NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A SURVEY OF REAL-TIME OPERATING SYSTEMS AND VIRTUALIZATION SOLUTIONS FOR SPACE SYSTEMS

by

Katherine K. Sheridan-Barbian

March 2015

Thesis Co-Advisors: Thuy D. Nguyen
Mark Gondree

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2015	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE
A SURVEY OF REAL-TIME OPERATING SYSTEMS AND VIRTUALIZATION SOLUTIONS FOR SPACE SYSTEMS

5. FUNDING NUMBERS

6. AUTHOR(S) Katherine K. Sheridan-Barbian

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	**12b. DISTRIBUTION CODE**
Approved for public release; distribution is unlimited	A

13. ABSTRACT (maximum 200 words)

The Department of Defense and the intelligence community rely on space systems for a broad spectrum of services. These systems operate in highly constrained environments (in terms of space, weight and power), making virtualization and resource sharing a desirable approach. Agencies are actively exploring new architectures, such as those employing virtualization, to support their growing space mission. In this thesis, we review how virtualization architectures claim to support the real-time requirements of their guests. We survey real-time systems and virtualization architectures proposed for use in space systems. Further, we investigate the behaviors of virtualized operating systems using a method of remote network-based fingerprinting with TCP timestamps. Our work provides insights into how guests, both general purpose and real-time, behave in virtualized environments. Our survey work and experimental analysis aim to further understanding of how virtualization can be securely incorporated into space systems.

14. SUBJECT TERMS	**15. NUMBER OF PAGES**
virtualization, hypervisor, real-time operating system, fingerprinting, space system	147
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540–01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

A SURVEY OF REAL-TIME OPERATING SYSTEMS AND VIRTUALIZATION SOLUTIONS FOR SPACE SYSTEMS

Katherine K. Sheridan-Barbian
Civilian, Department of Defense
B.A., Barnard College, 2004

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2015

Author: Katherine K. Sheridan-Barbian

Approved by: Thuy D. Nguyen
Thesis Co-Advisor

Mark Gondree
Thesis Co-Advisor

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Department of Defense and the intelligence community rely on space systems for a broad spectrum of services. These systems operate in highly constrained environments (in terms of space, weight and power), making virtualization and resource sharing a desirable approach. Agencies are actively exploring new architectures, such as those employing virtualization, to support their growing space mission. In this thesis, we review how virtualization architectures claim to support the real-time requirements of their guests. We survey real-time systems and virtualization architectures proposed for use in space systems. Further, we investigate the behaviors of virtualized operating systems using a method of remote network-based fingerprinting with TCP timestamps. Our work provides insights into how guests, both general purpose and real-time, behave in virtualized environments. Our survey work and experimental analysis aim to further understanding of how virtualization can be securely incorporated into space systems.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

APEX	applications/executive
ARLX	ARINC-653 Real-time Linux on Xen
AIR	ARINC-653 Interface in RTEMS
BCET	best-case execution time
CIM	common information model
COTS	commercial off-the-shelf
CPU	central processing unit
DAL	design assurance level
DCUI	direct console user interface
DOD	Department of Defense
GPOS	general purpose operating system
GSFC	Goddard Space Flight Center
GSWS	Galileo software standard
I/O	input/output
IMA	Integrated Modular Avionics
IMA-SP	Integrated Modular Avionics for Space
ISA	instruction set architectures
KSM	kernel same-page merging
KVM	kernel virtual machine
MMU	memory management unit
MSE	mean squared error
NASA	National Aeronautics and Space Administration
NRO	National Reconnaissance Office
NTP	network time protocol
OS	operating system
PMK	partition management kernel
RAD-HARD	radiation-hardened
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	real-time operating systems
RTP	real-time process

SMP	symmetric multiprocessing
SPAWAR	Space and Naval Warfare Systems Command
SWaP	size, weight and power
TCP/IP	Transmission Control Protocol/Internet Protocol
TSP	time-space partitioned
VCT	virtual machine configuration table
VM	virtual machine
VMM	virtual machine monitor
WCET	worst-case execution time

ACKNOWLEDGMENTS

I thank my family more than anything for supporting me over the past two years and for being patient while I worked on this thesis. I also thank my fellow classmates, especially Francisco Gutierrez-Villarreal, for helping me get through this program and helping refine my Python graphing skills. Lastly, I thank my thesis advisors for putting up with me and mentoring me through this process.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Virtualization has proven itself a viable approach to resource sharing in terrestrial systems but its use in space systems is a relatively new idea (Cudmore, 2013). Space systems have a number of unique needs, such as space, weight and power (SWaP) constraints and real-time requirements (Kang & Kim, 2014). There are many proposed virtualization architectures for space, but their ability to support systems with real-time requirements needs to be better understood. The role of virtualization in the context of real-time requirements is the focus of this thesis. In particular, we survey the design, implementation and performance of real-time operating systems and virtualization platforms proposed for space. We attempt to understand how these virtualization architectures claim to support the real-time requirements of the systems they host. We review the security properties of these systems, such as how process isolation is achieved. We also consider a practical aspect to security that has received little prior attention: we extend prior work on remote fingerprinting virtualized operating systems to consider fingerprinting real-time systems. The broad goal of this thesis is to express the relationship between virtualization and real-time systems, using space as a motivating context.

A. MOTIVATION

The use of space systems has grown dramatically since their inception. This motivates the development of new space system architectures able to support this demand. General William Shelton of Air Force Space Command claims that space was once a domain in which a single satellite orbited earth and is now one that supports nearly every United States military operation across the world (Garamone, 2014). The Department of Defense (DOD) relies on space systems for a broad spectrum of services, including communications, mission specific intelligence, operational awareness and weather analysis. The 2000 National Reconnaissance Office (NRO) Commission Report describes how the demand for data from NRO satellites has increased disproportionally to the resources provisioned, which is putting pressure on the office to meet all the

requirements from its customers ("Report of the National Commission for the Review of the National Reconnaissance Office," 2000). The DOD recognizes this strain on space system resources and is developing strategies to overcome such issues. One such strategy is the development of alternative architectures to make space systems more flexible, more secure and less costly. The 2011 National Security Space Strategy emphasizes the need to develop a "resilient, flexible, and healthy space industrial base" and states that it will "continue to explore a mix of capabilities with shorter development cycles to minimize delays, cut cost growth, and enable more rapid technology maturation, innovation, and exploitation" (Department of Defense [DOD], 2011).

At the same time, the functional requirements of embedded systems in the space domain and the hardware that supports them have become more complex over the past two decades (Andrews, Bate, Nolte, Otero-Perez, & Petters, 2005; Windsor, Deredempt, & De-Ferluc, 2011). Many systems are now moving to multicore processors instead of single core processors, which complicate the systems' ability to safely and securely support isolated real-time processes (Santangelo, 2013). As a result, efforts are being made to consolidate the code base of these complex systems and to design a robust management infrastructure to maintain temporal and spatial isolation between real-time applications and to limit security vulnerabilities (Joe et al., 2012; DaSilva, 2012; Windsor et al., 2011).

The DOD faces a number of challenges in the space domain given the numerous requirements for space systems vital to national security today. The DOD needs to incorporate the growing complexities of embedded systems in space while simultaneously cutting the costs of space missions and increasing the flexibility and adaptability of these systems. The U.S. space industry is exploring different ways to effectively address these needs (Cudmore, 2013). One solution that has gained considerable traction is to move away from federated system architectures, integrating software components into a tightly-coupled, modular architecture. The avionics industry paved the way to such an integrated architecture with its development of the *integrated modular avionics* (IMA) architecture. The space industry is now in the process of

developing an architecture similar to IMA that addresses the unique requirements of space systems (Windsor et al., 2011).

B. IMA AND IMA-SP

The IMA conceptual architecture centralizes the various functions and services involved in a complex avionics system onto a single set of physical resources (Rushby, 2000; DaSilva, 2012). IMA was introduced by the commercial avionics industry in the 1990s (Ramsey, 2007). The motivation for IMA was to reduce costs associated with distributed hardware systems while maintaining the ability to manage the software in avionics systems efficiently, safely and securely. IMA was also meant to make system development easier by enabling incremental validation and parallel development of components (Windsor & Hjortnaes, 2009). The two key principles of security in the IMA construct are spatial and temporal isolation (Parkinson, 2011). Spatial isolation is achieved through software partitions, which are implemented in order to handle fault containment. If a fault event occurs in one partition, it is isolated to that partition and does not affect the other partitions in the system (Rushby, 2000). Temporal isolation is achieved through a statically defined scheduling algorithm for each partition, which regulates the amount of processing power each partition receives (DaSilva, 2012). An attractive method for implementing the IMA concept is through virtualization. Instead of having a distributed network of hardware devices that are each dedicated to specific functions, virtualization allows applications running in different software partitions to share the same hardware resources. IMA's use is widespread throughout the commercial avionics industry (FAA, 2007) and its successful implementation has motivated the space industry to consider a similar conceptual framework (Diniz & Rufino, 2005).

To the best of our knowledge, the majority of work in developing an IMA construct for space has been done by the European Space Agency. Claudio DaSilva discusses the European Space Agency's work in developing the Integrated Modular Avionics for Space (IMA-SP) platform. He argues that the space industry lacks standardization and is in need of a partitioned software architecture like IMA. He notes, however, that there are several unique characteristics of the space domain that need to be

considered in the development of an IMA framework for space, including the limited power, mass and volume resources of space systems, which the ESA is currently studying (DaSilva, 2012). Windsor et al. (2011) also discuss the ESA's work in evaluating Integrated Modular Avionics for Space (IMA-SP) and the current work in defining and demonstrating the IMA-SP construct with other members of the space community. NASA is cognizant of the need for more modular software architectures in space and is currently researching the benefits of virtualization and partitioning architectures in space, with the same goals as IMA-SP (Cudmore, 2013; Rushby, 2011). Many U.S. companies developing software products for the aerospace industry are also aware of the movement towards integrated architectures in space and are developing products that adhere to the IMA architecture.

C. THESIS ORGANIZATION

This thesis is organized as follows. In Chapter II, we review requirements of real-time operating systems for space systems, software compliance standards for space systems and an overview of virtualization architectures. In Chapter III and IV, we survey several real-time operating systems and virtualization architectures designed for space systems. In Chapter V, we present our work in network-based fingerprinting of virtualized operating systems, and in Chapter VI, we conclude and discuss future work.

II. BACKGROUND

In this chapter, we review a number of topics that provide context for the real-time operating systems and virtualization architectures we survey later. First, we discuss real-time operating systems and the requirements for real-time operating systems in space. We review security criteria for space systems and software standards for space applications. Finally, we review common virtualization architectures and prior work relating to virtualization with real-time operating systems.

A. REAL-TIME OPERATING SYSTEMS

NASA defines a real-time operating system (RTOS) as a "preemptive multitasking operating system intended for real-time applications" and lists several features that an RTOS should have, which are summarized in Table 1.

Table 1. Characteristic Features of an RTOS (from "RTOS 101," n.d.)

Characteristics of an RTOS
Scheduling mechanism that guarantees response time
Task prioritization
Support for task synchronization
Priority inheritance
Hardware and software resource management
Guarantees tasks get completed by a deadline
Deterministic
Minimal latency
Minimal context switching

There are three primary categories for deadlines of real-time tasks: soft, firm and hard. Soft deadlines are those that are desirable but, if not met, will not cause serious damage to the system. If a firm deadline is missed, the system will not encounter total

failure but consecutive firm deadline misses could lead to system failure. Hard deadlines are ones that, if missed, result in catastrophic consequences to the system ("RTOS 101," n.d.).

What distinguishes an RTOS from a general-purpose operating system (GPOS) is the way it handles task scheduling and preemption in the kernel (Leroux, 2005). An RTOS schedules tasks based on their priority or deadline whereas a GPOS generally schedules tasks in a manner that maintains high throughput. An RTOS allows calls to the kernel to be preempted by user tasks that have higher priority whereas a GPOS requires that calls to the kernel be completed before another task can run, even if the task waiting is of higher priority than the task making the kernel call ("GPOS vs. RTOS," 2012).

B. REAL-TIME OPERATING SYSTEMS IN SPACE

RTOSs are used extensively in space operations due to the time-sensitive and safety-critical operations handled, such as attitude and orbit control, navigation, communications, critical payload management and power management (Keesee, 2003). Unlike those for terrestrial systems, RTOSs for space systems must perform their functions under harsh environments over the lifetime of the space mission, which can be over a decade in some cases (Air Force Space Command, 2013). Additionally, RTOSs must be compatible with space-qualified hardware. For example, a relatively small number of processors are designed to withstand the radiation present in space environments by being radiation-hardened (RAD-HARD) (Beus-Dukic, 2001; Ginosar, 2012). Further, efforts need to be taken to manage the size, weight and power (SWaP) of all space system components, including the operating system. Thus, RTOSs used for space systems often have a smaller memory footprint to accommodate SWaP constraints (Jones & Gross, 2014; Cudmore, 2013).

Beus-Dukic conducted a survey at the 1999 Eurospace conference among conference participants on the criteria they felt was most important when choosing a commercial RTOS for space system development. The survey found that most participants considered RTOS configurability, scalability and hardware compatibility to be essential features of an RTOS. Also of high importance were support for specific

programming languages and the availability of development tools. Unfortunately, there has not been a comparable survey since this, but their data gives us some insight into what criteria might be used by developers when choosing a commercial RTOS.

C. SOFTWARE COMPLIANCE IN SPACE SYSTEMS

The use of commercial off-the-shelf (COTS) products in space systems introduces some unique concerns in the space industry. NASA identifies some of these concerns, including the possible lack of documentation; the inability to examine the source code of proprietary software; the questionable development process of the code base; and the lack of required functionality or the addition of unnecessary functionality (NASA, 2004a). Since much of the software used in space systems is commercially developed, a series of standards and guidelines exist to help ensure software meets basic safety and security requirements for space systems. It should be noted that—while the U.S. space industry has a number of software standards and guidelines ("NASA Reference Documents," 2013)—much of the research in virtualization for space systems draws from avionics software guidelines (Windsor et al., 2011). The commercial avionics industry pioneered the idea of integrated modular avionics because of its potential to reduce costs and increase revenue and it has been successfully tested and deployed in a number of commercial aircraft (Prisaznuk, 2008). The space industry is drawing from the success of IMA and hardware consolidation using guidelines developed by the avionics industry to develop its own IMA architectures. In this section, we discuss existing standards and guidelines for the space industry, as well as relevant guidelines from the avionics industry.

1. DOD Standards

MIL-STD-498 is a U.S. military standard pertaining to software development released in 1994 (DOD, 1994). In 2008, it was largely superseded by IEEE 12207 (Moore, 1998), an international standard pertaining to the software life cycle process. It includes guidance involving the processes, activities and tasks included during the acquisition, service, supply, development, operation and maintenance of software products. It is intended for software acquisitions personnel, suppliers, developers,

operators, maintainers, managers and users of the software. IEEE 12207 is listed as a required standard for NASA mission-critical software ("NASA Software Guidelines," n.d).

DOD Instruction 8581.01 (DOD, 2010) is the DOD's information assurance policy for space systems. The instruction applies to all DOD space systems and space system components used to receive, store, process, display or transmit classified and unclassified data. The instruction lists information assurance directives with which DOD systems must comply and also mandates that information assurance requirements for software used in DOD systems be validated through the applicable military department.

a. IEEE 1228 and NASA-STD-8719.13B

IEEE 1228 is an international standard published in 1994 pertaining to software safety plans. IEEE 1228 is cited in NASA's own standard for software safety, NASA-STD-8719.13B, as an optional standard that can be used as an additional template when developing a software safety plan. NASA-STD-8719.13B is a NASA-specific technical standard, published in 2004. NASA-STD-8719.13B outlines software safety requirements for all NASA projects and details how to guarantee safety is built into software developed or acquired by NASA (NASA, 2004b). This standard applies to all COTS software, stating that all COTS software used in safety-critical systems needs to be thoroughly analyzed and evaluated. Interfaces to developed code, extra functionality, the ability to meet safety functions and the interaction of the software with other parts of the system need to be tested (NASA, 2004b, p. 28).

b. DO-178B

DO-178B, titled "Software Considerations in Airborne Systems and Equipment Certification," is primarily used by the Federal Aviation Administration but is sited extensively by NASA and others proposing virtualization architectures and real-time operating environments for space (Beus-Dukic, 2001; Vanderleest, 2013). DO-178B is not a mandatory standard but, rather, a set of guidelines to ensure the software used in airborne systems complies with airworthiness certification requirements. It is used in the international Avionics industry as the basis for software certification for commercial

aircraft (Nelson, 2003). DO-178B identifies five different design assurance levels (DALs), A through E, each representing the severity level of a software function. For example, the highest severity is level A and represents software that, if it fails, could cause the entire system to go into a failure state. The standard also identifies a total of 65 objectives for the software being tested. The set of objectives relevant to the software under test depends on its DAL rating (Rushby, 2011).

c. ARINC-653

ARINC-653 is a specification developed by the private entity Aeronautical Radio, Incorporated. This standard is used throughout the avionics industry and is gaining recognition in the space industry (ARINC Standards Store, n.d.; Rufino & Craveiro, 2008). ARINC-653 specifies a standardized interface between an RTOS and its applications (Diniz & Rufino, 2005) and defines a set of functional and certification requirements meant to ensure safety (Rufino & Craveiro, 2008). ARINC-653 is tightly connected to the concept of IMA since it is based on strict spatial and temporal partitioning rules. Spatial partitioning means that partitions have separate address spaces, which cannot be accessed directly by other partitions. Temporal partitioning means that only one application has access to system resources at any given time (Schoofs, Santos, Tatibana, & Anjos, 2009). Figure 1 illustrates the design of a system based on the ARINC-653 specification.

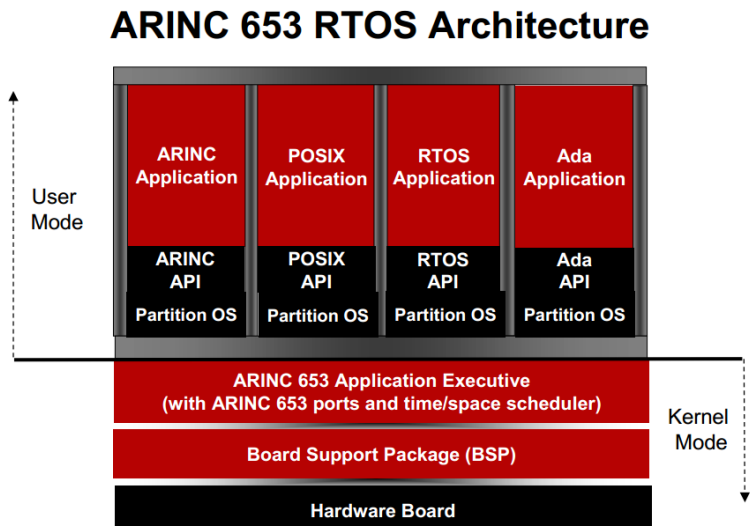

Figure 1. Example Application of the ARINC-653 Specification (from "ARINC 653," 2008)

At the heart of the ARINC-653 specification are two main concepts: the partition and the applications/executive (APEX) layer. The *partition* is intended to be a container for applications running on the operating system, ensuring applications are separated spatially and temporally from one another to avoid fault propagation (Gomes, 2012). Partitions can also be used for system services not available through the APEX interface, like fault management or device drivers (Samolej, 2011).

The *APEX interface* is a standardized application program interface (API) for services available to partitions. This enables hardware to be designed independently of software and allows software to be developed for ARINC-653 partitions, agnostic to the hardware providing this environment (Gomes, 2012). This increases the portability, reusability and modularity of systems, which are all goals of the IMA construct ("ARINC 653," 2008). The APEX has 51 routines that handle the following key functionalities: process management, time management, partition management, inter-partition and intra-partition communication management, and health monitoring ("ARINC 653," 2008). These functionalities enable each partition to manage its own tasks and processes. Communication across partitions is provided through requests to the operating system via the APEX API (Gomes, 2012). The APEX does not provide memory management

services; instead, it assumes memory is statically allocated to partitions at configuration time (Samolej, 2011).

Partitions consist of one or more processes, scheduled according to their priority. Process scheduling is based on the scheduling algorithm determined at configuration time (Han & Jin, 2011). Inter-partition communication is handled through the use of queuing and sampling port communication units, which are objects defined at system integration. *Sampling ports* allow a partition to access a sampling communication channel, in which messages are not stored but, rather, the most recent message overwrites any previous one. In contrast, *queuing ports* allow messages to be queued rather than over-written. Ports are connected via channels when the partitions are integrated, as defined in a configuration file. Intra-partition communication is handled using semaphores, blackboards and buffers; blackboards are similar to sampling ports, and buffers are similar to queues (Diniz & Rufino, 2005). The *health monitor* is a facility that monitors the hardware, OS and applications. The monitor can isolate faults by taking an action (such as restarting a partition) and prevent failures from propagating through the system (Samolej, 2011). The health monitor is meant to identify and manage errors within the system at the process level, the partition level or the module level. Errors are managed through the use of procedures defined by the system developer (Samolej, 2011).

D. VIRTUALIZATION BACKGROUND

In this section, we review definitions associated with virtualization and describe common architectures used to implement virtualization. The DOD's Enterprise Software Initiative defines virtualization as "the separation of a computer operating system's service request from the underlying physical delivery of that service by the hardware" (DOD ESI, n.d.). Tavernes et al. claim that virtualization can be implemented through three primary methods: hypervisor-based, microkernel-based and microvisor-based (Tavares et al., 2012). In this section, we first review the hypervisor approach, with examples of its implementation. We then briefly describe the microkernel and microvisor concepts.

1. Hypervisor Terminology

The term *virtualization* refers to the idea of creating a *software* environment on which multiple programs or operating systems can run, as if they were running on native hardware (Iqbal, Sadeque, & Mutia, 2009). This software environment is an abstraction layer that maps a hosted (guest) system's interface and resources onto an underlying interface and resources, belonging to a different "real" (host) system (Smith & Nair, 2005). The term commonly used to refer to this software abstraction layer is the *virtual machine monitor* (VMM) or *hypervisor*. The hypervisor acts as mediator between a host system's hardware and the various guest environments running on the hypervisor, called *virtual machines* (VMs). VMs are isolated from one another, coordinated in their resource use by the underlying hypervisor (Chiueh & Brook, 2005).

Popek and Goldberg (1974) define two primary types of hypervisors: type-1 (or native) and type-2 (or hosted). Type-1 hypervisors run directly above the host system's hardware and provide all VM resources. Type-2 hypervisors operate on top of a host environment and are dependent on this underlying OS for maintenance and distribution of resources. For example, type-2 hypervisors cannot boot until the host operating system has booted and, in the event the host operating system crashes, so too does the type-2 hypervisor (Jones, 2010). Figure 2 illustrates type-1 and type-2 hypervisors.

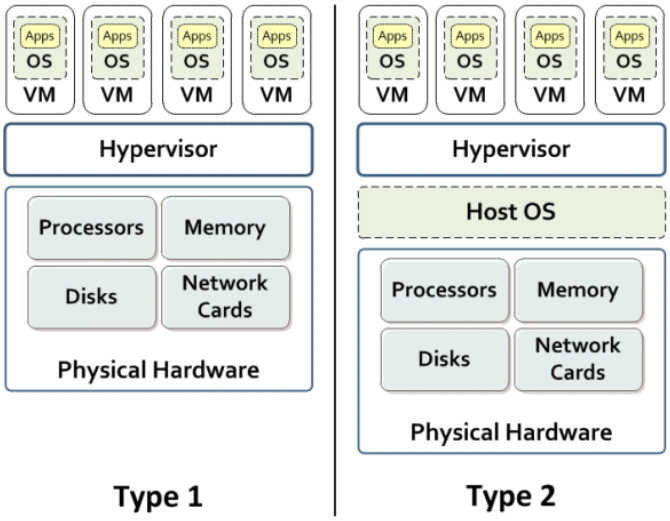

Figure 2. Example Type-1 and Type-2 Hypervisors (from Baliyase, 2014)

2. Full Virtualization Architectures

Full virtualization, illustrated in Figure 3, allows different operating systems to run unmodified on either type-1 (e.g., VMware ESXi) or type-2 (e.g., VMware Workstation) hypervisors. The hypervisor emulates the host platform, such that the VM and its applications run without any modification and without knowing that they are running on a virtualized platform (Jones, 2010). The hypervisor is responsible for emulating devices with which the VMs interact, providing VMs access to virtual hardware devices. When a VM wants to interact with a virtual device, requests from the VM are handled by the hypervisor (Kirch, 2007). The hypervisor, in turn, interacts with the hardware via a host operating system driver (for type-2 hypervisors) or a hypervisor driver (for type-1 hypervisors) (Sahoo, Mohapatra, & Lath, 2010).

In full virtualization binary translation converts privileged machine code from the VM to the hardware. Binary translation is a process whereby the hypervisor scans a VM's memory for privileged instructions before they are executed, and dynamically modifies these into code that the hypervisor can emulate for the hardware (Binu & Kumar, 2011). Full virtualization tends to have high overhead due to the need to translate machine code, and the frequency of traps between the VM and the hypervisor (Jeong, 2013).

