the Normal world without impacting the Secure world execution, if any.

B. Self-tests

Self-tests are periodically performed during the VOSYSmonitor runtime, to detect any randoms faults that may impact the execution of the safety critical domain. The self-test execution period can be configured at compilation time.

1) Self-test core: A dedicated core is allocated to the self-tests execution (called Self-test core). This core cannot be the same core where the safety critical application is assigned (called Safety core), since the hard real-time requirements of the safety-critical tasks would not be guaranteed.

However, the Self-test core can be used by the Normal world application when a power-up sequence has been requested (through the PSCI convention) by the Normal world. In this context, the co-execution between self-tests and non-critical applications is similar to the Safety core. Indeed, the core is scheduled for the Normal world when no self-tests are executed, while the Secure Timer interrupt is configured as secure in order to preempt the Normal world execution and return in VOSYSmonitor for the self-tests execution.

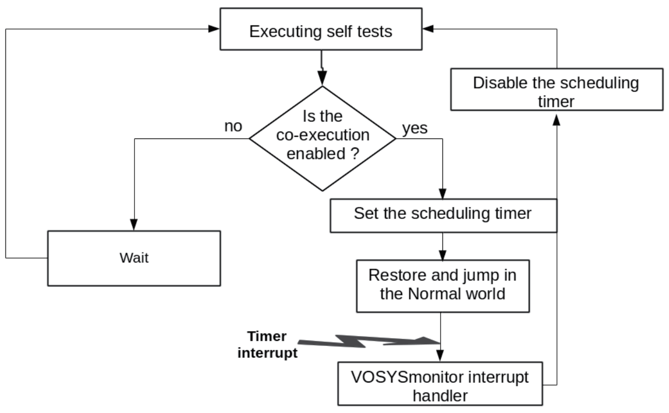
Executing self tests
Disable the scheduling timer
no
Is the co-execution enabled ?
yes
Set the scheduling timer
Wait
Restore and jump in the Normal world
Timer interrupt
VOSYSmonitor interrupt handler

Fig. 5. Self-tests execution

The self-tests executed are:

- **Memory isolation:** Verify the memory regions allocated to VOSYSmonitor and the safety critical domain are correctly configured as Secure (i.e. the Normal world application cannot read or write in this memory area).
- **Peripherals isolation:** Verify the drivers used by VOSYSmonitor are correctly isolated as Secure.
- **VOSYSmonitor code integrity:** Verification of VOSYSmonitor Read-Only code section by calculating and comparing a checksum. In this context, a reference checksum is generated during the VOSYSmonitor initialization. Then, the self-test re-calculates the checksum and compares with the reference one to detect any memory corruption.
- **VOSYSmonitor execution speed:** Check that the clock frequency of the Safety core does not drop down a certain threshold defined at compilation time. Due to limitation of the software, the frequency cannot be measured directly, therefore, the configuration registers generating the clock are read (PLL, divider, etc.), meaning random faults cannot be prevented if there are no hardware mechanism implemented.

2) Self-test performed on Safety core: Due to hardware limitations, some faults can only be detected if the self-test is performed on the Safety core. In order to mitigate the impact on the safety critical applications, the self-tests are executed only during a context switch to reschedule the core for the Normal world application through VOSYSmonitor (i.e., when the safety critical domain has no real-time tasks pending). Unlike the self-tests performed on the Self-test core, the execution of the Normal/Secure worlds applications are not preempted for the execution of these specific self-tests since the safety critical tasks have the highest priority.

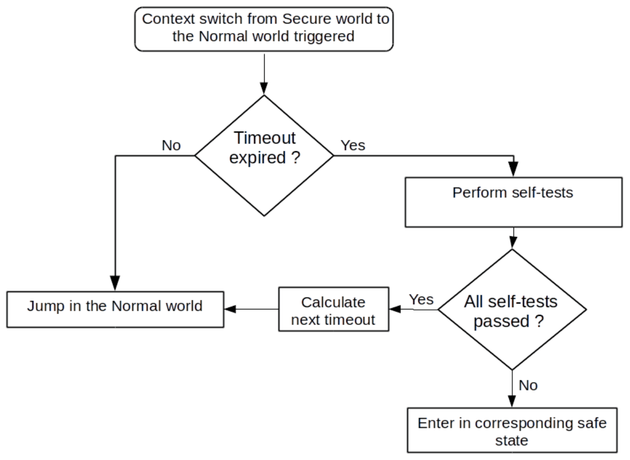

Fig. 6. Self-tests execution on Safety core

The self-tests executed are:

- **TrustZone configuration:** Ascertain the configuration of the Secure Monitor mode registers configured by VOSYSmonitor.

