A Safe Graphics Rendering Solution for Consolidated Operating Systems

Angelos Mouzakitis*, Kevin Chappuis*, Julian Vetter*, Michele Paolino*, Youssef Kamoun* and Daniel Raho*
*Virtual Open Systems, Grenoble, France
Email: {a.mouzakitis,k.chappuis,j.vetter,m.paolino,y.kamoun,s.raho}@virtualopensystems.com

***Abstract*—New breakthroughs in the automotive domain, such as Advanced Driver Assistance Systems (ADAS), 5G Vehicle to Everything (V2X) connections and In-Vehicle Infotainment (IVI) systems have made a significant impact on the automotive industry. Virtualization plays a key role in this trend, since it provides the ability to consolidate services with different levels of criticality, such as for instance ADAS functions and IVI or 5G connectivity services. Today, one scenario that arises with this new trend is the consolidation of a safety critical digital instrument cluster which displays safety metrics, e.g., speed, torque, etc. along with an IVI system. In such an architecture, the Graphical Processing Unit (GPU) is of central importance to ensure an efficient implementation. However, utilizing the GPU in both compartments raises safety concerns, and poses the question whether the strict isolation implemented by the virtualization layer can be upheld. Therefore, in this paper, we investigate this issue, and address it, by proposing a solution that consolidates a safety critical digital cluster along with an IVI system. We present the design of a safety mechanism to isolate the GPU rendering in both compartments, called "split-display", leveraging the ARM® TrustZone® technology. In our design, the secure world hosts a Real-Time Operating System (RTOS), which handles the GPU rendering in order to protect mission-critical tasks (e.g., speedometer and warning icons) from potential failures occurring in the IVI system. The mechanism provides safety guarantees for the GPU rendering of the RTOS. Our prototype "split-display" solution for mixed-criticality systems is implemented on the Renesas R-Car H3 platform. To validate our prototype implementation, we performed a number of experiments and evaluate the performance impact that occurs due to the consolidation. The results show that our implementation ensures at least 30 frames per second (fps) which is in line with the ISO 15005 safety standards. This number can even be achieved if a failure occurs in the IVI system.**

Keywords–Graphics; Split-Display; Mixed-Criticality; Real-Time; VOSYSmonitor

I. INTRODUCTION

Road vehicles nowadays are host to a huge number of embedded processors, executing millions of lines of code. However, the maintenance of these large code bases is tedious and error-prone for the vehicle manufacturers. Therefore, the manufacturers try to use off-the-shelf software wherever possible to facilitate and streamline their development process. The availability of existing Real Time Operating System (RTOS) solutions proves itself useful in this respect. Especially, since the OSEK/VDX [1]–[3] consortium certified some of these RTOSs as suitable for the use in vehicular embedded control units. Such OSEK/VDX-conforming RTOSs address the needs of the vehicle manufacturers in almost all concerns. They only fall short in a small but critical number of domains such as In-Vehicle Infotainment (IVI), security and safety. To address these shortcomings alternative RTOSs for the high-end automotive domain are available today.

But not only leveraging off-the-shelf software helps to streamline the development process of the vehicle manufacturers, also the integration of multiple components of different criticality (forming a so called Mixed Criticality System (MCS) [4]) lowers the number of embedded controllers in the car and consequently reduces cost, space, weight, heat generation and power consumption. By leveraging virtualization, the vehicle manufacturers can now run multiple systems on a single embedded controller, from a highly reliable RTOS for mission-critical functions to a highly customizable Linux-based system for IVI services.

Although, virtualization enables numerous new applications, the trend also poses new challenges on the software stack. One such challenge is the proper sharing of hardware resources. Since all software components access the same hardware components of the embedded controller, special care must be taken when integrating such an architecture. In the past, researchers have already shown how to undermine the isolation enforced by the OS using shared hardware resources (e.g., caches [5], DMA devices [6], etc). Thus, a resilient software architecture needs to be in place.

In this context we investigate the sharing of a common display among two components with different levels of criticality. This is not only challenging because the involved components (e.g., Video Signal Processor) are powerful embedded devices in itself, which have to be handled with care, to not jeopardise the system integrity. But the task is also urgent because a rich automotive user interface calls for the integration of an IVI along the instrument cluster.

In this paper we present a solution, integrated into an open source RTOS which composes multiple visual elements and renders them to a common display to meet the needs of future automotive applications in areas like IVI or security/safety. The system relies on the isolation properties enforced by a highly privileged software component called VOSYSmonitor [7], which guarantees isolation between peripherals and memory of both OSes using ARM® TrustZone®.