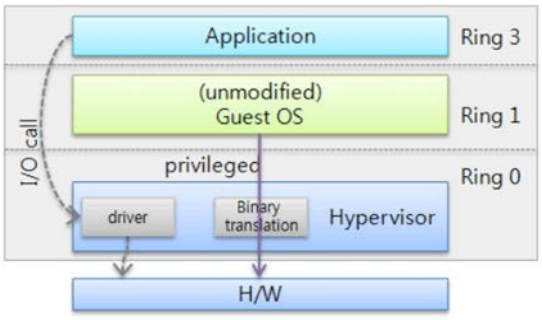

Figure 3. Example of Full Virtualization (from Jeong, 2013)

3. Paravirtualization Architectures

Paravirtualization differs from full virtualization in the way communication between the VMs and devices is handled. In full virtualization, the hypervisor fully

emulates devices and translates privileged instructions without the guest OS being modified; in paravirtualization, the guest OS has been modified to run virtualized. This modification allows the VM to relay instructions through the hypervisor without requiring that the hypervisor first translate them. For example, paravirtualization, illustrated in Figure 4, can be implemented using a privileged VM to handle input/output (I/O) requests from other guest VMs. The privileged VM is equipped with a "back-end" driver that can access the hardware, while the other VMs are equipped with "front-end" drivers (Binu & Kumar, 2011). When a VM wants to execute an I/O instruction, it uses its front-end driver proxies to relay the instruction to the back-end driver. The hypervisor does not need to scan for privileged instructions during operation; instead, the paravirtualized guest has been modified to send requests to the back-end driver.

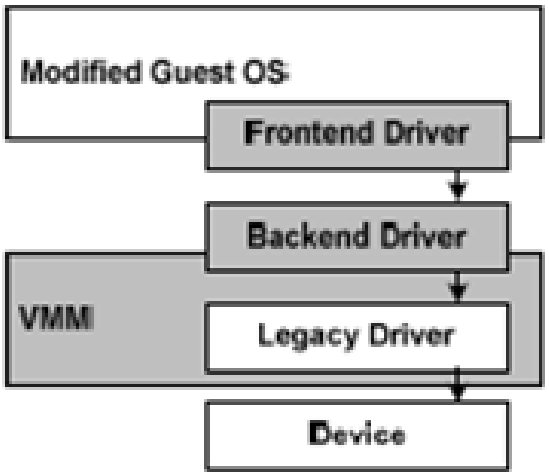

Figure 4. Implementation of Paravirtualization (from Binu & Kumar, 2011)

The modified instructions used by paravirtualized guest OSs are called *hypercalls*. Hypercalls are software traps from the VM's virtual driver to the hypervisor (LeVasseur et al., 2005; "Xen Hypercall," n.d.). Paravirtualization tends to be simpler and faster than full virtualization but has considerable engineering cost, since each guest OS is modified to be aware that it does not run on native hardware (Barham et al., 2003).

4. Software Emulation Architectures

Emulation is a process whereby the physical hardware platform, such as ARM or PowerPC, is emulated by the hypervisor (Murphy, n.d.) as illustrated in Figure 5. Here, the hypervisor emulates different instances of hardware, such as the processor and I/O devices, used by separate VMs. The hypervisor translates the instruction set architectures (ISA) of an emulated processor into the ISA of the underlying platform. In software emulation, every instruction issued by the VM is interpreted by the emulator layer (Chiueh & Brook, 2005; Jones, 2010).

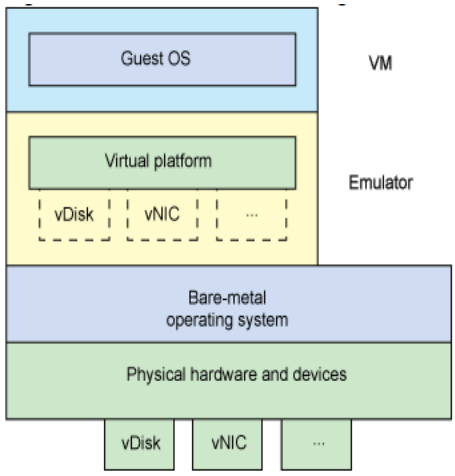

Figure 5. An Illustration of the Emulation Concept (from Jones, 2011).

5. Hardware-Assisted Virtualization Architectures

Hardware-assisted virtualization refers to changes that have been made directly in hardware to better accommodate virtualization. With hardware-assisted virtualization, extensions have been added to CPUs and their ISAs so that certain virtualization procedures, such as binary translation or paravirtualization via hypercalls, are unnecessary. Instead, privileged instructions can be trapped and emulated by the hardware directly, instead of by the hypervisor (Jones, 2010; "Understanding Full Virtualization," 2007).

6. Example Architectures

In this section, we review the landscape of virtualization technologies. We discuss five well-known virtualization products—VMWare Workstation, Xen, Qemu, KVM and VMWare ESXi—employing these as points-of-comparison in our survey, later.

a. VMWare Workstation

VMWare Workstation is VMWare's full virtualization architecture, designed to run on individual PCs. VMWare Workstation runs as a type-2 hypervisor and is designed to work with x86 host systems. When VMWare Workstation is installed, three components are created: the VM Driver, the VMM and the VMWare Application or VMApp (see Figure 6). Both the VMM and the VM Driver operate at the same privilege level as the host OS, while the VMApp runs at the level of the guest VM (above the VMM). The VMM is an application running on the host OS. When a user executes the VMApp, it works with the VMDriver to load the VMM into the host's kernel memory. Once loaded, the host operating system is only cognizant of the application and the driver and not the VMM. The VMM communicates directly with the hardware, and the host operating system via the VMDriver (Munro, 2001).

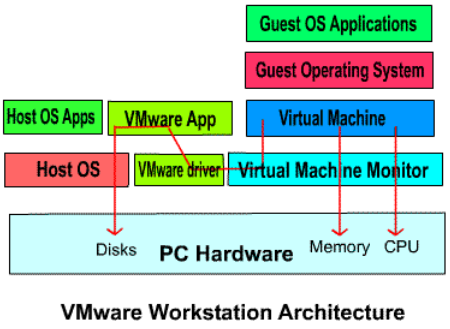

Figure 6. VMWare Workstation Architecture (from Munro, 2001).

Non-privileged instructions executed on the guest OS are sent through the VMM directly to the host system to be processed. Privileged instructions, however, are trapped by the VMM and translated via binary translation. The VMDriver then facilitates a transfer so that the VMM can communicate with the host OS. Once in the "host world,"

the VMApp-translated instructions are communicated via the VMApp to the host OS, which executes the instruction (Rosenblum & Garfinkel, 2005; Chiueh & Brook, 2005; USENIX, 2001).

b. XEN

Xen, illustrated in Figure 7, is an open-source, type-1 hypervisor for x86 platforms that utilizes paravirtualization but also supports full virtualization and hardware-assisted virtualization. Xen operates directly on top of the host hardware, in a higher privilege level than all but one of its VMs. The Xen hypervisor creates a distinguished VM at boot time, the Domain 0 or Dom0 VM, which is privileged and responsible for various management tasks (see Figure 7). The Dom0 VM, through its ability to interact directly with host hardware and provide interfaces for other VMs, is able: to create and kill other VMs, to control their physical memory allocations, to control a VM's access to various underlying physical resources, such as the hard disk and shared network devices, and to manage the I/O of each VM. The Dom0 VM is the only domain that is able to access the hardware directly ("DomU," n.d.).

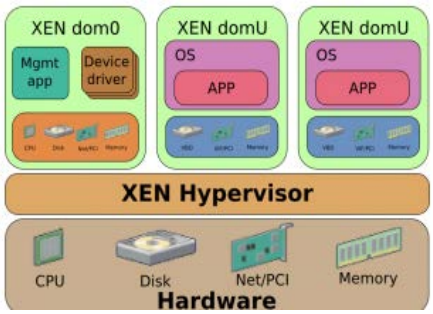

Figure 7. Xen Architecture (from "Virtualization," 2013)

To more efficiently handle privileged instructions from a guest OS instance to the VMM, Xen requires that each paravirtualized guest OS is modified so that privileged instructions are replaced with calls to the Xen hypervisor. VMs communicate directly with the Xen hypervisor through hypercalls to perform privileged operations (Barham et al., 2003; Binu & Kumar, 2011).

c. QEMU

Qemu is a software-based hardware emulator that can run multiple instances of itself on top of a host operating system. Each instance of Qemu can be viewed as a hypervisor, emulating a system (see Figure 8). Qemu is capable of emulating several different CPUs, including x86, PowerPC, ARM and SPARC. Qemu consists of several subsystems, including a CPU emulator, emulated devices (VGA display, the mouse, keyboard, network card), a user interface and a debugger. These subsystems allow for the complete simulation of an unmodified guest running on top of emulated hardware.

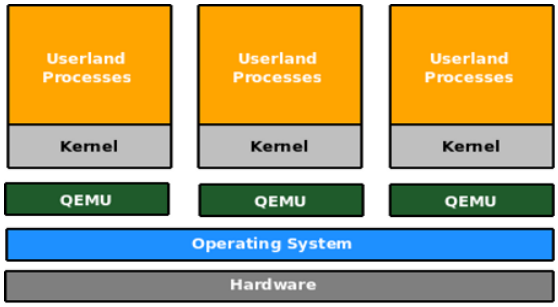

Figure 8. Qemu Architecture (from Hussein, 2009)

Emulation in Qemu is carried out using a process called *dynamic binary translation*, to translate guest CPU instructions into host instructions. Translation occurs at runtime and the result is stored in a fixed-size cache for reuse later. By using a cache, frequent instructions do not need to be translated multiple times. The process by which frequently used instructions are saved for reuse, to avoid translation overhead, is called *dynamic recompilation* (Landley, 2009). There are several steps in Qemu's dynamic translation process. First, guest instructions are broken into "micro operations." The purpose of this is to simplify the translation logic, allowing for repeated use of translated micro-operations. Each micro operation is implemented individually, written in C and compiled by GCC to create native, object files. The object files are used by Qemu's *dyngen* utility, a compile time tool that uses the object file as input to a dynamic code generator. The code generator is invoked at runtime to create the machine code used by the host (Bellard, 2005; Chiueh & Brooks, 2005).

d. KVM

Kernel virtual machine (KVM) is an open-source, hardware-assisted virtualization architecture that supports paravirtualization (see Figure 9). KVM requires Intel VT-X or AMD-V enabled CPUs and makes use of their CPU extensions (Habib, 2008). The KVM VMM is essentially a modified Linux kernel module designed to operate as a hypervisor. Each VM running on KVM is a Linux process, which can be managed like any normal Linux process. Whereas normal Linux processes operate in either user mode or kernel mode, KVM enables a third "guest mode." Processes in guest mode run from within the KVM VM (Habib, 2008). Since each VM is a Linux process, they can leverage all the features available within the Linux kernel. For example, SELinux and sVirt can be employed to implement security features to constrain KVM VMs (processes). KVM VMs use Qemu for I/O (Qemu, n.d.), which is employed as a user-space process inside the VM (Habib, 2008). Memory for each VM can be shared by using the Kernel same-page merging (KSM) feature, which scans each VM's memory space and consolidates identical memory pages (Zhang et al., 2010).

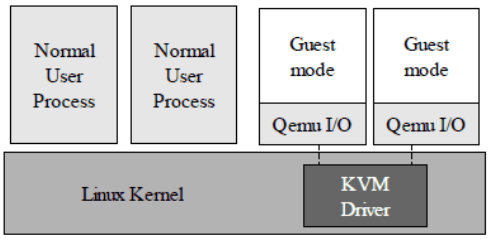

Figure 9. KVM Architecture (from Virtualization Station, 2008)

e. VMware ESXi

VMware ESXi is a type-1 hypervisor (vSphere ESXi, n.d.). The primary component of VMware ESXi is the VMKernel (see Figure 10). This controls all interaction with the hardware, and is designed with the sole purpose of managing and controlling the VMs. In addition to the VMs that run above VMKernel, several processes also run on top of the VMKernel to help with VM management. One of these processes is a VMM process, which provides the execution environment for the guest operating

system running within each VM. There is one VMM instance per VM. The VMM process is an intermediary process allowing guests to interact with the resources controlled by the VMKernel (Mishchenko, 2010). Each instance of the VMM process utilizes a helper process, called the VMX, which handles I/O to non-critical devices and communicates with the user interfaces and remote consoles ("VMWare Knowledge Base," n.d.). Additional processes that run above the VMKernel are the Direct Console User Interface (DCUI) and the Common Information Model (CIM) server. The DCUI is a low-level management interface used for initial configuration of the ESXi hypervisor. The CIM server enables remote monitoring of the ESXi server and the VMs it manages, implementing a standard CIM API for remote CIM clients (Fujitsu, 2010).

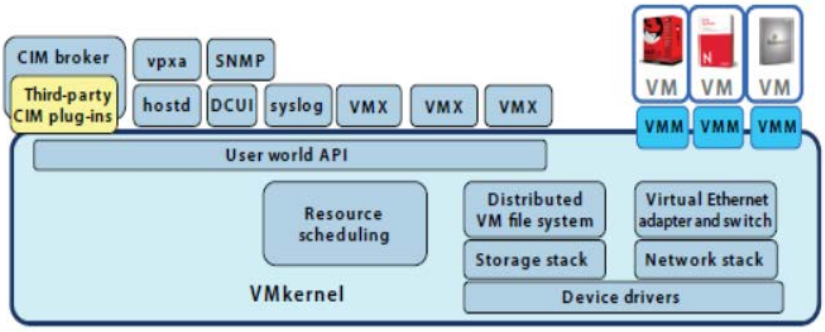

Figure 10. ESXi Architecture (from "The Architecture," n.d.).

7. Microkernel and Microvisor

A microkernel is a small software layer over hardware, providing services to processes and operating systems in a less privileged domain ("Microkernel Architecture," n.d.; Douglas, 2010). A hypervisor's main responsibility is to implement virtual machines that run at a lower privilege level than the hypervisor; in contrast, a microkernel is a small base on which other systems can be built (General Dynamics, 2008). In particular, a hypervisor may be implemented on top of a microkernel.

Armand and Gien suggest that the use of microkernels is motivated by the increasing complexity of operating systems (Armand, 2009). Microkernels are well suited for use in embedded systems, which are often not designed to support a full-featured,

monolithic kernel. Microkernels allow systems to be designed in less complex ways and in a more modular fashion since less functionality is included at the kernel level (Armand, 2009). Security is another motivation for the development of microkernels. Iqbal et al. observe that microkernels support the principle of least privilege: functionalities at higher privilege levels are as limited as possible (Iqbal et al., 2009). Only essential tasks, such as low-level address space management, thread management and inter-process communication are handled by the microkernel.

The term “microvisor” is used to refer to a microkernel that supports virtualization (Iqbal et al., 2009; General Dynamics, n.d.). The term first appears in reference to the OKL4 microvisor in 2010. The OKL4 microvisor is designed to support both full operating systems, as well as applications, and can support real-time and non-real-time software (Heiser &Leslie, 2010). Next, we discuss the real-time operating systems that can be used in space systems.

THIS PAGE INTENTIONALLY LEFT BLANK

III. REAL-TIME OPERATING SYSTEMS FOR SPACE

In this chapter, we survey real-time operating systems currently being used or proposed for use in the space domain. We discuss how each RTOS is used in space and review the high-level design of the operating system, task or process management, scheduling and memory management. We then discuss the compatibility of each RTOS with virtualization architectures, its compliance with space standards and offer an analysis of its use in space system development.

A. SCOPE

The purpose of our survey work is to review fundamental RTOS designs and identify different methods of implementing key functionalities (see Table 2). Some RTOSs have been excluded from this study, due to lack of industry adoption or lack of available system information. This includes eCos ("Home Page," n.d.), ThreadX ("ThreadX," n.d.), Wind River Linux ("Wind River Linux," n.d.), QNX (QNX, n.d.), Deos, HeartDeos ("A Time," n.d.), and Salvo ("Welcome," n.d.). Table 2 summarizes the pertinent attributes of an RTOS.

Table 2. RTOS Attributes Chart

RTOS	License	Supported Languages	Supported APIs	Relevant Standards	Hardware Support	Security Modes	Memory Footprint (kernel)	Memory Protection	Scheduling	Performance Evaluation	Task[1] Execution Mode
RTEMS	Open-source (GNU GPL)	C, Ada, C++, Java, Go, Lua	POSIX, BSD Sockets, SAPI, Classic RTEMS API	None (Space Qualified version is GSWS qualified)	ERC32, LEON, ARM, Pentium, x86, MIPS, PowerPC	supervisor (On-Line Applications Research Corporation, 2013)	~1200MB (Evans, 2007)	None	Round robin, fix priority, earlierst deadline first, constant bandwidth, simple SMP, partitioned/cluster scheduler	Yes	Privileged
FreeRTOS	Open-source (Modified GNU GPL)	C	FreeRTOS API	None (SafeRTOS is DO178-B certified)	x86, Xilinx, ARM, PIC, Freescale	user, supervisor (PowerPC)	~5-10KB	Use of hardware MPU on Cortex-M3 and ARM processors	Priority based preemptive, cooperative, hybrid	No	Privileged
PREEMPT_RT	Open-source (GNU GPL)	All Linux	POSIX	None	all Linux	user, kernel	~100MB	None	FIFO, Round Robin, Batch, Idle, Other	Yes	User
RTLinux	Open-source (GNU GPL) or Commercial	All Linux	POSIX	None	all Linux	user, kernel	~9MB (Computer as a controller, n.d.).	None (Pettersson & Svensson, 2006)	FIFO, Round Robin, Batch, Idle, Other	Yes	Kernel

[1] The term “task” refers to the basic unit of execution for an RTOS.

RTOS	License	Supported Languages	Supported APIs	Relevant Standards	Hardware Support	Security Modes	Memory Footprint (kernel)	Memory Protection	Scheduling	Performance Evaluation	Task[1] Execution Mode
RTAI	Open-source (GNU GPL)	All Linux	RTAI Native API, POSIX	None	ARM, x86, PowerPC	user, kernel	~4.5MB (size of latest tar file)	None (though use of LXRT[2] module allows applications to be written in user space) (Contributing Editor, 2001)	FIFO, round robin, dual scheduler (RT-microkernel and userland non-RT kernel)	Yes	Kernel
Xenomai	Open-source (LGPL)	All Linux	Xenomai Native API, POSIX (skin)	None	ARM, BlackFin, x86, PowerPC, Nios 11 ("Embedded Hardware, n.d.)	user, kernel	~20MB (size of stable release tar file)	Mmap POSIX facility	FIFO, Round Robin, Sporadic, TP, other	Yes	Primary, secondary[3]
VxWorks	Commercial	C, C++, Ada, Java	VxWorks API, POSIX	Customizable to be DO-178B certified	ARM, FreeScale, MIPS, Pentium, x86, etc.	user, kernel	~20KB	Hardware MMU support configuration options; stack protection; POSIX	Round Robin, preemptive priority-based	Yes	Privileged or user (RTP tasks[4] run in user mode)
VxWorks 653	Commercial	C, C++, Ada, Java	POSIX, VxWorks API, ARINC-653	ARINC-653	FreeScale; PowerPC, Intel IA-32	User, kernel	Unknown	POSIX memory lock facility	ARINC time-preemptive scheduling; priority-preemptive scheduling	No	Supervisor, user (partitions run in user mode)

[2] LXRT is an RTAI module that allows real-time tasks to be developed and run in user space. LXRT processes can be migrated to kernel space.

[3] Primary mode is equivalent to kernel mode and secondary mode is equivalent to user mode.

[4] See "VxWorks" section where RTPs are discussed.

RTOS	License	Supported Languages	Supported APIs	Relevant Standards	Hardware Support	Security Modes	Memory Footprint (kernel)	Memory Protection	Scheduling	Performance Evaluation	Task[1] Execution Mode
INTEGRITY-178B	Commercial	C, C++, Ada	ARINC-653; Integrity Kernel API	DO178-B; SKPP High Robustness	x86, PowerPC, ARM, MIPS, FreeScale etc.	supervisor, user	Unknown	Acccess verification; processor MMU support	ARINC-partition scheduler (preemptive scheduler)	No	Privileged or user
LithOS	Open-source (Unknown)	Unknown	ARINC-653	Unknown	x86	Unknown	Unknown	unknown	Whatever is defined at configuration	No	Unknown
LynxOS-178	Commercial	C, C++	ARINC-653; POSIX	DO-178B	x86, PowerPC	User, kernel	Unknown	POSIX memory lock facility	FIFO, round robin, priority-based quantum (proprietary)	No	User, kernel

B. VXWORKS

VxWorks is a proprietary suite of software products designed for embedded systems with real-time requirements. VxWorks is developed and maintained by Wind River Systems. Wind River offers software "platforms" tailored in different ways for specific industries, such as aerospace ("6.9 Platform," n.d.); for certain architectures, such as MILS and ARINC653 ("MILS Platform," n.d.; Parkinson, n.d.); or for certain certifications, such as DO-178B ("Cert Platform," n.d.). These platforms all include the VxWorks Operating System, a development environment called the VxWorks Workbench, and optional middleware based on the platform. Both the MILS platform and the ARINC653 platform include modified versions of the standard VxWorks Operating System. VxWorks is compatible with over 124 different processors ("Board Support Packages," n.d.) including the MIPS and PowerPC processor families. VxWorks has its own API but is also fully POSIX compliant.[5] VxWorks provides an IPv4/IPv6 network stack that has undergone third party testing and validation to ensure high performance ("6.9 Platform," n.d.). This network stack was cited as being a key factor in the ESA's use of VxWorks on the European Geostationary Navigational Overlay System, a navigational space satellite mission (Parkinson, n.d.).

Over the past 20 years, NASA has used VxWorks in a number of its missions ("VxWorks Space," n.d.). VxWorks 5.3.1 was used on a MIPS processor by the Mars Exploration Rover ("VxWorks," n.d.). Other versions of the operating system are being used on other missions including the Cygnus Spacecraft, an unmanned cargo transport vessel where VxWorks is running on the main flight computer ("Genesis," n.d.). VxWorks is also being used to control the flight computer of the MESSENGER probe, an unmanned spacecraft orbiting Mercury ("Messenger," n.d.; "VxWorks Space," n.d.). SpaceX, the private space travel company, uses an unspecified VxWorks platform on its Dragon reusable spacecraft ("SpaceX," n.d.).

[5] Supports the 1003.1 standard but does not provide process creation capability with fork() or exec() or file ownership and file permissions.

The VxWorks operating system is tightly coupled with the additional software products designed for embedded systems that Wind River offers. As such, the operating system is compatible with the Wind River Hypervisor. VxWorks can also run as an unmodified guest operating system on the Green Hills Multivisor ("Integrity Multivisor," n.d.).

Wind River offers a suite of highly customizable and modular software products with different design features based on the certifications or architectures required. As such, there is no set of standards with which the core VxWorks operating system alone complies. Wind River offers separate products, such as VxWorks653 that complies with the ARINC-653 specification, and the VxWorks CERT platform that complies with the DO-178 standard ("Profiles," n.d.).

1. Design

Conceptually, VxWorks reflects the "process model" similar to UNIX and Linux, whereby kernel space and user space are clearly delineated and the applications that run in these two spaces run at different privilege levels ("6.9 Guide," n.d.). VxWorks can be configured as a micro-kernel, a basic kernel or as a full-featured operating system. It is unclear which versions of the operating system are commonly used in spacecraft but documentation does confirm that VxWorks has been used in space systems of different sizes, such as microsatellites (Teston, Vuilleumier, Hardy, & Bernaerts, 2004) and unmanned spacecraft ("CIRA," n.d.), which might indicate the use of different VxWorks configurations in space systems. Figure 11 illustrates the various capabilities included in each configuration.

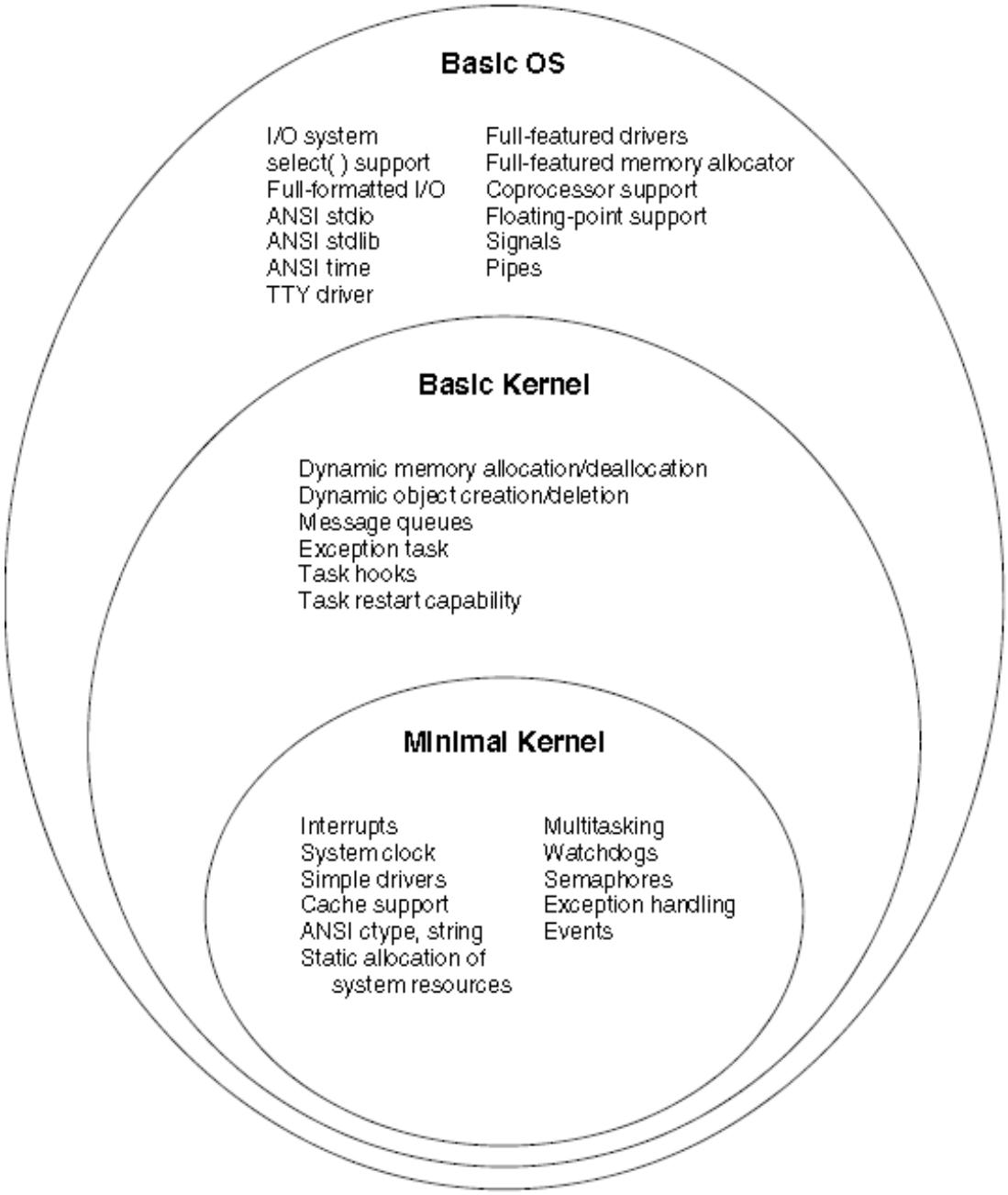

Figure 11. VxWorks Kernel Scale Options (from "6.9 Guide," n.d.)

a. Task Management

The basic unit of execution in VxWorks is a thread, which VxWorks refers to as a task. VxWorks refers to processes as Real-Time Processes or RTPs, which are a collection of tasks grouped by function. Support for RTPs is an optional configuration in

VxWorks. If RTPs are supported then tasks within RTPs run in user mode. If RTPs are not supported then VxWorks tasks run at the highest privilege on the processor[6] ("VxWorks Architecture," 2005).

b. Scheduling Management

VxWorks supports three types of task schedulers, listed in Table 3. For all schedulers, the default scheduling option is priority-based preemptive scheduling in which a higher priority task can preempt a lower priority task to run.

