CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

BEng Thesis

Exploring Xen/KVM in prototyping an automotive use-case

Orientação científica: David Pereira, Coorientação: Cláudio Maia

Johann Knorr

CISTER-TR-191212

Exploring Xen/KVM in prototyping an automotive use-case

Johann Knorr

CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

https://www.cister-labs.pt

Abstract

Due to increasing autonomy in vehicles, the automotive industry is searching for solutionswhich allow the reduction of engineering costs resulting from increasing hardware andsoftware requirements. To solve the problem, one of the solutions being studied is theutilization of virtualization to allow multiple systems to coexist on the same hardware,allowing the systems to function as if they are isolated, while reducing hardware costs. Usingvirtualization, the system becomes capable of hosting services of mixed criticality safely,reducing interference between the services through temporal and spatial isolation.In this project we analysed the Xen hypervisor in an effort to better comprehend how itsinclusion would impact the execution of a system with real-time requirements. After a brieflook at the current state of the art we describe how Xen behaves on the 32-bit ARMarchitecture, how it shares peripheral devices between virtual machines and how interdomaincommunication can be achieved. Following that, we present an example of how to deploy Xenon a Banana Pi SOC. After studying how best to establish a continuous connection betweendifferent virtual machines we tested our solution for message transfer jitter, deadline failurerate impact and deployed a prototype of an automotive application to verify how the systemwould behave in a real-world scenario.As planned, a system of mixed criticality was deployed on the hardware using the Xenhypervisor, PREEMPT RT was deployed on one virtual machine with real-time requirements,while Linux was deployed on a general-purpose virtual machine. The test showed that on theselected hardware Xen was unable to guarantee temporal isolation, showing significantperformance drops in deadline failures. The communication between virtual machines andthe deployment of the real-world application, however, were successful.

Exploring Xen/KVM in prototyping an automotive use-case

CISTER

2018 / 2019

1160996 Johann Knorr

Exploring Xen/KVM in prototyping an automotive use-case

CISTER

2018 / 2019

1160996 Johann Knorr

Licenciatura em Engenharia Informática

Setembro 2019

Orientadores ISEP: **David Pereira e Cláudio Maia**

To Ana Silva, for all the support and love you give me

Acknowledgments

Here I would like to thank my family for helping me become the person I am today and my girlfriend, Ana Silva, for all the support and love you give me.

I thank the friends I made here at ISEP, specifically Bruno Reis, Flávio Costa, Francisco Machado, João Cardoso, João Neves, Miguel Ramos and Rui Oliveira, for your companionship, for sharing your knowledge and for our mutual support during these three years.

Lastly, I want to thank my supervisors, Cláudio Maia and David Pereira and my colleagues at CISTER for guiding me and assisting me in the development of this project.

Johann Knorr

Abstract

Due to increasing autonomy in vehicles, the automotive industry is searching for solutions which allow the reduction of engineering costs resulting from increasing hardware and software requirements. To solve the problem, one of the solutions being studied is the utilization of virtualization to allow multiple systems to coexist on the same hardware, allowing the systems to function as if they are isolated, while reducing hardware costs. Using virtualization, the system becomes capable of hosting services of mixed criticality safely, reducing interference between the services through temporal and spatial isolation.

In this project we analysed the Xen hypervisor in an effort to better comprehend how its inclusion would impact the execution of a system with real-time requirements. After a brief look at the current state of the art we describe how Xen behaves on the 32-bit ARM architecture, how it shares peripheral devices between virtual machines and how interdomain communication can be achieved. Following that, we present an example of how to deploy Xen on a Banana Pi SOC. After studying how best to establish a continuous connection between different virtual machines we tested our solution for message transfer jitter, deadline failure rate impact and deployed a prototype of an automotive application to verify how the system would behave in a real-world scenario.

As planned, a system of mixed criticality was deployed on the hardware using the Xen hypervisor, PREEMPT RT was deployed on one virtual machine with real-time requirements, while Linux was deployed on a general-purpose virtual machine. The test showed that on the selected hardware Xen was unable to guarantee temporal isolation, showing significant performance drops in deadline failures. The communication between virtual machines and the deployment of the real-world application, however, were successful.

Keywords (Theme) **Virtualization, Automotive, Embedded Systems, Mixed-Criticality, Real-Time**

Keywords (Technologies) **Xen, Robot Operating System, 32-Bit ARM, Banana Pi, Linux**

Index

List of Images

List of Tables

Glossary

ARAMiS - Accidental Risk Assessment Methodology for Industries

AUTOSAR – AUTOmotive Open System ARchitecture

CISTER – Research Centre in Real-Time & Embedded Computing Systems

ECU – Electronic Control Unit

EDF – Earliest Deadline First

EE – Erika Enterprise

FP – Fixed Priority

GPOS – General Purpose Operating System

HVM – Hardware Virtual Machine

ISEP – Instituto Superior de Engenharia do Porto

ISO - International Organization for Standardization

KIT - Karlsruhe Institute of Technology

KVM – Kernel Virtual Machine

LEI – Licenciatura em Engenharia Informática

OIL – OSEK Implementation Language

OS – Operating System

OSEK/VDX – Offene Systeme und deren Schnittstellen fur die Elektronik im Kraftfahrzeug

PESTI – Projeto/EstágioR&D – Research and Development

ROS – Robot Operating System

RTOS – Real-Time Operating System

SOC – System-on-a-Chip

VDX - Vehicle Distributed eXecutive

VM – Virtual Machine

VMM – Virtual Machine Monitor

1 Introduction

The project entitled “Exploring Xen/KVM in prototyping an automotive use-case”, explored in this report, is part of the curricular unit named Projeto/Estágio (PESTI) of the Licenciatura em Engenharia Informática (LEI) of the Instituto Superior de Engenharia do Porto (ISEP). This project was proposed by the Research Centre in Real-Time & Embedded Computing Systems (CISTER), a Research and Development (R&D) centre devoted to the study of real-time and embedded systems, that is part of ISEP.

1.1 Context

Due to the advances in vehicle autonomy through computerization, the automotive industry is searching for solutions that can keep engineering costs low while increasing performance and guaranteeing the safety of critical systems within the vehicles [1]. The introduction of new features, including increased multimedia capabilities, advanced driving assistance and safety measures increase the costs of the required hardware and software. Several of these features require the use of embedded systems, i.e., computer systems that dynamically interact with the external world, these systems can influence but not control the environments they are embedded in [2]. These systems are also frequently used in scenarios where timeliness is a concern [3], meaning that the computation has not only to be logically correct but also complete within a certain time bound. These systems are then an example of real-time systems.

Examples of embedded systems in the automotive industry are electronic control units (ECUs) that are responsible for controlling one or more electronic systems of the vehicle, as for instance the vehicles emission rates or managing the information displayed on the dashboard [3]. In fact, in 2015 cars contained around 110 ECUs responsible for the emission and cooling systems, driving assistance features and engine operation [4]. Due to the demand in new features and electronic components to control them, the automotive industry is interested in decreasing the time to market while keeping the development and integration costs low. One possible solution to overcome this challenge is the adoption of software virtualization solutions.

Virtualization is the technique by which several systems can execute on the same hardware, while keeping their isolation and effectively behaving as if executing on different machines. Each of these systems is known as a Virtual Machine (VM) and their manager/monitor is known as the hypervisor. A hypervisor, or virtual machine monitor (VMM), is a tool that allows one or more VMs to execute on top of the hardware. It guarantees that either machine does not affect the other machines' performance and creates increased security between machines. Several different implementations of hypervisors exist on the market, both commercial and open source variants. Some of the more well-known hypervisors are Xen [5], KVM [6], Jailhouse [7], Microsoft Hyper-V [8], Oracle Virtual Box [9].

Through the use of hypervisors in the automotive use case, the system can become capable of supporting systems of mixed criticality on the same platform, recurring to temporal and spatial partitioning thereby reducing interference between systems and creating fault containment zones [10] [11]. For instance, in a mixed criticality system, a multimedia system with low criticality can coexist safely on the same hardware with an assisted driving system with high criticality, without the multimedia system representing a liability to the assisted driving system.

1.1.1 Framing in the context of the Vortex collaborative laboratory

The project being reported here is part of a task that is framed in the R&D activities that are part of the Vortex collaborative laboratory agenda. Vortex, which is led by Altran Portugal and that has as partners ISEP, INESC TEC, NovaLINCS, and Beta-i, is one of the Portuguese collaborative laboratories that were approved by the Portuguese Government and associated research and innovation partners, and that started their activities during the year of 2019.

The work performed in this project contributes specifically to activities that were selected by the Vortex governing body to tackle the challenges of the automotive industry in what concerns the usage of virtualization solutions to provide increased support for assisted driving features, with particular focus on visualization and having the Robot Operating System (ROS) as the integration framework.

1.2 Problem Description

This project aims to answer the question: *Can an open source hypervisor fit the needs of the automotive industry?* It is unknown how real-time systems will behave under the influence of Xen or KVM, two of the most used open source hypervisors, that is, how their timeliness varies and how their access to the system's resources can be guaranteed. The real-time system's requirements, namely predictability and deadline achievement can easily be jeopardized by excessive utilization of the systems resources.

If any VM occupies the system's CPU for too long or accesses memory that another VM was making use of, this can have adverse effects on the RTOS's timeliness, rendering it unusable if no measures are taken to prevent it. It is therefore important to confirm if said measures exist in the hypervisor and verify how well they behave under stress.

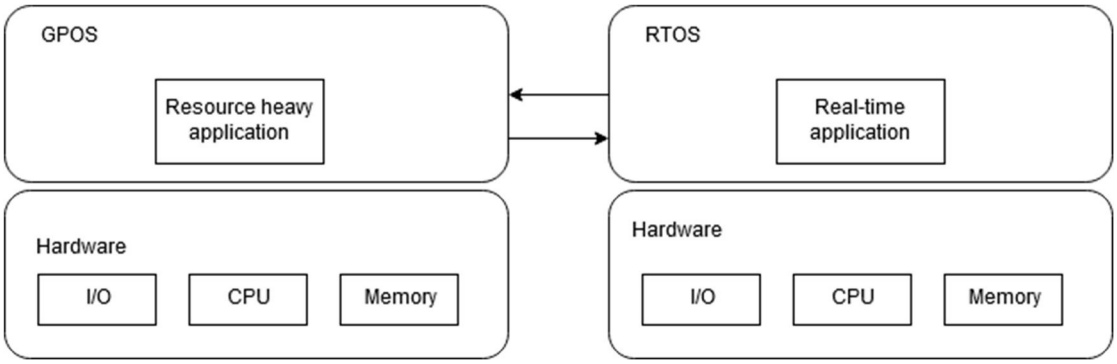

Figure 1 The current solution in the automotive sector

In the current approach used by the automotive industry, depicted in Figure 1, two computer systems are used. While their independence is guaranteed, for every different real-time service a new piece of hardware must be used, and as such it increases both material and energy costs, as well as complicating the deployment.

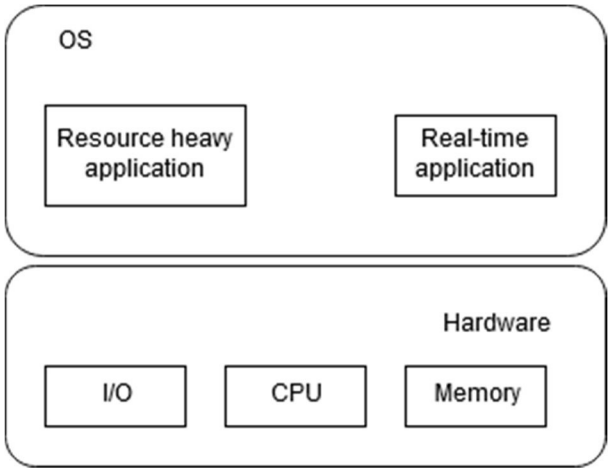

Figure 2 Solution without hypervisors

Should both systems be joined in the same hardware, without the use of a hypervisor, as depicted in Figure 2, the energy consumption and material costs are reduced, but a security vulnerability is exposed as there is no way to guarantee that the real-time application, responsible for safety-critical applications can meet the deadlines, since the systems resources will be heavily used by the other systems.