In particular, we make the following new contributions:

- We design a mixed criticality "split-display" solution to allow a mission critical instrument cluster to run side-by-side with an IVI system.
- we evaluate existing solutions, depict their advantages and drawbacks and position our novel approach among them.
- We implement a full prototype of our architecture and evaluate its performance on an automotive grade evaluation board (Renesas R-Car H3).

 ISBN: 978-1-61208-626-2

The rest of this paper is structured as follows. In Section II, we give background on virtualization, ARM® processor extensions and hypervisors in general. Then, in Section III we present related work and emphasize the advantages and drawbacks of existing solutions compared to our design. Our system architecture is described Section IV. Section V outlines our design. We focus on the peculiarities of our driver and automotive application in Section VI. We evaluate our design in Section VII. We conclude our work and present future work in Section VIII.

II. Background

In the following sections we give a brief overview of ARM® TrustZone®, ARM® Virtualization Extensions (VE) as well as the different types of hypervisors. We conclude this section with a concise introduction to VOSYSmonitor, which is the underlying firmware layer, providing the necessary isolation building blocks.

A. ARM TrustZone

Under the term TrustZone® [8], [9], ARM® introduced a new separation concept that is orthogonal to ELs (exception levels). This new concept provides two worlds, a "secure world" with the same set of ELs and a new mode, called monitor mode, to switch between this new and the classical "non-secure world" (Figure 1). The goal of TrustZone® is to keep the non-secure world fully backward compatible. Thus, the world separation and switching between worlds is almost fully implemented in hardware (with new sets of banked registers, etc). A new processor instruction was introduced with TrustZone®, the `smc` (Secure Monitor Call) instruction [10], to provide a way of interaction between both worlds. The isolation between the worlds is enforced in combination with other cooperating system components. A TrustZone® compliant memory controller announces the current security state ("secure" or "non-secure") for every bus transaction via the AXI AxPROT signal. Also, a new TrustZone® controller was introduced to allow for the configuration of certain ranges of physical memory as secure, preventing the non-secure world from accessing them. Moreover, the standard ARM® interrupt controller (GIC) supports the classification of interrupt sources into groups, allowing them to be routed to either the secure or non-secure world.

But it is important to note that the design of TrustZone® aims at a large degree of autonomy of both worlds, without the possibility (and perceived need) of close interaction. This means, as opposed to ARM VE, TrustZone® does not provide the ability to trap certain instructions, provide a nested paging mechanism or allow the direct injection of interrupts into specific virtual machines.

B. ARM Virtualization Extensions

ARM® added full virtualization support as an optional feature in ARMv7 [11]. Systems with these extensions have an additional execution mode, hypervisor mode (hyp). This mode is located in the new privilege level EL2, placed below EL0 and EL1. In addition, to having full access to all system control registers that exist in EL1, software executing in EL2 is provided with additional control registers for reconfiguring execution in EL0 and EL1, by, e.g., trapping certain instructions. 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.

C. 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 a 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 [13] or Hyper-V [14].

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. The host operating system however, has to corporate with the hypervisor process and, e.g., reflect specific types of exceptions back to this process. Famous examples for such a hypervisor are VirtualBox [15] or Parallels [16].

Kernel-based Virtual Machine (KVM) [17] 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.

D. VOSYSmonitor

VOSYSmonitor is a firmware that runs in the Secure Monitor mode (EL3) of ARMv8-A processors. It enables the native concurrent execution of two operating systems, such as, e.g., a safety critical RTOS along with a GPOS (General Purpose Operating System). The execution of both, 32-bit and 64-bit applications is possible and their isolation ensured by the means of TrustZone®.

Since VOSYSmonitor runs in EL3, the GPOS can still opt for a solution such as Linux-KVM, and leverage the ARM® VE to instantiate multiple VMs (Virtual Machines). Yet, the RTOS running in Secure world, is completely isolated from these applications executing in the normal world.

To ensure an efficient context switching between the two worlds hardware exception mechanisms, such as interrupts are used. Additionally, both OSs 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 world.

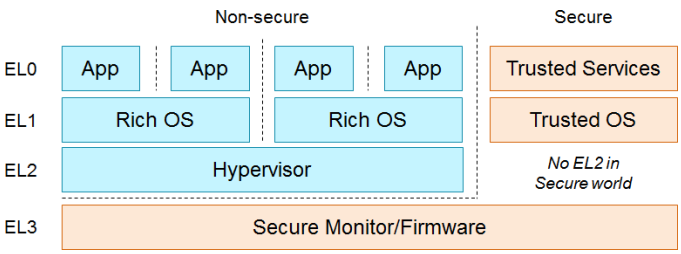

Figure 1. ARMv8 processor architecture [12]

III. Related work

In the following section we present a number of projects or solutions which are relevant for this work.