Table 3. VxWorks Supported Schedulers (from "6.9 Guide," n.d., p. 138)

Task Scheduler	**Use**
Traditional VxWorks Scheduler	Scheduling policy enforced across the system (kernel and user mode) with either a priority-based, preemptive policy or a round-robin policy
POSIX Thread Scheduler	Schedules POSIX threads (pthreads) within real-time processes and applies scheduling policies on a thread-by-thread basis
Custom Scheduler	Developer can define own scheduler

c. Memory Management

VxWorks supports memory protection on processors with or without MMUs. RTPs have their own region of virtual address space that is not shared with any other process; this allows memory to remain isolated if VxWorks is running on a processor that does not have an MMU ("6.9 Guide," n.d.).

VxWorks also offers a proprietary mapping facility called *sdLib*, which enables RTP applications to share memory through a shared data region. Once established, user-mode applications and kernel tasks have access to these shared data regions ("6.9 Guide," n.d., p. 66).

[6] This applies to ARM, Intel and SuperH processors. On MIPS processors, if RTPs are not supported, tasks run in kernel mode.

2. Analysis

VxWorks is a legacy RTOS that has proven its ability to perform on space missions for a number of years. Reliability is a major decision factor for use in space systems given the time and money involved in validating a new system. Space system developers tend to choose VxWorks due to its proven reputation on high profile space missions (“CIRA, n.d.; Volpe et al., 2000, p. 30).

VxWorks is a modular RTOS that can be configured in various ways depending on the processor on which it runs and the applications it hosts. This is evident in the different options available for task management, scheduling and memory management. Flexible configuration options are another factor that space system developers cite as being essential when choosing an RTOS to use for space system development (Beus-Deukic, 2001).

VxWorks also offers a familiar development environment, which NASA’s Joint Propulsion Laboratory cited as being a factor in the Mars Curiosity Rover mission success. The VxWorks programming interface that is similar to UNIX, and its POSIX compatibility helped NASA developers develop and debug during development since their work took less time and existing code could be reused (“NASA’s Mars,” n.d.).

C. REAL-TIME LINUX

There are several projects dedicated to making Linux capable of handling real-time requirements (“Introduction to Linux,” 2002). These projects offer different solutions to making Linux an RTOS. One approach taken by the RTLinux, Xenomai and RTAI projects is to develop a software layer below the Linux kernel that handles real-time requirements. A second approach, taken by the CONFIG_PREEMPT_RT community (“Real-Time Linux Wiki,” n.d.), is to improve the existing Linux kernel to meet real-time requirements with the PREEMPT_RT patch (McKenney, 2005; Opdenacker, 2004; Clark, 2013). Each version of real-time Linux comes in the form of a patch to the standard Linux kernel. With this approach, the portability of these RTOSs to various hypervisors is comparable to main line Linux.

To the best of our knowledge, the different implementations of real-time Linux run on all of the virtualization architectures surveyed in this thesis. The implementations of real-time Linux do not comply with any space standards and the developers are open about the fact that there are no guarantees with the real-time Linux code.

In 1999, NASA initiated a project called FlightLinux to assess Linux's readiness for space systems. Though the program ended in 2002, many advantages to Linux's use in space were identified, including the ease of developing applications required for missions and the relative ease of developing features, such as adding fault tolerance into the existing software (Katz & Some, 2003). Since the FlightLinux project, Linux has been used in a number of space missions (Edge, 2013; "Five Ways NASA," n.d.). RTLinux was used in a hurricane data system for NASA's Goddard Space Flight Center. In this project, RTLinux was responsible for aircraft attitude correction and a number of other tasks related to data collection (Wright & Walsh, 1999). RTAI is currently being used by NASA's McDonald Laser Ranging Station for its range control activities, including locating satellites in orbit (Ricklefs, n.d.). Xenomai is used by NASA's robotics developers to develop a robotic machine to perform tasks in space (Krüger, Schiele, & Hambuchen, 2013).

In a 2013 presentation, Keven Scharpf of the PTR group cited the PREEMPT_RT patch as a viable solution to hard-real-time requirements for space systems. The PTR group has worked on a number of space missions, including the Tacsat-2 microsatellite mission, which was the first mission to use Linux in space (Scharpf, 2013). Wind River also makes use of the PREEMPT_RT patch in its WindRiver Linux 4 and 6 products ("Wind River Linux 4," n.d.; "Wind River Linux 6," n.d.).

1. RTLinux, Xenomai, and RTAI

RTLinux, Xenomai, and RTAI are all designed as "dual kernel" configurations. These operating systems have some minor differences, but their fundamental approach to making Linux real-time is the same. We will focus on the architecture of RTLinux for the remainder of this section.

In RTLinux (see Figure 12), a microkernel extension is added to the Linux kernel (Opdenacker, 2004). This extension is a set of Linux kernel modules that deal specifically with real-time tasks by providing a subset of the POSIX API ("RTLinux," n.d.). With this alteration to the standard Linux kernel, a second real-time microkernel, i.e., RTLinux Kernel, is placed under the standard Linux kernel, which runs as an idle task on top of the RTLinux Kernel (Balasubramaniam, n.d.). Real-time applications are created as modules that run on the RTLinux Kernel and are written using a subset of the POSIX API, based on the POSIX Minimal Realtime System Profile, or PSE51 (Terrasa, Garcia-Fornes, & Espinosa, 2002).

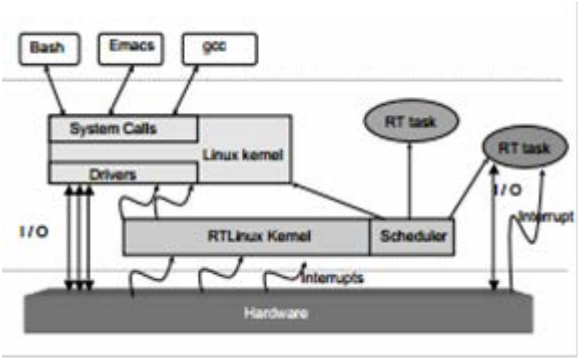

Figure 12. Illustration of RTLinux Design (from Balasubramaniam, n.d.)

a. Task Management

All real-time tasks run at kernel privilege level and have direct access to the hardware. All interrupts are intercepted by the RT-microkernel, which decides what to do. If these interrupts have real-time handlers, then the RT-microkernel schedules them first (Yodaiken, 1999).

b. Scheduling Management

The RT-microkernel has its own scheduler that is responsible for scheduling both real-time and non-real-time tasks (Yodaiken, 2001). This scheduler is generally a preemptive priority based scheduler with tasks having their priority statically determined.

c. Memory Management

In RTLinux and Xenomai, real-time tasks are allocated fixed amounts of memory for data and code (Balasubramaniam, n.d.) and do not use virtual memory (Yodaiken, 2001). RTAI on the other hand, uses dynamic memory allocation (Balasubramaniam, n.d.). For all three dual-kernel configurations of Linux, the real-time applications running on top of the RT-microkernel share a common address space (Haas, n.d.).

2. PREEMPT_RT

The PREEMPT_RT patch (see Figure 13) makes the Linux kernel fully pre-emptible through optimizations inside the kernel. The patch is sometimes referred to as RT-PREEMPT, PREEMPT-RT, CONFIG_PREEMPT_RT or CONFIG_PREEMPT ("Real-Time Linux Wiki," n.d.). Unlike RTLinux, RTAI and Xenomai, PREEMPT_RT does not include a separate kernel to handle real-time tasks. The goal of the PREEMPT_RT project is to make the existing Linux kernel 100% pre-emptible (Rostedt & Hart, 2007, pp. 161–172).

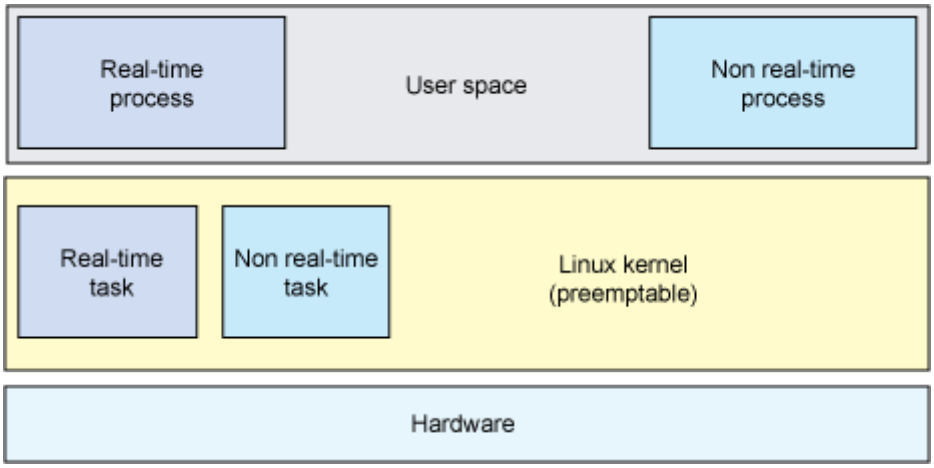

Figure 13. Illustration of PREEMPT_RT Modification to Linux Kernel (from Jones, 2008)

a. Design

The PREEMPT_RT patch allows the Linux kernel to become a predictable and deterministic operating system (Rostedt & Hart, 2007). This is done by doing two things:

using threads to service selected device interrupts and replacing existing spin locks with mutexes that are preemptive and support priority inheritance (Fayyad-Kazan, 2014).

b. Task Management

Using separate threads to service device interrupts reduces interrupt latencies, allowing a higher priority task to not be significantly affected by a lower priority task, which causes heavy I/O interrupts (Rostedt & Hart, 2007). Mutexes in the PREEMPT_RT patch prioritize the tasks waiting for the resource (Moore, 2005).

c. Scheduling Management

The PREEMPT_RT patch does not include any modification to the schedulers already available in the standard Linux kernel.

d. Memory Management

The PREEMPT_RT patch does not include any additional memory management functionalities that are not already in use in the standard Linux kernel.

3. Analysis

There is a lot of discussion within the space community regarding Linux's suitability for space systems. Prieto et al. (2004) cite a number of reasons why Linux is an attractive operating system for space. One factor is the time that can be saved in testing and debugging since developers are very familiar with the software environment. Another reason Linux is attractive is because the development platform for building applications can mirror the actual software environment in space. The open source community's involvement in software debugging and problem solving is also a resource that Prieto claims can be incredibly helpful (Prieto et al., 2004).

The Naval Research Laboratory cited that Linux was used on its TacSat-1 spacecraft, primarily because accessibility to source code was vital for debugging purposes and because of the ease of migrating development software on x86 platforms to the actual PowerPC space processor. The TacSat-1, however, did not have any hard real-

time requirements, which was a reason why NRL chose Linux as opposed to a proprietary RTOS like VxWorks (Huffine, 2005).

Linux is an attractive operating system for space systems given its widespread use and legacy reliability in terrestrial systems. The real-time Linux projects surveyed offer features like task prioritization and bounded latencies that provide useful determinism for space systems. The projects however, have not been certified to any space standard and the developers make no guarantee that the real-time Linux projects are suitable for hard real-time systems. Key safety features like memory protection or static scheduling policies (in PREEMPT_RT) are only as good as the standard kernel.

D. GREEN HILLS INTEGRITY-178B

INTEGRITY-178B is a proprietary, ARINC-653 compliant, DO-178B Level A certified RTOS developed and maintained by Green Hills Software. The INTEGRITY 178B separation kernel was certified to be compliant to the Separation Kernel Protection Profile under the U.S. Common Criteria evaluation scheme (Green Hills, 2008).

It is unclear from published literature what hypervisors INTEGRITY-178B can run on as a guest. Green Hills Software has a virtualization platform called INTEGRITY-Multivisor (see Chapter IV). In no descriptions of this platform is INTEGRITY-178B mentioned as a possible guest VM ("Integrity Multivisor," n.d.).

NASA selected INTEGRITY-178B to operate the flight control module and the backup emergency controller on the Orion Crew Exploration Vehicle, a space vessel designed to carry astronauts to the moon. NASA chose INTEGRITY-178B since it was considered the most mature RTOS and was the most cost-effective ("NASA's Orion," 2008). INTEGRITY-178B is also used on NASA's Pad Abort Demonstrator, a test bed platform meant to evaluate emergency abort scenarios for spacecraft crewmembers on the International Space Station ("Green Hills Software," 2003; "Pad Abort," 2003).

1. Design

Green Hills INTEGRITY-178B's design is based on a secure separation architecture, which implements five different principles: minimal implementation,

componentization, least privilege, secure development process and independent expert validation ("Secure Separation," n.d.). The INTEGRITY-178B separation kernel (see Figure 14) separates resources into partitions and isolates these partitions from one another. Applications of different criticality level can run within these partitions and the kernel ensures that a lower priority application cannot interfere with a higher priority application.

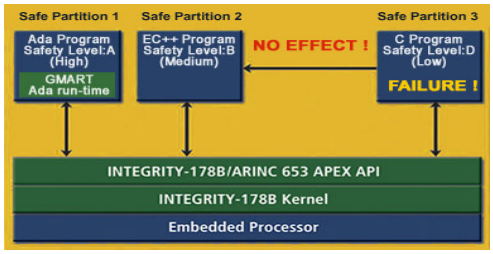

Figure 14. INTEGRITY-178B Design (from "Safety Critical Products," n.d.)

a. Task Management

INTEGRITY-178B is an object-oriented OS, meaning that the various functionalities are treated as objects as opposed to actions. The core objects supported by INTEGRITY-178B and their purpose are listed in Table 4. Each task (subject) is associated with a single AddressSpace, which is a block of memory addresses.

Table 4. INTEGRITY-178B Objects

OBJECT	PURPOSE	HOW DEFINED
AddressSpace	Defines a partition; supports task management	Statically
Task	Task management	Statically
MemoryRegion	Memory management	Statically
Link	Access management	Statically
IODevice	I/O management	Statically
Connection	Synchronous and asynchronous communications	Statically
Activity	Task management; asynchronous communications	Statically or dynamically
Semphore	Task synchronization	Statically or dynamically
Clock	Time Management	Statically or dynamically

All tasks associated with a partition (i.e., an AddressSpace) have an identifier that links it to its AddressSpace. This task identifier is used for authentication purposes. The task identifier is used to enforce authorized information flow and resource sharing. Tasks within a partition can freely access resources allocated to the partition, but if a task tries to access resources from a different partition, the task will be terminated. Access policies for each AddressSpace are defined at configuration time.

b. Scheduling Management

The INTEGRITY-178B scheduler manages the execution of the tasks allocated to the configured partitions. Since INTEGRITY-178B is ARINC-653 compatible, it adheres to a partition schedule that is statically defined. Each partition is allocated a block of time in which its tasks can be executed. AddressSpaces can be allocated a specific block of processor time or can be combined with other partitions that then share processor time.

c. Memory Management

The INTEGRITY kernel runs in a physical address space and leverages the processor MMU to manage the virtual address spaces allocated to the partitions. Each partition has its memory and data statically assigned. INTEGRITY does not support dynamic memory allocation.

2. Analysis

INTEGRITY-178B is the only RTOS surveyed that has a separation kernel that has undergone formal verification and been proven to perform at “high robustness” levels by the National Information Assurance Partnership evaluation scheme. Security and safety design considerations, such as memory protection, ARINC-653 scheduling compliance and access policies for tasks are built into the RTOS, which make it suitable for safety-critical missions. The RTOS is also a proven RTOS for space systems, given its use in NASA missions.

Arguably, INTEGRITY-178B offers less in the way of flexibility than VxWorks or Linux. The RTOS relies on the processor MMU for memory protection so is not suitable for processors without an MMU, such as the ERC32 and does not support

dynamic memory allocation. INTEGRITY-178B also does not have as extensive a track-record within the space community compared to VxWorks (Cudmore, 2007).

E. FREERTOS

FreeRTOS is an open-source RTOS developed and maintained by the British company Real Time Engineers LTD. FreeRTOS is available under a modified GNU general public license, which allows applications developed with the FreeRTOS API to remain closed source ("FreeRTOS," n.d.). SafeRTOS is another version of FreeRTOS developed by the company HighIntegritySystems, which is DO-178B certified ("Safety-critical RTOS," 2013). To our knowledge, SafeRTOS has not been deployed in any space systems so will only be discussed briefly. FreeRTOS is specifically tailored for microcontrollers and is portable to 35 different architectures, including FreeScale, x86 and ARM. FreeRTOS uses its own API and does not support POSIX.

FreeRTOS has been used primarily in small satellite deployments. It is an attractive choice because of the number of ports available for microcontrollers and because it is free (Holmstrøm, 2012). The private company GOMspace uses it on its Nanomind computer processor, which is designed to control small satellite missions ("NanoMind Computers," n.d.). CubeSatShop.com advertises a flight-qualified processor called the ISIS on board computer that includes FreeRTOS ("The One-Stop-Shop," n.d.). FreeRTOS has been used in a number of academic satellite projects including an Indian nanosatellite project called STUDSAT-2, which is India's first nanosatellite project. FreeRTOS is used as the on board computer of STUDSAT-2 and controls the central workings of the satellite (Rajulu, Dasiga, & Iyer, 2014). The firm Surrey Satellite Technology Ltd. and the University of Surrey in England used FreeRTOS in their experimental nanosatellite, Strand-1, which was the first of a series of cooperative satellite missions aimed at technological innovation in the small satellite domain. Strand-1 used a GomSpace on board computer, which ran FreeRTOS (Kenyon et al., 2011).

FreeRTOS can run as a paravirtualized guest machine on the X-hyp embedded hypervisor ("Para Virtualized Quests for Xhyp," n.d.). In 2014 a project to port

FreeRTOS to Xen on ARM was introduced by the Oregon based company Galois (Daugherty, 2014).

1. Design

The kernel itself is only composed of three C source files: queue.c, (queue structures), list.c, (linked list used in the queue structure) and tasks.c (task and scheduling logic) (Douglas, 2010).

a. Task Management

Tasks are defined as basic C functions and are the unit of execution. Applications that run on FreeRTOS are treated as a set of independent tasks (Real Time Engineers, Ltd., 2014). FreeRTOS supports one to one mapping of resources to tasks through the use of "gatekeeper tasks," which are tasks that have sole ownership of a resource. Only this task can communicate with the resource directly; other tasks needing the resource need to communicate with the resource's gatekeeper (via a queue) which will then make the resource available.

b. Scheduling Management

The FreeRTOS scheduler uses a fixed priority, pre-emptive scheduling algorithm, but also supports a cooperative scheduling model whereby tasks are never preempted and tasks with the same priority do not share processing time equally. The priority assigned to a task is not static and can be changed by the task itself.

c. Memory Management

FreeRTOS applications are able to allocate memory differently, depending on their requirements. If tasks or other facilities such as queues or semaphores are created before the scheduler starts running, then memory is dynamically allocated by the kernel and stays allocated for the duration of the application.

FreeRTOS supports a macro that is used to allocate protected regions on the ARM memory protection unit (MPU) regions, but this requires the specific port of FreeRTOS

to run on processors that support an MPU such as the ARM Cortex-M3 (Real Time Engineers, Ltd., 2014).

2. Analysis

The fact that manufacturers of microprocessors for small satellites are including FreeRTOS on their chip sets indicates that FreeRTOS has a legacy in the small satellite domain ("NanoMind Computers," n.d.). Being open-source also makes FreeRTOS an attractive option for missions with limited budgets. FreeRTOS is well documented and its core code development is maintained separately from community contributions, which makes revisions to the code consistent and traceable. The proprietary SafeRTOS version of FreeRTOS offers potential flexibility to developers who might be interested in a more secure version of the RTOS.

FreeRTOS however, does not provide much in the way of security for its applications. The small code base of the kernel limits the potential vector for security breaches but protection mechanisms, such as memory protection are not consistently available for all versions of the RTOS. Furthermore, tasks can execute at the same privilege level as the kernel.

F. LYNXOS-178

LynxOS-178 is a DO-178B certified proprietary RTOS developed by Lynx Software Technologies. LynxOS-178 runs primarily on the x86 platform but also supports some PowerPC platforms ("Board Support," n.d.). LynxOS-178 is not advertised as being completely ARINC-653 compliant since it does not support the ARINC-653 standard for inter-partition communications (Leiner, 2007) but it does incorporate some of the ARINC-653 functionalities.

LynxOS-178 is currently being used to monitor signals and transmit navigation data in the ESA's Galileo mission, a global navigation system that consists of thirty satellites (Howard, 2011). NASA has referenced LynxOS as a partitioning operating system worth studying for deployment in NASA space missions (Cudemore, 2013). LynxOS was used by NASA on the McDonald Laser Ranging Station to control tracking,

ranging and timing starting in the early 1990s but switched to RTAI in 2011 for cost reasons (Ricklefs, n.d.). To the best of our knowledge, LynxOS-178 can only run as a guest OS on the LynxSecure Microkernel hypervisor ("LynxOS-178," n.d.).

1. Design

LynxOS-178 is fully POSIX compliant and uses POSIX as its native interface (see Figure 15). LynxOS-178 also includes some ARINC-653 functions, such as health monitoring, partition management, time and process management and the ARINC-653 API.

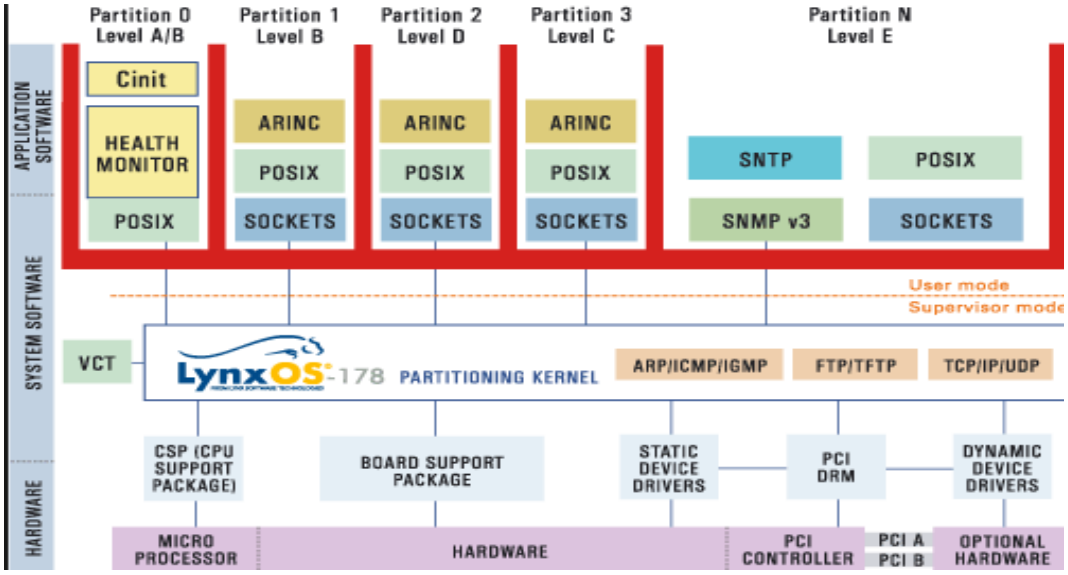

Figure 15. Illustration of LynxOS-178 (from "LynxOS-178," n.d., p. 2)

a. Task Management

POSIX threads are the basic scheduling entity. A task in LynxOS-178 is a group of threads Tasks run within partitions which are spatially isolated blocks of memory allocated by the processor's MMU. LynxOS-178 uses a patented approach called "priority tracking" to prevent priority inversion. Each task has two priority values associated with it, one for kernel threads and one for user threads. Kernel threads that handle interrupts do so "in step" with the user thread that actually requires the interrupt

("Linux Software," n.d.). This allows kernel threads to have their priority dynamically changed so that they always have higher priority than user tasks (Carlgren & Ferej, n.d.).

b. Scheduling Management

The LynxOS scheduler is preemptive and supports FIFO, round-robin and priority-based quantum scheduling polices. The priority-based quantum policy is a proprietary scheduling policy, similar to round-robin but with dynamic time slices (Carlgren & Ferej, n.d.). Each partition running on LynxOS-178 is scheduled according to a fixed cyclic scheduling policy and is statically assigned processor time. Partitions are able to schedule their own tasks using priority-based preemptive scheduling. Priority inheritance and the priority ceiling protocol are supported to prevent priority inversion within a partition (Leiner, 2007).

c. Memory Management

LynxOS requires the processor's MMU to do to memory protection. Neither memory nor resources are shared between partitions. Memory is statically allocated to partitions as defined in a configuration file called the virtual machine configuration table (VCT).