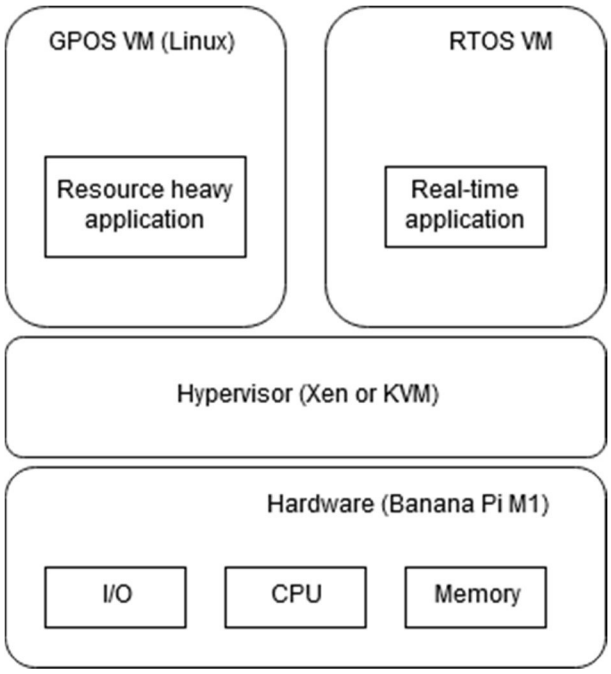

Figure 3 Proposed use case

In the proposed solution, which is shown Figure 3, the hypervisor solves the problem related to the access to the systems resources, by guaranteeing their separation throughout the VMs, making it so that one VM cannot access the resources allocated to another VM. By providing temporal and spatial isolation, the hypervisor enables the system to maintain the real-time application's timeliness.

1.2.1 Objectives

This project proposes to verify the feasibility of the usage of the Xen or the KVM hypervisor in an embedded environment. The project will study the impact of executing systems of mixed criticality in the chosen hypervisor and proceed to establish how different VMs are separated and how they can eventually communicate with each other.

1.2.2 Approach

In order to properly address the problem at hand, a preliminary study of the concept of hypervisor and virtualization must be performed. This research focuses on the characteristics and different types of hypervisors as well as differences in approaches to virtualization. The next step focuses on the specific characteristics and design of both the Xen and KVM hypervisors, allowing to identify advantages and disadvantages in their use and establishing which should be used in the proposed use-case.

Having chosen a hypervisor, the next milestone becomes its execution on a Banana Pi, a system on a chip (SOC) that serves as the embedded environment. For the correct execution of this step, several pieces of software have to be compiled from source: the bootloader, U-Boot, due to it being the de facto bootloader for embedded systems, the Linux kernel, serving as the hypervisors host OS and the chosen hypervisor for this project.

Once the system's initial configuration is finished, two VMs must be created. The installation of the first machine contains a general-purpose operating system (GPOS), while the second contains a real-time operating system (RTOS).

After both virtual machines are working correctly the system has to undergo a series of tests to establish how the VMs affect each other's functioning.

1.2.3 Contributions

Through this project, it is expected that Xen or KVM's place in the automotive sector is either corroborated or rejected as a viable hypervisor for the solution through virtualization. This work highlights the processes by which deployment can be achieved and provides benchmarks of the chosen hypervisor's performance in the selected use case. Using a prototype application from the Vortex Colab, we will test the communication between two VMs and verify the behaviour of the VMM in a real-world scenario. By the end we expect to know if Xen or KVM can safely support a system of mixed criticality, guarantee the timely execution of safety-critical tasks and reduce the energy consumption and deployment efforts.

1.2.4 Planning

The planning phase is of utmost importance for the success of any project, even more so for R&D projects, where it is unknown if the final objective can be achieved. So being, having well traced and delimited minor goals, and respecting them to all possible extent, helps in maximizing the efficiency of all involved parties.

For this project, several major milestones were identified, which are represented in bold in the table below. Minor goals are identified when the complexity of the milestone warrants it.

Table 1 The project's tasks and their expected duration

Task	Time
Hypervisor deployment	**42d**
Study on hypervisors and their use in embedded systems	14d
Perform KVM experiments	14d
Perform Xen experiments	14d
Write Paper on Hypervisors in Embedded Systems (Optional)	**21d**
Research on related works	14d
Writing of the paper	7d
Virtual Machine Deployment	**10d**
Compilation and deployment of the GPOS VM	3d

Compilation and deployment of the RTOS VM	7d
Write progress report	**2d**
M1 – Progress Report	
Development of a real-time application	**7d**
Development of the prototype	**22d**
Study of the interdomain interference impact	**14d**
Write final report	**22d**
M2 – Report Delivery	

2 State of the Art

2.1 Related Work

Several other projects have studied the requirements, advantages and disadvantages of integrating hypervisors in the automotive realm. In the following sections, some of the previously made studies are presented and their findings discussed.

2.1.1 MultiPARTES: Multicore virtualization for Mixed-criticality Systems

The MultiPARTES [12] project's goal was to promote "mixed-criticality integration for embedded systems based on virtualization". The MULTIPARTES authors discussed the certification requirements in the automotive industry, specifically ISO/IEC 61508 and ISO 26262, and the difficulty of having software appraised due to conservative assumptions of the certification authorities that also lead to poor usage of the system's potential. The authors then list a set of requirements the system should display in order to qualify for independent certification. First, the partitions should be isolated both spatially and temporarily. Next, it should be possible to analyse partitions in an isolated fashion and methods for said analysis must be provided. Lastly, the scheduling of one partition must not impact the scheduling of another.

The team also presented the challenges that arose from designing a system with virtualization capabilities, as each application had to be assigned to partitions and dependability and security requirements had to be taken into consideration [12].

The MultiPARTES team brought up a series of important concerns, yet their project differs from the one presented here since it does not cover the communication or partitioning aspects, nor offers an implementation of their proposed methodology.

2.1.2 Integrating Linux and the real-time ERIKA OS through the Xen hypervisor

In this paper, the authors test the compatibility between ERIKA Enterprise [13] , an RTOS, and the Xen hypervisor on an embedded environment. The authors study a similar use case to the one proposed in this project with the execution of an RTOS and a GPOS on the same hardware. They alert the reader for the risks of mixed criticality without isolation guarantees, namely failures in the GPOS, which might negatively affect the execution of safety-critical tasks on the

RTOS and malfunctions on the RTOS which, through errors in the file descriptors, might pollute the Linux memory area.

The authors proceed to describe their approach to communication between partitions, having the communication starting from Erika Enterprise's partition as to guarantee that the communication does not negatively affect the scheduling of the other real-time tasks, as the RTOS is responsible for attributing the priority it deems fit for the communication.

Finally, the team lists the limitations and drawbacks from their implementation, specifically the fact that whenever the RTOS needs to signal the GPOS it has to send first a hypercall to the Xen hypervisor, effectively creating a bottleneck for the communication and presenting a risk to the RTOS proper functioning, as the machine could stay in hypervisor mode for a period of time that would compromise the deadline of real-time operations. The authors also argue that isolation between the two partitions should be improved as to limit interference between them

Nonetheless, the authors show that it is possible to execute Erika Enterprise on top of the Xen hypervisor and that communication is possible, suggesting that more work should be put into that research [14].

As the authors had suggested, more research must be done on the subject, the project described relied on outdated versions of both Xen and Erika Enterprise, and both have come a long way since the release of this experiment.

2.1.3 ARAMiS II

The Accidental Risk Assessment Methodology for Industries (ARAMiS) is a project funded by the German government and coordinated by the Karlsruhe Institute of Technology (KIT) aiming to promote research on safety-critical application in the automotive and avionics industry. It is a follow up to the original ARAMiS project, which ended in 2015. The project mostly focuses on developing methodologies, tools and architectures to further facilitate the development and deployment of safety-critical applications on multicore embedded systems [15]. Several use cases are presented by ARAMiS, ranging from automotive to avionics and industrial automation [16].

2.2 Existing Technologies

2.2.1 Hypervisors

A hypervisor, also known as VMM, is a piece of software that allows a VM to execute on top of the hardware. To be classified as a VMM it must follow three basic characteristics. Firstly, it must allow for programs to be run as if they were being run right on top of the original machine. Secondly, all programs run on the VM must have a minuscule impact on their performance, compared to the original machine. Lastly, the VMM must be in control of the entirety of the systems resources, meaning it is in control of which VM can use which resources and to what extent [17].

Commonly, hypervisors are separated into two categories. The first category, referred to as type 1, bare-metal or native hypervisor, encompasses the hypervisors that execute directly on top of the hardware. These hypervisors must include code to perform the scheduling and resource allocation. The second category, called type 2 or hosted hypervisor, runs on top of an already existing OS [18]. It should be noted that type 2 hypervisors suffer from considerable overhead due to the existing OS executing underneath. The architectural differences between type 1 and type 2 hypervisors are depicted in Figure 4.

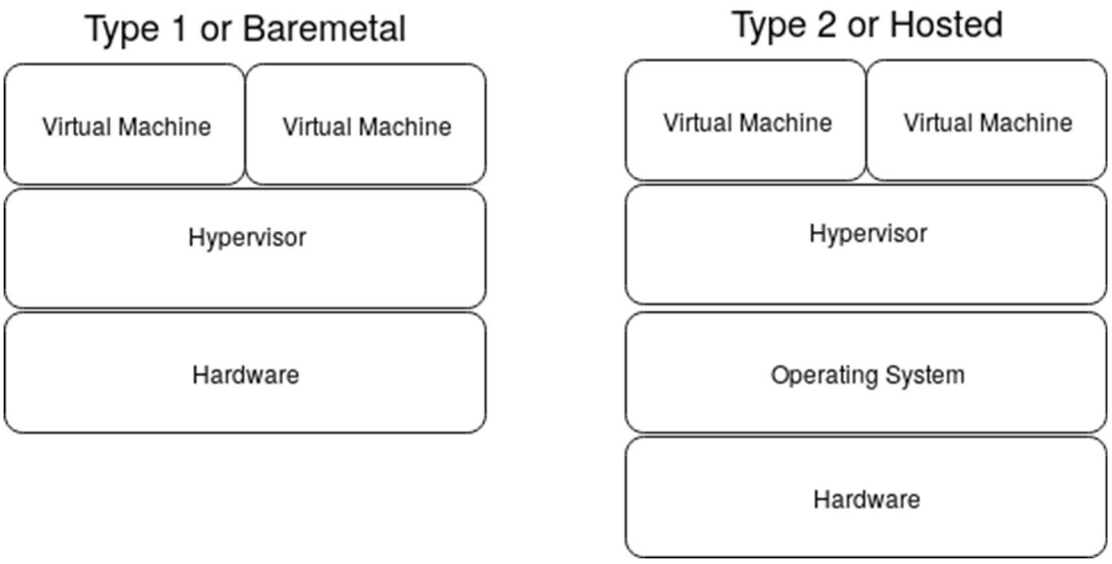

Figure 4 VMM Types Architecture

When discussing virtualization approaches, two fundamental techniques can be identified, full virtualization and para-virtualization (PV), and some approaches that make use of characteristics from both.

Full virtualization consists of the creation of a fully virtualized system through the abstraction of the physical system. The OS is not aware of the virtualization and will, therefore, run just as it would on the physical system [19].

Para-virtualization, on the other hand, involves modifying the OS, by exposing a virtual architecture, so that it is aware of the ongoing virtualization, making it more efficient, yet breaking backwards compatibility with existing code for said OS. Para-virtualized systems usually add instructions, devices or registers to the architecture, improving overall performance [20].

2.2.1.1 Xen

Xen first came to be in 2003 as a platform for the instantiation of several smaller VMs, each running on its own copy of an OS, by partitioning the underlying hardware to support their concurrent execution, isolating the VMs from one another, guaranteeing both security and performance identical to the underlying physical system. The original Xen hypervisor functioned only via para-virtualization, as the hardware at the time made full virtualization very costly from a performance perspective. The x86 architecture did not include virtualization capabilities [21]. With the appearance of hardware virtualization extensions, that facilitated full virtualization, Xen incorporated full virtualization resulting in a feature known as hardware virtual machine (HVM), making it so that unmodified OSs could be run on top of the Xen hypervisor. Currently, the Xen team is working on merging the PV working mode a working mode and the fully virtualized mode to “simplify the interface between OSs with Xen Support and the Xen Hypervisor” [22], an approach dubbed as PVH, as a mixture of PV and HVM.