- **Interrupt controller configuration:** Ascertain the correct Interrupt controller configuration to ensure the scheduling policy. Special care is taken to the interrupt used for the Secure world scheduling.

C. Safe state

If one of the self-tests failed, VOSYSmonitor is able to select an appropriate safe state procedure, depending on the failure severity, in order to preserve the execution of the safety-related system running in the Secure world. VOSYSmonitor supports three different safe state modes:

1) Safety application only: VOSYSmonitor stops the co-execution in order to run the safety critical system in stand-dalone mode on the Safety core by preventing SMC call at EL1/PL1 and above. In addition, a sync event is generated to all secondary cores, which are executing the Normal world applications, in order to reboot the Normal world only on the cores, which are not assigned to the safety critical domain.

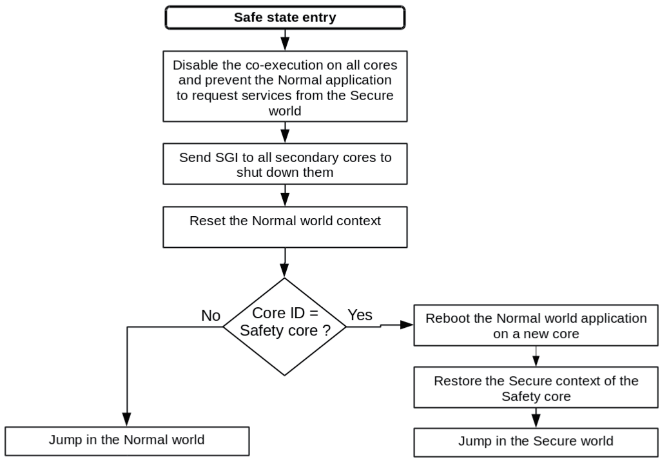
Safe state entry
Disable the co-execution on all cores and prevent the Normal application to request services from the Secure world
Send SGI to all secondary cores to shut down them
Reset the Normal world context
Core ID = Safety core ?
No
Yes
Reboot the Normal world application on a new core
Restore the Secure context of the Safety core
Jump in the Normal world
Jump in the Secure world

Fig. 7. Safety critical OS preservation

2) Normal world powered off: VOSYSmonitor stops the co-execution in order to run the safety critical system in standalone mode on the Safety core by preventing SMC call at EL1/PL1 and above. In addition, a sync event is generated to all secondary cores, which are executing the Normal world applications, to shut them down in order to ensure that the safety critical domain, running in the Secure world, will not be corrupted in case of spatial and/or temporal isolation failure.

3) Safety application migration: VOSYSmonitor migrates the execution of the safety critical from the Safety core to another core, which is not located into the same cluster. Once the new core has been identified, VOSYSmonitor populates the context of the previous Safety core to the selected core, then it updates the ”Execution state” structure with this new information. In addition, all secure interrupts (FIQ), which targeted the previous Safety core, are migrated to the new core. It is important to notice that VOSYSmonitor is able to power on the core identified for the migration if it is not already up and running. Finally, the previous Safety core is either redirected to the Normal world execution or shut down if the Normal world did not power-up previously.