2. Analysis

LynxOS-178, like VxWorks and INTEGRITY-178B is safety certified and proven in the space domain. Unlike VxWorks and INTEGRITY-178B however, LynxOS-178 is limited in the number of processor families it supports, which currently consists of the x86, PowerPC and Pentium processors. LynxOS-178, like INTEGRITY-178B, also requires the use of a MMU for memory management and does not provide support for processors without an MMU. LynxOS-178's POSIX compliance and support of multiple development languages, such as C++, were cited by the ESA as being some of the reasons LynxOS-178 was chosen to manage navigation functionality in the Galileo satellite mission (Howard, 2007).

G. RTEMS

The Real-Time Executive for Multiprocessor Systems (RTEMS) is an open-source hard real-time operating systems designed for embedded systems and available under the GNU General Public License ("RTEMS Community," n.d.). RTEMS is compatible with a wide variety of processors including the ERC32, Leon, ARM, Pentium, various members of x86 architecture, MIPS and PowerPC (Silva, 2009), RTEMS supports a number of open standard APIs including POSIX and BSD sockets. Applications can be written in C/C++ using the POSIX API; additional languages supported are Ada, Java, Go and Lua (Bloom, 2013).

RTEMS has been and continues to be used in many different space projects. RTEMS was used on the FedSat, a research microsatellite developed by an Australian cooperative research group composed of university, commercial and government organizations ("Operating Systems," 2008; "Fed Sat 1," n.d.) between 2003 and 2006. RTEMS was also used on the Galileo GIOVE-A, ESA's first prototype for a navigation satellite ("Galileo Pathfinder," 2010). RTEMS is a supported operating system on NASA's SpaceCube satellites (Seagrave, 2008) and is being used on NASA's Mars Reconnaissance Orbiter (Komolafe & Sventek, 2006/07; "Mars Reconnissance Orbiter," n.d.).

RTEMS version 4.8.1 has been ported to run on the XtratuM hypervisor as a para-virtualized guest OS. The ported code includes board support packages for the LEON2 and LEON3 processors ("RTEMS," n.d.). RTEMS can also run on the PikeOS microkernel developed by Sysco ("SYSGO's Safe and Secure," 2010) and on the AIR microkernel. RTEMS is the basis for the hardware abstraction layers of AIR but can also run as a client partition alongside the ARINC-653 API (Schoofs, 2011).

1. Space Standards Compliance

The European Space Agency used version 4.8.0 of RTEMS to develop a "space-qualified" version of RTEMS that was qualified under the Galileo software standard (GSWS) to work on the ERC32, LEON2 and LEON3 processors. The GSWS is a space system software compliance policy that sets standards for the development, integration

and testing of software used specifically in NASA' Galileo Spacecraft. GSWS requires independent module/unit testing to ensure software safety and assurance (Feldt, Torkar, Ahmad, & Raza, 2010). The ESA considered validating RTEMS with DO-178B but decided GSWS was a more complete standard at the time hence its use. The space-qualified version of RTEMS is comprised of a series of scripts and patches that when applied to RTEMS code will delete some managers and will add others, making the system qualified up to a GSWS Development Assurance level B, which means that the OS does not contain any unused code (Silva, 2009).

RTEMS has continued to evolve and as of version 4.10 ESA's version is not maintained in the main RTEMS repository (Lee, 2012), which makes consistent development a challenge. ESA's goal was to make RTEMS a building block in space missions but it first needed to get RTEMS TRL6 certified ("Definition of Technology," n.d.). To achieve this goal, the ESA decided to focus on the components of RTEMS that were relevant to ESA space missions and enlisted the firm Edisoft to establish an RTEMS maintenance center that dealt only with the RTEMS developments being made by ESA instead of the general RTEMS community ("Operating Systems," 2008). This diversion has led to some confusion and frustration amongst developers who are unclear on which version of RTEMS to work with for space projects (Lee, 2012).

2. Design

RTEMS (see Figure 16) supports dynamic memory allocation, inter-task communication and synchronization, various scheduling configurations, priority inheritance, responsive interrupt management and symmetric multi-processing across multiple cores. Such services are implemented by a set of "resource managers." Core functions that are used by multiple managers, such as scheduling and object management are maintained as part of the "SuperCore" (Bloom, 2013).

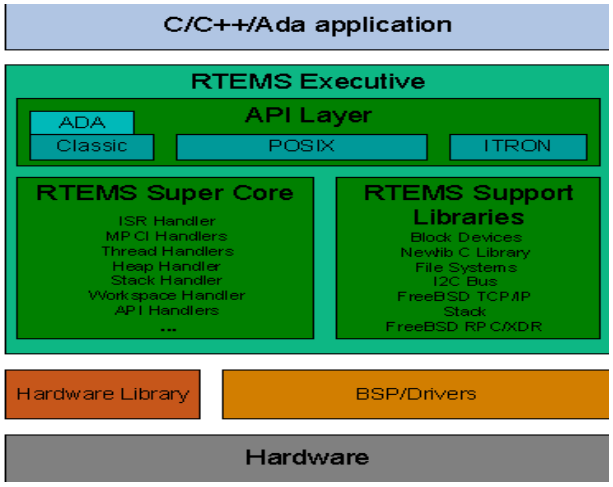

Figure 16. RTEMS Conceptual Architecture (from "RTEMS Architecture," n.d.)

a. Task Management

Tasks are defined in RTEMS as the "smallest thread of execution that can compete on its own for resources" (On-Line Applications Research Corporation, 2013, p. 64). When a task is created, it is allocated a task control block data structure. The TCB is the only RTEMS internal data structure that an application can access and modify. Tasks have a priority assigned to them when they are initially created (On-Line Applications Research Corporation, 2013).

b. Scheduling Management

The RTEMS scheduler is in charge of managing a given set of tasks in the ready state and determining when tasks get executed. The default scheduling algorithm is a priority-based scheduler, however, developers can also work with the following: a simple priority scheduler that maintains a single linear list--meant for small applications, earliest deadline first scheduler, constant bandwidth server scheduler (each task is given a CPU budget and if the budget is exceeded then a callback is invoked), simple SMP (symmetric multiprocessing) or a partitioned/clustered scheduler, which allows developers to choose different policies for different cores (On-Line Applications Research Corporation, 2013).

c. Memory Management

RTEMS uses a flat memory model and does not support virtual memory allocation, segmentation or MMU hardware support. The partition manager creates and deletes partitions and dynamically allocates memory to them in fixed-sized units (On-Line Applications Research Corporation, 2013). The POSIX mprotect() function can be used to protect regions of memory ("RTEMS 4.10.99.0 On-line Library," 2014).

3. Analysis

RTEMS is proven in the space community given its use in many different space applications and its use by the ESA to develop a "space qualified" version of the RTOS. Its compatibility with many different processors, as well as its extensive documentation makes it an attractive RTOS for space system developers.

As stated previously, there is some confusion within the space systems development community over which versions of RTEMS to work with since the space qualified version of RTEMS is based on an older version of the RTOS and is not part of the mainline RTEMS code development tree. RTEMS also does not support memory protection other than what POSIX offers and leaves it up to the developer to incorporate such features.

H. ADDITIONAL REAL-TIME OPERATING SYSTEMS

The following real-time operating systems are worth surveying due to their compatibility with key virtualization architectures despite the fact that there is limited documentation on how they function. We discuss the important attributes of each RTOS.

1. LithOS

LithOS is developed and maintained by the Spanish company Fentiss. LithOS is an ARINC-653 compliant para-virtualized RTOS designed to run as a partition on the XtratuM hypervisor (see Figure 17). Though there is no documentation of LithOS's deployment in space, it was designed specifically to support systems requiring strict

spatial and temporal isolation that run on the XtratuM hypervisor, which the ESA is using to evaluate virtualization and IMA-SP.

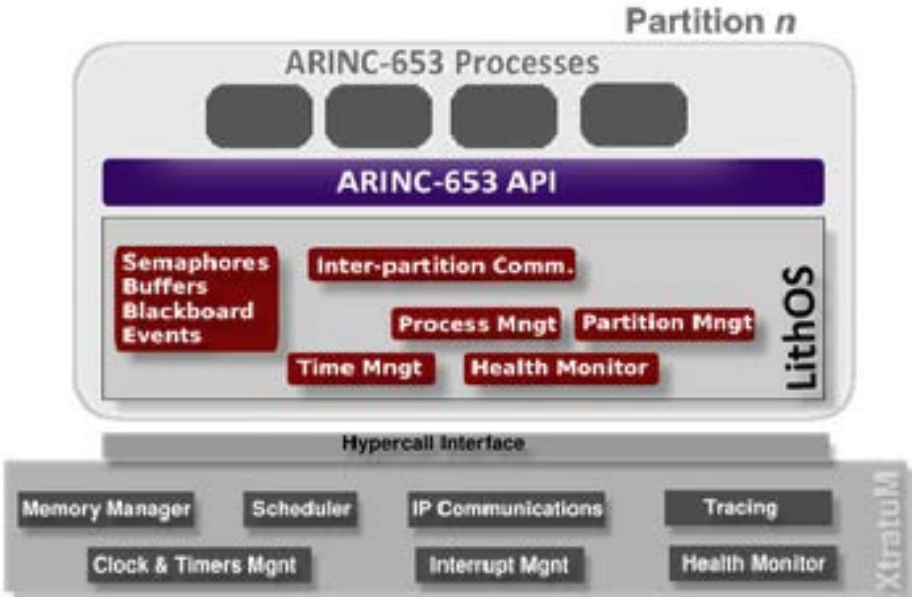

Figure 17. LithOS Architecture, Running As an XtratuM Partition (from "LithOS," n.d.)

The XtratuM hypervisor (see Chapter IV) incorporates many of the ARINC-653 spatial and temporal isolation mechanisms. LithOS leverages these when running as a virtual machine on the hypervisor. Additionally, LithOS provides support for multi-processing, intra-process communication and process scheduling, which are services that XtratuM does not provide.

LithOS follows the ARINC-653 standard and implements the ARINC-653 API, as well as its own native API. LithOS also includes a few non-portable services relating to time and partition management that the ARINC-653 API does not include, which are non-portable.

2. VxWorks 653

VxWorks 653 is a version of VxWorks that NASA has recognized as being a potential operating system for future projects (Raines, 2012; Barry, 2009; Jaekel, 2014).

VxWorks 653 is an ARINC-653 certified operating system that is comprised of the module OS and the partition OS. The module OS is the supervisor-mode OS that enforces time-space partitioning through memory management services and static schedules to ensures fault isolation. The partition OS is designed to run within a VxWorks 653 user-mode partition, which is a virtualized run-time environment that supports applications. The partition OS is also known as "vThreads," a multi-threading system based on VxWorks 5.5, which includes additional libraries that support the ARINC-653 APEX and POSIX APIs. Each instance of vThreads also contains its own scheduler. Figure 18 illustrates the architecture of VxWorks 653.

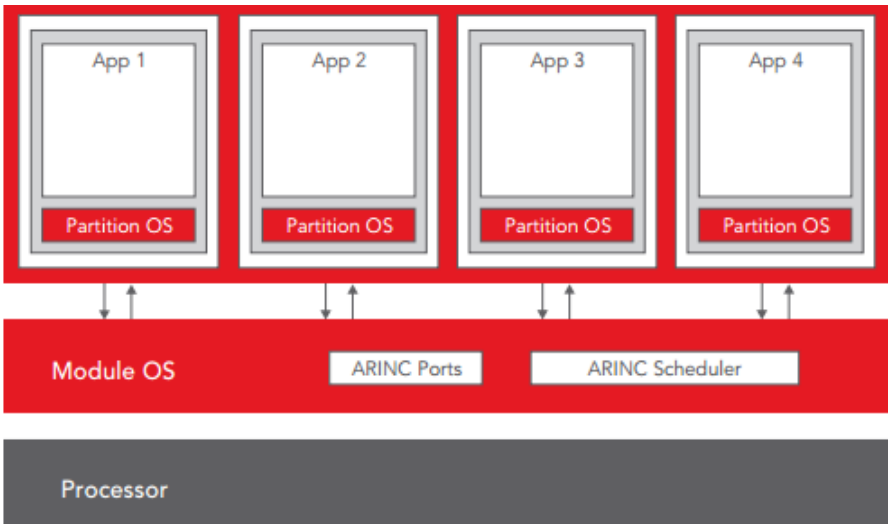

Figure 18. VxWorks 653 Architecture (from Parkinson & Kinnan, n.d.)

Next, we discuss the virtualization architectures that are designed for, or are applicable to the space domain.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. VIRTUALIZATION ARCHITECTURES USED IN SPACE

In this chapter, we survey a selection of virtualization architectures that are applicable to the space domain (see Table 5). These architectures have been selected based on their use in space or their consideration by the space community for future use. We discuss how each architecture is being used by the space community, its basic design and what processors and operating systems it supports. We focus on how each architecture supports the real-time requirements of its applications and provide comparative analysis. We also review some virtualization architectures that have space-relevant attributes—such as real-time process support, high assurance properties or space-qualified RTOS compatibility—that make them worth surveying for possible space application.

Our analysis of each virtualization architecture is based on a variety of characteristics including its maturity within the space community, licensing, documentation availability, standards compliance and hardware and software capability. We also consider the functionality of the virtualization layer with regards to its memory footprint and trusted computing base.

Table 5. Summary of Virtualization Architecture Key Attributes

Hypervisor	License	Internal Design	Development Tools	Documentation	Hardware Support	API and Guests supported	Standards	Footprint (kernel)	Performance Evaluation	Space use status
INTEGRITY Multivisor	Proprietary	Security Kernel	Wind River Workbench	Unavailable openly	(see INTEGRITY RTOS)	All guests (designed to be OS agnostic)	DO-178B, ARINC-653, EAL 6+	Unknown	No	No
VxWorks hypervisor	Proprietary	Configurable	Yes	Unavailable openly	(see VxWorks Hypervisor)	All guests (designed to be OS agnostic)	None	Depends, highly modular	No	No
XtratuM	Open-source GPL or proprietary	Monolithic kernel	No	Yes	X86, ARM, PowerPC	LithOS, paRTiKle, Linux, RTEMS	Unknown	10K lines of code	ESA	ESA
ARLX	Permissive after subscription	Xen-based	No	Some	ARM, x86	All guests supported by Xen	DO-178C	~70K	Yes	Yes
PikeOS	proprietary	Microkernel	Yes	Some	X86, MIPS, PowerPC, ARM, SPARC V8/LEON	Linux; RTEMS; POSIX, Ada	DO-178B, MILS and ARINC-653	Unknown	NASA	NASA
AIR	Open-source	Microkernel	Unknown	No	All	All guests(designed to support most OSes)	ARINC-653	Unknown	Yes (ESA) Current status unknown	Unclear
NOVA	Open-source	Separation kernel	No	Yes	X86	All guests (via emulation)	None	9k lines of code	No	No
X-hyp	proprietary	Unknown	Unknown	No	ARM-9, Cortex	FreeRTOS, Linux, RTEMS	None	Unknown	No	No
Proteus	Unknown		No	No	PowerPC	All guests (via full virtualization)	None	15 Kb	No	No
RT-Xen	Open-source	Xen-based	No	No	All Xen	Linux guests (unspecified versions)	None	Unknown	No	No

A. XTRATUM

XtratuM is an open-source[7] type-1 hypervisor that uses paravirtualization (Crespo et al., 2014) and is designed to provide temporal and spatial isolation for safety critical applications ("XtratuM Hypervisor," 2012). XtratuM is based on the concept of robust partitioning and allows processes of different security levels to run concurrently. XtratuM is designed to reflect ARINC-653 standards, but is not fully ARINC-653 compliant because of the responsibilities delegated to partitions. The XtratuM hypervisor works with the x86, ARM and PowerPC processors ("XtratuM Product," n.d.; Zhou, 2009), as well as the LEON2, 3, and 4 implementations of the SPARC processor. XtratuM can host LithOS, RTEMS, PaRTiKle, and Linux operating systems.

To the best of our knowledge, XtratuM has yet to be deployed in space however considerable research is in progress focusing on its ability to support space systems. Since 2012, the ESA has been conducting a set of studies to evaluate the effectiveness of using time-space partitioned (TSP) architectures in space, using XtratuM as the base for this research. These studies are conducted under the ESA's EagleEye virtual space mission intended for software testing ("New-generation Aircraft," n.d.). As of 2013, the EagleEye TSP project has tested XtratuM version 3.4 with support for the LEON3 processor with a memory management unit (Bos et al., 2013). In 2014, Carrascosa et al. (2014) documented porting XtratuM to the LEON4 multicore processor in support of the ESA's ongoing efforts to test and evaluate XtratuM's performance with multicore processors.

NASA (n.d.) also carried out some research with the XtratuM hypervisor during the 2012 Internal Research and Development Program (IRAD) that sought to demonstrate the benefits of virtualization on the LEON 3 flight processor.

[7] Commercial license also available.

1. Design

XtratuM, illustrated in Figure 19, is designed as a "monolithic, non-premptive kernel" ("XtratuM Product," n.d.). The entire hypervisor layer operates in supervisory mode and no process may preempt it (see Figure 18). The hypervisor layer is responsible for virtualizing the machine's CPU, memory, interrupts and other peripheral devices.

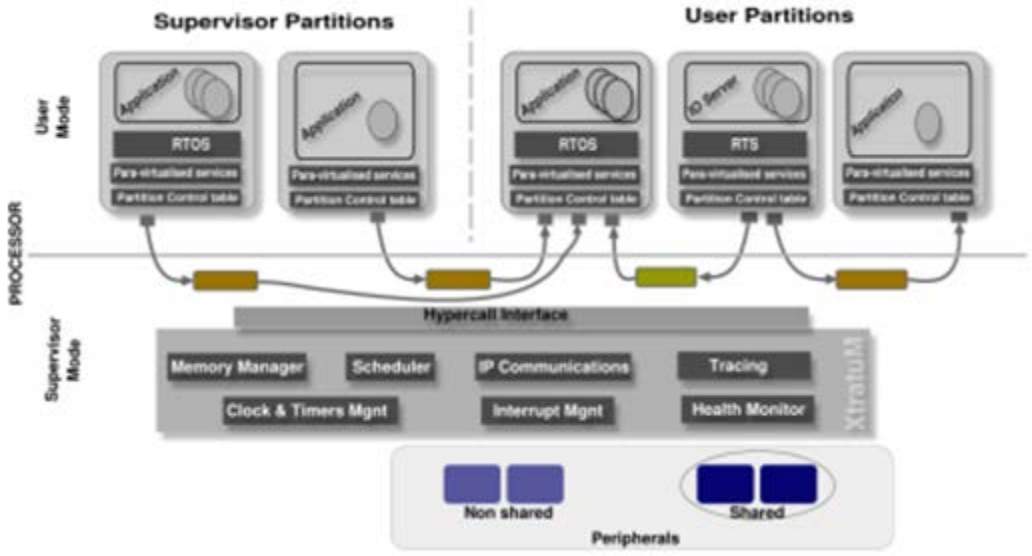

Figure 19. XtratuM Architecture (from "XtratuM Hypervisor," 2011).

2. Partition Management

Partitions are the independent execution environments managed by the XtratuM hypervisor. Partitions can be an application, an RTOS or a general purpose operating system. XtratuM partitions do not share any of their address space. Partitions are started automatically after the XtratuM hypervisor completes the initial boot sequence. There are two types of partitions supported by XtratuM: *system partition* and *user partition.* System partitions are also referred to as supervisor partitions in some literature, but the developers changed this terminology to *system partition* to avoid confusion with hardware modes ("XtratuM Hypervisor," 2011). System partitions are able to suspend, resume, halt or reset the execution state of user partitions (Masmano, Ripoll, Peiró, & Crespo, 2010) through specific hypercalls. This activity is regulated by resource and

inter-process communication policies defined at configuration time for each partition. System partitions are able to manage the system but still rely on the hypervisor to access hardware. For multi-thread applications, the operating system or run-time support libraries on which the applications run must support threads ("Xtratum Hypervisor," 2011). This is different from the ARINC-653 specification for partitions, which isolates and manages threads and processes inside a partition through the use of a defined API.

3. Memory Management

XtratuM is designed to enforce spatial isolation with or without an MMU, though full spatial isolation is only guaranteed on versions of the hypervisor ported to processors with an MMU. For processors without an MMU (namely the Leon2), XtratuM uses the processor's write protection registers to enforce isolation through memory write protection policies, which deny partitions the ability to write into another partition's memory space. For MMU supported versions of XtratuM, a kernel memory manager module uses the MMU to enforce spatial isolation between partitions. If specified at configuration time, the memory manager can support authorized memory sharing between partitions for inter-partition communication (Masmano et al., 2010).

4. Scheduling Management

XtratuM implements an ARINC-653 cyclic scheduling policy in which time slots for partitions to interact with the processor are statically defined. XtratuM takes into account the overhead incurred with context switches that occur when one partition's time slot is over and another partitions' time slot begins. Since XtratuM offers deterministic processing, it knows the worst-case execution time (WCET) and the best-case execution time (BCET) of context switches and hypercalls. In order to make context switches as efficient as possible, XtratuM's scheduling use the empirically determined BCET and WCET of context switches and any in-progress hypercalls to calculate worst-case delay. With this cost in mind, XtratuM can adjust execution time by factoring it into the allotted time slot ("XtratuM Hypervisor," 2011).

5. Analysis

The fact that XtratuM is part of an ESA initiative to research virtualization in space and many of its refinements have been motivated by space research distinguishes XtratuM from some of the virtualization architectures we survey here. Its open-source license and well-documented API make it an attractive architecture for projects with limited budgets. The one main drawback is the limited support it provides for both hardware and software.

B. ARLX

ARINC-653 Real-time Linux on Xen (ARLX) is a hypervisor developed and maintained by the Michigan-based company Dorner Works Ltd. (DornerWorks, n.d.). ARLX is a type-1 hypervisor based on the open-source Xen hypervisor, but with extensions DornerWorks claims make it a high safety and security assurance system. ARLX was designed based on the DO-178C[8] certification standard and there is an initiative to get ARLX formally verified to Common Criteria Evaluation Assurance Level 6+ (Studer, 2014). Some formal method analysis on ARLX has been performed, discussed later in this section.

ARLX is available via subscription under a permissive license, meaning that with an initial purchase all source code is available and can be modified. ARLX is compatible with ARM and x86 family processors and supports any operating systems compatible with Xen (VanderLeest, Greve, & Skentzos, 2013). A Navy-fielded deployment of ARLX runs VxWorks and Integrity in guest domains (Santangelo, 2013).

ARLX is currently being used on unspecified platforms by the Joint Tactical Networking Center, which is managed by the Navy's Space and Naval Warfare Systems Command (SPAWAR). ARLX is also being used by the company sci_Zone, a NASA small business innovation research awardee, on its QuickSAT project. QuickSAT is a space-hypervisor that supports virtualized payloads and systems on CubeSATs and

[8] DO-178C replaced DO-178B in 2012.

MicroSATs. QuickSAT is being used in NASA research centers and by the Air Force Research Laboratory's University NanoSat program (Santangelo, 2013).

1. Design

ARLX core architecture follows that of Xen, but it modifies the kernel and adds another privileged domain in addition to Dom0 to support input and output. The code base of ARLX is 30–50% smaller than the generic code base of Xen, which DornerWorks claims is over 150,000 lines of code. The designers of ARLX point out that ARLX is still a work-in-progress and that the hypervisor is in heavy development. As a result, some features, like minimized partition memory footprints, optimized partition switching mechanisms and full ARINC-653 compliance are still future projects (Greve & VanderLeest, 2013). The current status of these projects is unknown.

In ARLX, the Xen kernel is modified so that it implements time and space partitioning according to the ARINC-653 standard. The typical Xen scheduler is replaced with the ARINC-653 scheduler. Additionally, an ARINC-653 memory manager replaces the traditional Xen memory manager in the Xen kernel. To the best of our knowledge, ARLX requires an MMU to enforce spatial isolation. The inter-partition ARINC-653 API is added to Xen's communication architecture, which allows for ARINC-653 compliant inter-partition communication mechanisms (Greve & VanderLeest, 2013). The developers of ARLX define five security policy *domains* that are used to enforce information flow between partitions. Security domains refer to information flow levels and not to the guest domains running on top of Xen. These security domains are listed in Table 6.