Xen is a traditional type 1 hypervisor as it is in direct communication with the hardware. On a system boot, Xen automatically creates a privileged guest, named Domain-0. This guest is responsible for providing device drivers and running management applications. As with the unprivileged guests (Domain-U), Domain-0 possesses its own virtual CPU and virtual memory. Since Domain-0 is responsible for all the I/O tasks, the Domain-U guests only communicate

indirectly with peripheral devices, delegating those tasks to Domain-0 [23]. The basic architecture of a Xen system is displayed in Figure 5.

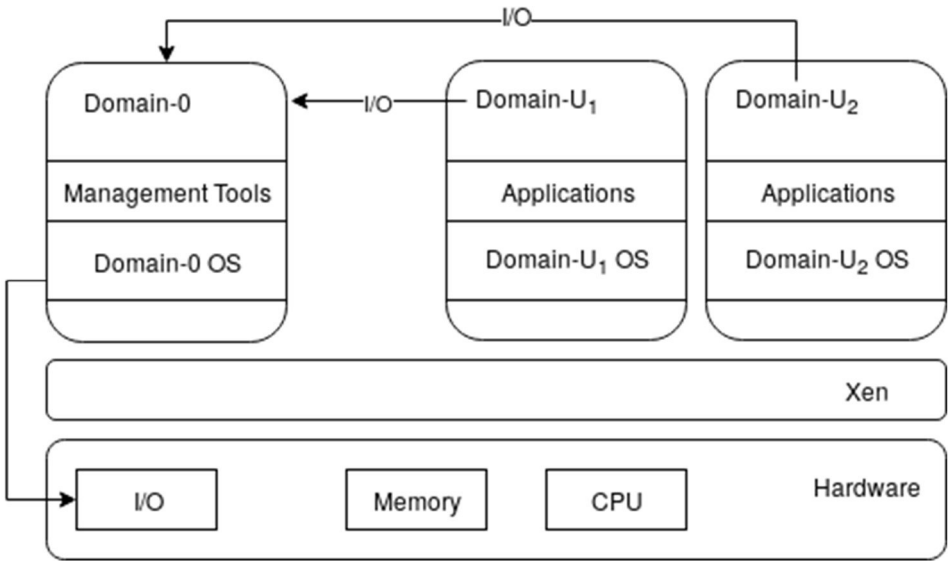

Figure 5 Xen Architecture

2.2.1.2 KVM

Having made its debut in 2006, the Kernel-based Virtual Machine (KVM) seeks to add hypervisor capabilities to the mainline Linux kernel. Being embedded into the Linux Kernel, as a module, KVM does not need to implement any functions to perform scheduling or resource allocation, like a traditional type 1 hypervisor, making use of the already existing capabilities of the Linux Kernel. With KVM, a virtual machine is, from the perspective of the system, identical to any other process [24]. To achieve virtualization, KVM makes use of hardware extensions on the CPU and virtualises I/O devices through Virtio, a technology similar to Xen's para-virtualized device drivers, enabling faster I/O operations [25].

KVM is often labelled a type 2 hypervisor. At first glance, it seems that KVM is running on top of the OS, as well as the fact that it does not deal with scheduling and memory management, like other hosted hypervisors. Despite those circumstances, KVM effectively transforms the OS into a type 1 hypervisor, as it is a part of the OS and not software running on top [26]. The functioning of KVM is shown in Figure 6.

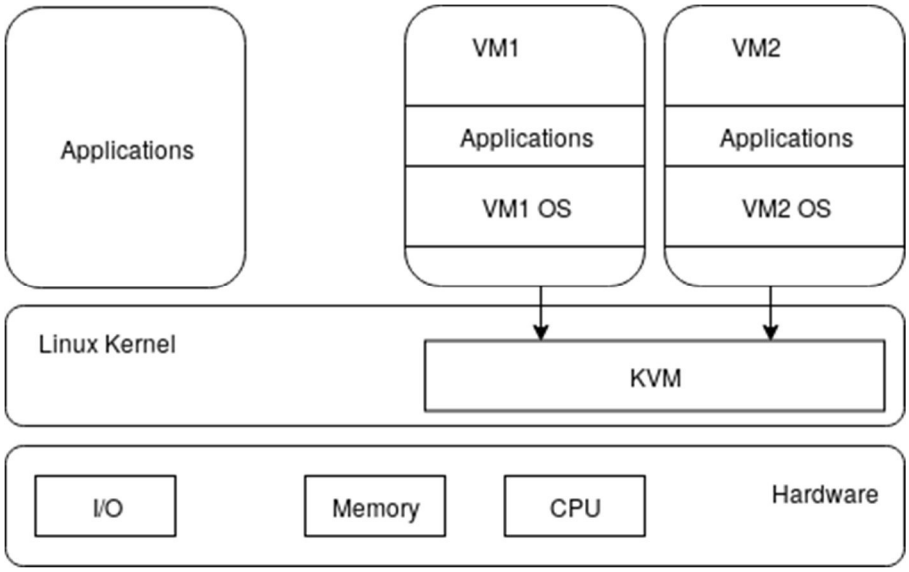

Figure 6 KVM Architecture

Processes in Linux have commonly two execution modes, kernel and user mode. With KVM, a third mode, named guest mode, joins user mode and kernel mode. While running in guest mode, the process has all the same privileges as any process in the regular operating modes, with the addition that it can request that certain accesses to registers or instructions be trapped. Under KVM, a virtual machine will run most of the time in guest mode, changing to kernel mode when it must deal with events like external interrupts and changing to user mode when requiring an I/O operation [24].

2.2.1.3 Jailhouse

Jailhouse is a partitioning hypervisor released in 2013. Isolation is achieved via hardware-assisted virtualization [27]. Due to its alternative approach to virtualization, Jailhouse offers no scheduler, as a CPU cannot be shared between guests, called cells in the Jailhouse jargon. Jailhouse is launched through Linux, the original system becomes then the root-cell, comparable to Xen's Dom-0. The root-cell is responsible for distributing resources to the other cells in a process called “shrinking”. In this process the root cell cedes control over some of its CPU, memory or peripheral devices, and allocates them to the new cells [28].

2.2.2 Real-Time Computing

A real-time computer system is a system which correctness is not measured only in its logically correct computations but also in the timely arrival of said computations. It should be noted that the aim is not to minimize the average response time but to meet the deadline of each task [29].

When considering real-time deadlines, three types can be identified, as shown in Table 2. Hard deadlines are those where if the results are produced after the deadline, consequences might be catastrophic. Soft deadlines are those where if the results are produced after the deadline the system will have degradation in performance, but the result is still of some use to the system. Firm deadlines are those where if the results are produced after the deadline the results are simply discarded [30] [31].

Table 2 Impact of missed deadlines on real-time systems

Type	Usefulness of missed deadlines	Consequences of missed deadlines
Hard Real-Time	Not useful	Catastrophic
Soft Real-Time	Useful	Performance Degradation
Firm Real-Time	Not useful	Performance Degradation

To improve the chances of deadlines being met, every RTOS must implement a scheduling algorithm which enables it to meet the established deadlines for each application. These real-time scheduling algorithms select a method by which the next task is chosen, among others, some are:

- Shortest job first
- Static priority
- Earliest deadline first

In the case of an eventual system overload, the algorithm must also determine a strategy to solve the issue, which usually is achieved through either reducing or eliminating the execution of tasks of lower priority [32].

Therefore, a RTOS is a system capable of executing real-time applications while providing the aforementioned characteristics.

2.2.2.1 ERIKA Enterprise

Erika Enterprise (EE) is an OSEK/VDX and AUTOSAR compliant RTOS designed for single-chip microcontrollers. EE provides a multithreading environment supporting real-time scheduling algorithms and stack sharing [33]. The scheduling algorithms supported by the EE kernel include Fixed Priority (FP), Earliest Deadline First (EDF) and Contract-based scheduling (FRSH). This way EE can offer both traditional and innovative scheduling algorithms out of the box.

EE uses the OSEK Implementation Language (OIL), created by the OSEK/VDX consortium, to define RTOS objects to be used by the application. Due to the complexity of the language and to facilitate configuration, EE comes bundled with RT-Druid, thus providing a visual interface capable of generating the needed configuration files [34].

OSEK/VDX

"Offene Systeme und deren Schnittstellen fur die Elektronik im Kraftfahrzeug" (OSEK) is a partnership of the German automotive industry whose goal is to standardise the architecture for distributed control units in the automotive sector. OSEK merged with the Vehicle Distributed eXecutive (VDX) project in 1994, transforming the official name to OSEK/VDX [35]. The partnership is responsible for many standards, which have since been incorporated by the International Organization for Standardization (ISO).

AUTOSAR

Founded in 2003, AUTOSAR (AUTOmotive Open System ARchitecture) is a consortium, created by major automotive manufacturers, aiming to standardise the software architecture for automotive embedded systems, improving safety and reducing energy consumption and software reuse. AUTOSAR specifies several standards and guidelines that are highly regarded in the industry [36] [37].

2.2.2.2 FREERTOS

FreeRTOS is an RTOS designed for embedded devices, supporting over 35 architectures [38]. FreeRTOS supports both pre-emptive and cooperative scheduling and features FP and round-robin scheduling algorithms.

FreeRTOS uses C header files for configuration, therefore every application must have a FreeRTOSConfig.h file. This file specifies the kernel's characteristics for that application and is therefore tied to the application instead of the kernel [39].

2.2.2.3 SAFERTOS

Having been created as a fork of the FreeRTOS project, SafeRTOS inherited most of the features from FreeRTOS. In addition, SafeRTOS puts a high value on deterministic priority-based scheduling, This, in turn, brought it the certification of TÜV SÜD, having been certified with IEC 61508-3 SIL and ISO 26262 ASIL [40].

2.2.2.4 PREEMPT_RT

The PREEMPT_RT patch for the Linux kernel aims increase the preemption on said kernel by reducing the code that is non-preemptible and changing the least code necessary to provide real-time capabilities [41]. By transforming regular Linux into an RTOS, any program that runs on the stock kernel also runs on PREEMPT_RT, thus providing combability with already existing software and providing a familiar environment for further development [42].

2.2.3 **Das U-Boot**

The "Universal Bootloader", also known as "Das U-Boot" or just U-Boot, is an open-source bootloader created for embedded systems. It originated from a similar project called PPCBoot in 2002 and is used as the default bootloader by a multitude of board vendors. The bootloader supports a variety of architectures, both ARM and x86 [43] [44]. A bootloader in a computer system is the application which loads the OS's kernel into memory and proceeds to its execution. The bootloader generally allows for a selection of the kernel to execute, in case multiple kernels are present on the system. On embedded systems, a bootloader is also responsible for programming the system's memory controllers, initializing processor caches, managing hardware peripherals and enabling network support. The responsibilities of the bootloader on embedded systems are increased due to a lack of extensive firmware, generally found on regular desktops or server systems [45].

2.3 Conclusion

After analysing the available technologies and ranking them in order of suitability for the project at hand, it was decided that Xen would be the most suitable hypervisor to use. Xen

has more extensive documentation and reports of projects of similar nature on this hypervisor, when compared with KVM.

Regarding the choice of the RTOS, Erika Enterprise proved to be the most suitable for this project, due to the existing certification and previous deployment on the Xen hypervisor. However, due to developments on the hypervisor since that deployment, Xen no longer supports Erika Enterprise, and although efforts were made to correct this, they proved to be fruitless.

Knowing that Linux fully supports Xen, PREEMPT_RT was chosen as the RTOS, despite being a soft real-time OS it serves as proof of concept for this project.

3 Xen

In this chapter, we provide a comprehensive explanation of Xen's internal mechanisms. Here we will draw the distinctions between Xen's implementation on the ARM architecture versus its x86 counterpart, listing the most important differences, followed by an explanation on how peripheral sharing and interdomain communication are achieved and the steps required to deploy the hypervisor and, finally, how to set up an unprivileged domain.

3.1 Xen on ARM

As seen in Chapter 2, Xen is a type 1 hypervisor. Xen manages all the accesses to the system's resources, except for the peripheral devices, which are controlled by Domain-0, as well as the scheduling of all the different tasks. This description of Xen is true for both the x86 and ARM architectures. There are, however, a few key differences in how Xen operates on ARM.

While the Xen project exists since 2003, it was not until 2013 that the ARM architecture became supported. Facing the challenge of porting a 10-year-old project to a new architecture, the Xen team seized the opportunity to remove any unnecessary code. Most notably, the ARM version of Xen has no support for emulation, meaning that all references to emulation have been substituted in favour of virtualization extensions present in the hardware or paravirtualized interfaces, in the case of input/output devices.