In [18], Lee et al. describe an architecture called VADI which enables the execution of a digital instrument cluster on a consolidated hardware platform. The architecture is able to concurrently process graphic commands for two isolated execution domains using one Graphical Processing Unit (GPU) device that renders the frames on one display [19]. VADI implements a GPU sharing mechanism, while protecting the GPU and display device from non-trusted software applications by using TrustZone®.

Since physical virtualization is vulnerable to device driver failures, they leverage a virtualization solution called SASP, to consolidate the digital instrument cluster along with the IVI system. An RTOS, such as AUTOSAR [20] and a GPOS (e.g., Linux) can execute concurrently.

According to the evaluation phase, VADI maintains a performance of 30 frame per second (FPS) [21] [22], which is in line with the requirements for automotive control software. In addition, VADI ensures the execution of the digital instrument cluster component even in case of an unrecoverable failure of the other execution domain.

A framework that allows the sharing of a single GPU among different Virtual Machines (VMs), has been presented by Qi et al. [23]. The framework called VGRIS provides a GPU command queue for each VM in the main memory of the host OS. When an application calls a GPU API function the host OS intercepts the call and converts it into a specific command which is stored in the GPU command queue. The host OS runs a GPU scheduler which selects a specific queue and sends the commands to the GPU.

The evaluation of the Service Level Agreements (SLA) scheduling shows that the average FPS rates of the tested games, running in independent VMs are around 30 FPS. The drawback of VGRIS is, it depends on a Type 2 hypervisor because it is implemented in the host OS.

The VAGS architecture proposed by Gansel et al. [24] is an automotive display consolidation architecture which implements GPU virtualization in a vehicle. VAGS performs all graphic processing on a consolidated GPU device and draws the rendered frames on several display devices, such as the head unit screen, the center console, and the digital cluster.

VAGS consists of a Window Manager, which is based on a hierarchical access control management for display areas and input events. Therefore, the applications create, delete and move their windows through a dedicated Window Manager API. Depending on their permissions and priorities, applications are allowed to display their windows in dedicated display areas in order to avoid the overlapping of windows applications with different priorities.

The entire design from the virtualization manager to the graphical applications is based on Linux. However, VAGS only presents the design without any implementation and evaluation details since it is a work in progress solution.

The VMGL solution by Andres et al. [25] is the most popular GPU virtualization mechanism for virtual machines. VMGL uses a network device to send graphic commands to a virtualized GPU driver. The solution uses a standard network interface, such as a socket, and thus guarantees independence from the virtual platform. VMGL performance results closely resemble the results of their native counterparts.

The drawbacks of VMGL are, the following. Complex system configurations with multiple high-end applications, which concurrently render, may impose an aggregate bandwidth demand on VMGL of several Gbit/s. Moreover, the network communication requires a specific device and increases the transmission time of graphic commands between VMs since additional network processing and data copies are needed. In addition, automotive systems require device isolation to protect devices from errors caused by non-critical applications. Therefore, a system with VMGL must provide an additional isolation mechanism for the network device. Moreover, the VMGL mechanism is implemented on Xen [13] and VMware for the x86 architecture. But graphic libraries for embedded systems such as OpenGL ES and EGL are very different from libraries for x86 desktop systems like OpenGL and GLUT, thus further complicating the integration of VMGL into an automotive software stack.

IV. Architecture

The study of related work already highlights the requested functionality for rich graphical automotive applications, and also stresses the challenges that arise when consolidating such a system. Based on these requirements, we present our solution in the following section.

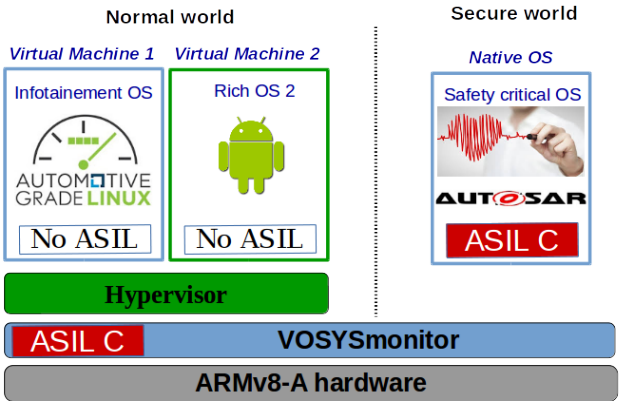

Figure 2. Automotive Architecture

Our architecture allows the rendering of content from both execution domains on a common physical screen. To achieve a high responsiveness of the digital instrument cluster we ported an open-source RTOS (i.e., FreeRTOS [26]) to run on top of VOSYSmonitor in the secure world while we rely on the high customizability of Linux in the normal world. Apart from ensuring a strict spatial and temporal isolation of the RTOS running in the Secure world, VOSYSmonitor also ensures a proper execution of the RTOS, when the non-critical application in the normal world fails.