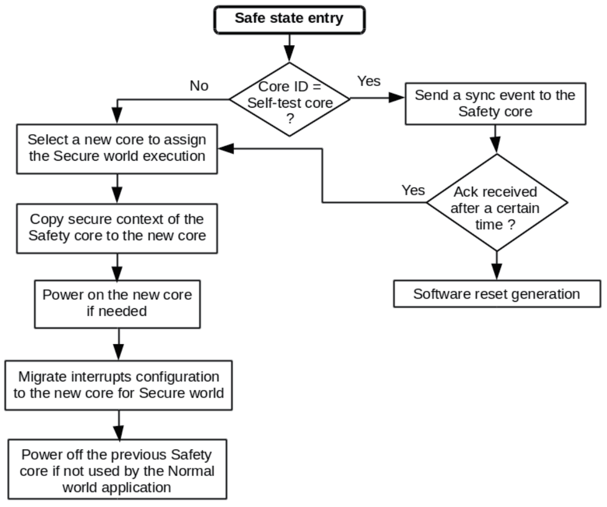
Safe state entry
Core ID = Self-test core ?
No
Yes
Send a sync event to the Safety core
Select a new core to assign the Secure world execution
Yes
Ack received after a certain time ?
Copy secure context of the Safety core to the new core
Software reset generation
Power on the new core if needed
Migrate interrupts configuration to the new core for Secure world
Power off the previous Safety core if not used by the Normal world application

Fig. 8. Safety critical OS migration

V. EVALUATION

This section aims to highlight the execution process upon detection of a failure. Three metrics in particular are evaluated:

- **Self-test Execution Time (ST):** Time required for VOSYSmonitor to run the corresponding self-tests sequence on the Self-test core and the Safety core.
- **Fault Detection Time (FDT):** Time required for VOSYSmonitor to detect a failure.
- **Recovery Time (RT):** Time required for VOSYSmonitor to update the world scheduling in order to preserve the mission-critical tasks running in the Secure world when a failure is detected.

Fig. 9 depicts the execution process when a failure is detected during the self-tests execution. The process might slightly differs for the self-test execution on the Safety core since they might be impacted by the scheduling of the safety critical application but the concept remains the same. For the evaluation purpose, we will assume that the safety critical application is scheduling in order to not impact the self-test execution on the Safety core.

The evaluation uses the ARMv8 Performance Monitoring Unit (PMU) [7], which counts the number of CPU clock cycles consumed. The tests have been performed on the Renesas R-Car M3 Salvator-X board, which includes a Cortex A-57 (1.5 GHz) and a Cortex A-53 (1.3 GHz) cluster. As results may vary depending on the core where VOSYSmonitor is operating, all tests have been executed on both A-53 and on A-57. In addition, the number of CPU clock cycles are listed in the results in order to provide an estimate of the self-tests latency if another board with a different core frequency is used. Finally, the tests are realized in the best performance conditions since only VOSYSmonitor is using the CPU caches (i.e., A57 - L1 cache of 48KB, L2 cache of 1MB; A53 - L1 cache of 32KB; L2 cache of 512KB). As such, the number of cache misses might differ compared to a real use-case where the applications running in the Normal/Secure worlds might flush the caches

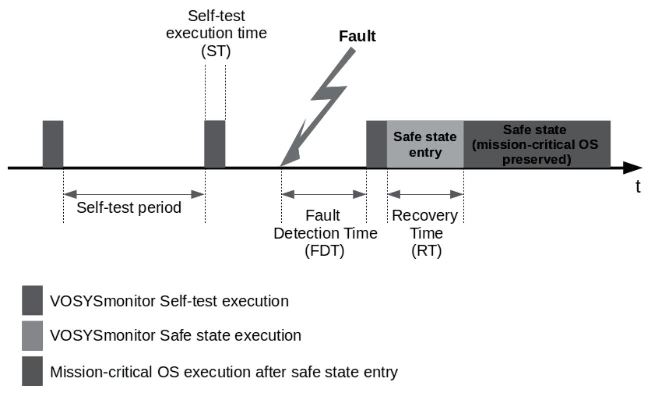
Self-test execution time (ST)
Fault
Safe state entry
Safe state (mission-critical OS preserved)
t
Self-test period
Fault Detection Time (FDT)
Recovery Time (RT)
VOSYSmonitor Self-test execution
VOSYSmonitor Safe state execution
Mission-critical OS execution after safe state entry

Fig. 9. Safety features execution overview

lines used by VOSYSmonitor and the performance might be affected.

A. Self-test execution time

The goal of this section is to evaluate the Self-test Execution Time (ST in Fig. 9). As said in Section IV-B, self-tests are executed on two distinct cores: the Self-test core and the Safety core. The evaluation consists in measuring the execution of all self-tests in the worst case (i.e. taking the more clock cycles).