Table 6. ARLX Security Domains (from Greve & VanderLeest, 2013)

SECURITY DOMAIN	CONTENT
ARLX_INIT	Initialization read-only data for system startup
ARLX_CONFIG	Configuration data only written at system initialization. Read-only while system is running
ARLX_XEN	State of Xen hypervisor
ARLX_DOM0	State of Xen Dom0 (privileged Domain)
ARLX_DOMU	State of Xen DomU's (non-privileged)

Information flows from top down with two exceptions: Dom0 and Xen are able to communicate freely and each DomU can communicate if specified by configuration. There is no domain defined for the privileged I/O domain.

2. Partition Management

The ARLX Dom0 is designed to be as small and intended to be formally verifiable. Dom0 is still implemented by a Linux-based OS but the developers are considering using FreeRTOS instead—which has a smaller code base and has a certified version (SafeRTOS)—or some other certified OS, like INTEGRITY or VxWorks (Studer, 2014).

ARLX features a privileged domain, separate from Dom0, which is responsible for I/O management between partitions. In standard Xen, Dom0 provides this functionality. ARLX reduces the trusted computing base of the Dom0 by isolating the I/O responsibilities into a separate domain. This design decision is based on the idea of Dom0 disaggregation, which takes control logic out of Dom0 and distributes it throughout different domains, with the idea of making each domain small and easily verifiable (Murray, 2008). The privileged I/O domain also regulates bandwidth usage between partitions that share I/O devices. This feature is not required by the ARINC-653 standard, but the ARLX developers felt it was valuable since it can incorporate determinism into bandwidth usage for each partition. ARLX handles shared I/O by splitting shared I/O device drivers up, with half of the driver residing in the I/O domain and half residing in another DomU. The DomU's portion of the driver contains the API to communicate with

the I/O domain. The I/O domain's portion of the driver provides access to memory and registers through memory mapping. The architecture of ARLX is illustrated in Figure 20.

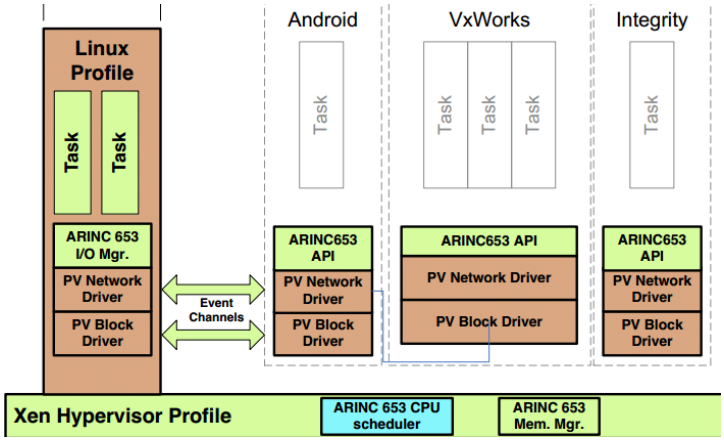

Figure 20. ARLX Hypervisor Environment (from Santangelo, 2013)

3. Analysis

ARLX is a unique architecture due to its integration of ARINC-653 compliance into a well-known open-source hypervisor. To our knowledge, ARLX is also one of the few virtualization architectures surveyed that has actually been deployed in space, i.e., via the QuickSat/Xen program. ARLX's licensing is also an attractive feature since it allows developers to access and modify source code. ARLX, however, is still a work in progress as its developers attest, and lacks all the features of other architectures, including being fully ARINC-653 certified.

DornerWorks is pursuing formal verification of the ARLX hypervisor, since its target is use in safety critical embedded systems. As part of this effort, it conducted an initial analysis of the security properties for the system (VanderLeest et al., 2013). This study found that there was substantial work required for the system to be considered high assurance. Though there are many benefits to leveraging Xen for ARLX, there are also drawbacks. There have been several high profile vulnerabilities exposed against the Xen hypervisor over the years (Kovacs, 2014; Apecechea, 2014) related to the fact that it was not built from the onset to be high assurance. Its code base is also much larger than other surveyed virtualization architectures, which have smaller code footprints to limit the

trusted computing base. In particular, ARLX's addition of a second privileged domain under Xen increases its TCB.

C. PIKEOS

PikeOS is a proprietary virtualization architecture developed and maintained by the company Sysgo. PikeOS is a separation kernel-based type-1 hypervisor that supports paravirtualization and hardware-assisted virtualization.[9] PikeOS is DO-178B, MILS and ARINC-653 compliant and is in the process of becoming formally verified, a requirement for Common Criteria EAL 6 certification ("Publishable Summary," 2012). PikeOS is portable to the PowerPC, x86, ARM, MIPS and SPARC V8/LEON processor families. PikeOS can run Linux and RTEMS as guest operating systems. It supports multiple APIs including POSIX, Ada and RTEMS. PikeOS is also compatible with a certifiable IP stack and offers communication encryption and binary verification ("Products PikeOS Hypervisor," n.d.).

PikeOS (see Figure 21) was used as the hypervisor in NASA's 2013 Internal Research and Development Program. This program explored flight hardware virtualization for science data processing, to consolidate multiple physical processors to reduce their size, weight and power consumption and to increase security on flight systems ("Fall 2013," 2013). Their test configuration consisted of PikeOS run on a LEON3 processor, supporting ElinOS (Sysgo's verison of embedded Linux) in one partition and custom Goddard Space Flight Center (GSFC) software running in another partition. The ElinOS VM was used to do non-critical science data processing that did not have real-time requirements, and the GSFC partition was used as the core flight executive that handled critical functions with hard real-time requirements. The project demonstrated that when the ElinOS partition crashed, it had no effect on the GSFC partition. The 2013 tests with PikeOS also demonstrated that multiple flight processors can be booted in virtual machines and that virtual machines can be rebooted individually mid-flight (NASA, n.d.; Cudmore, 2013).

[9] Paravirtualized virtual machines can also leverage hardware assisted virtualization if the processor supports it.

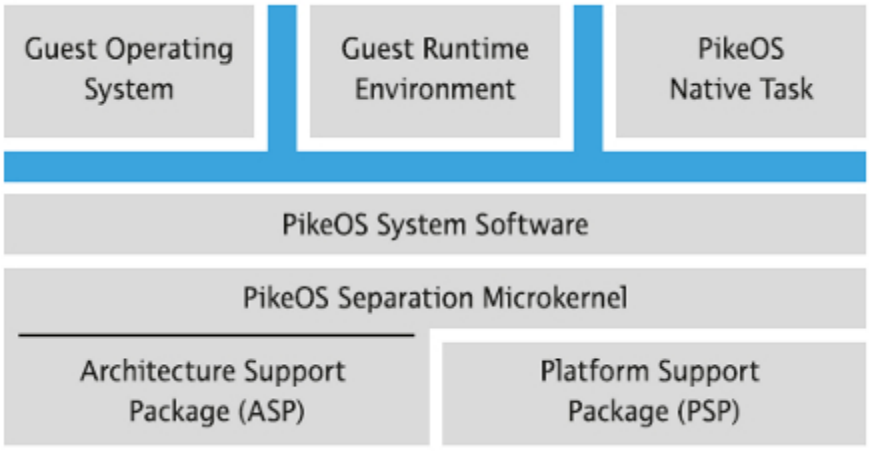

Figure 21. PikeOS Architecture (from Lehrbaum, 2013)

1. Design

There are two layers to the PikeOS architecture: the microkernel layer and the virtualization layer (see Figure 3). The microkernel is responsible for managing address space separation, partition scheduling, inter-partition communication and enforcing communication control and access measures for threads and tasks (Tverdyshev, 2011; Müller, Paulitsch, Tverdyshev, & Blasum, 2012). The virtualization layer is responsible for implementing the API for partitions and guest applications.

PikeOS has two primary abstractions: tasks and threads. Threads are always associated with a task and execute based on the task's state. Tasks consist of a virtual address space, threads and other resources that they might be allocated. The microkernel controls all resources in the system, is responsible for managing communication for tasks and threads and delegating use of resources to partitions based on the security policy set at configuration time (Tverdyshev, 2011; Baumann, Bormer, Blasum, & Tverdyshev, 2011).

2. Partition Management

Each partition consists of a set of tasks, threads and communication ports (as defined in the ARINC-653 API). It is the job of the virtualization layer to instantiate these

partitions, mediate communication with other partitions based on a pre-defined security policy and control access to system resources.

3. Memory Management

The microkernel has a memory manager that assigns address space through memory pages to the partitions. Partition memory pages are statically defined at configuration time and assigned to partitions by the memory manager at run-time. At run-time, each partition can dynamically store data and allocates memory to its applications through these memory pages (Baumann et al., 2011).

4. Scheduling Management

PikeOS supports a combination of scheduling methods including priority-based, time-driven and proportional sharing scheduling. Using a combination of scheduling methods ensures that hard real-time threads get scheduled first and prevents low-priority threads from being starved out of processing time (Kaiser, 2007).

Partitions running on PikeOS are statically assigned a priority level, by which the microkernel schedules partitions based on this priority. In addition, PikeOS uses what are referred to as “time domains” in which priority-based scheduling of threads is based on their “class,” i.e., time-driven, event-driven or non-real-time. Event-driven and time-driven threads are assigned a higher priority than other threads. Threads are grouped into time domains and can only execute when their time domain is active, no matter their priority.

There are two types of time domains: foreground and background domains. The foreground domain is always running, and the background domain is scheduled by the microkernel based on a static schedule determined at configuration. The background domain can run at the same time as one other domain. Event-driven threads are assigned to the background domain. The highest priority task between the two active domains gets scheduled first. Low priority threads get executed when all event and time-driven threads within their time domains are completed (Kaiser, 2007; Kaiser, 2009).

5. Analysis

PikeOS exhibits many of the properties required to support multiple software environments in space. It has a small trusted computing base with limited complexity, it is certified and compliant with many of the primary standards recognized by the space community and has demonstrated its robustness by protecting highly critical domains when another domain fails. PikeOS is also compatible with a number of relevant space processors, including the MIPS processor, which is not supported by XtratuM. PikeOS is proprietary and it is unclear how accessible its source code is to developers, which, if limited, might be a drawback. Sysgo also does not appear to have a substantial presence in the space community with its other products, including a modified version of Linux for embedded systems, which might limit PikeOS's use if other virtualization architectures from recognized vendors are deemed more compatible with legacy systems.

D. AIR

ARINC-653 Interface in RTEMS (AIR) (Rufino & Filipe, 2007), is a virtualization architecture that supports the execution of safety critical real-time applications and non-real-time applications concurrently. AIR, like XtratuM, was a project initiated by the ESA as part of their assessment of adapting time and space partitioning software for space systems. The original AIR was a proof-of-concept project to build an ARINC-653 system specifically for the space domain. A final report for the AIR project published in 2007 provided the foundational architecture for an ARINC-653 compliant system for space. Since then, AIR-II seeks to evolve AIR from proof-of-concept to a deployable product (Rufino, Craveiro, Schoofs, Tatibana, & Windsor, 2009). As of 2014, AIR is referenced as an open-source product offered by the international aeronautics company, GMV ("air Robust," n.d.). According to GMV, AIR is currently TRL level 5, which means the system has been tested and prototyped in a relevant environment. The status of AIR testing and on which space systems AIR may be considered for deployment in the future are both unknown. AIR is designed to be hardware and software independent.

1. Design

AIR is comprised of three primary components: the AIR partition management kernel (PMK), the real-time operating system kernel for each partition (POS), and the ARINC-653 APEX API. The PMK is a microkernel responsible for partition scheduling and inter-partition communication. Each POS kernel is abstracted through the POS adaptation layer which allows the architecture to be kernel independent.

2. Scheduling Management

AIR has an ARINC-653 scheduling manager within the PMK that ensures priority-based partition scheduling, as well as POS schedulers that are responsible for scheduling processes within each partition. AIR also includes "timeliness enhancement mechanisms" within the PMK layer, which are meant to further ensure robust scheduling within the system (Rufino et al., 2009). One enhancement mechanism is mode-based scheduling, which give the option of switching to different scheduling modes for a partition. Another is process deadline monitoring, whereby the PMK verifies that earliest deadline tasks in a partition are completed by when they are intended. If they are not, then the PMK reports this to the ARINC-653 compliant health monitor.

3. Memory Management

AIR accounts for memory protection and management with the use of the processor's MMU or MPU. Each partition has its own page directory. Memory pages and shared libraries can be shared between partitions. POS and APEX code can also be shared across partitions ("Air Overview," 2011). Memory and code sharing between partitions is done based on pre-defined inter-partition communication policies established at configuration time (Rosa, 2011).

4. Analysis

Notable, attractive attributes of the AIR virtualization architecture, illustrated in Figure 22, include the fact that it is processor and operating system/application agnostic, and that it is open-source. The fact that it incorporates the ARINC-653 functionalities and API make it a robust virtualization architecture to consider in the space domain. The

main drawback of the AIR project is its status as a prototype, though it does appear from GMV documentation that the hypervisor is being actively maintained and developed. There is no current documentation on the AIR hypervisor being fielded in any space system.

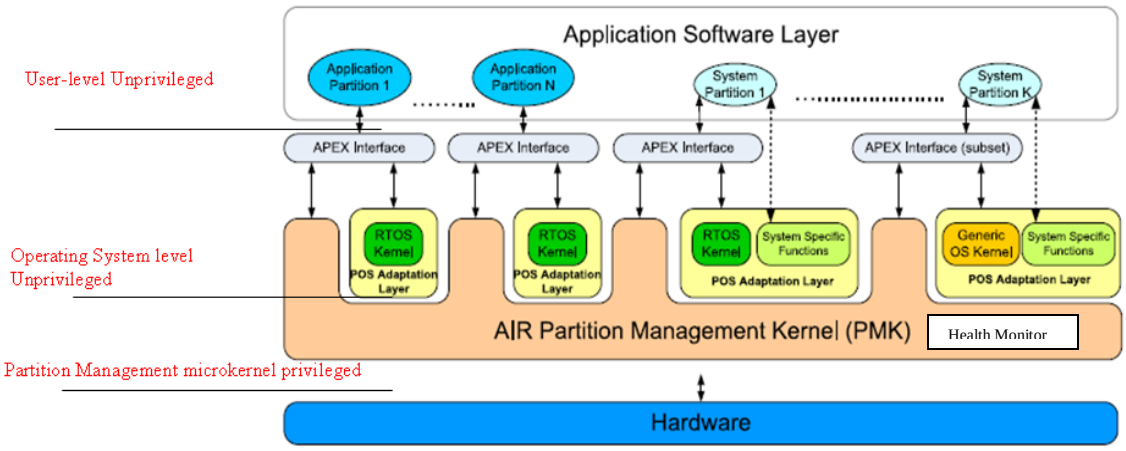

Figure 22. AIR Architecture (from Rosa, 2011; Rufino et al., 2009).

E. ADDITIONAL VIRTUALIZATION ARCHITECTURES

The following section briefly surveys several virtualization architectures worthy of mention, despite lack of consideration by the space community and/or lack of sufficient documentation to survey adequately.

1. Green Hills Multivisor

Green Hills Software's INTEGRITY multivisor is a separation kernel based virtualization architecture based on the INTEGRITY RTOS kernel, but with the added ability to support paravirtualized operating systems and leverage hardware virtualization assistance to fully virtualize operating systems. The multivisor can support multiple operating systems—including Windows, Linux, VxWorks and Android—and multiple processors, both single and multicore, including IntelVT, ARM, FreeScale and any processor supported by the INTEGRITY RTOS. To our knowledge, the INTEGRITY Multivisor has not been deployed in space systems. It is marketed primarily to the telecommunications and avionics industries; however, the use of the INTEGRITY kernel

is space systems makes the INTEGRITY multivisor a potentially deployable architecture in the future (“Integrity Multivisor Datasheets,” n.d.).

2. Wind River Hypervisor

The Wind River hypervisor is an embedded type-1 hypervisor (“Wind River Hypervisor,” n.d.) designed to host operating systems and applications of mixed criticality and with different timing requirements, from hard real-time to general-purpose on single or multicore processors. The Wind River hypervisor supports full and paravirtualization of operating systems and can leverage hardware-assisted virtualization features on processors. It is designed to host any operating system or application through the use of the VxWorks API and is designed to run on top of a variety of different processor families, including ARM, PowerPC, and Intel.

The hypervisor, like the VxWorks operating system, is highly configurable and offers different scheduling options on single or multicore processors, different means of configuring external devices and different ways to virtualize each partition (full or partial virtualization). The hypervisor is responsible for scheduling partitions (called virtual boards) and uses time-slice or priority-driven methods. Threads are completely event-driven, meaning they are only executed when an event prompts them. Developers have the customization option of replacing the hypervisor scheduler. External device driver management is also configurable: drivers can be located within partitions or within the hypervisor and can be shared or private resources (“Wind River Hypervisor,” n.d.).

The hypervisor is not known to be compliant with any relevant standards, though Wind River offers a separation kernel for systems requiring high assurance (not part of this survey). The relationship between these two products is unclear. To our knowledge, the Wind River hypervisor has not been deployed or considered for deployment in any space system. The hypervisor is marketed primarily to the industrial control and telecommunications industries.

3. SafeHype

SafeHype is a prototype small, lightweight satellite hypervisor being designed by Intelligence Automation, Inc. based out of Rockville, Maryland. In 2013, the firm was awarded a $150,000 grant by the Defense Advanced Research Projects Agency in order to develop the hypervisor ("SBIR SafeHype," n.d.). Unfortunately, there is not an extensive amount of information available through the open literature. SafeHype is a small hypervisor meant to virtualize satellite payloads and support dynamic provisioning of virtual machines mid-flight ("SBIR SafeHype," n.d.). The hypervisor is designed to make use of hardware support and paravirtualization. It is claimed the code base of the hypervisor is small enough to be formally verified ("Intelligent Automation," n.d.). SafeHype is a project that may yield a viable virtualization system for future spacecraft. The mechanism for dynamically provisioning virtual machines mid-flight is interesting though, unfortunately, there is no information on how this is accomplished.

4. NOVA

NOVA is a research "microvisor" developed by Udo Steinberg and Bernhard Kauer from the Technical University of Dresden in Germany. Though not designed for space systems, it is an interesting architecture that has some attributes that are important to the space community, including a small size and spatial and temporal isolation ("NOVA Virtualization," n.d.).

NOVA is a small hypervisor, or "microvisor," that runs on x86 processors that support the Advanced Configuration and Power Interface, an open industry standard. It can also run under QEMU as a virtual machine. NOVA has its own kernel and application program interface and makes use of hardware virtualization support available on the x86 processor.

The NOVA environment consists of three layers: a microhypervisor running in kernel mode, the user-level environment and the VM layers or domains. Security and performance-critical functionalities are handled inside the microvisor. All other functionalities run in user mode outside of the microvisor. The microvisor is responsible for interrupt handling, scheduling and memory management. NOVA uses an object-

oriented interface to delegate and regulate access to resources. There are five basic kernel objects (see Table 7). When one of these objects is created by a domain, it gets associated with a "capability" that belongs to the domain creating the object. Depending on that domain's policy, access to these objects can be shared with other domains.

Table 7. The Five Kernel Objects in the NOVA Microvisor (from Steinberg & Kauer, 2010)

Kernel Object	**Function**
Protection Domain	Spatial Isolation
Execution Context	Protection Domain Thread and CPU execution
Scheduling Context	Temporal Isolation
Portals	Intra-partition (domain) communication
Semaphores	Execution synchronization

What makes NOVA an interesting virtualization solution for space is the fact that it has a small code base, has a means of controlling access to critical resources through its capability-based interface and is open-source. The object-oriented approach to access control employed by the NOVA microvisor is similar to the proprietary INTEGRITY kernel, which regulates information flow through statically defined policies for subjects and objects (discussed in Chapter III). The main draw backs of the microvisor are the fact that it is only compatible with the x86 processor, relies on processor virtualization support and does not make any claim to support real-time systems.

5. Proteus

Proteus was designed as a research project of the Heinz Nixdorf Institute in Germany as an open-source type-1 hypervisor able to run general-purpose operating systems and real-time operating systems concurrently. Proteus supports both full and paravirtualization on PowerPC multicore processors (Gilles, Groesbrink, Baldin, & Kerstan, 2013) and does not rely on hardware support for virtualization. Figure 23 illustrates the architectures of the Proteus Hypervisor.

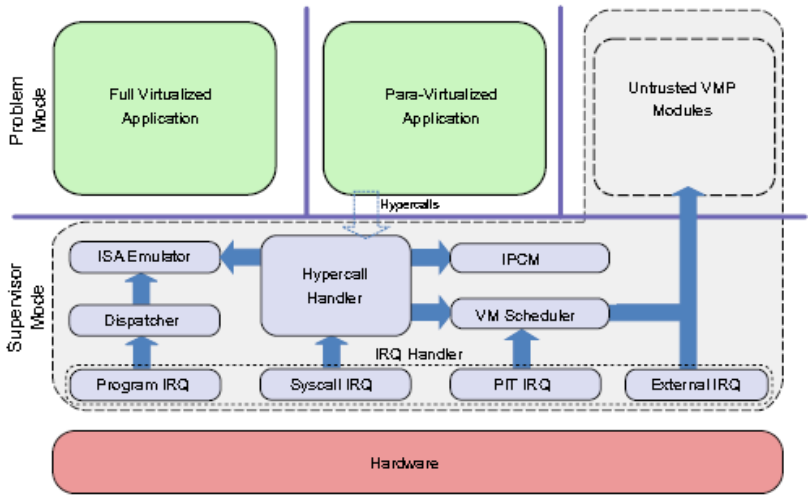

Figure 23. The Proteus Hypervisor Architecture (from Baldin & Kerstan, 2009)

There are two execution modes on the PowerPC processor that the hypervisor uses: applications run in *problem mode*; interrupts, the virtual machine scheduler and the inter-partition communication manager run in *supervisor mode*. Device drivers and other non-critical resources are run in a separate partition on top of the hypervisor and run in problem mode. Problem mode is subdivided into two logical modes: VM privileged mode and VM problem mode. System calls made by the virtual machine are executed in the VM privileged mode.

Proteus uses the PowerPC MMU for memory management and each VM running on the hypervisor has its own dedicated address space that is statically defined. For temporal isolation, Proteus supports different configurations of core support for virtual machines. VMs can be dedicated to one core or can be divided among multiple cores. The hypervisor uses a fixed time slice approach to scheduling, based on statically assigned priorities.

Proteus is an interesting virtualization architecture to consider for space due to its claimed support for real-time systems, the fact that it does not rely on hardware virtualization support and its compatibility with the PowerPC processor family, which is a common space system processor (Ginosar, 2012). It is unclear however, what type of real-time systems Proteus can actually support and whether or not the hypervisor is suitable for hard real-time applications.

6. X-Hyp

X-Hyp is a type-1 embedded hypervisor that supports paravirtualization and is designed specifically for real-time systems. The hypervisor comes with a paravirtualized version of FreeRTOS and supports Linux, RTEMS and µcos and is compatible with the ARM-9 and Cortex processors. X-Hyp is available under both commercial licensing and open-source. The hypervisor has its own API with 54 hypercalls for the ARM processor. Figure 24 shows the basic architecture of X-Hyp. X-Hyp has little documentation but supports some valuable characteristics that make it worth mentioning, including its support for three RTOSs used in space and its availability as an open-source product.

Figure 24. The Basic X-Hyp Architecture (from "X-hyp Paravirtualized," n.d.)

7. RT-Xen

RT-Xen is a Washington University project focused on incorporating soft real-time guarantees into the open source Xen hypervisor. The Office of Naval Research awarded a three-year grant to make RT-Xen a real-time virtualization architecture for embedded systems ("RT-Xen Project," 2013). RT-Xen incorporates *resource reservations* into domain scheduling and adds real-time schedulers at the kernel and the domain level. The kernel scheduler is responsible for managing the scheduling of each domain based on configuration data provided by Dom0. The configuration data includes priority levels of the domains, their allotted time slice, the processor cores they are allowed to run on and the amount of processor power they get allocated. This scheduler uses either an earliest deadline first or rate monotonic policy to manage domain scheduling. Within each domain, there is another real-time scheduler responsible for scheduling its own processes (Xi, Wilson, Lu, & Gill, 2011; "Xen Project: RT-Xen," n.d.).

Though not designed for space systems, RT-Xen is an interesting technology that might be considered in conjunction with ARLX. Whereas ARLX is designed for high assurance, RT-Xen is designed for real-time guarantees, both of which are attributes required for mission-critical space systems. RT-Xen, however, is only meant to meet soft real-time requirements and suffers from the same drawbacks as ARLX, namely that it is based on a large, legacy code base not intended for high assurance applications.

THIS PAGE INTENTIONALLY LEFT BLANK

V. REMOTE FINGERPRINTING OF VIRTUALIZED OPERATING SYSTEMS

In this chapter, we discuss our work in measuring and comparing fingerprints for virtualized operating systems, employing methods explored previously by Chen et al. (2008). We use TCP timestamp measurements to derive a timestamp skew, which prior work shows can be used to characterize some operating systems remotely. Our work focuses both on (1) validating prior experiments with fingerprinting general-purpose operating systems under different virtualization scenarios, and (2) extending these results to real-time systems, using Real-Time Linux (i.e., Linux with the PREEMPT_RT patch enabled) as a target.

A. MOTIVATION

The ability to remotely fingerprint a guest operating system running as a virtual machine is valuable for the reconnaissance phase of system exploitation. Since remote fingerprinting does not require direct access to the machine, the fingerprint of virtualized operating systems can help detect virtual honeynets and enable adversaries to exploit hypervisor-specific vulnerabilities, if the fingerprint of the guest OS leaks information about their underlying hypervisor. To the best of our knowledge, there is no prior work measuring TCP timestamp skew on real-time operating systems, either running directly on hardware or as a guest on a hypervisor.