The IVI system is composed of several components. It provides a rich interactive map that shows detailed route information, allows route planning, and also offers direction indicators for the driver. Along the map application which is integrated into a Linux system, we execute a virtualized Android VM to provide an off-the-shelf car infotainment system, paired

 ISBN: 978-1-61208-626-2

with controls for interacting with the air conditioning system, a calling application or a video player. The digital instrument cluster on the other hand is implemented in the RTOS and is responsible for rendering mission critical information, such as the accelerometer, the tachometer and different warning icons for the temperature, missing seatbelts, etc. Figure 2 shows our architecture and the placement of the different components in the MCS. VOSYSmonitor constitutes the highest privileged component in the system, on top of which two operating systems are consolidated.

V. Design and Implementation

In Section IV, we described the high-level system architecture, and the placement of the components. Now we take a closer look, on how a safe interaction between the IVI and the instrument cluster is ensured by our architecture.

Figure 3 gives a detailed view on the vital system components. In Figure 3, it can be noticed that the GPU driver is

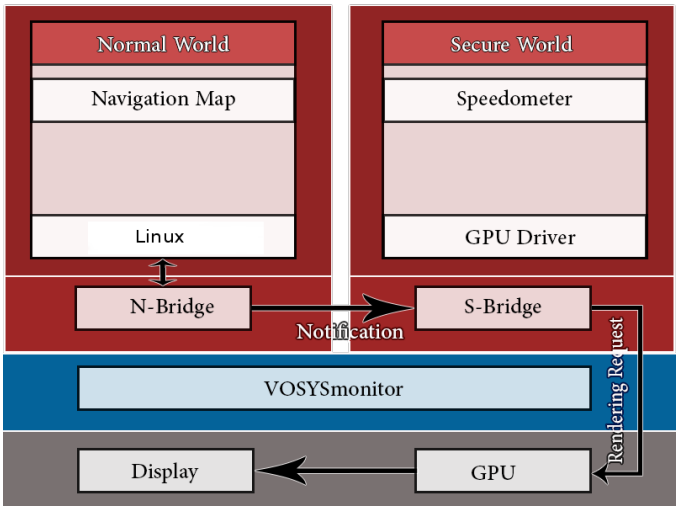

Figure 3. System Architecture

only available to the secure domain, thus requiring a routing mechanism to allow the normal world applications to interact with the graphic buffer as well. To achieve this, we designed two components, the S-Bridge for the secure and the N-Bridge for the normal world. These two components allow for an efficient message transfer between the normal and secure world. Notifications that are issued in the normal world, are routed to the secure world via the `smc` instruction. On top of which we use the concept of Remote Procedure Calls (RPC), to allow the normal world to invoke the desired functionality. That is, accessing GPU resources from the normal world, relies on invoking notifications from the N-Bridge to the S-Bridge. The Secure world then forwards the rendering requests to the GPU, which is in charge to render graphical content to the non-secure framebuffer.

While calls to the GPU are forwarded to the secure world, the planes are managed individually by the according OS (either RTOS or Linux). But, there are two dedicated display areas for the secure and normal world which are mapped to a depth level representing the priority of the plane. The concept is depicted in Figure 4. The figure shows the placement of the two framebuffers (planes) on the display. The plane with the