1) Self-test core: The worst case for the self-tests execution is when the self-test core is currently scheduled by the Normal world. Indeed, in this case, the Normal world context has to be saved before starting the self-tests execution.

The table below lists the time consumed for the self-test execution, starting from the preemption of the Normal world execution by the Secure Timer interrupt (see Fig. 5), to the end of the self-test sequence.

TABLE I. SELF-TESTS LATENCY ON SELF-TESTS CORE

Processor	Frequency (Ghz)	Clock cycles	Latency (ns)
Cortex-A53	1.3	63508	48.9
Cortex-A57	1.5	61347	40.9

The results show that the Cortex-A57 has better performance whether the CPU clock cycles or the latency is considered. However, it is important to analyze the impact of self-tests execution on the Normal world application since this latter is preempted. The shortest self-test period, which might be selected at the compilation time, is 1ms (any lower input value is rejected). As a result, the worst case scenario (i.e., self-test period of 1ms) on the self-test core consumed 4.4% of the processing time for the self-test execution. Thus, the impact on the Normal world application is considered minor, furthermore in the case of a multi-processor application. Indeed, the self-tests execution impact would be further attenuated as only one core (i.e., the Self-test core) is concerned. However, in order to prevent any excessive latency on the Normal world execution, especially during the initialization phase, VOSYSmonitor cannot schedule the self-tests execution on the Primary core (i.e. the core being powered up first during a reset).

2) Safety core: As previously mentioned, these self-tests are only executed when the Safety critical OS has no pending tasks and gives the control to the Normal world application. In this context, only the self-tests execution is measured, meaning the latency induced for preserving the world context is not included below.

TABLE II. SELF-TESTS LATENCY ON SAFETY CORE

Processor	Frequency (Ghz)	Clock cycles	Latency (ns)
Cortex-A53	1.3	56	0.04
Cortex-A57	1.5	99	0.07

We can observe that the latency induced by the self-test sequence on the Safety core is insignificant, especially if we compared this value with the measurement of the self-test sequence on the Self-test core. This result can be explained by the self-tests sequence, which is minimalist on the Safety core. In addition, a standard context switch is performed in 1.14 us on the Renesas R-Car M3 Salvator-X (Cortex-A57). Therefore, when the self-test sequence is executed during the context switch, an overhead of 6% is observed, which increases the full context switch time to 1.21ns. Although this overhead is minor, it is important to notice that the impact on the safety critical application is none since the self-test sequence is only performed during a switch from the Secure world to the Normal world.

B. Fault Detection Time

It is not possible to estimate the FDT since it is dependent from the error occurrence as well as the self-test period. Indeed, as specified in Section IV-B, the self-test period can be selected by the user at the compilation time. However, it is possible to give an estimate of the worst-case FDT.

At the time of writing, the biggest self-test period configurable is 10s. In such a case, if the error occurs just after a self-test sequence, it implies that the error will be detected on the next self-test sequence, therefore, the FDT will be around 1Os in this worst scenario. On the other hand, the latency induced by the self-test execution might be also considered. Indeed, the error may occur during the self-tests execution, and might be not detected if the self-test in charge has already been executed. However, this latency is considered negligible according to the result in table V-A1.

C. Recovery Time

As discussed in Section IV-C, three safe states are available according to the error triggered. This section lists the different latencies by measuring the delay between a safe state request and its completion.

1) Safety application only: This safe state is the most time-consuming, which is due to the Normal world reboot that may vary depending on the application. In this test case, Linux is running in the Normal world after being flashed by U-boot. Therefore, it is needed to flash U-boot binary during the reboot procedure, thus causing this relatively huge latency. This operation might be avoided (and the performance improved) if the U-boot binary can be re-executed after a first execution.

In case this delay is unacceptable for the use-case, it is possible to use the safe state *Normal world powered off* (see section IV-C2), which offers better latency.

TABLE III. Safety application only latency

Processor	Frequency (Ghz)	Clock cycles	Latency (ns)
Cortex-A57	1.5	931740	621.2

2) Normal world powered off: This safe state requires less time to perform its sequence but the counter-part is that the Normal world application is completely powered off. However, such implementation is needed since it allows the preservation of the safety critical application in case of spatial and/or temporal isolation failure. Most of the latency is due to synchronization between cores. Indeed, the safe state will not conclude until all secondary cores, not allocated to the safety features (i.e., Self-test core and Safety core), are powered down.