On the x86 architecture, Xen supports two kinds of guests, paravirtualized guests, which must be modified to execute on top of the hypervisor and fully virtualized guests, which make use of emulation, do not need modification but have reduced performance. On ARM, due to the lack of emulation, Xen only supports one type of guest, usually referred to as ARM Guest.

This guest type functions as an amalgamate of the previous two types, managing to avoid modifying the overlying OS without suffering performance loss, which is achieved by making use of the virtualization extensions present in the hardware to avoid the need for emulation, thus not suffering the performance loss characteristic to the latter. The only things Xen requires of the OS are custom device drivers for the peripheral devices [46].

3.2 Peripheral Sharing

While it is the purpose of the hypervisor to guarantee the separation between the different VMs both spatially and temporally, it may sometimes be necessary to allow more than one machine access to the same piece of hardware. For instance, to establish a working internet connection, the VMs must access, and thus share, the network adapter. Without the possibility of peripheral sharing, the machines would either not have access to the service or the hardware would need to include one device for each domain, which would inevitably raise the hardware cost of the system.

To control which domains, have access to which devices, Xen allows assigning the different devices between VMs. By default, all available devices are assigned to Domain-0, through Domain-0 the devices can then be distributed to different unprivileged domains, according to their needs. Should more than one machine need access to the device though, Xen offers the solution through the use of split device drivers.

The split device drivers achieve their function by, as the name suggests, being distributed between two domains. One domain, the one which needs access to the device but does not control it, uses the FrontendDriver to establish a connection to the domain which holds the device. The device controlling domain, in turn, uses the BackendDriver to relay the previous domains request to the device in question. Thus, the BackendDriver serves as a broker between all the FrontendDrivers requests and the shared device. This behaviour is depicted in Figure 7.

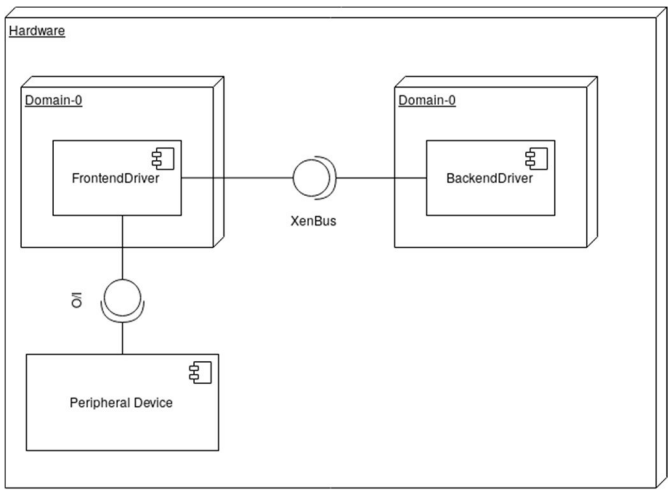

Figure 7 Peripheral Sharing Deployment Diagram

3.3 Inter domain communication

In order to determine the medium by which the domains communicate, a communication mechanism must be chosen. Xen offers two immediate solutions for interdomain communication. One solution is based on memory sharing between the domains, while the other solution is to delegate the communication to the OS.

In terms of raw performance, it would undoubtedly be beneficial to use a mechanism based on shared memory. Xen supports memory sharing with grant tables, where one VM offers selected memory pages to be accessed by a different VM. Alas, Xen only supports this feature on fully virtualized machines executing in an x86 environment [42], rendering this solution unpractical at the current time.

Making use of Xen's ability to share peripherals, the communication can be handled on the OS level and, in this case, established through TCP. In order to allow each domain to appear in the network as its own device, Xen can be configured to use bridging. Using this mechanism, a network bridge can be created in the domain that controls the network device, usually Domain-0, where all the other domains can connect to, thus becoming available in the network.

3.4 Xen Installation

To deploy Xen on an ARM based system, a few steps must be taken. In this section we present the necessary steps to set up Xen and Domain-0 on a Banana Pi SOC. For this process, one should guarantee all the necessary toolchains are present.

```
$ sudo apt install gcc-arm-linux-gnueabihf bc build-essential git
device-tree-compiler ncurses-dev
```

Having acquired the necessary toolchains, we can start by collecting all the required source files for the bootloader, Xen, the Linux Kernel and the latest Linaro release based on Ubuntu.

```
$ mkdir workspace
$ cd workspace
$ git clone git://git.denx.de/u-boot.git
$ cd u-boot
$ git checkout tags/v2019.01
$ cd ..
$ git clone
git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
```

```
$ mv linux-stable linux
$ cd linux
$ git checkout tags/v4.19.37
$ cd ..
$ git clone git://xenbits.xen.org/xen.git
$ cd xen
$ git checkout tags/RELEASE-4.11.0
$ wget
http://releases.linaro.org/ubuntu/images/developer/15.12/linaro-
vivid-developer-20151215-714.tar.gz
```

Having acquired all the necessary files, the next step becomes compiling the downloaded source codes and creating any necessary configuration files

While building the system from the ground up, the first program that needs to be compiled is the bootloader. U-Boot already includes a default configuration for the Banana Pi, thus removing any unnecessary complexity to this process. The boot.src file created in this step will be used to configure the bootloader and designate the program to load on boot as well as the arguments to pass to said program, in this case Xen [47].

```
$ cd u-boot
$ make CROSS_COMPILE=arm-linux-gnueabihf- Bananapi_config
$ make CROSS_COMPILE=arm-linux-gnueabihf -j ${nproc}
$ mkdir boot
$ cd boot
$ echo '
setenv fdt_addr 0x7ec00000
setenv fdt_high 0xffffffff
setenv kernel_addr_r 0x6ee00000
setenv xen_addr_r 0x6ea00000
fatload mmc 0 ${xen_addr_r} /xen
setenv bootargs "console=dtuart dtuart=serial0 dom0_mem=512M"
fatload mmc 0 ${fdt_addr} /sun7i-a20-bananapi.dtb
fdt addr ${fdt_addr} 0x40000
fdt resize
fdt chosen
fdt set /chosen \#address-cells <1>
fdt set /chosen \#size-cells <1>
fatload mmc 0 ${kernel_addr_r} /zImage
fdt mknod /chosen module@0
fdt set /chosen/module@0 compatible "xen,linux-zimage"
"xen,multiboot-module"
fdt set /chosen/module@0 reg <${kernel_addr_r} 0x${filesize} >
fdt set /chosen/module@0 bootargs "console=hvc0 ro
root=/dev/mmcblk0p2 rootwait clk_ignore_unused"
bootz ${xen_addr_r} - ${fdt_addr}
' > boot.cmd
$ mkimage -C none -A arm -T script -d "boot.cmd" "boot.scr"
```

```
$ cd ../../
```

The following program to be compiled is the hypervisor itself, besides cross compiling the code to be executable on ARM one must also specify the target architecture, as we have seen that Xen for x86, and ARM have different sources.

```
$ cd xen
$ make dist-xen XEN_TARGET_ARCH=arm32 CROSS_COMPILE=arm-linux-
gnueabihf- -j {nproc}
$ cd ..
```

Finally, we have to compile the Linux kernel to include all the necessary modules. Using the menuconfig option we gain access to an interface that facilitates the selection of modules as well as providing categorization and a description for each [48].

```
$ cd linux
$make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- multi_v7_defconfig
$make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
```

System Type -> Support for the Large Physical Address Extension
Kernel Features -> Xen guest support on ARM
Networking Support -> Networking options -> 802.1(d) Ethernet Bridging
Device Drivers -> Multiple devices driver support (RAID and LVM) -> Device mapper support
Device Drivers -> Block Devices -> Xen block-device backend driver
Device Drivers -> Network device support -> Xen backend network device
Virtualization
Virtualization -> Host kernel Accelerator for virtio net

```
$make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage dtbs
modules -j {nproc}
```

Having all the binaries ready we must prepare an SD card to move them too. We'll divide the card into two partitions the first partition will be our boot partition and will hold the u-boot and Xen binaries, the Kernel and the device tree blob, which we get through U-Boot. The second partition will hold the files from the Linaro release as well as the modules selected during the Kernel compilation phase.

```
$ sudo blockdev --rereadpt /dev/mmcblk0
$ sudo sfdisk /dev/mmcblk0 <<EOT
1M,16M,c
,4G,L
,,8e
EOT
$ sudo mkfs.vfat /dev/mmcblk0p1
$ sudo mkfs.ext4 /dev/mmcblk0p2
```

```
$ sudo dd if=u-boot/u-boot-sunxi-with-spl.bin of=/dev/mmcblk0
bs=1024 seek=8
$ sudo mkdir /mnt/mmc1
$ sudo mkdir /mnt/mmc2
$ sudo mount /dev/mmcblk0p1 /mnt/mmc1
$ sudo mount /dev/mmcblk0p2 /mnt/mmc2
$ sudo cp u-boot/boot/boot.scr /mnt/mmc1/
$ sudo cp linux/arch/arm/boot/zImage /mnt/mmc1/
$ sudo cp linux/arch/arm/boot/dts/sun7i-a20-bananapi.dtb /mnt/mmc1
$ sudo cp xen/xen/xen /mnt/mmc1/
$ sudo tar zxf linaro-vivid-developer-20151215-714.tar.gz -C
/mnt/mmc2 --strip 1
$ sync
$ sudo cp /mnt/mmc1/zImage /mnt/mmc2/root/zImage
$ sudo cp xen /mnt/mmc2/root/xen
$ sudo umount /mnt/mmc1
$ cd linux
$ sudo make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
INSTALL_MOD_PATH=/mnt/mmc2 modules_install
$ echo' /dev/mmcblk0p2 / ext4 rw,defaults 0 0 none /tmp tmpfs
defaults 0 0 ' > /mnt/mmc2/etc/fstab
$ touch /mnt/mmc2/machine-id
$ sudo umount /mnt/mmc2
```

From here we insert the card into the Banana Pi's card reader and boot it. All the following steps are executed on the Banana Pi either with a monitor and keyboard or a serial connection with an UART cable.

```
$ echo 'deb http://ports.ubuntu.com/ubuntu-ports/ xenial main
universe restricted multiverse
deb-src http://ports.ubuntu.com/ubuntu-ports/ xenial main universe
restricted multiverse' > /etc/apt/sources.list
$ echo' auto lo
iface lo inet loopback
auto eth0
iface eth0 inet dhcp
> /etc/network/interfaces
$ service networking restart
$ sudo apt update
$ sudo apt dist-upgrade
$ cd /root/xen
$ make clean
$ apt install build-essential
$ apt build-dep xen
$ ./configure
$ make install
$ echo '/usr/local/lib' >> /etc/ld.so.conf
$ ldconfig
$ sudo apt install lvm2
```

```
$ sudo apt install debootstrap
$ sudo apt install openssh-server
$ apt-get install bridge-utils
$ brctl addbr br0
$ echo '
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet manual
auto br0
iface br0 inet dhcp
  bridge_ports eth0' > /etc/network/interfaces
$ service networking restart
$ echo 'none /proc/xen xenfs defaults 0 0' | sudo tee -a /etc/fstab
$ /etc/init.d/xencommons start
```

This concludes the necessary steps to set up Xen and Domain-0. The system is now set up to host other VMs and can be accessed remotely via SSH. The system is also set up to be able to share the network adapter with any other domain since we will be using the network to establish a connection between the domains.

3.5 Domain Setup

Now that the system is set up, we can engage in the process of creating our unprivileged domains. These domains, just like Domain-0, require a Kernel and a partition with the files from any Linux distribution. Unlike Domain-0, however, these domains also require a configuration file, which will, amongst other things, list the amount of memory allocated to the domain, the CPU cores it can access, and its name. In this section we will present an example on how to set up a domain on the previously configured system.