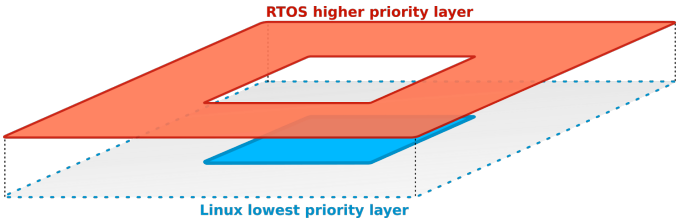

Figure 4. Display controller composition

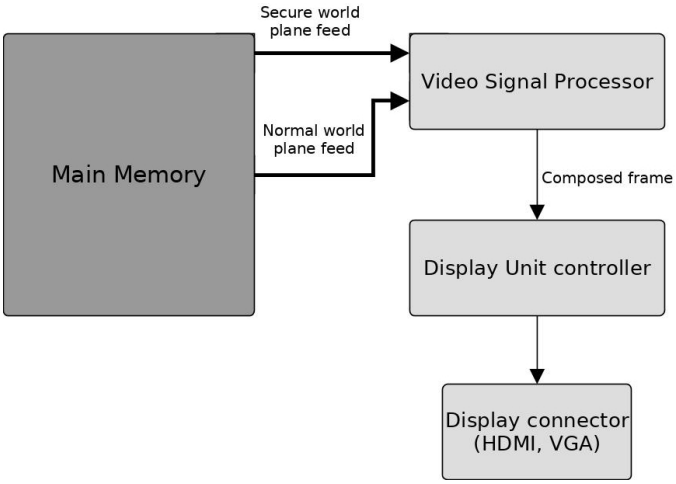

Figure 5. Hardware peripherals involved in the graphics pipeline

higher priority is displayed on top of the one with the lower priority. So in our case, the RTOS is able to render on top of the Linux (which is displayed on the lower plane). Such a hierarchical management for display areas ensures that mission critical information are never cloaked by IVI content.

To summarize the design of our "split-display" architecture, the overall system implementation relies on the isolation capabilities of VOSYSmonitor (which makes sure that hardware peripherals needed for graphic rendering are only accessible from the secure world), a set of modified drivers that operate in the Secure world as well as the S-Bridge and N-Bridge. The design of the drivers allows both compartments to share the hardware peripherals that are involved in the graphics pipeline, while respecting the constraints imposed by the ISO 15005 standard for a safe display solution.

VI. Drivers & Automotive Application

In this section we discuss the different driver components as well as their implementation. But in order to understand the interaction of the driver components, we first have to take a look at the hardware peripherals that are involved in the graphics pipeline.

A. Hardware peripherals

Figure 5 illustrates the high level overview of the hardware peripherals that are used by the graphics rendering pipeline. The Renesas R-Car H3 contains multiple instances of these hardware components in order to forward rendered content to up to four different physical display connectors (VGA, LVDS and two HDMI) in parallel.

 ISBN: 978-1-61208-626-2

Our architecture ensures that all hardware peripherals and resources that are affected by the graphics rendering pipeline (direct or indirect), are isolated and protected from malicious activity of the normal world. VOSYSmonitor ensures this by configuring the memory address space of the peripherals that are driven by the RTOS using TrustZone®, to only allow "secure" accesses to the memory. Moreover, VOSYSmonitor also applies memory protection to peripherals related to the power domain configuration, clock generation, the system reset hardware block as well as the memory area used by the RTOS framebuffer. This system configuration ensures a correct rendering behaviour for the safety critical information at all times.

B. Video Signal Processor driver

The Video Signal Processor (VSP) peripheral is the core hardware peripheral to achieve the concept of the "split-display". The VSP is capable to read data from the main memory, through an intermediate data compression peripheral, through up to five independent read channels. Each read channel transfers data, which corresponds to a different plane. The device composes a frame based on the internal configuration of the composed planes.

The VSP driver operates in the RTOS and initializes the device with a static configuration for the operational clock and for properties of each read channel. Properties of the read channel include, e.g., the framebuffer's starting address, the resolution, the color depth, the plane's size and position in the display. For demonstration purposes, the scenario that we evaluate in Section VII consists of an RTOS plane, which has the following properties: width = 1920, height = 1080 and is placed at (0, 0) on the screen. Similarly, the Linux's plane has the following configuration: width = 640 height = 480 and it is placed at (640, 300). In this context, Linux is able to render at its own plane, that is initialized by the RTOS, and thus preventing further configuration changes. Moreover, the isolation of the planes for the RTOS and the Linux system as well as the frame composition ensures that the normal world cannot render at a part of the display that is owned by the secure domain.

C. Display unit driver

The Display Unit (DU) peripheral is the component in the graphics pipeline, which receives the output from a VSP module and produces the desired output (i.e., frame) regarding the internal configuration timings. It is important to note that the RTOS drives the peripheral with the same security policy as the VSP module. In this scenario, the physical connector for the split-display screen output is the Video Graphics Array (VGA) and the resolution is configured to 1920x1080x32bpp utilizing the planes of both OSs.