B. TEST METHODOLOGY

In our experiment, we compare the TCP timestamp skew variation between operating systems running on bare metal and on a virtualized platform. We do this by replicating prior work in TCP timestamp fingerprinting.

1. TCP Timestamp Option

The TCP timestamp option (*TSopt* field) is an optional 32-bit field in the TCP packet header that was first introduced in 1992 in RFC 1323. Its purpose was to improve performance and provide reliable operation over paths with high speed (Jacobson, 1992).

The timestamp is a number that represents the perception of time for each party in every packet of a TCP flow. RFC 1323 states that the timestamp measurement should be taken from a virtual clock that "must be at least approximately proportional to real time" (Jacobson, 1992, section 3.3). The virtual clock is not required to be synchronized with the system clock and is often independent of a system's adjustments if network time protocol (NTP) is enabled. This virtual clock is usually reset every time a system is rebooted. The TCP timestamp clock increases monotonically with a predefined frequency between 1 and 1,000 Hz.

The timestamp option is enabled if the initiator of the TCP flow includes a TSopt payload with a timestamp value in its original SYN packet and if the reply indicates that both hosts implement the option. For the fingerprinting methodology we employ, we require the remote host to support the TCP timestamp option and have open ports that can be used to initiate a TCP session.

2. Prior Work

Chen et al. (2008) extend techniques originally introduced by Kohno et al. (2005) for remote OS fingerprinting. Chen et al. (2008) examine timestamp skew behavior between (unspecified versions of) Windows and Linux, both running on bare metal and running as virtualized guest operating systems on either VMWare or Xen. In their experiment, they send several hundred SYN packets to the target host for an unspecified amount of time. They calculate the frequency at which the TCP timestamp clock increases and use this to calculate the skew of the target's time source. This is achieved by comparing the actual time the target's response packet is received and the time recorded in the response's TCP options. The perceived skew is measured over time and used to generate a mean squared error (MSE) or *randomness indicator* associated with the target. They compare the MSEs associated with bare metal and virtualized targets, concluding that virtualized operating systems can be fingerprinted based on MSE behavior. In particular, Chen et al. (2008) suggest skew can be used to distinguish virtualized systems from bare metal systems, and to distinguish identical guest OSes hosted on different hypervisors.

C. TEST PLAN

We conduct all tests in an isolated environment on a small local network. Our test environment consists of five Optiplex 755 desktop machines with Intel Duo Core CPUs and 8GB of RAM. One of these machines, called *sniffer*, serves as the active host performing remote fingerprinting. The remaining machines (M1, M2, M3, M4) act as targets in various configurations (see Table 8). Details of the versions of the hypervisors and operating systems used in the M1–M4 host configurations are summarized in Table 9. The *sniffer* machine employs the same version of Fedora 19 used in the target host configurations. All virtualized configurations are run in full virtualization mode, meaning the guest operating system is unaware that it is being virtualized. Xen supports full virtualization by using Qemu (see Chapter II).

Table 8. Target Host Configuration Summary

NOTATION	CONFIGURATION	Type of Virtualization	IP ADDRESS	MACHINE
[F]	Fedora 19 bare metal	-	10.10.10.2	M1
[F/F]	Fedora 19 running VMWare with Fedora 19 guest	Full	10.10.10.21	M1
[W/F]	Fedora 19 running VMWare with Windows 7 guest	Full	10.10.10.22	M1
[RT/F]	Fedora 19 running VMWare with PREEMPT_RT guest	Full	10.10.10.23	M1
[X]	Xen bare metal	-	10.10.10.3	M2
[F/X]	Xen running Fedora 19 guest / DomU	Full	10.10.10.31	M2
[W/X]	Xen running Windows 7 guest / DomU	Full	10.10.10.32	M2
[RT/X]	Xen running PREEMPT_RT guest / DomU	Full	10.10.10.33	M2
[RT]	PREEMPT_RT bare metal	-	10.10.10.4	M3
[W]	Windows 7 bare metal	-	10.10.10.5	M4
[F/W]	Windows 7 running VMWare with Fedora19 guest	Full	10.10.10.51	M4
[W/W]	Windows 7 running VMWare with Windows 7 guest	Full	10.10.10.52	M4
[RT/W]	Window 7 running VMWare with PREEMPT_RT guest	Full	10.10.10.53	M4

Table 9. Target Host Software Summary

Name	VERSION
Fedora 19	32-bit 3.14.23-100.fc19.i686.PAE Linux kernel
Windows 7	32-bit Windows 7 Professional 6.1.7601 Service Pack 2
Linux with PREEMPT_RT patch	Ubuntu 12.04.3-desktop-i386 with the Linux 3.12.1-rt4 kernel
VMWare	VMWare Workstation 10.0.3 build-1895310
Xen	Xen-3.0-x86_64
Xen Dom0	Debian 3.2.0-4-amd64

In the test environment: all machines are connected to a local switch; IP addresses are statically assigned; firewalls and Network Time Protocol services are disabled on operating systems and all hypervisors use bridged devices for networking.

1. Hardware and Software Decisions

The intent of our test environment and target host configurations is to replicate prior work as closely as possible; however, Chen et al. (2008) did not indicate the specific versions of operating systems or hypervisors they employ. Further, Kohno et al.'s (2005) experiments, cited by Chen et al. (2008), employ software that (presumably) was current circa 2005. We had no selection criteria beyond VMWare Workstation, Xen and some Linux distribution, considered current as of 2005 or 2008. Thus, selecting newer software was not a criterion for us.

Hardware decisions were based on the availability of five machines with identical physical profiles. We chose VMWare Workstation 10 because we were unable to obtain an older version of VMWare. Our choice of Windows 7 Service Pack 2 was based on its compatibility with VMWare Workstation 10 and its status as an older but still heavily used Windows distribution. We chose Xen release 3.0 with Debian running in Dom0 because installation instructions were readily obtainable. We chose Fedora 19 because one of our planned[10] target configurations used RTEMS, whose build instructions required Fedora 19. We chose real-time Linux using the PREEMPT_RT patch because it is open-source and readily available. Our decision to build real-time Linux using Ubuntu 12.04-LTS with the PREEMPT_RT patch was based on forum recommendations (Ask Ubuntu, n.d.) suggesting this is a stable distribution for which the patch works, and based on availability of patch instructions.

2. Test Execution

For each test configuration, we capture two separate TCP sessions with *sniffer*, one 90 minutes long and one 10 minutes long. For each session, we probe each host

[10] Later, we abandoned employing RTEMS in our experiments, due to difficulty in configuring the RTOS to run on our physical machine profile.

configuration through banner grabbing with *netcat*. During each session, we capture all traffic using *tcpdump*. Table 10 summarizes the ports used for each operating system. Chen et al. (2008) only capture SYN packets, whereas we capture all packets in the session.

Table 10. Ports/Services Used to Generate TCP traffic

OS	PORT	SERVICE
Fedora	22	SSH
Windows	445	Active Directory
Xen (Debian Dom0)	111	RPC
PREEMPT_RT	22	SSH

To obtain TCP timestamp values from a TCP session, we employ a Python script (*tcp_skew.py*) written by Russell Fink of the University of Maryland, Baltimore (Fink, n.d.) to parse the packet capture. For each packet, this script extracts the time recorded in the options field of the TCP packet (*T*) and the timestamp recorded by tcpdump running on *sniffer* (*t*). Figure 25 shows sample output from this script.

	t	***T***
source IP address	Packet receipt time (measurer time) Unix Timestamp (03 Dec, 2014)	TCP timestamp value (sender time)
10.10.10.21	1417643046.053868055344	191846072
10.10.10.21	1417643046.054200887680	191846072
10.10.10.21	1417643046.065603017807	191846083
10.10.10.21	1417643046.065854072571	191846084
10.10.10.21	1417643046.065918922424	191846084
10.10.10.21	1417643048.146640062332	191848165
10.10.10.21	1417643048.147005081177	191848165
10.10.10.21	1417643048.164706945419	191848182
10.10.10.21	1417643048.165000915527	191848183
10.10.10.21	1417643048.165066957474	191848183
10.10.10.21	1417643050.260829925537	191850279

Figure 25. Example Output from tcp_skew.py Code

We normalize measurements for each session by subtracting the time associated with the start of packet capture, using another script (*tcp_clock.py*). In particular, this script calculcates (T_i - T_0) for TCP timestamps and (t_i - t_0) for tcpdump timestamps. Using these values, we adapt the formula of Chen et al. (2008) for calculating the target's clock

frequency. Chen's original formula is F = $(T_1 - T_2) / (t_1 - t_2)$. We use F = $(T_{last} - T_0) / (t_{last} - t_0)$, believing this may provide a similarly accurate reading. We validate this assumption in testing (see Observation 2).

We use the derived frequency *F* for each operating system to generate clock readings. We translate TCP timestamps into a set of clock readings following Chen et al. (2008), by calculating $(T_i - T_0)/F$. There are two clocks that can be compared with these values: the time elapsed locally ($x_i = t_i - t_0$) and the time elapsed on the target ($w_i = (T_i - T_0)/F$). For each configuration, we generate a scatter plot of the target's skew, plotting time elapsed on *sniffer* (x_i) on the x-axis and the skew ($y_i = w_i - x_i$) on the y-axis. Appendices A through G include all graphs generated for our experiment.

Given the calculated skew, we use Chen et al.'s (2008) method to calculate the MSE for each configuration. We use linear least-squares fitting to find a best-fit line, *f(x)* for the timeseries data. We calculate the MSE for the best-fit line by adding the squares of the offsets and dividing by the number of TCP packets in the traffic capture, *N* (See Figure 26). Chen et al. (2008) characterize the MSE as a *randomness indicator*, to be used as the baseline for comparison between bare-metal and virtualized operating systems.

$$\frac{\sum_i \left[f(x_i) - y_i \right]^2}{N}$$

Figure 26. MSE Equation

3. Test Notation

Given the number of configurations we are testing, we require a simplified means of characterizing our observations. We have therefore developed configuration notation for the purposes of summarizing individual tests and sets of tests (see Tables 8 and 11). In cases where we describe multiple configurations, we use variables. For example, MSE[*A*/X] is equivalent to the set MSE values MSE[F/X], MSE[W/X], MSE[RT/X]. Comparing the MSE values MSE[*A*/F] and MSE[*B*] is equivalent to comparing all pairs between sets {MSE[F/F], MSE[W/F], MSE[RT/F]} and {MSE[F], MSE[W], MSE[X], MSE[RT]}. The notation MSE[RT-*S*] indicates MSE[RT-1FF] and MSE[RT-1RR].

Similarly, comparing MSE[RT-*S*/F] and MSE[RT-*T*/W]} is equivalent to considering all pairwise comparisons between sets {MSE[RT-1FF/F], MSE[RT-1RR/F]} and {MSE[RT-1FF/W], MSE[RT-1RR/W]}. When considering sets of MSE values S1 and S2, we abuse notation: S1≈S2 means 'the MSE values of S1 are similar to those of S2', S1>S2 means 'the MSE values of S1 are large compared to those of S2,' and S1≠S2 means 'the MSE values of S1 are dissimilar to those of S2.'

Table 11. Experiment Notation Summary

Notation	Meaning
T_i	TCP timestamp *i*
t_i	tcpdump timestamp *i*
F	Frequency of target configuration
x_i	Time elapsed on *sniffer* (x-axis of scatter plot)
w_i	Time elapsed on target configuration (based on TCP timestamp)
y_i	TCP clock skew (y-axis of scatter plot)
RT-1FF	PREEMPT_RT configured with *sshd* process prioirty 1, FIFO scheduling
RT-1RR	PREEMPT_RT configured with *sshd* process prioirty 1, round-robin scheduling

D. ANALYSIS

We validate many of Chen et al.'s (2008) original findings; however, we find one of their conclusions—that virtualized operating systems can be easily fingerprinted because of their dramatically different TCP time skew variation—is not entirely convincing in light of our experimentation with some (previously unevaluated) configurations. We divide the analysis that follows into a series of individual observations.

1. Observation 1: MSE Is Not Sensitive to Session Length

Chen et al. (2008) do not specify the amount of time they run each packet capture but state that experiments conclude within "a few minutes." We want to determine if the length of the packet capture has any impact on the MSE calculation. We do this by comparing two packet captures for each bare metal target ([F], [W], [X] and [RT]). We find that that the average MSE difference between 10 minute and 90 minute captures is 0.026ms, leading us to conclude that the capture length does not have a significant impact

on MSE calculation. Figure 27 shows the time series data for [F] under both time frames (see Appendix A and B for other configurations). The time values on these two packet captures are different since packet times vary for each packet capture. This explains the visually incongruous lines in Figure 27. The skew behavior however is comparable. We conclude that Chen et al.'s (2008) "a few minutes" timeframe provides a relatively stable MSE calculation, as longer time frames do not significantly impact these calculations. Based on this observation, we conduct all subsequent tests using 10-minute packet captures.

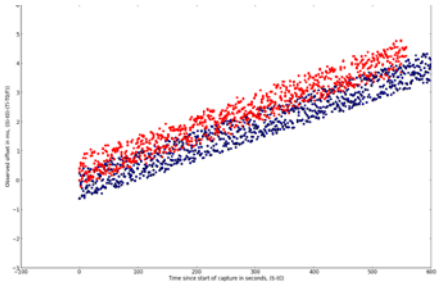

Figure 27. Configuration [F], Skew vs. Time, 1.5 hour Capture (Blue) and 10-Minute Capture (Red)

2. Observation 2: Frequency Calculation Appears Relatively Stable with Respect to Packet Selection

Chen et al. (2008) present a method for measuring the operating systems' TCP clock frequency remotely. As explained in Section 3, we modify their equation by looking at the first and last timestamps: $F = (T_{last} - T_0) / (t_{last} - t_0)$. We verify that our modified equation has no impact to this calculation after rounding the result to the nearest real frequency interval, as Chen et al. (2008) suggest. To confirm that the choice of packets used to calculate frequency is arbitrary, we calculated $(T_j - T_i) / (t_j - t_i)$ for every $j>i>0$. These calculations also have no impact to the frequency calculation. For our Linux configurations we compare our result to the actual operating systems' clock frequency configuration by looking at the kernel configuration file. We do not do this for our Windows configurations because, to our knowledge, this information is not accessible within Windows. Table 12 summarizes our frequency results.

Table 12. Frequency Results

Operating System	**Calculated Frequency (Hz)** $(T_{last} - T_0) / (t_{last} - t_0)$	**Calculated Frequency (Hz)** $(T_j - T_i) / (t_j - t_i)$	**Reported Host TCP Clock Frequency (Hz)**
[F]	1000	1000	1000
[W]	100	100	N/A
[X]	250	250	250
[RT]	250	250	250

3. Observation 3: MSE[*A*] ≠ MSE[*B*] (for all *A*≠B, except [RT])

Chen et al. (2008) observe different MSE behaviors for Windows running on bare metal and for Linux running on bare metal. They did not record a bare metal MSE value for Xen's Dom0. Chen et al. (2008) found the MSE value for bare metal Windows is very high, attributing this to that configuration yielding the lowest measured frequency value (10Hz) among all target configurations. Our results match Chen's as illustrated in Table 13. Excluding [RT] configurations, all our bare metal configurations exhibit dissimilar MSE behavior. In agreement with Chen et al.'s observations, our [W] configuration has the highest MSE value, possibly due to its low clock frequency compared to that of other configurations (see Table 13).

Table 13. Bare Metal MSEs Excluding [RT]

CONFIGURATION	**MSE (ms)**
[W]	8.156
[F]	0.086
[X]	1.427

4. Observation 4: No Obvious Difference in MSE Behavior between Virtualized and Bare Metal Configurations

Chen et al. (2008) conclude that virtualized hosts have "more perturbed clock skew behavior" than bare metal hosts which they claim is observable through MSE. Our results also reflect a difference between bare metal and virtualized MSE but less pronounced than in prior work.

a. Observation 4a: MSE[F/A] ≈ MSE[F]

Chen et al. (2008) conclude that virtualized instances of Linux exhibit orders of magnitude larger MSE than Linux running on bare metal. In particular, their results show almost 300,000% change between bare metal Linux and Linux on VMWare, and 173% change (Chen et al., 2008)[11] between bare metal Linux and Linux running on Xen. We find our virtualized Fedora configurations demonstrate at least one order of magnitude change compared to the bare metal configuration, but the changes are smaller than Chen et al.'s observations suggest (see Table 14).

Table 14. Linux MSE Results

CONFIGURATION	MSE (ms)	DIFFERENCE (ms) from MSE[F]	CHANGE % from MSE[F]
[F]	0.086	-	-
[F/F]	0.221	-0.134	-156%
[F/W]	0.756	-0.67	779%
[F/X]	1.586	-1.5	-1744%

b. Observation 4b: MSE[W/A] ≈ MSE[W]

Chen et al. (2008) find noticeable differences in MSE behavior among virtualized and bare metal Windows configurations. In particular, they observe a 22% change between bare metal Windows and Windows running on VMWare and an 8% change between bare metal Windows and Windows running on Xen. Chen et al. (2008) claim these changes are statistically meaningful under Z-test analysis, making "the randomness introduced by VMM very obvious." We find, however that there is not a substantial difference between [W] and its virtualized counterparts ([W/W], [W/F], [W/X]) in terms of MSE. In fact, comparing [W] with [W/W], changes in MSE behavior appears fairly negligible (see Table 15).

[11] See 0.083 ms^2 MSE for baseline Linux and 245.8 ms^2 MSE for Linux on VMWare; we calculate difference as ((0.083-245.8)/0.083)*100.

Table 15. Windows Configuration MSEs

CONFIGURATION	MSE (ms)	DIFFERENCE (ms) from MSE[W]	CHANGE % from MSE[W]
[W]	8.156	-	-
[W/W]	8.066	0.09	1.1%
[W/F]	6.873	1.283	15.7%
[W/X]	8.658	-0.502	-6%

5. Observation 5: MSE[*A*/F] ≠ MSE[*A*/W]

Chen et al. (2008) do not clarify what configuration of VMWare they use in their experiment and do not comment on any difference in behavior of VMWare on Windows vs. VMWare on Linux. We find that configurations [*A*/W] and [*A*/F] appear different in terms of MSE, suggesting that the host OS for VMWare Workstation impacts fingerprinting substantially (see Table 16).

Table 16. MSE[*A*/W] vs. MSE[*A*/F]

[*A*/W] CONFIGURATION	MSE (ms)	[*A*/F] CONFIGURATION	MSE (ms)
[F/W]	0.756	[F/F]	0.221
[W/W]	8.066	[W/F]	6.873

6. Observation 6: MSE[*A*/X] > {MSE[*A*/F], MSE[*A*/W], MSE[*A*]} for all *A* ≠ [RT]

Chen et al. (2008) observe that Windows on Xen and Linux on Xen exhibit smaller MSE values than Linux on VMWare and Windows on VMWare. They suggest "Xen introduces much less randomness than VMWare does, probably because they have different algorithms for firing software interrupts." In contrast, we observe [F/X] demonstrates higher MSE than [F], [F/W] or [F/F] (see Table 14); also, [W/X] demonstrates higher MSE than [W], [W/W] or [W/F] (see Table 15). This contradicts Chen et al.'s observations that Xen introduces less randomness than VMWare. It is, however, in-line with their larger observation that one can observe MSE differences among hypervisors, albeit somewhat more limited.

7. Observation 7: MSE[RT] ≠ MSE[*A*] for all *A* ≠ RT

We extend the prior work of Chen et al. (2008) to consider fingerprinting an RTOS. Our [RT] configuration's MSE value is different from the MSE values of other bare metal configurations as illustrated in Table 17. This result agrees with our findings in Observation 3; i.e., that bare metal configuration MSE behaviors are dissimilar from one another.

Table 17. Bare Metal MSE with [RT] Configuration

CONFIGURATION	Calculated MSE (ms)
[W]	8.156
[F]	0.086
[X]	1.427
[RT]	1.337

8. Observation 8: MSE[RT] ≈ MSE[RT/F] ≈ MSE[RT/W] ≈ MSE[RT/X]

We observe that, relative to configuration [RT], its virtualized counterparts demonstrate the lowest MSE difference among all our virtualized configurations. When these differences are translated into percentages however, it appears that virtualized instances of [RT] do not display any substantial change compared to other [RT/*A*] configurations. Table 18 summarizes our findings with the [RT] configuration i.e., using sshd's default service priority[12] and the default scheduling policy, Linux Completely Fair Scheduler (SCHED_NORMAL). This observation agrees with Observation 4, where we find no obvious difference between [*A*] configurations and their [*A/B*] counterparts, contrary to Chen et al.'s findings.

[12] Default priority is 0 since there is no prioritization associated with SCHED_NORMAL, which is the default universal time-sharing scheduler policy in our configuration.

Table 18. PREEMPT_RT Configuration MSEs

CONFIGURATION	MSE	DIFFERENCE (ms) from MSE[RT]	% CHANGE from MSE[RT]
[RT]	1.337	-	-
[RT/F]	1.395	-0.058	-4.3%
[RT/W]	1.788	-0.451	-34%
[RT/X]	1.297	0.04	2.99%

9. Observation 9: MSE[RT] ≈ MSE[RT-1FF] ≈ MSE[RT-1RR]

We observe differences in MSE behavior when altering the PREEMPT_RT configuration in terms of target service priority and scheduling policy (Round Robin vs. FIFO).

For configuration [RT-1FF], we make the following two changes: we adjust the priority of sshd using the *chrt* command to be priority 1, i.e., the highest process priority level; we change the scheduling class to FIFO. For configuration [RT-1RR] we make the same changes but use Round Robin scheduling class instead of FIFO. We find these [RT-*S*] configurations have similar MSE behavior relative to our [RT] configuration. Table 19 summarizes our findings for the FIFO configurations and Table 20 summarizes our findings for the Round Robin configurations.

Table 19. PREEMPT_RT with sshd Priority 1, FIFO Scheduling Class

CONFIGURATION	MSE	DIFFERENCE (ms) from MSE[RT-1FF]	%CHANGE from MSE[RT-1FF]
[RT-1FF]	1.387	-	-
[RT-1FF/F]	1.343	0.044	3.17%
[RT-1FF/W]	12.669	-11.282	-813%
[RT-1FF/X]	1.343	0.044	3.17%

Table 20. PREEMPT_RT with sshd Priority 1, Round-Robin Scheduling Class

CONFIGURATION	MSE	DIFFERENCE (ms) from MSE[RT-1RR]	% CHANGE from MSE[RT-1RR]
[RT-1RR]	1.315	-	-
[RT-1RR/F]	1.404	-0.089	-6.7%
[RT-1RR/W]	11.349	-10.034	-763%
[RT-1RR/X]	1.317	-0.002	-0.15%

10. Observation 10: MSE[RT-*S*/W] > {MSE[RT], MSE[RT-*T*], MSE[RT/*A*]}

We observe our [RT-*S*/W] configurations result in a much higher MSE than all other [RT] configurations, indicating that Windows 7 has an impact on our [RT] configuration when scheduling class and process priority are altered. Table 21 lists MSEs for other configurations not listed in Tables 19 and 20 as points of comparison.

Table 21. MSE[RT-*S*/W] vs. other MSE[RT] Configurations

CONFIGURATION	MSE (ms)
[RT-1FF/W]	12.669
[RT-1RR/W]	11.349
[RT]	1.337
[RT/W]	1.788
[RT/F]	1.395
[RT/X]	1.297

11. Observation 11: MSE[RT-*S*/*A*] ≈ MSE[RT-*T*] ≈ MSE[RT] for *A* ≠ W

We observe that, aside from our [RT-*S*/W] configurations, MSE for [RT-*S*/*A*] are similar to both [RT] and [RT-*S*] configurations. This observation agrees with Observations 4, 8 and 9. Continuing the trend in Observations 4 and 8, we see no obvious difference in MSE between [RT-*S*/A] and [RT-*S*]. Combined with Observation 9 on the similarity between [RT-*S*] and [RT-*T*], this implies the similarity in MSE for all configurations [RT-*S*/*A*] compared to [RT] (see Tables 19, 20 and 21).

12. Observation 12: MSE[RT-*S*/*A*] ≈ MSE[RT-*T*/B] for A, B ≠ W

We observe that, aside from our [RT-*S*/W] configurations, the MSE behavior for all virtualized [RT-*T*] configurations is similar. In fact, our results show identical MSE for [RT-1FF/X] and [RT-1FF/F] (see Tables 19, 20 and 21).

13. Observation 13: [*A/B*] is more like [*A*] than [*B*] for *A* ≠ B and *A*≠F

Chen et al. (2008) do not report MSE comparing the virtualized guest and its bare metal host. We extend this work by investigating which MSE virtualized guests most closely resemble. We find that (with the exception of our [F/*B*] configurations) all [*A/B*] configurations more closely resemble the MSE of [*A*] instead of [*B*]. Table 22 summarizes our findings.