Firstly, we will be creating the partition to which the domain will have access. For this purpose, we will be using a logical volume created through LVM.

```
$ lvm
      lvm> pvcreate /dev/mmcblk0p3
      lvm> vgcreate vg0 /dev/mmcblk0p3
      lvm> vcreate -L 2G vg0 --name linux-guest-1
      lvm> exit
$ mkfs.ext4 /dev/vg0/linux-guest-1
```

Next, we will be mounting the created partition and bootstrap our chosen distribution on it, meaning we will gather all the files present in the distribution and deploy them on the partition.

```
$ mount /dev/vg0/linux-guest-1 /mnt
$ debootstrap --arch armhf xenial /mnt http://ports.ubuntu.com/
```

Finally, we will access the partition and prepare it for its use. We will be configuring the network as well as the file systems table and installing an SSH server, so the domain can be accessed remotely. Furthermore, we need to attribute a password to the root user and change the hostname, as it will default to the hostname of Domain-0.

```
$ mount /dev/vg0/linux-guest-1 /mnt
$ chroot /mnt
$ passwd
$ echo 'domU' > /etc/hostname
$ echo 'auto eth0
iface eth0 inet dhcp' > /etc/network/interfaces
$ apt install openssh-server
$ echo '/dev/xvda / ext4 rw,relatime,data=ordered 0 1' >> /etc/fstab
$ exit
$ umount /mnt
```

Having prepared the partition, and using the same kernel as Domain-0, this VM only requires a configuration file.

```
$ echo 'kernel = "/root/zImage"
memory = 256
name = "domU1"
vcpu = 2
serial = "pty"
disk = [ "phy:/dev/vg0/linux-guest-1,xvda,w" ]
vif = ["bridge=br0"]
extra = "console=hvc0 xencons=tty root=/dev/xvda' > domU.cfg
```

In this example configuration file, we attributed 256 mb of memory to the domain, named it “domU1”, specified the partition it is able to access, granted it access to the network through the network bridge, and gave it access to all 2 of the systems CPU cores. Xen allows for more configuration options, for instance, we can attribute specific cores of the CPU to each domain. The virtual CPU is identified by an index and can be retrieved through the tools stack using:

```
$ xl vcpu-list
```

Considering that our system’s CPU only possesses two cores, we considered it to be beneficial to allow the system to use all the available cores, as tests showed a higher responsiveness for each VM when the cores were shared between our three VMs, the three VM setup is further analysed in section 4.3.

Finally, to start the VM we use the xl toolstack to create the domain. The created domain can then be accessed through domain-0’s console or via an SSH connection. In this example we

will be accessing the new domain at its creation through the domain-0 console with the “-c” option, to resume control of the domain-0 we press “CRTL +]” to escape the new domain’s console.

```
$ xl create domU.cfg -c
```

This finalizes the creation of our VM. The machine is now running and ready to be given the use it was created for.

4 Analysis and Design

Having acquired enough information on the hypervisor's functioning, we can now progress to the analysis of the use case. Here we will offer a brief explanation on the proposed use case, the requirements to fulfil as well as the limitations we incur. For each requisite we will present different solutions and the reasoning that led to the adoption of one solution over the other.

4.1 Use-case

The system must allow one VM, hosting a real-time OS, to transmit previously gathered sensor data to another VM, hosting a general-purpose OS, so that it can proceed to the visualization of the transmitted data.

Requirements:

- Create one VM with a GPOS;
- Create one VM with an RTOS;
- Establish a continuous connection between the GPOS VM and the RTOS VM;

Limitations:

- The chosen GPOS and RTOS must be compatible with Xen;
- The chosen GPOS and RTOS must execute on a 32-bit ARM environment;
- The chosen GPOS and RTOS must be capable of handling the interdomain communication.

4.2 Interdomain Communication

Facing the problem of interdomain communication and the limitations Xen imposes, one must determine how the connection will be established. Being barred of using a shared memory mechanism, the next-best option is to establish a communication channel through TCP. The next hurdle to overcome is the discovery of the other domain, as each domain is unaware of the other's address and availability. When problems of continuous communication between different system are faced, we can use one of two notable patterns: the observer pattern or the publish-subscribe pattern.

4.2.1 The Observer Pattern

In this pattern, we admit the existence of an object, commonly known as the subject, responsible for the maintenance of a list of all the objects registered for notification on a certain event, the observers. When the event happens, the subject automatically notifies all the registered observers which can then act upon the event.

A common use for this pattern is scenarios where there is a dependency between two roles, where encapsulating between the two allows for a higher code reutilization, and the change of an object belonging to one of the roles forces changes of all the objects of the other role.

In summary, this pattern allows for abstract coupling between the subject and the observer since the subject only knows its list of observers conforming to the specified interface, and broadcast communication, the event that fires the notification does not need to specify the receiver, since all the subscribed observers are notified [49]. Figure 8 presents an example on how the observer pattern's operation is accomplished.

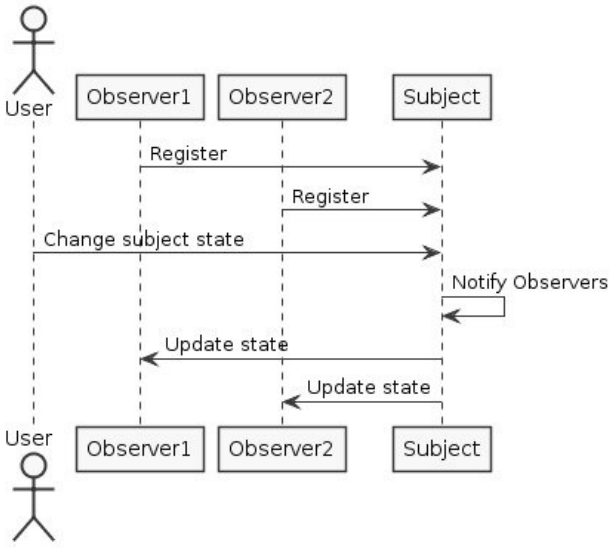

Figure 8 Observer Pattern Sequence Diagram Example

The Observer pattern enables a one-to-many relationship between the message sender and the receivers, but in a system in need of a many-to-many relationship, where the senders notify several receivers and the receivers get notified by several senders demands for a more appropriate pattern to be applied.

4.2.2 The Publish-Subscribe Pattern

The Publish-Subscribe pattern serves the same purpose as the Observer pattern in so far as that they both enable the communication between different entities. In the case of the Publish-Subscribe pattern, we admit the existence of at least three entities: one or more publishers and subscribers and the message broker. The message broker is an entity responsible for holding the information of all the topics being advertised and serves as an intermediary in the communication process. The publisher sends a notification on a certain topic which is then added to the list of topics on the message broker. The subscriber, an entity similar to an observer, notifies the message broker of its interest in the topic of the publisher and so the message broker informs the two of each other's existence. By serving as an intermediary between the communication of the publisher and the subscriber, the message broker diminishes the coupling in the system and enables a many-to-many relationship between the communicating nodes [50]. This process is depicted in Figure 9.

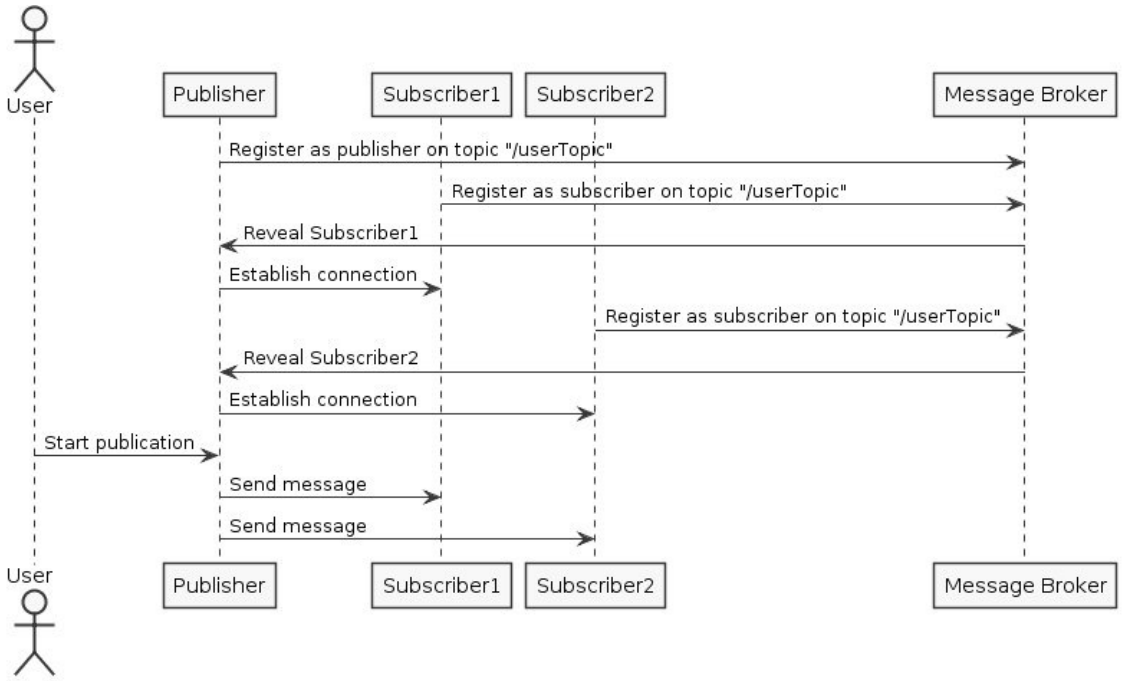

Figure 9 Publish Subcrbie Pattern Sequence Diagram Example

Since the proposed solution intends to emulate the same behaviour as the components in an automotive vehicle, where a many-to-many relationship between certain components is often

required, we decided that the Publish-Subscribe pattern would be more adequate for this solution.

4.3 Architectures

Having decided how the domains will communicate, the next step becomes distributing responsibilities between the domains. As per the use case's requirements, the RTOS domain is the one publicising information, meaning that it must hold the responsibilities of the publisher. Equally, the GPOS domain is on the receiving end and must therefore host the subscriber. One responsibility remains however, since we have yet to decide which VM should have the responsibility of hosting the message broker. In our case, we have three available candidates, each with its advantages and drawbacks:

- The RTOS domain: Hosting the message broker on this domain limits us to two actively used domains. However, it also increases the risk of failing to meet deadlines for real-time applications, hence this domain should host as little responsibilities as possible;
- The GPOS domain: Equally to the RTOS domain, using this machine to serve as message broker limits the actively used machines to two. Using this domain reduces the risk of deadline failure as it frees the RTOS domain to pursue the activities with deadlines to follow. This domain however is the least secure and most prone to failure as it is the one the end user will be communicating with, the most.
- Domain-0: Using Domain-0 to be the message broker increases the actively used machines to three, meaning that each of the other domains will have less processor time. Domain-0 already is responsible for enabling the communication through TCP between the other domains which would make it a good choice in terms of consolidating similar responsibilities. Using Domain-0 frees the RTOS domain CPU time and is more secure than the GPOS domain as it is not accessed by the end user.

Making a compromise between performance and security, we decided that the best option would be to use Domain-0 to be the message broker.

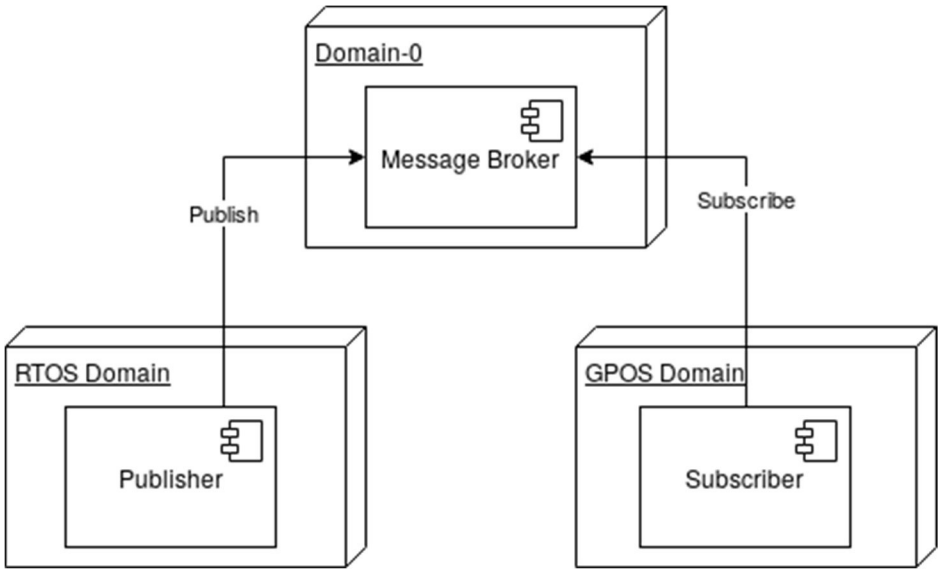

Figure 10 Deployment Diagram of the chosen architecture

In conclusion, as seen in Figure 10, the proposed architecture consists of three VMs. Domain-0, which serves as message broker between the publishers and subscribers, the RTOS domain, which registers as a publisher on Domain-0, responsible for transmitting the sensor data, and the GPOS domain, which registers as a subscriber on Domain-0, responsible for interpreting the received data.