D. Automotive application

The automotive application is highly decomposed. Figure 6 shows the individual components and how the aggregated image is generated. For the secure world, we adapted a digital instrument cluster to run in the RTOS. The adaption also required us to first convert the images used by the instrument cluster (i.e., speedometer, tachometer and warning icons) from an ordinary image format into a binary format (according to the properties required by the framebuffer), referencable as symbols in the data section of the application. In the normal world we further decomposed the IVI components and added an additional layer of isolation by using the KVM hypervisors.

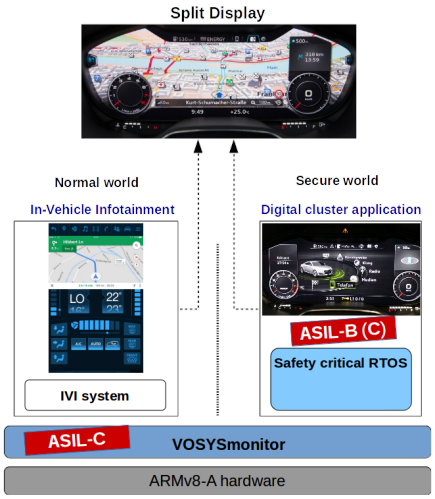

Figure 6. Automotive Application

Our final setup is shown in Figure 7. Whereby, we routed the output of different components to different hardware ports. Table I gives an overview were each component IVI and instrument cluster was routed to.

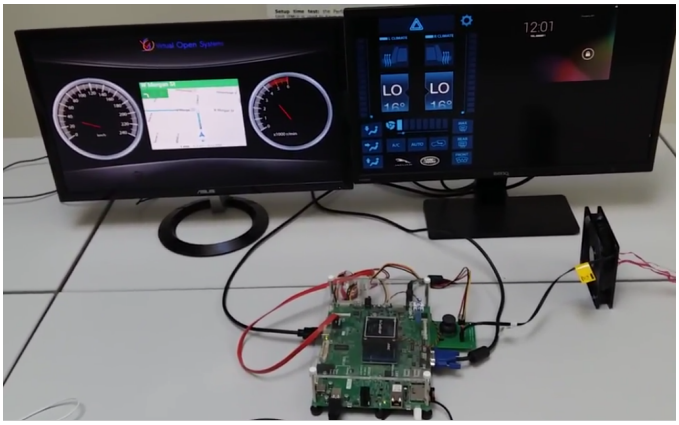

Figure 7. Evaluation board and display device

TABLE I. THE DIFFERENT APPLICATIONS OF THE IVI AND INSTRUMENT CLUSTER, WHERE THEY ARE PLACED IN THE SOFTWARE STACK AND THEIR OUTPUT DISPLAY.

Application	Component	Output Port
Secure application (Speedometer, Tachometer)	Instrument cluster (secure)	VGA
Navigation map	IVI (non-secure)	VGA
Control application for heating ventilation and air conditioning (HVAC)	IVI (non-secure)	HDMI 1
Android Lollipop (Hardware accelerated)	IVI (non-secure)	HDMI 2

 ISBN: 978-1-61208-626-2

VII. Evaluation

Our evaluation setup consisted of the following software components. In the normal world we executed a system based on Linux kernel version 4.4, while in the secure world we executed a FreeRTOS with a modified driver stack. We generated two realistic graphic workloads (as described in Section IV). The speedometer along with the tachometer in the secure world continuously rendered the vehicle's speed and engine torque. On the other hand we had the navigation map in the normal world showing detailed directions to the driver. These two applications represented our instrument cluster and IVI system, respectively. The entire graphical workload was being rendered using the CPU since FreeRTOS did not support GPU drivers for hardware acceleration.

Figure 7 depicts our hardware test setup. It consisted of a Renesas R-Car H3 [27] evaluation board. The board offers an automotive grade solution and features multiple ARM®-Cortex®-A57/A53 cores. The processing performance achieved by the hardware platform was 40,000 DMIPS (Dhrystone million instructions per second).

As shown in Figure 7, we split the display into two planes, one for the secure world and another one for the normal world. The upper plane displayed the speedometer, tachometer alongside the warning icons which were rendered by the Secure world, while the lower plane presented the navigation map of the normal world.

A. Rendering Time

This metric determines the number of frames that the CPU is able to render on the display. An important requirement in automotive industry is to ensure a minimum frame rate for critical information, such as the speed indicator and the warning icons. The purpose of this evaluation is to ascertain isolation between two graphical applications running in both worlds and that if a crash occurs in the normal world, performance of the Secure world is not impacted.