Table 22. MSE Comparisons (Blue Indicates Most Similar MSE Based on %)

CONFIG	MSE	DIFFERENCE from MSE[W]		DIFFERENCE (ms) /CHANGE (%) from MSE[F]		DIFFERENCE (ms) /CHANGE (%) from MSE [X]		DIFFERENCE (ms) /CHANGE (%) from MSE [RT]	
		(ms)	%	(ms)	%	(ms)	%	(ms)	%
[F/W]	0.756	7.404	90.780	-0.670	-779.068	-	-	-	-
[F/X]	1.586	-	-	-1.500	-1744.186	-0.166	-11.638	-	-
[W/F]	6.873	1.287	15.780	-6.787	-7891.860	-	-	-	-
[W/X]	8.658	-0.498	-6.106	-	-	-7.238	-509.218	-	-
[RT/F]	1.395	-	-	-1.309	-1522.093	-	-	-0.058	-4.338
[RT/W]	1.788	6.372	78.127	-	-	-	-	-0.451	-33.732

E. DISCUSSION

The purpose of our study is to replicate the work of Chen et al. (2008) to investigate if their observations appear relatively stable, and generalize to real-time systems under virtualization. Overall, our experiments show that fingerprinting behavior of VMWare Workstation guests is dependent on the underlying host operating system. Our work also shows that when using MSE as a metric to compare virtualized operating systems, there is no easily observable difference between operating systems running on different hypervisors.

As with Chen et al. (2008), our work suggests there is a strong correlation between an operating system's TCP clock frequency and its fingerprint. Operating systems with lower frequency values (e.g., Windows) have higher MSE values and lower percentages of difference between baseline and experimental MSE values. Operating

systems with higher frequency values (e.g., Fedora) have lower MSE values and high percentages of difference between baseline and experimental MSE values.

Our work also reveals some interesting behavior of virtualized operating systems, particularly in the [RT-1FF/W] and [RT-1RR/W] configurations. The MSE behavior for these configurations is dramatically different from [RT], [RT-*S*] and [RT-*T/A*] configurations. Of note is the observation that only the [F/F] and [F/W] configurations have MSE behavior that more closely resembled the host OS instead of the guest. We investigate the reason for this behavior as future work.

There are several limitations to our experiment that may have impacted the generality of our results. Our setup lacked extraneous network and CPU load, as host and guest had limited background processes running and had exclusive use of a local network. As future work, these experiments may be re-run on a typical network for an enterprise or in a setting with multiple processes competing for CPU time to see if the results change. We also do not run our experiments on multiple physical machine profiles. To confirm the generality of our observed behaviors one would re-run these experiments on different physical machine profiles, i.e., to investigate how much TCP timestamp skew variation can be attributed to the operating system and how much can be attributed to the hardware. Also, all the tested virtualized configurations are based on full virtualization. We suggest re-running our tests with different virtualization settings, such as paravirtualization and hardware-assisted virtualization to see how MSE behavior compares.

A possible limitation of our work is the use of tcpdump to label time of receipt for each TCP packet at the *sniffer* machine. We suggest re-running these experiment to employ a system clock timestamp, rather than relying on a user-land application's perception of time. Our experiment could also benefit if the operating system choices were more consistent. We chose a different version of Linux with a different frequency to run our Xen Dom0 ([X] configuration) compared to our other Linux configurations. We suggest standardizing these software choices for consistency and comparison. We further suggest experimenting with different Linux distributions and different kernel versions. It would be interesting to see how our results compare to newer operating systems. Finally,

additional research should consider statistical metrics for comparison to see if they offer more insight into the behavior of different hypervisors and virtualized operating systems in the context of fingerprinting.

Our work is an attempt to capture the TCP timestamp skew behavior of a set of general-purpose and real-time operating systems in an isolated, controlled environment. Our results differ from Chen et al. (2008) and suggest that hypervisor and operating system fingerprinting is not clearly predictable from MSE. We propose some future work to carry this research forward.

VI. CONCLUSION AND FUTURE WORK

Virtualization is a promising field of research for the space community, and its implementation in space research projects indicates that it is a technology that the space community appears committed to utilizing. In this thesis we have sought to highlight some key security-relevant properties of real-time operating systems and virtualization architectures for space systems. Our work has revealed the diversity of architectures supporting virtualized for the space domain, and the ways in which these virtualization architectures handle real-time requirements of guests. Our work highlights some tradeoffs associated with security, flexibility, popularity and compatibility with other systems and hardware. The purpose of our survey was to explain, at a high level, the fundamental differences and similarities between real-time operating systems and virtualization solutions for space. A limitation of this survey was that we did not analyze the implementation of consequential security features in the surveyed systems. We leave as future work the analysis of enforcement mechanisms for key security functionality, such as memory management or spatial isolation. For unevaluated systems, penetration testing may be warranted to investigate these security properties.

We have also presented an experimental investigation of remote fingerprinting based TCP timestamp skew for virtualized operating systems. This extended prior work, considering timestamp skew behavior for the Linux PREEMPT_RT patch running on bare metal, on Xen and on VMWare Workstation. We suggest (see Chapter V) continuation of this work is warranted, by re-running experiments on a public network, on different hardware, with different virtualization setttings etc. We also leave as future work the inclusion of other real-time operating systems in this evaluation, such as RTEMS and FreeRTOS, as well as alternative virtualization platforms such as XtratuM and NOVA and others surveyed in this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. BARE METAL, 1.5-HOUR RUN

In this appendix, we provide data associated with experiments discussed in Chapter V. Figures 28–31 show the results of experiments with bare metal configurations ([F], [X], [W], [RT]) after 1.5-hour packet capture.

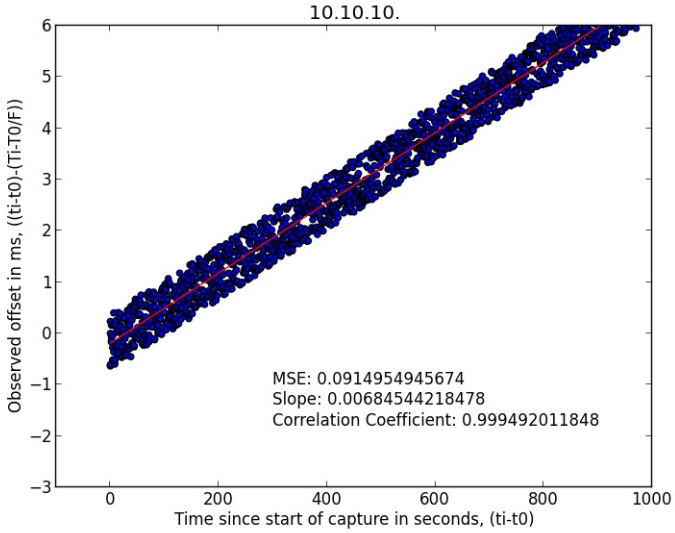

Figure 28. Configuration [F], Skew vs. Time, 1.5 Hour Packet Capture

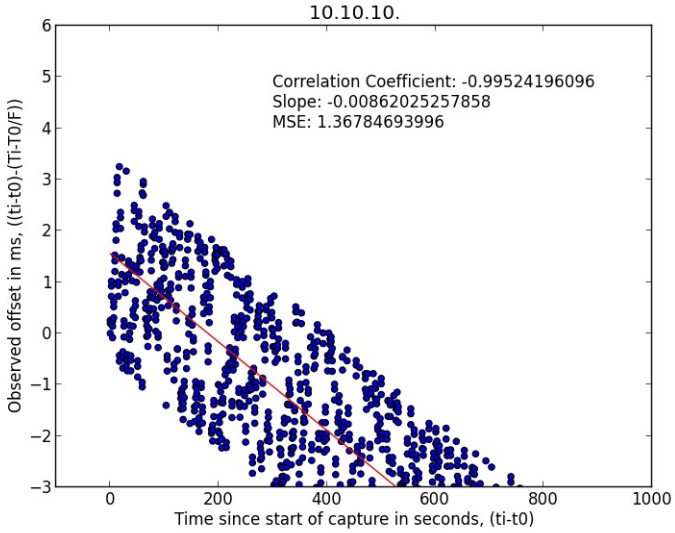

Figure 29. Configuration [X], Skew vs. Time, 1.5 Hour Packet Capture

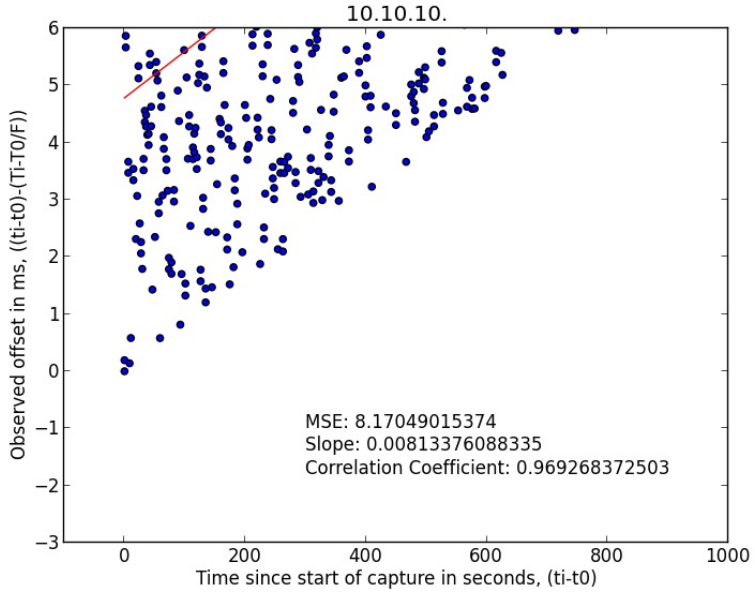

Figure 30. Configuration [W], Skew vs. Time, 1.5 hour Packet Capture

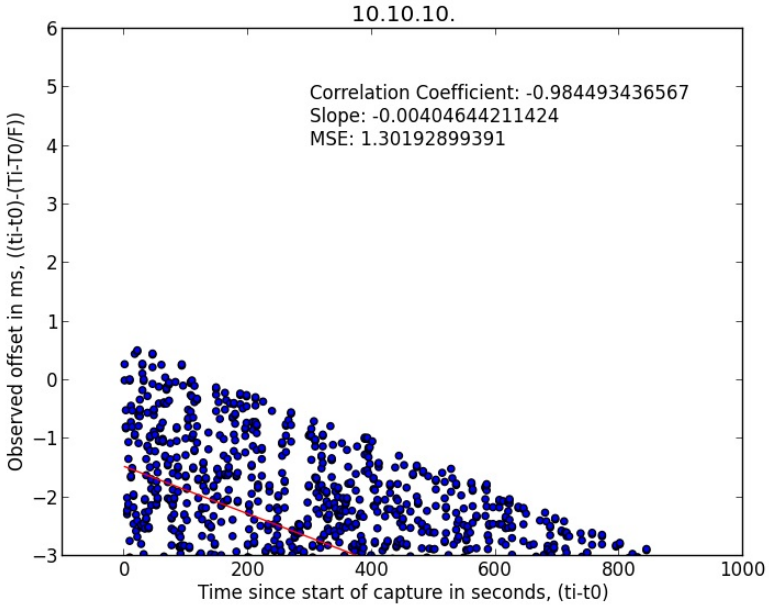

Figure 31. Configuration [RT], Skew vs. Time, 1.5 Hour Packet Capture

APPENDIX B. BARE METAL, 10-MINUTE RUN

In this appendix, we provide data associated with experiments discussed in Chapter V. Figures 32–35 show the results of experiments with bare metal configurations ([F], [X], [W], [RT]) after 10-minute packet capture.

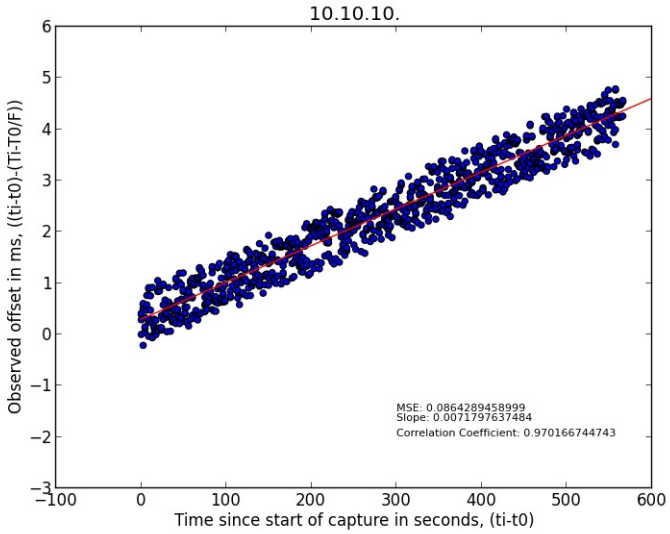

Figure 32. Configuration [F], Skew vs. Time, 10-Minute Packet Capture

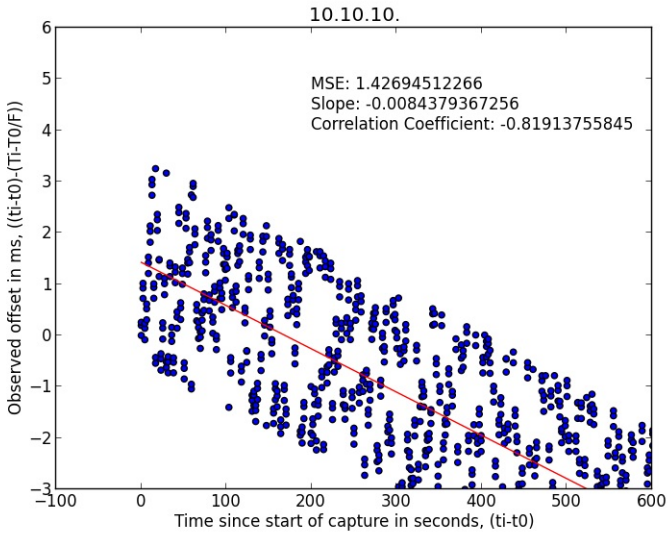

Figure 33. Configuration [X], Skew vs. Time, 10-Minute Packet Capture

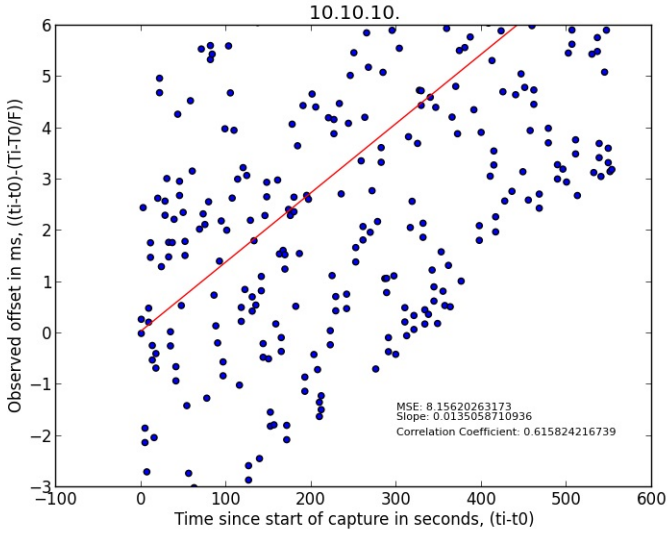

Figure 34. Configuration [W], Skew vs. Time, 10-Minute Packet Capture

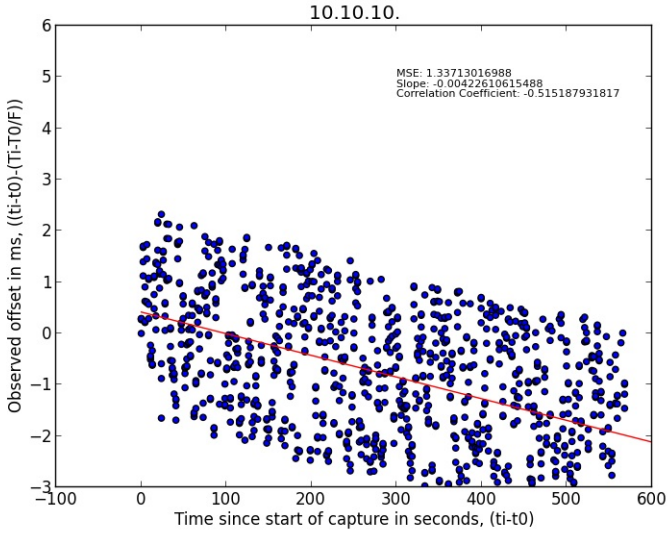

Figure 35. Configuration [RT], Skew vs. Time, 10-Minute Packet Capture

APPENDIX C. VIRTUALIZED LINUX

In this appendix, we provide data associated with experiments discussed in Chapter V. Figures 36–38 show the results of experiments with virtualized Linux configurations ([F/F], [F/W], [F/X]).

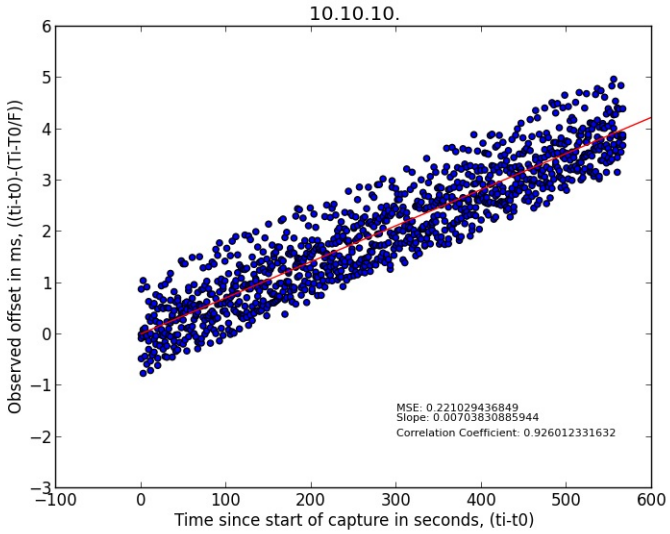

Figure 36. Configuration [F/F], Skew vs. Time

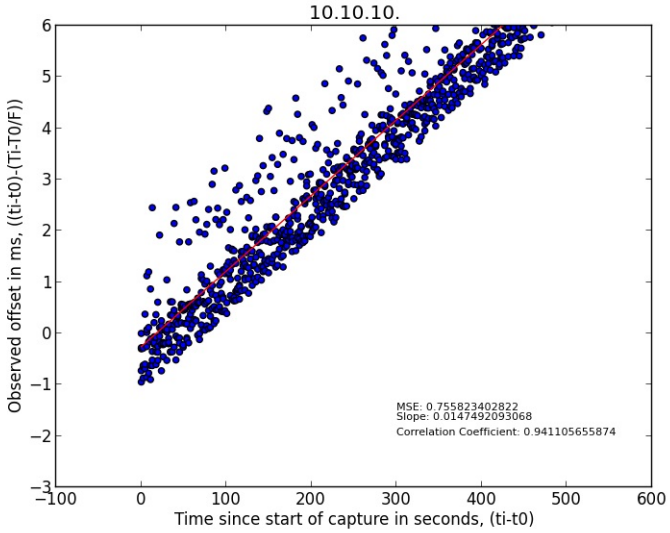

Figure 37. Configuration [F/W], Skew vs. Time

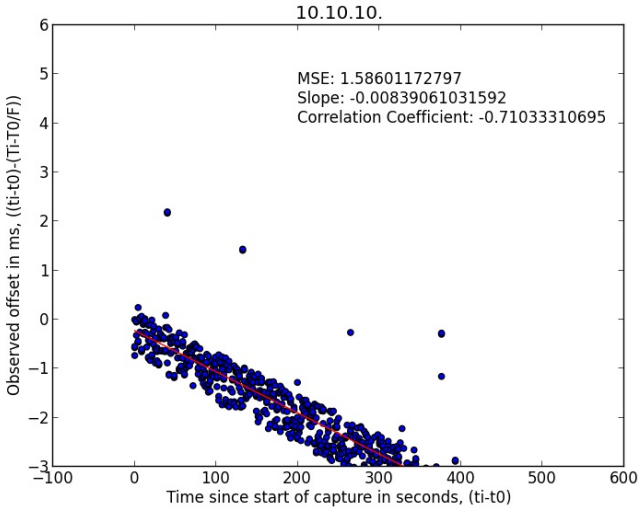

Figure 38. Configuration [F/X], Skew vs. Time

APPENDIX D. VIRTUALIZED WINDOWS

In this appendix, we provide data associated with experiments discussed in Chapter V. Figures 39–41 show the results of experiments with virtualized Windows configurations ([W/F], [W/W], [W/X]).

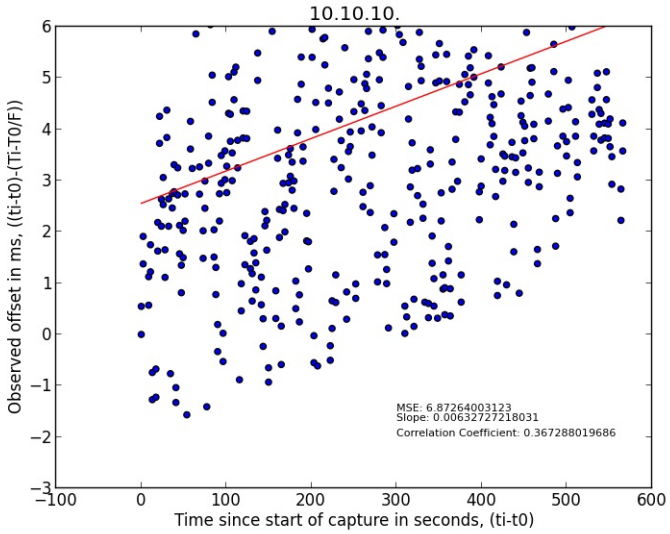

Figure 39. Configuration [W/F], Skew vs. Time

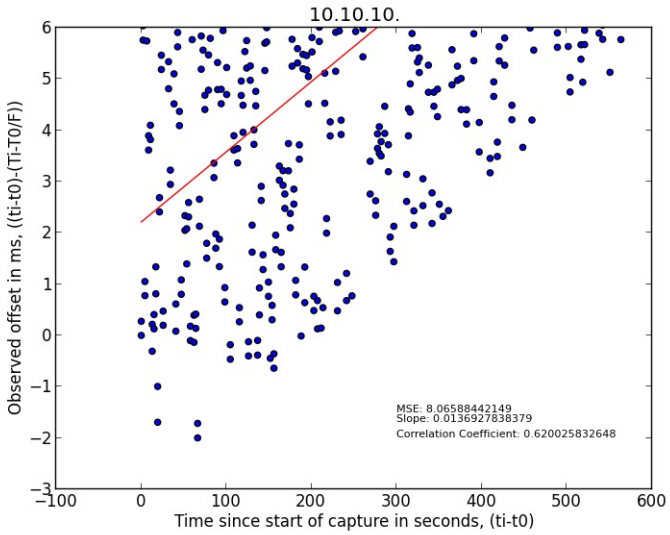

Figure 40. Configuration [W/W], Skew vs. Time

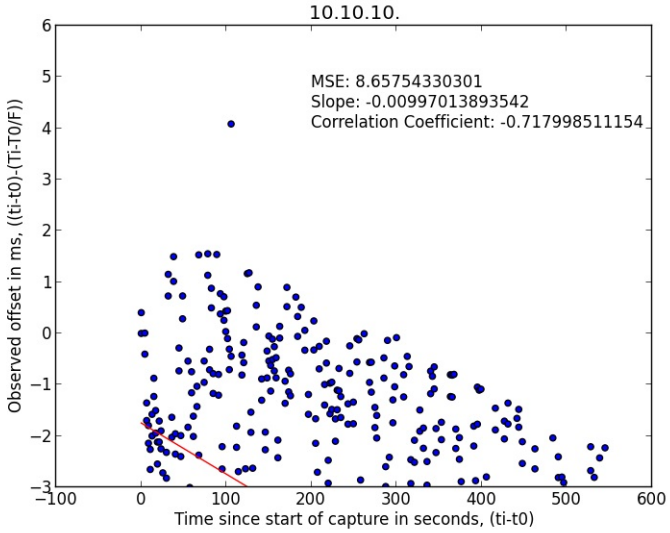

Figure 41. Configuration [W/X], Skew vs. Time

APPENDIX E. VIRTUALIZED PREEMPT_RT

In this appendix, we provide data associated with experiments discussed in Chapter V. Figures 42–44 show the results of experiments with virtualized PREEMPT_RT configurations ([RT/F], [RT/W], [RT/X]).