5 Implementation

The current chapter's scope is to delineate the implementation of the use case. We will be covering the OS selection for each VM and the framework chosen to handle the data transmission.

5.1 OS Selection

To answer the problem of the OS selection, several candidates were studied. All candidates are limited by two factors. First, they must present a version compatible with the 32-Bit ARM architecture. Second, they must be compatible with Xen. In the case of the OS for the real-time domain, the OS also must provide real-time capabilities.

Considering the machine with real-time requirements, the first RTOS to be considered was ERIKA Enterprise, due to the previous projects that combined Xen and the RTOS on hardware similar to the Banana Pi [14] and the certifications this RTOS possesses [51] [52]. Unfortunately, developments on the Xen API required modifications on ERIKA's end, making it so that this option was discarded due to time constraints. The second choice was FreeRTOS, a highly popular RTOS that also was at one time compatible with Xen. Being an older project, it also suffered the same fate as ERIKA. As the Xen API changed, FreeRTOS drivers had to change too, and the project's maintainers deeming it not important enough to maintain the compatibility, so it was eventually lost. Finally, we resorted to Preempt-RT, a modification of the Linux Kernel which outfits it with real-time capabilities, transforming Linux into a soft real-time system. While it is arguably not the best option, it serves the purposes of the use case well enough to illustrate the capabilities of the hypervisor regarding the maintenance of spatial and temporal isolation.

Having chosen Linux as being the OS of the RTOS domain, a distribution had to be chosen as well, thus, the chosen OS for the RTOS domain is a release of Ubuntu 16.04 with a Preempt-RT modified kernel.

Turning now to the OS for the GPOS domain, the solution is very straightforward. Xen is developed with Linux in mind, Linux is the native Xen OS and officially supported by the community. The only limiting factor when choosing a distribution is the target architecture,

the distribution must have a release for the 32-Bit ARM architecture, having already chosen Ubuntu 16.04 for the RTOS, the same choice was made for the GPOS, as it facilitates configuration and installation, and is compatible with the chosen data transmission framework.

5.2 Data Transmission Framework

Having chosen a communication mechanism, and how to apply it, a framework was chosen which would enable the use of the publish-subscribe pattern in the interdomain communication, ROS.

ROS [53] is a framework which, with its tools and libraries, allows us to easily establish a communication channel between our domains. Each domain runs an application which registers as a node on the ROS Master, the message broker. These nodes can then publish or subscribe to certain topics. Every node connects to its master through TCP, needing thus either the domain's IP or hostname, the URL for the master is saved in an environment variable and therefore independent of the application.

In this case we deployed the ROS Master on our Domain-0 and created two nodes, one for each unprivileged domain. On the RTOS domain, the node was created as a simple publisher, repeatedly sending a message. On the GPOS domain, the subscriber node would receive the message and print it to the screen. After this preliminary test it was shown that the communication was working as designed.

6 Experiments

Having verified that it is indeed possible to transmit data between VMs using ROS on Xen, it is important to calculate the costs of using our solution to achieve our goal and contemplate the behaviour of a real-world application on the system. For this purpose, we implemented two test cases and deployed a prototype of a real-world application developed by the Vortex Colab.

- The first test is designed to measure the resulting jitter, the difference in message transfer time, from sending information between ROS nodes on different domains in comparison to two ROS nodes on the same domain.
- The second test measures the deadline failure rate impact of the GPOS domain on the RTOS domain, as we expect the hypervisor to guarantee temporal isolation.
- The Vortex prototype application is used to interpret the transferred sensor data and perform its visualization through a web browser.

6.1 Message delivery time jitter

This test was created to measure the impact of Xen's peripheral sharing in the transmission time of data between the domains. Using ROS to handle the publishing and subscribing of messages, a test was designed where the publisher would send a message to the subscriber containing the messages index. The subscriber would then reply with the same index, allowing the publisher to register both departure and arrival time of the message.

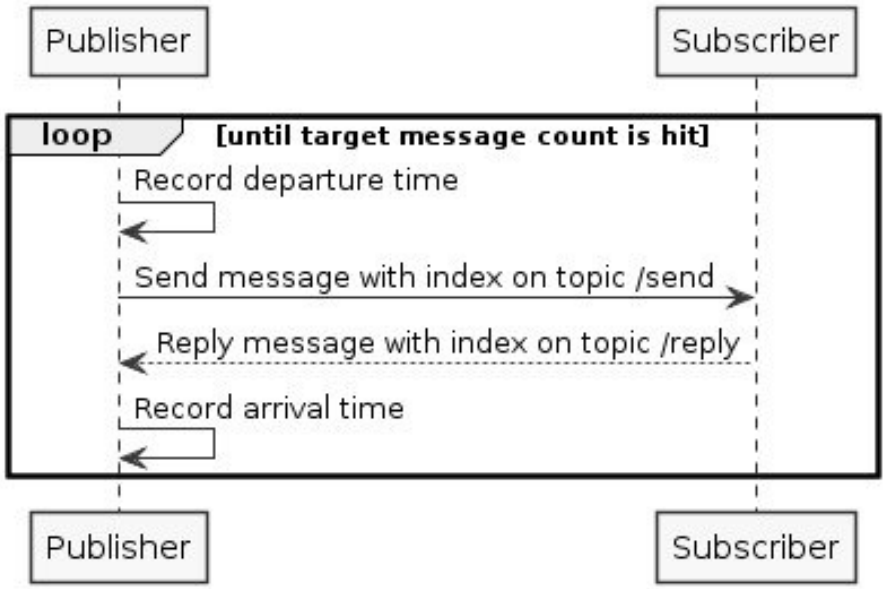

Figure 11 Jitter time test sequence

After recording the departure and arrival time, the time difference is calculated and the next message is sent, repeating the process until the target number of messages is hit. This process is illustrated in Figure 11.

To measure the jitter, first 100 thousand messages were sent from the publisher on Domain-0 to the subscriber on Domain-0, the same experiment was then performed sending messages from the publisher on the RTOS domain to the subscriber on the GPOS domain, resulting in a total of 400 thousand messages and 200 thousand time records, one for every pair of messages.

In order to obtain a proper visualization of the data, allowing us to make judgements on the results, we will be making use of a box plot. A box plot allows us to visualize numeric data through its quantiles, meaning that it represents the minimum, the maximum, the mean, the first quantile and the third quantile of the numeric data. We can then visualize the precision in the data gathered, meaning how far the values deviate from one another. In this case we will be removing the outliers, the values which are more than one and a half standard deviations away from the third quantile, as these values are not representative of the normal functioning and are considered experimental errors.

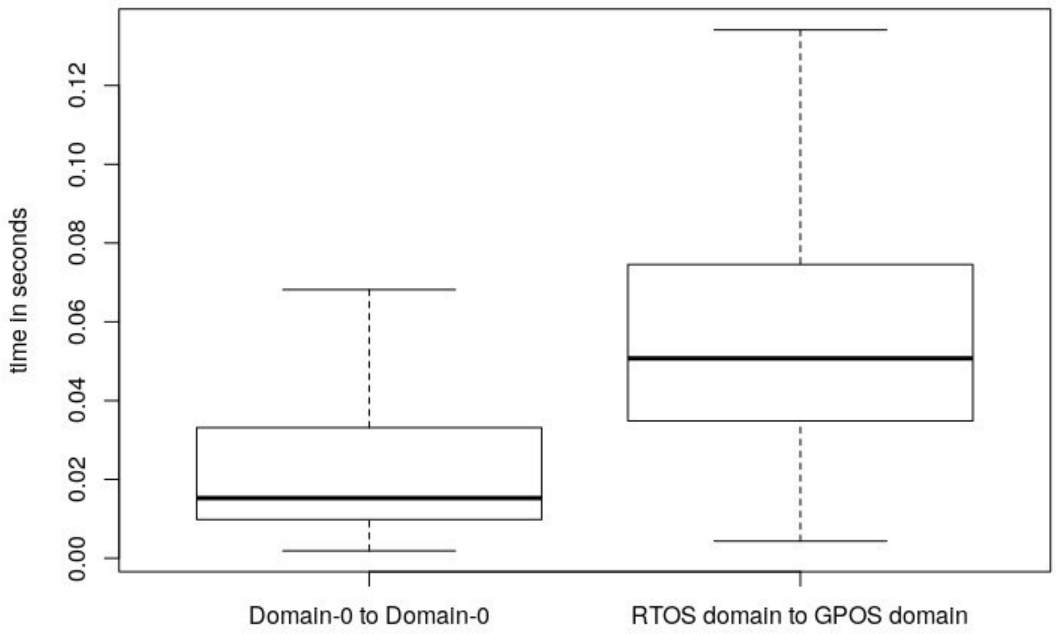

Figure 12 Boxplot on round-trip time on Domain-0 and between the RTOS domain and the GPOS domain

In Figure 12, we can observe that the median for the RTOS-GPOS dataset lies outside of the box of the Domain-0 dataset, which suggests that the two groups are likely to be different. The boxes represent the interquartile range for each dataset, meaning the interval between the first and the third quantile. The values that lead to the plot can be analysed in Table 3.

Table 3 Jitter Test Quantiles

Dataset	Domain-0 to Domain-0	RTOS domain to GPOS domain
Minimum	0.001863956	0.004333973
First Quantile	0.009802461	0.034840465
Median	0.015256524	0.050739527
Third Quantile	0.033153057	0.074551940
Maximum	0.06817695	0.134117127

To be certain that the existing difference between the two datasets is statistically significant and not simply a result of chance, we will finalize this experiment with a student's t-test [54]. A student's t-test allows us to determine if the mean difference between the two groups is statistically significant, thus allowing us to make conclusions based on the datasets. Using a two tailed student's t-test with a confidence level of 95% we obtain a p-value lower than $2.2e^{-16}$, which is lower than our significance level, thus rejecting the null hypothesis that the true difference in means is 0 and concluding that the means are significantly different.

Having determined that the difference is statistically significant, we can now analyse the means to determine the performance loss incurred from using two domains. With the mean of the Domain-0 dataset being 3.208 milliseconds and the mean of the RTOS-GPOS dataset being 6.062 milliseconds we verify an increase of 88.97% in round trip time.

This test allows us to conclude that using Xen's split driver peripheral sharing model results in a significant overhead in the message transfer time. It should be noted, however, that all messages were delivered successfully.

6.2 Deadline failure

One of the most important features of a real-time system is that it is predictable, meaning that we can expect a task to have a certain computation time and that it will arrive in a defined time window. The rear end of the time window is defined to be the deadline, any tasks which computation ends after the deadline is considered a failure. To measure the impact in the deadline failure rate on our RTOS domain resulting from our GPOS domain's utilization, we fashioned a test which would first measure the deadline failure without the GPOS domain and afterwards the deadline failure with the GPOS domain.

For this test a simple real-time application was written, which would perform an access to the system's memory every 6 milliseconds. If the operation took over 6 milliseconds, our defined deadline, we considered it a failure, otherwise we considered it to have been successful. The operation was then repeated ten thousand times and the successes and failures recorded, as depicted in Figure 13.

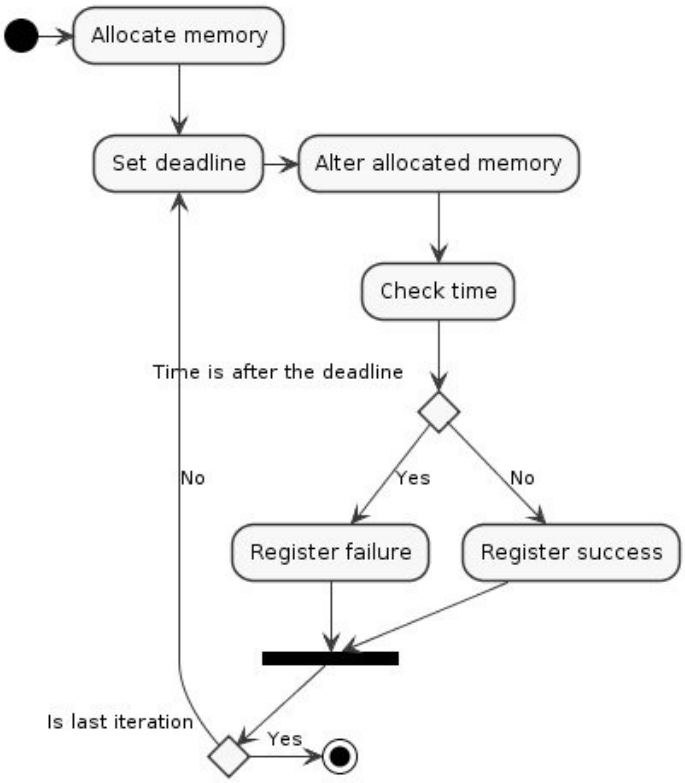

Figure 13 Real time application flowchart

Running the application without interference from the GPOS domain resulted in 40 failures and 9960 successes, meaning that the failure rate was of 0.4%.