For the evaluation, the CPU rendering of the RTOS and the navigation map in the normal world are processed simultaneously, while a RTOS thread monitors the frames rendered per second by the RTOS.

Figure 8 indicates the performance of the Secure world running alongside the normal world and Figure 9 shows a result of the Secure world while the normal world remains idle. As shown in Figure 8, the frame rate of the digital cluster decreases whenever there is an animation and it is high when the animation stops. The animation requires frequent frame updates because it has to trace all new locations of the corresponding item (e.g., icons and the needle of the speedometer). Figure 9 shows that the secure world maintains a minimum frame rate of 30 fps even with a fault in the normal world. This means that the critical tasks running in the Secure world are completely isolated from the tasks running in the normal world. Moreover, in this evaluation, despite the fact that the navigation map is halted due to an unrecoverable fault (e.g., kernel fault) occurring in Linux, the speedometer remains running without any impact on its performance.

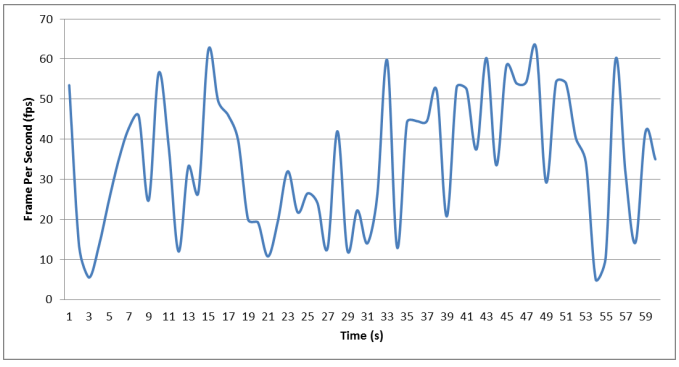

Figure 8. Performance of Digital cluster

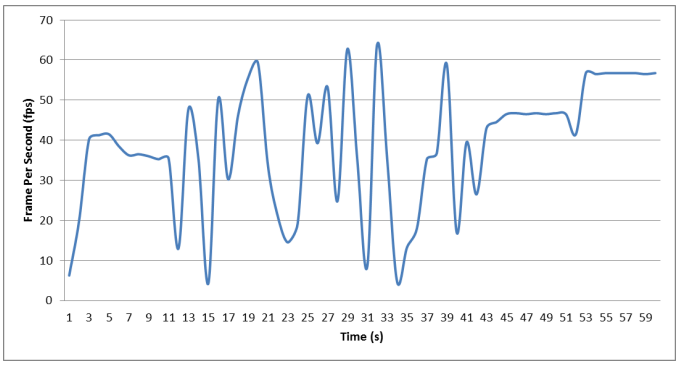

Figure 9. Performance of digital cluster when Linux fails

The above figures show that the Secure world maintains an average fps rate of 36.5, even when a fault occurs in the normal world, which is higher than the 30 fps minimum frame rate that is required to be able to have a smooth and fluid animation. The obtained results prove that there is a complete isolation between the two worlds without any world impacting the other's performance.

VIII. Conclusions and Future Work

In this paper we presented the "split display" architecture, a novel approach to tackle the issue of security and safety aware display sharing. Our architecture is based on VOSYSmonitor a highly privileged entity on top of which we integrated a Linux and an open source RTOS, functioning as our IVI and mission critical instrument cluster, respectively. Our architecture provides a strict isolation of the graphic applications running in the RTOS from the non-critical IVI running in the Linux. Although, our applications have been tested without graphical acceleration, our performance numbers still confirm our design decisions and clearly indicate the capability of our architecture to render mission critical information smoothly along with non-critical applications like a navigation map. We showed that even in the event of an unrecoverable failure in the normal world the mission critical instrument cluster is able to operate.

Our efforts included (but were not limited to) the design of two components called S-Bridge and N-Bridge to forward GPU rendering requests from the normal to the secure world, in the style of RPCs. Moreover, we ported several drivers critical for graphical rendering from Linux to FreeRTOS. We also developed and integrated a prototype of our architecture on an Renesas R-Car H3 evaluation board.

Since we implemented this prototype, there is no doubt about the feasibility of our approach.

 ISBN: 978-1-61208-626-2

In the future we want to focus our efforts on the following open points. First, sharing the graphic CPU rendering implementation with the FreeRTOS community. Second, improving the application to use a dedicated GPU and thus enabling graphical acceleration and increasing the overall FPS. Last, supporting the QT [28] graphical framework for the open source RTOS, to ease the creation of graphical applications.