TABLE IV. Normal world powered off latency

Processor	Frequency (Ghz)	Clock cycles	Latency (ns)
Cortex-A57	1.5	9010	6.0

3) Safety application migration: This safe state execution is relatively short compared to the *Safety application only* one since the Normal world application is not rebooted. Indeed, during the context switch from the Normal to Secure world, a new core is selected to migrate the Secure world context. The latency is mainly due to the population operation of the interrupt controller configuration. Indeed, all secure interrupts routed to the previous Safety core must be redirected to the new selected core. As such, all interrupts must be checked to see if they are assigned to the Secure world, and the resulting latency is proportionate to the number of interrupts implemented (i.e., 511 on the Renesas R-Car M3 Salvator-X for instance).

TABLE V. Safety application migration latency

Processor	Frequency (Ghz)	Clock cycles	Latency (ns)
Cortex-A57	1.5	28082	18.7

VI. Conclusion

The ISO 26262 safety standard allows for the certification of a SEooC. This means the standard is applied on an element, where the exact context and scope of the element is not known, yet. VOSYSmonitor, a secure monitor layer, which is developed for a range of mixed-criticality applications constitutes such an SEooC.
This paper, demonstrated the process of applying the ISO 26262 on VOSYSmonitor. In particular, the safety properties that VOSYSmonitor provides have been laid out and it's performance numbers measured (e.g., latency of a self-test). While the safety features presented, are primarily tailored towards VOSYSmonitor, they still provide general insights on how to design an ISO 26262 certified element which is not developed in the context of a particular system or vehicle.
Further, it is important to mention, that the ISO 26262 standard puts a particular focus on the **safety manual**. However, while the SEooC provides generic safety features, the hazards for a specific use-case cannot be listed and thus, no mitigation can be proposed. In this case instead, the safety manual lists Safety Related Application Conditions (SRAC). These requirements then have to be fulfilled by the *system integrator* (i.e. the person/company responsible for the item).

VII. Acknowledgment

This work was supported by the dReDBox project. This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 687632. This work reflects only the authors' view and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] Virtualbox. http://www.virtualbox.com, March 2018.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. 37(5):164–177, 2003.

[3] The Liunx fondation. The automotive grade linux software defined connected car architecture. Whitepaper, June 2018.

[4] Yutaka Haga, Kazuhide Imaeda, and Masayuki Jibu. Windows server 2008 r2 hyper-v server virtualization. *Fujitsu Sci. Tech. J*, 47(3):349–355, 2011.

[5] Green Hill. Integrity multivisor. *URl: http://www. ghs. com/products/rtos/integrity_virtualization. html*, 2016.

[6] ARM Limited. Building a secure system using trustzone technology. Whitepaper, Avril 2009.

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

[8] HORIBA MIRA Ltd. Misra - the motor industry software reliability association. URL: https://www.misra.org.uk.

[9] Pierre Lucas, Kevin Chappuis, Michele Paolino, Nicolas Dagieu, and Daniel Raho. Vosysmonitor, a low latency monitor layer for mixed-criticality systems on armv8-a. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[10] Roberto Mijat and Andy Nightingale. Virtualization is coming to a platform near you. Whitepaper, January 2011.

[11] OPENSYNERGY. Coqos hypervisor sdk. https://www.opensynergy.com/coqos-hypervisor-sdk/.

[12] Parallels. Parallels workstation, parallels desktop. http://www.parallels.com, March 2018.

[13] Qumranet. Kernel-based virtual machine for linux. http://qumranet.com/kvm, March 2018.

[14] Daniel Rossier. Embeddedxen: A revisited architecture of the xen hypervisor to support arm-based embedded virtualization. Whitepaper, June 2012.

[15] Daniel Sangorrin, Shinya Honda, and Hiroaki Takada. Dual operating system architecture for real-time embedded systems. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Brussels, Belgium*, pages 6–15, 2010.

[16] Dr. Henrik Thane. Testing and safety standards. http://swell.weebly.com/uploads/1/4/3/4/1434953/swell_safety_and_verification_20111007d.pdf.