In order to tests the GPOS domain 's impact, another application was written, similarly to the real-time application, memory accesses are performed, however at a larger rate. For each iteration of this programs functioning, five sub processes would perform memory accesses simultaneously, in an attempt to occupy both the CPU and the memory. This program would then execute for all ten thousand iterations of the real-time application.

Performing the same experiment as before, now with the GPOS domain consuming the system's resources, the test resulted in 1737 failures and 8263 successes, resulting in a deadline failure rate of 17.37%.

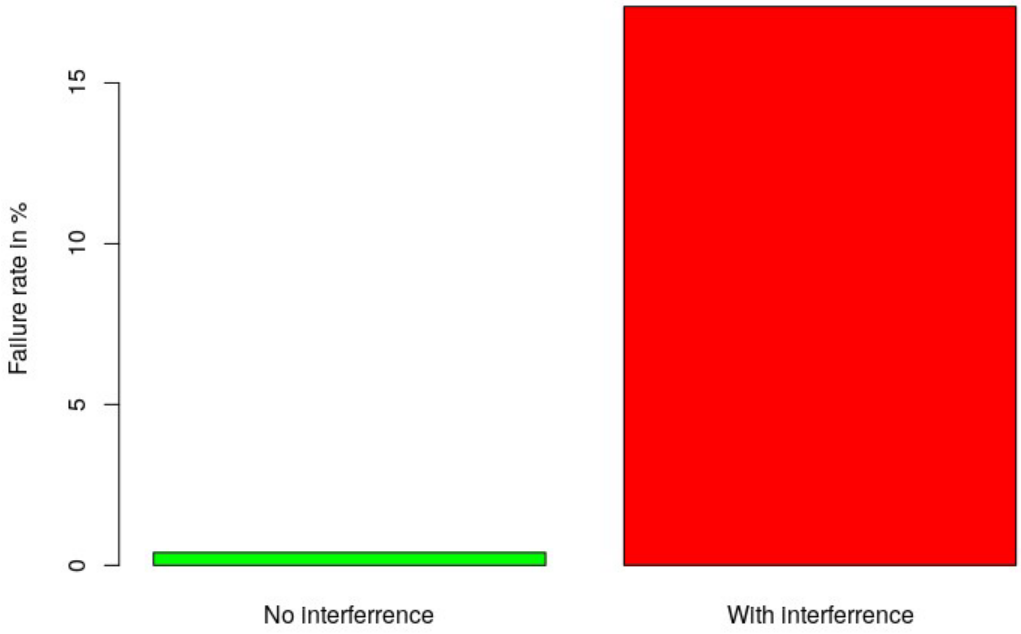

Figure 14 Deadline failure rate comparison

Comparing the two deadline failure rate, represented in Figure 14, 0.4% when the RTOS domain is working alone and 17.37 when it is under the influence of the GPOS domain, we verify an increase in 4242.5%. This leads us to conclude that our solution fails to guarantee temporal isolation, this failure is likely due to the domain's shared CPU cores, a problem which can only be solved by performing the test on a piece of hardware featuring a CPU with at least three cores, one for each domain.

6.3 Real-world application

The most important milestone for this project is confirming if, despite Xen's performance drawbacks, the deployed solution is capable of handling the execution of an application which can be used in a real-world scenario. For this purpose, we resorted to using an application made available to us through to the Vortex Colab. This application is being developed for the automotive industry to allow the visualization of sensor data collected by the vehicle.

Just like our system, this application relies on ROS to perform the transfer of data, which is then interpreted and made available for visualization through a web browser. The sensor data transferred is available in its raw form, courtesy of the Karlsruhe Institute of Technology [55], but can only be used after conversion to the ROS bag file format [56]. We use this generated

bag file on our RTOS domain and initiate publication through the rosbag tool. Through ROS's rosbag tool we can easily initiate the gradual transfer of the sensor data at any given rate, meaning we can accelerate or decelerate the transmission. The setup is illustrated in Figure 15.

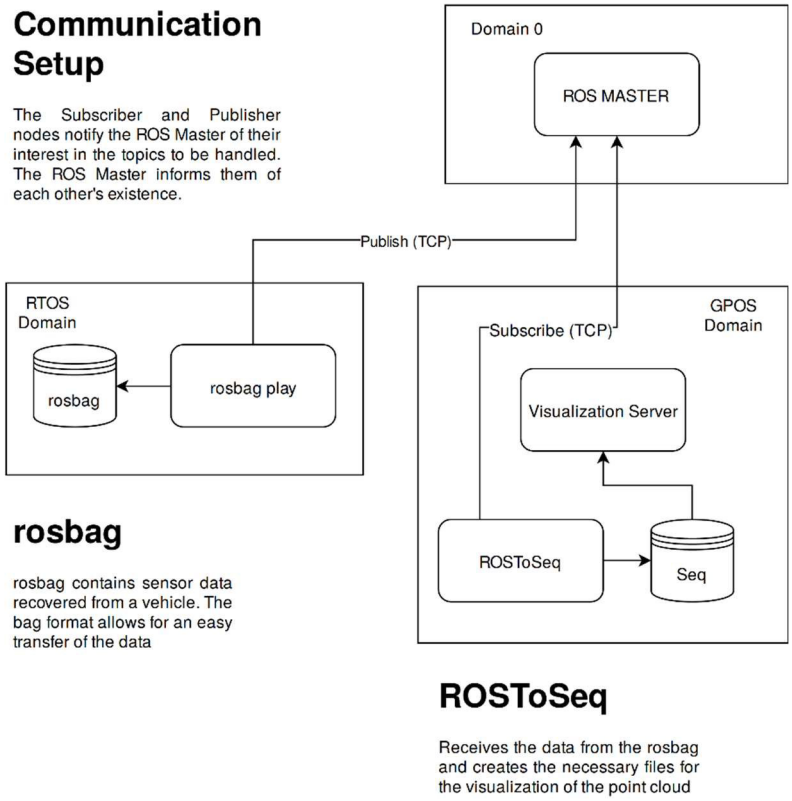

Figure 15 Deployment of the Vortex Application

On our GPOS domain we can find the subscriber, labelled ROSToSeq. This program allows us to transform the received messages into a format, called sequences, which can be interpreted by the virtualization server. Lastly, the virtualization server provides a web interface through which all the created sequences can be played at the user's leisure.

In Figure 16 we can see on the left, the transmission of the contents of the bag file, listing the total time of the file and the current instant being transmitted. On the top right corner, we

can see index of the last created file as well as some indicators of the conversion time. Finally, on the bottom right we can follow the execution of the ROS Master.

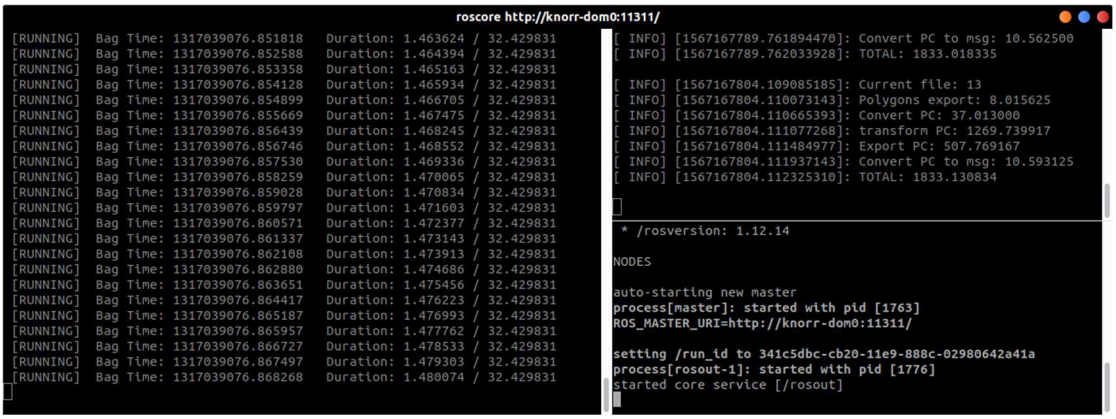

Figure 16 Execution of the Subscriber, Publisher and Message Broker

After the files have been created and transmitted, we can proceed to the visualization of said files through the visualization server. As shown in Figure 17, after selecting the sequence to reproduce, the program will load the sequence and display all the entities captured by the sensor on the vehicles travel, as well as the trajectory the vehicle followed, as can be seen in Figure 17.

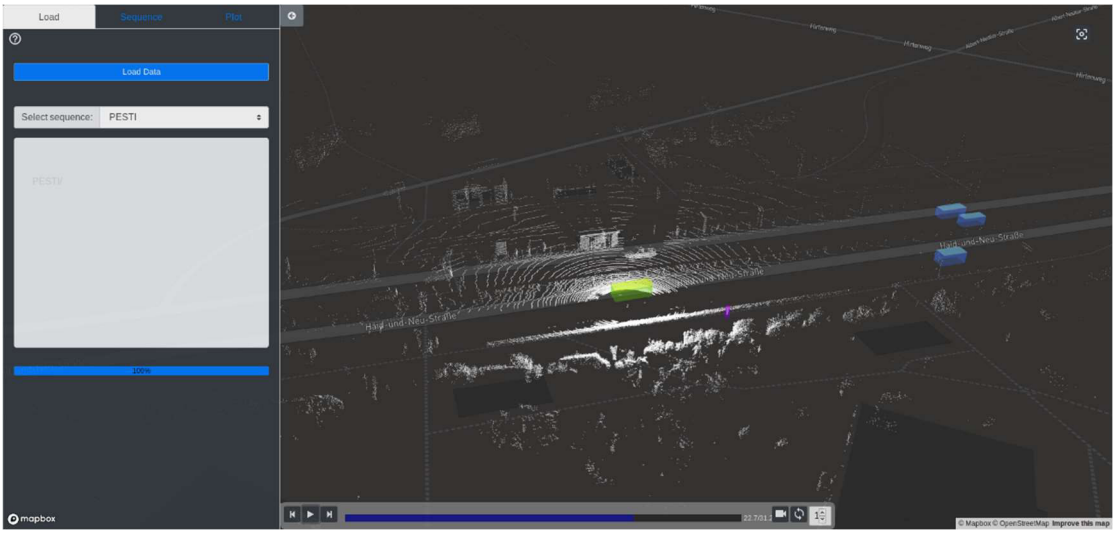

Figure 17 Visualization of the sensor data

Having executed a prototype of an application to be used in the automotive industry for a real-world scenario, we can conclude, through this experiment, that our system is capable of achieving the proposed goal, while virtualizing all the required machines with Xen. The result of this test represents the first experiment, for the Vortex Colab, in a series of attempts to use Xen for virtualizing automotive systems, however, further testing should be done on hardware which is generally used in the industry to confirm if the solution is scalable and can meet the requirements of the Vortex Colab.

7 Conclusions

In this chapter we will look back at all the work done during the development of this project, making judgements on the advantages and drawbacks of the implemented solution. First, we will list the objectives which were achieved, followed by a brief description of the implemented solution and finishing with the limitations the systems incurs through our solution and the selected hardware. Having done this analysis, a few suggestions will be made on how to conduct further research on this topic as well as a description of limiting factors in the conduction of this research.

7.1 Proposed vs achieved goals

Here we will list the initial objectives proposed by the project:

- Study the partitioning aspect of the Xen or KVM hypervisor
- Study the communication aspect of the Xen or KVM hypervisor
- Study the impact of deploying systems of mixed criticality on the Xen or KVM hypervisor

Having listed all the proposed objectives, we will now list what was indeed achieved:

- Two VMs were deployed on the Banana Pi SOC. To each VM was an independent storage area and memory were allocated. The processor cores were shared due to their limited number. Peripheral devices are assigned to Domain-0 but can be shared with other domains through the use of split drivers.
- The two VMs were connected through TCP on the OS level as, on ARM, Xen does not support memory sharing.
- Three experiments were performed on the deployed solution. One to measure message transfer jitter, one to measure the impact the GPOS machine has on the RTOS machines' deadline failure rate and one to verify how the solution would behave in real-world conditions.