IX. ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under the NGPaaS project (grant agreement No. 761557).

REFERENCES

[1] A. Zahir and P. Palmieri, "Osek/vdx-operating systems for automotive applications," in IEE Seminar on OSEK/VDX Open Systems in Automotive Networks (Ref. No. 1998/523), Nov 1998, pp. 4/1–418.

[2] D. John, "Osek/vdx history and structure," in IEE Seminar on OSEK/VDX Open Systems in Automotive Networks (Ref. No. 1998/523), Nov 1998, pp. 2/1–214.

[3] C. Hoffmann et al., "Osek/vdx network management," OSEK/VDX Open Systems in Automotive Networks (Ref. No. 1998/523), IEE Seminar, November 1998.

[4] A. Burns and R. Davis, "Mixed criticality systems - a review," Department of Computer Science, University of York, Tech. Rep, 2013, pp. 1–69.

[5] C. Percival, "Cache missing for fun and profit," BSDCan, 2005.

[6] R. Wojtczuk, J. Rutkowska, and A. Tereshkin, "Xen 0wning trilogy," Invisible Things Lab, 2008.

[7] P. Lucas, K. Chappuis, M. Paolino, N. Dagieu, and D. Raho, "Vosys-monitor, a low latency monitor layer for mixed-criticality systems on armv8-a," Euromicro Technical Committee on Real-Time Systems 2017, June 2017.

[8] T. Alves and D. Felton, "Trustzone: Integrated hardware and software security-enabling trusted computing in embedded systems (july 2004)."

[9] "ARM Security Technology - Building a Secure System using TrustZone Technology," Whitepaper, ARM Limited, April 2009.

[10] "Smc calling convention system software on arm platforms," Whitepaper, ARM Limited, November 2016.

[11] R. Mijat and A. Nightingale, "Virtualization is coming to a platform near you," ARM Limited.

[12] Armv8 processor architecture. [Online]. Available: https://community.arm.com [retrieved: April, 2018]

[13] P. Barham et al., "Xen and the art of virtualization," vol. 37, no. 5, 2003, pp. 164–177.

[14] Y. Haga, K. Imaeda, and M. Jibu, "Windows server 2008 r2 hyper-v server virtualization," Fujitsu Sci. Tech. J, vol. 47, no. 3, 2011, pp. 349–355.

[15] "Virtualbox," http://www.virtualbox.com, Oracle, March 2018.

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

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

[18] C. Lee, S. W. Kim, and C. Yoo, "Vadi: Gpu virtualization for an automotive platform," IEEE Transactions on Industrial Informatics, vol. 12, no. 1, 2016, pp. 277–290.

[19] C. Patsakis, K. Dellios, and M. Bouroche, "Towards a distributed secure in-vehicle communication architecture for modern vehicles," Computers & Security, vol. 40, 2014, pp. 60–74.

[20] S. Martínez-Fernández, C. P. Ayala, X. Franch, and E. Y. Nakagawa, "A survey on the benefits and drawbacks of autosar," in Proceedings of the First International Workshop on Automotive Software Architecture. ACM, 2015, pp. 19–26.

[21] F. Chao, S. He, J. Chong, R. B. Mrad, and L. Feng, "Development of a micromirror based laser vector scanning automotive hud," in Mechatronics and Automation (ICMA), 2011 International Conference on. IEEE, 2011, pp. 75–79.

[22] V. Milanovic, A. Kasturi, and V. Hachtel, "High brightness mems mirror based head-up display (hud) modules with wireless data streaming capability," vol. 9375, 2015, p. 93750A.

[23] Z. Qi et al., "Vgris: Virtualized gpu resource isolation and scheduling in cloud gaming," ACM Transactions on Architecture and Code Optimization (TACO), vol. 11, no. 2, 2014, p. 17.

[24] S. Gansel, S. Schnitzer, F. Dürr, K. Rothermel, and C. Maihöfer, "Towards virtualization concepts for novel automotive hmi systems," 2013, pp. 193–204.

[25] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. De Lara, "Vmm-independent graphics acceleration," 2007, pp. 33–43.

[26] FreeRTOS. Open source real time operating system. [Online]. Available: http://www.freertos.org [retrieved: January, 2003]

[27] Renesas. Rcar-h3. [Online]. Available: https://www.renesas.com/en-us/solutions/automotive/products/rcar-h3.html [retrieved: January, 2015]

[28] Qt. Cross-platform application framework. [Online]. Available: https://www.qt.io/ [retrieved: January, 1995]

 ISBN: 978-1-61208-626-2