7.2 The solution

To test the partitioning and communication aspects of the Xen hypervisor, the system was deployed on a Banana PI SOC. For the deployment of Domain-0 and the GPOS domain Ubuntu 16.04 was chosen as the OS, on the RTOS domain the OS was equally Ubuntu 16.04, however with a PREEMPT_RT modified kernel, turning it into a sot real-time system. After the deployment of the system a communication channel was set up through TCP with ROS using the publish-subscribe pattern, as it was deemed more appropriate for the proposed scenario. The solution features thus three VMs, Domain-0, which serves as the message broker for our communication, the GPOS domain which serves as the subscriber and receives information and the RTOS domain which serves as the publisher and is responsible for the transmission of data.

The solution was later tested for message transfer jitter and deadline failure. In the message transfer test, the system suffered an 88.97% decrease in velocity in comparison to two services on the same domain. In the deadline test, a 4242.5% increase on missed deadlines was measured when the GPOS domain was allowed to consume the system's CPU and memory resources simultaneously with the RTOS domain.

Finally, to test if the system could be used in the automotive industry a prototype of a real-world application was used to simulate the transmission and visualization of sensor data between two VMs, a test which proved to be successful and without noteworthy problems.

7.3 Future work and limitations

In the conduction of this research, the most limiting factor were the missing, outdated and often miscategorised parts of the Xen documentation. Other limiting factors included limited support for the chosen hardware on the part of the chosen OSs and the extensive time consumed by compilating all the necessary parts.

For future work, different hardware should be used, as 32-Bit is on the decline and support is dropping. Equally, for better testing the partitioning aspect, principally in terms of the CPU, the hardware used should feature at least six cores, to comfortably host all three VMs without interference.

8 References

[1] The Linux Foundation Automotive Grade Linux (AGL) Virtualization Expert Group (EG-VIRT), The Automotive Grade Linux Software Defined Connected Car Architecture, The Linux Foundation, 2018.

[2] L. P. Kaelbling, Learning in Embedded Systems, A Bradford book, 1993.

[3] A. Massa and M. Barr, Programming Embedded Systems, O'Reilly, 2006.

[4] T. Holstein and J. Wietzke, Contradiction of Separation through Virtualization and Inner Virtual Machine Communication in Automotive Scenarios.

[5] Xen Project, “Home - Xen Project,” The Linux Foundation, [Online]. Available: https://xenproject.org/. [Accessed 30 May 2019].

[6] K. contributors, “KVM,” KVM, 07 November 2016. [Online]. Available: https://www.linux-kvm.org/page/Main_Page. [Accessed 30 May 2019].

[7] Siemens, “GitHub - siemens/jailhouse: Linux-based partitioning hypervisor,” [Online]. Available: https://github.com/siemens/jailhouse. [Accessed 31 May 2019].

[8] S. Cooley and H. Lohr, “Hyper-V on Windows 10 | Microsoft Docs,” Microsoft Corporation, 05 February 2016. [Online]. Available: https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/. [Accessed 30 May 2019].

[9] “Oracle VM VirtualBox,” Oracle, [Online]. Available: https://www.virtualbox.org/. [Accessed 30 May 2019].

[10] P. Modica, A. Biondi, G. Buttazzo and A. Patel, “Supporting Temporal and Spatial Isolation in a Hypervisor for ARM Multicre Platforms,” in *2018 IEEE International Conference of Industrial Technology (ICIT)*, Lyon, France, 2018.

[11] T. J. M, *Virtualization for embedded systems,* developerWorks, 2011.

[12] S. Trujillo, A. Crespo and A. Alonso, “MultiPARTES: Multicore Virtualization for Mixed-Criticality Systems,” *2013 Euromicro Conference on Digital System Design,* pp. 260-265, 2013.

[13] Evidence, “Erika Enterprise | Open Source RTOS OSEK/VDX Kernel,” [Online]. Available: https://www.erika.tuxfamily.org/drupal. [Accessed 5 Semptember 2019].

[14] A. Avanzini, P. Valente, D. Faggioli and P. Gai, “Integrating Linux and the real-time ERIKA OS through the Xen hypervisor,” *10th IEEE International Symposium on Industrial Embedded Systems (SIES),* 2015.

[15] ARAMiS II, “Projekt - ARAMiS II,” [Online]. Available: https://www.aramis2.org/projekt/. [Accessed 11 May 2019].

[16] ARAMiS II, “Demonstratoren - ARMAIS II,” [Online]. Available: https://www.aramis2.org/demonstratoren/. [Accessed 11 May 2019].

[17] R. P. Goldberg and G. J. Popek, “Formal requirements for virtualizable third generation architectures,” *Communications of the ACM,* vol. 17, no. 7, pp. 412--421, 1974.

[18] R. P. Goldberg, “Architectural Principles for Virtual Computer Systems,” 1972.

[19] T. Abels, P. Dhawan and B. Chandrasekaran, “An overview of xen virtualization,” *Dell Power Solutions,* vol. 8, pp. 109--111, 2005.

[20] A. Whitaker, M. Shaw and S. D. Gribble, “Denali: Lightweight virtual machines for distributed and networked applications,” 2002.

[21] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield, “Xen and the art of virtualization,” *ACM SIGOPS operating systems review,* vol. 37, no. 4, pp. 164--177, 2003.

[22] Xen Project, “Xen Project Software Overview,” [Online]. Available: https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview. [Accessed 20 March 2019].

[23] P. Jayshri and N. Koli, “A technical review on comparison of Xen and KVM hypervisors: An analysis of virtualization technologies,” *International Journal of Advanced Research in Computer and Communication Engineering,* pp. 8828-8832, 12 2014.

[24] A. Kivity, Y. Kamay, D. Laor, U. Lublin and A. Liguori, “kvm: the Linux Virtual Machine Monitor,” *Proceedings of the Linux Symposium,* vol. 1, pp. 225--230, June 2007.

[25] M. T. Jones, “Virtio: An I/O virtualization framework for Linux,” *IBM White Paper,* 29 January 2010.

[26] Red Hat, Inc., “What is KVM?,” [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-KVM. [Accessed 20 March 2019].

[27] J. Kiszka, “[ANNOUNCE] Jailhouse: A Linux-based Partitioning Hypervisor,” 18 November 2013. [Online]. Available: https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg538651.html. [Accessed 30 April 2019].

[28] S. Valentine, “Understanding the Jailhouse hypervisor, part 1 [LWN.net],” 1 January 2014. [Online]. Available: https://lwn.net/Articles/578295/. [Accessed 30 April 2019].

[29] H. Kopetz, Real-Time Systems Design for Distributed Embedded Applications, 2011.

[30] K. Juvva, “Real-Time Systems,” *Topics in Dependable Embedded Systems,* no. 28, 1998.

[31] T. Kaldewey, C. Lin and S. Brandt, “Firm real-time processing in an integrated real-time system,” *REPORT-UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS,* vol. 398, p. 6, 2006.

[32] University of California, Los Angeles, “Real Time Scheduling,” 1 April 2016. [Online]. Available: http://web.cs.ucla.edu/classes/spring16/cs111/supp/realtime.html. [Accessed 30 April 2019].

[33] OSRTOS, “OSRTOS | Erika Enterprise,” [Online]. Available: https://www.osrtos.com/rtos/erika-enterprise/. [Accessed 30 April 2019].

[34] Evidence, ERIKA Enterprise Minimal API Manual, 2012.

[35] D. John, “OSEK/VDX history and structure,” *IEE Seminar on OSEK/VDX Open Systems in Automotive Networks ,* 1998.

[36] Autosar, “FAQ - AUTOSAR,” [Online]. Available: https://www.autosar.org/faq/. [Accessed 30 April 2019].

[37] Autosar, “History - AUTOSAR,” [Online]. Available: https://www.autosar.org/about/history/. [Accessed 30 April 2019].

[38] OSRTOS, “OSRTOS | FreeRTOS,” [Online]. Available: https://www.osrtos.com/rtos/freertos/. [Accessed 30 April 2019].

[39] FreeRTOS Team, “FreeRTOS - The Free RTOS configuration constants and configuration options - FREE Open Source RTOS for small real time embedded systems,” [Online]. Available: https://www.freertos.org/a00110.html. [Accessed 30 April 2019].

[40] WITTENSTEIN, “SAFERTOS, the safety certified RTOS - available pre-certified to IEC 61508,” WA&S Ltd, [Online]. Available: https://www.highintegritysystems.com/safertos/. [Accessed 30 April 2019].

[41] P. McKenny, “A realtime preemption overview [LWN.net],” 10 August 2005. [Online]. Available: https://lwn.net/articles/146861. [Accessed 27 June 2019].

[42] The Linux Foundation, “Intro to Real-Time Linux for Embedded Developers - The Linux Foundation,” 2019 March 2013. [Online]. Available:

https://www.linuxfoundation.org/blog/2013/03/intro-to-real-time-linux-for-embedded-developers/. [Accessed 27 June 27].

[43] W. Denk, “History < U-Bootdoc < DENX,” [Online]. Available: http://www.denx.de/wiki/view/U-Bootdoc/History. [Accessed 11 May 2019].

[44] W. Denk, “Geocrawler.com - ppcboot-users - [PPCBoot-users] Halloween release of PPCBoot: 2.0.0 - the Final Release.,” [Online]. Available: https://web.archive.org/web/20040127050919/http:/www.geocrawler.com/archives/3/4205/2002/10/0/10043434/. [Accessed 11 May 2019].

[45] K. Yaghmour, J. Masters, G. Ben-Yossef and P. Gerum, Building Embedded Linux Systems, 2008.

[46] Xen Project, “Xen ARM with Virtualization Extensions whitepaper - Xen,” [Online]. Available: https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper. [Accessed 30 July 2019].

[47] R. Mortier, “xen-arm-builder/u-boot.sh at master · mirage/xen-arm-builder · GitHub,” 17 November 2016. [Online]. Available: https://github.com/mirage/xen-arm-builder/blob/master/u-boot.sh. [Accessed 12 August 2019].

[48] Xen Project, “Mainline Linux Kernel Configs - Xen,” 26 January 2017. [Online]. Available: https://wiki.xenproject.org/wiki/Mainline_Linux_Kernel_Configs. [Accessed 12 August 2019].

[49] E. Gamma, R. Helm, R. Johnson and J. Vlissides, “Observer,” in *Design Patterns Elements of Reusable Object-Oriented Software*, Westford, Massachusets, Addison-Wesley, 2009, pp. 293-296.

[50] YanqingWu, “Master - ROS Wiki,” Open Source Robotics Foundation, 15 January 2018. [Online]. Available: wiki.ros.org/Master. [Accessed 26 August 2019].

[51] Evidence, “Misra compliance - ErikaWiki,” [Online]. Available: https://www.erika.tuxfamily.org/wiki/index.php?title=Misra_compliance. [Accessed 11 September 2019].

[52] Evidence, “OSEK VDX certifications - ErikaWiki,” [Online]. Available: https://www.erika.tuxfamily.org/wiki/index.php?title=OSEK_VDX_certifications. [Accessed 11 September 2019].

[53] Open Source Robotics Foundation, “ROS.org | Powering the world's robots,” [Online]. Available: https://www.ros.org. [Accessed 28 August 2019].

[54] Encyclopaedia Britannica, “Student's t-test | statistics ! Britannica.com,” 20 July 1998. [Online]. Available: https://www.britannica.com/science/Students-t-test. [Accessed 12 September 2019].

[55] A. Geiger, P. Lenz, C. Stiller and U. Raquel, “Vision meets Robotics: The KITTI Dataset,” *International Journal of Robotics Research (IJRR),* 2013.

[56] T. Krejci, “GitHub - tomas789/kitti2bag: Convert KITTI dataset to ROS bag file the wasy way!,” [Online]. Available: https://github.com/tomas789/kitti2bag. [Accessed 26 August 2019].

[57] Xen Project, “Support statement for this release,” [Online]. Available: https://xenbits.xen.org/docs/4.11-testing/SUPPORT.html#memory-sharing. [Accessed 7 July 2019].