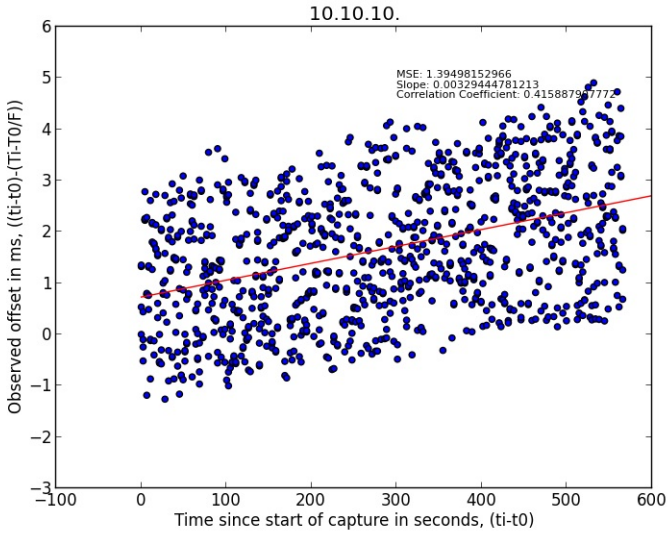

Figure 42. Configuration [RT/F], Skew vs. Time

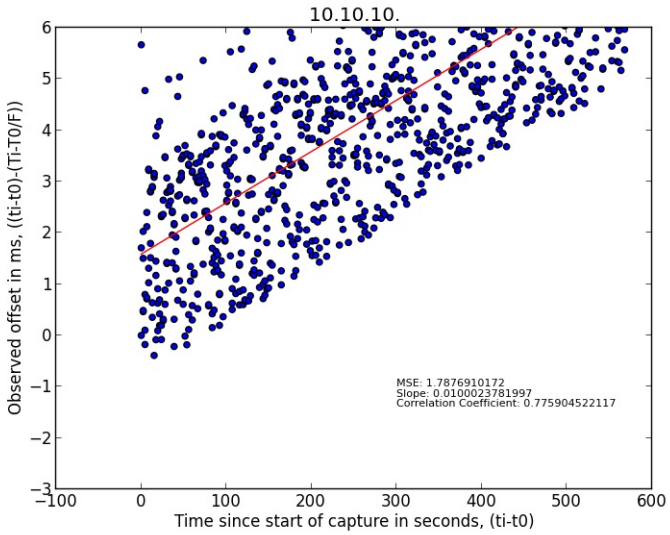

Figure 43. Configuration [RT/W], Skew vs. Time

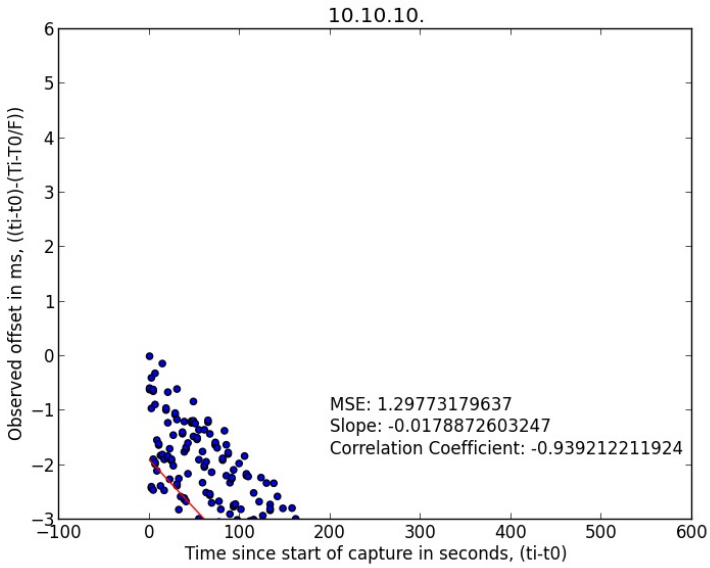

Figure 44. Configuration [RT/X], Skew vs. Time

APPENDIX F. PREEMPT_RT, FIFO SCHEDULING

In this appendix, we provide data associated with experiments discussed in Chapter V. Figures 45–48 show the results of experiments with PREEMPT_RT with FIFO scheduling and sshd priority 1 ([RT-1FF], [RT-1FF/F], [RT-1FF/W], [RT-1FF/X]).

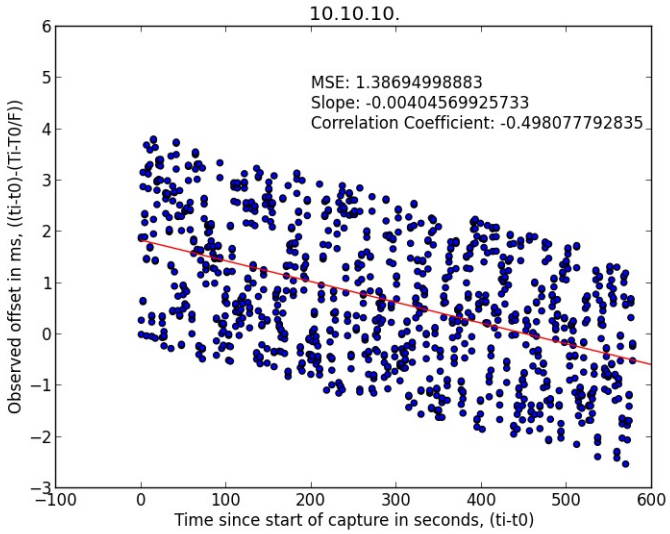

Figure 45. Configuration [RT-1FF], Skew vs. Time

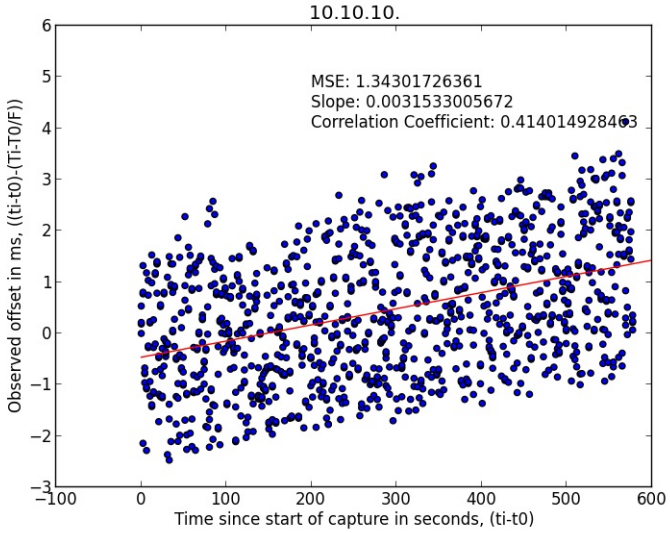

Figure 46. Configuration [RT-1FF/F], Skew vs. Time

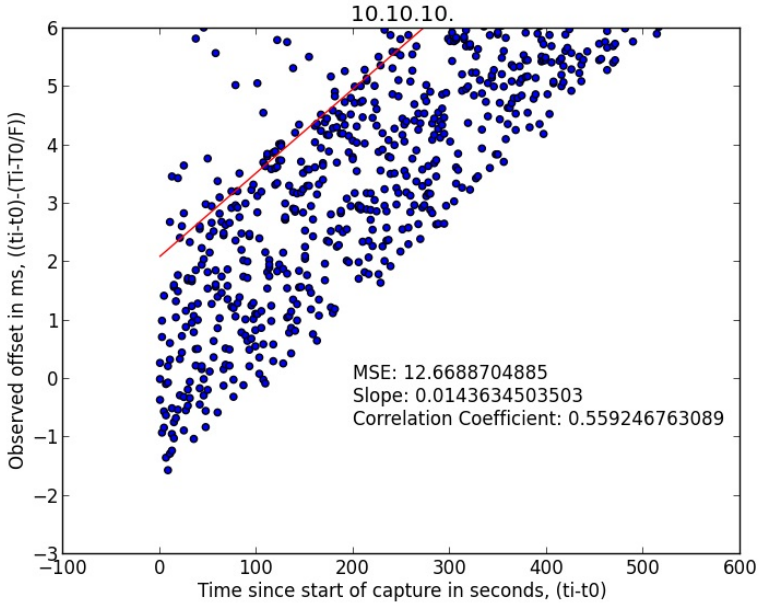

Figure 47. Configuration [RT-1FF/W], Skew vs. Time

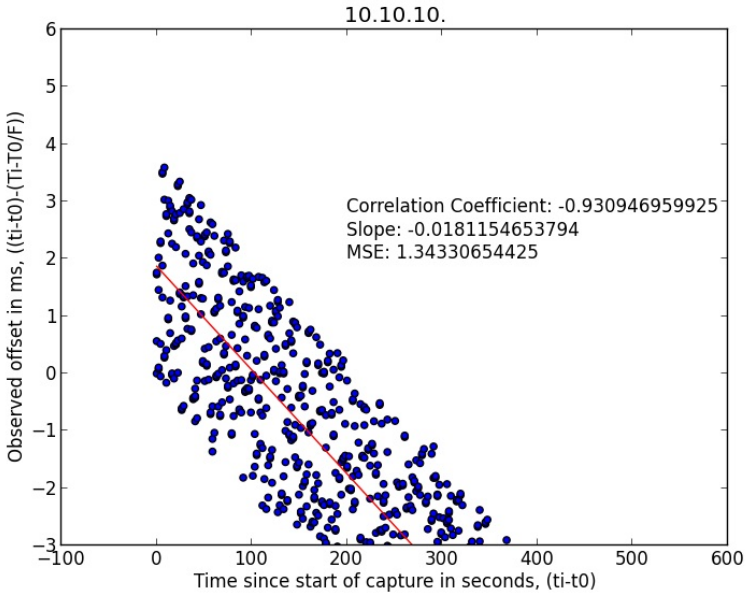

Figure 48. Configuration [RT-1FF/X], Skew vs. Time

APPENDIX G. PREEMPT_RT, ROUND ROBIN SCHEDULING

In this appendix, we provide data associated with experiments discussed in Chapter V. Figures 49–52 show the results of experiments with PREEMPT_RT with round robin scheduling and sshd priority 1 ([RT-1RR], [RT-1RR/F], [RT-1RR/W], [RT-1RR/X]).

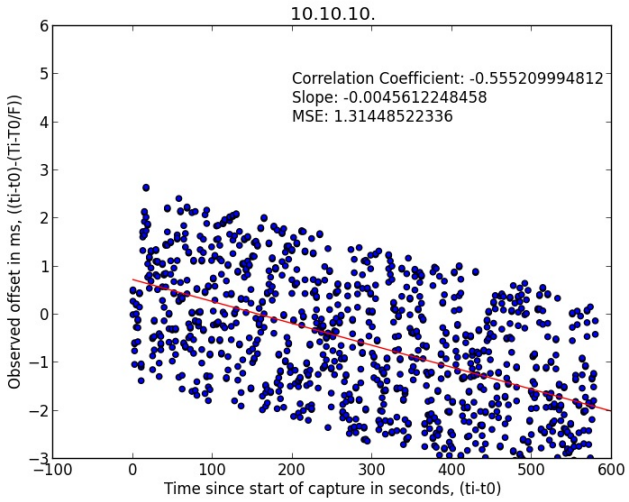

Figure 49. Configuration [RT-1RR], Skew vs. Time

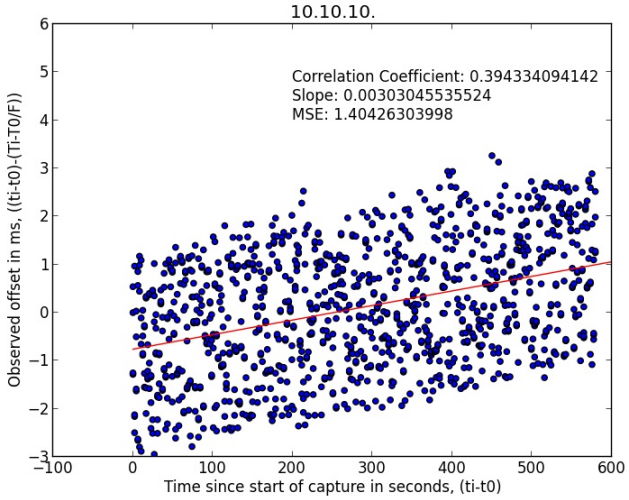

Figure 50. Configuration [RT-1RR/F], skew vs. time

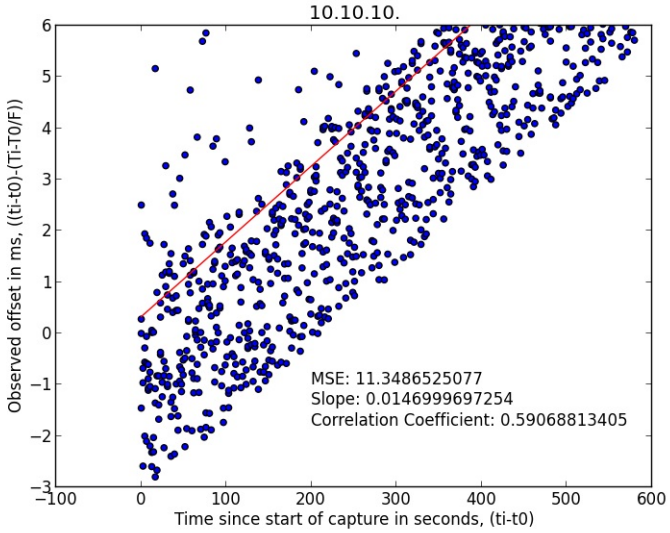

Figure 51. Configuration [RT-1RR/W], Skew vs. Time

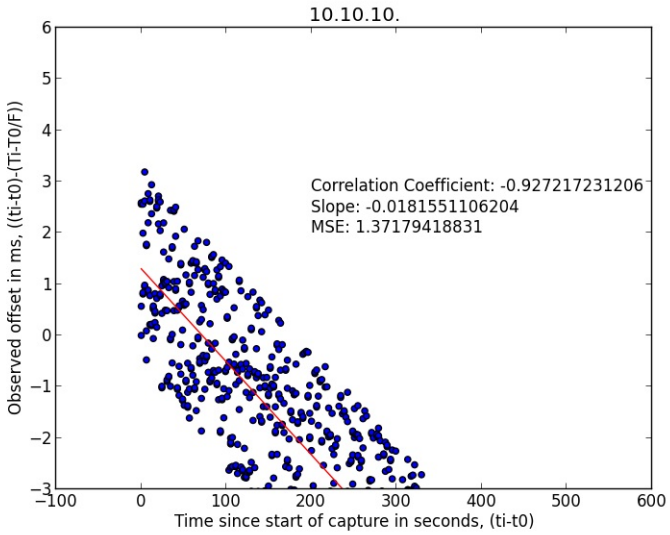

Figure 52. Configuration [RT-1RR/X], Skew vs. Time

SUPPLEMENTAL

Code to run the experiment and generated data from Chapter V is available in the CISR Archive, which may be accessed at the Computer Science Department of the Naval Postgraduate School.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

6.9 guide. (n.d.). Retrieved December 26, 2014, from http://read.pudn.com/downloads154/ebook/679838/vxworks_kernel_programmers_guide_6.6.pdf

6.9 platform. (n.d.). Retrieved May 4, 2014, from http://www.windriver.com/products/product-notes/PN_VE_6_9_Platform_0311.pdf

A time & space partitioned DO-178 level A certifiable RTOS. (n.d.). Retrieved January 15, 2015, from http://www.ddci.com/products_deos.php

Air Force Space Command. (2013). Resiliency and disaggregated space architectures. Retrieved from http://www.afspc.af.mil/shared/media/document/AFD-130821-034.pdf

Air overview. (2011). Retrieved from http://www.gmv.com/export/sites/gmv/DocumentosPDF/air/Presentation_GMV-AIR-1.1.pdf

air Robust. (n.d.). Retrieved February 4, 2015, from http://www.gmv.com/en/aeronautics/products/air/

Andrews, D., Bate, I., Nolte, T., Otero-Perez, C., & Petters, S. M. (2005, July). Impact of embedded systems evolution on RTOS use and design. *1st International Workshop Operating System Platforms for Embedded Real-Time Applications (OSPERT'05).*

Apecechea, G., Inci, M. S., Eisenbarth, T., & Sunar, B. (2014). Fine grain cross-VM attacks on Xen and VMware are possible. Retrieved from https://eprint.iacr.org/2014/248.pdf

Architecture of VMware ESXi, The. (n.d.). Retrieved March 12, 2014, from http://www.vmware.com/files/pdf/ESXi_architecture.pdf

ARINC 653. (2008). Retrieved from http://www.computersociety.it/wp-content/uploads/2008/08/ieee-cc-arinc653_final.pdf

ARINC Standards Store. (n.d.). ARINC standards 600 series. Retrieved from http://store.aviation-ia.com/cf/store/catalog.cfm?prod_group_id=1&category_group_id=3

Armand. F. (2009, November). Real time virtualization. Retrieved from http://www.emn.fr/z-info/jobjet/uploads/file/Slide%20Francois%20Armand.pdf

Ask Ubuntu. (n.d.). How can I install a real-time kernel? Retrieved March 18, 2014 from http://askubuntu.com/questions/72964/how-can-i-install-a-realtime-kernel

Balasubramaniam, M. (n.d.). Introduction to real-time operating systems. Retrieved December 3, 2014, from http://www.cis.upenn.edu/~lee/06cse480/lec-RTOS_RTlinux.pdf

Baldin, D., & Kerstan, T. (2009). Proteus, a hybrid virtualization platform for embedded systems. In *Analysis, Architectures and Modelling of Embedded Systems* (pp. 185–194).

Baliyase. (2014, March 1). MBR vs GPT. Retrieved from http://blbaliyase.blogspot.com/

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., & Warfield, A. (2003). Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, *37*(5), 164–177.

Barry, M., & Horvath, G. (2009, July). Prototype implementation of a goal-based software health management service. In *Space Mission Challenges for Information Technology, 2009. SMC-IT 2009. Third IEEE International Conference* (pp. 117–124).

Baumann, C., Bormer, T., Blasum, H., & Tverdyshev, S. (2011, March). Proving memory separation in a microkernel by code level verification. In *Object/ Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium* (pp. 25–32).

Bellard, F. (2005, April). QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (pp. 41–46).

Berens, K. (n.d.). Real-time Linux evaluation. presentation. Retrieved January 5, 2015 from http://webcache.googleusercontent.com/search?q=cache:TyVujS9lhhcJ: www.nasa.gov/centers/ivv/ppt/172526main_Kalynnda_Berens_Real-time_Linux_ Evaluation.ppt+&cd=1&hl=en&ct=clnk&gl=us

Beus-Dukic, L. (2001). COTS real-time operating systems in space. *Safety Systems: The Safety-Critical Systems Club Newsletter*, *10*(3), 11–14.

Binu, A., & Kumar, G. S. (2011). Virtualization techniques: A methodical review of XEN and KVM. In A. Abraham et al., (Eds.), *Advances in Computing and Communications, Communications in Computer and Information Sciences, 190* (pp. 399–410). Berlin, Heidelberg: Springer-Verlag.

Bloom, G., & Sherrill, J. (2014). Scheduling and thread management with RTEMS. *ACM SIGBED Review*, *11*(1), 20–25.

Board support. (n.d.). Retrieved March 12, 2014, from http://www.lynx.com/board-support-2/

Board support packages. (n.d.). Retrieved January 17, 2015, from https://bsp.windriver.com/index.php?bsp&on=list&type=platform&value=VxWorks:%206.8%20-%20Wind%20River%20Workbench%203.2

Bos, V., Mendham, P., Kauppinen, P. K., Holst, N., Crespo Lorente, A., Masmano, M., ... & Zamorano Flores, J. R. (2013). Time and space partitioning the EagleEye reference mission. Data Systems in Aerospace (DASIA 2013), May 14, 2013–May 16, 2013, Porto, Portugal.

Carlgren, H., & Ferej, R. (n.d.). Comparison of CPU scheduling in VxWorks and LynxOS. Retrieved January 2, 2015 from http://class.ece.iastate.edu/cpre584/ref/embedded_OS/vxworks_vs_lynxOS.pdf

Carrascosa, E., Coronel, J., Masmano, M., Balbastre, P., & Crespo, A. (2014). XtratuM hypervisor redesign for LEON4 multicore processor. *ACM SIGBED Review*, *11*(2), 27–31.

Cert platform. (n.d.). Retrieved May 4, 2014, from http://www.windriver.com/products/product-overviews/vxworks-cert-product-overview.pdf

Chen, X., Andersen, J., Mao, Z. M., Bailey, M., & Nazario, J. (2008, June). Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks with FTCS and DCC, 2008. DSN 2008. IEEE International Conference* (pp. 177–186).

Chiueh, S. N. T. C., & Brook, S. (2005). A survey on virtualization technologies. *RPE Report*, 1–42.

CIRA. (n.d.). Retrieved December 29, 2014, from http://www.windriver.com/customers/customer-success/documents/CaseStudy_CIRA.pdf

Clark, Libby. (2013, March 21). Intro to real-time Linux for embedded developers. Retrieved from https://www.linux.com/news/featured-blogs/200-libby-clark/710319-intro-to-real-time-linux-for-embedded-developers

Computer as a controller. (n.d.). Retrieved December 13, 2014, from http://people.sabanciuniv.edu/~onat/Files/RTLinux.htm

Contributing editor. (2001, July 23). Create hard read-time Tasks with precision under Linux. Retrieved from http://electronicdesign.com/embedded/create-hard-real-time-tasks-precision-under-linux

Crespo, A., Masmano, M., Coronel, J., Peiró, S., Balbastre, P., & Simó, J. (2014). Multicore partitioned systems based on hypervisor. Preprints of the 19th World Congress. The International Federation of Automatic Control, Cape Town, South Africa, August 24–29, 2014.

Cudmore, A. (2007, November). Flight software workshop 2007 (FSW-07). Retrieved from http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20080040872.pdf

Cudmore, A. (2013, September). Flight processor virtualization. Presented at NASA Goddard Space Flight Center, Greenbelt, MD.

da Silva, C. D. C. (2012). Integrated modular avionics for space applications: Input/output module. Retrieved from https://fenix.tecnico.ulisboa.pt/downloadFile/395144691307/resumo.pdf

Daugherty, J. (2014, August, 19). Porting FreeRTOS to Xen on ARM. Retrieved from http://www.slideshare.net/xen_com_mgr/free-rtos-xensummit

Definition of technology readiness levels. (n.d.). Retrieved October 21, 2014, from http://esto.nasa.gov/files/trl_definitions.pdf

Department of Defense. (1994). *Software development and documentation*. Retrieved from http://www.letu.edu/people/jaytevis/Software-Engineering/MIL-STD-498/498-STD.pdf

Department of Defense. (2010). *Information assurance (IA) policy for space systems used by the Department of Defense*. Retrieved from http://www.dtic.mil/whs/directives/corres/pdf/858101p.pdf

Department of Defense. (2011). *National security space strategy*. Retrieved from http://www.defense.gov/home/features/2011/0111_nsss/docs/NationalSecuritySpaceStrategyUnclassifiedSummary_Jan2011.pdf

Department of Defense ESI. (n.d.). IT virtualization technology and its impact on software contract terms. Retrieved from http://www.esi.mil/contentview.aspx?id=273

Diniz, N., & Rufino, J. (2005). ARINC 653 in space. In *Dasia 2005*, EUROSPACE, Edinburgh, Scotland.

DomU. (n.d.). Retrieved from http://wiki.xen.org/wiki/Dom0

DornerWorks. (2014, October 28). DornerWorks wins SBIR phase 2 award from DARPA. Retrieved from http://dornerworks.com/about/news

Douglas, H. (2010). *Thin hypervisor-based security architectures for embedded platforms* (master's thesis). Retrieved from http://soda.swedish-ict.se/3865/1/Thesis%2520 20100226.pdf

Edge, J. (2013, March 6). ELC: SpaceX lessons learned. Retrieved from http://lwn.net/Articles/540368/

Embedded hardware. (n.d.). Retrieved December 17, 2014, from https://xenomai.org/embedded-hardware/

Evans, P. (2007, February 27). How big is RTEMS? Retrieved from http://lists.rtems.org/pipermail/users/2007-February/015838.html

Fall 2013 colloquium series. (2013). Retrieved from http://istcolloq.gsfc.nasa.gov/fall2013/speaker/cudmore.html

Fayyad-Kazan, H., Perneel, L., & Timmerman, M. (2014). Linux PREEMPT-RT v2. 6.33 versus v3. 6.6: Better or worse for real-time applications?. *ACM SIGBED Review*, *11*(1), 26–31.

Fed Sat 1. (n.d.). Retrieved January 10, 2015, from http://space.skyrocket.de/doc_sdat/fedsat-1.htm

Federal Aviation Administration. (2007). Real-time operating systems and component integration considerations in integrated modular avionics systems report. Retrieved from https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/media/AR-07-39_ROSI-IMA.pdf

Feldt, R. Torkar, R., Ahmad, E., & Raza, B. (2010). Challenges with software verification and validation activities in the space industry. Retrieved from http://www.cse.chalmers.se/~feldt/publications/feldt_2010_icst_space_vav_challenges.pdf

Fink, R. (n.d.). Retrieved October 20, 2014, from http://userpages.umbc.edu/~rfink1/skew/

Five ways NASA is using Linux OS to run. (n.d.). Retrieved July 20, 2014, from http://www.100tb.com/blog/?p=485

FreeRTOS. (n.d.). Retrieved October 4, 2014 from http://www.freertos.org/

Fujitsu. (2010). Architecture of VMWare ESXi 4. Retrieved from http://www.rolva.com.tr/files/files/ROLVA_VMware_Architecture%20of%20VMware%20ESXi%204.pdf

Galileo pathfinder achieves five years in orbit. (2010, December 28). Retrieved from http://www.esa.int/Our_Activities/Navigation/Galileo_pathfinder_GIOVE-A_achieves_five_years_in_orbit

Garamone, J. (2014, January). Shelton discusses importance of space defense. Retrieved from http://www.defense.gov/utility/printitem.aspx?print=http://www.defense.gov/news/newsarticle.aspx?id=121443

General Dynamics. (n.d.). OKL4 microvisor. Retrieved from http://www.ok-labs.com/products/okl4-microvisor

General Dynamics. (2008, April). Microkernels vs. hypervisors. Retrieved from http://www.ok-labs.com/blog/entry/microkernels-vs-hypervisors/

Genesis. (n.d.). Retrieved December 28, 2014, from http://genesismission.jpl.nasa.gov/

Gilles, K., Groesbrink, S., Baldin, D., & Kerstan, T. (2013). Proteus hypervisor: Full virtualization and paravirtualization for multi-core embedded systems. In *Embedded Systems: Design, Analysis and Verification* (pp. 293–305).

Ginosar, R. (2012). Survey of processors for space. In *Data Systems in Aerospace (DASIA). Eurospace*.

Gomes, A. O. (2012, March). *Formal specification of the ARINC 653 architecture using circus*. (master's thesis). Retrieved from http://etheses.whiterose.ac.uk/2683/

GPOS vs RTOS for an embedded system. (2012). Retrieved from http://www.circuitstoday.com/gpos-versus-rtos-for-an-embedded-system

Green Hills software to power spaceflight crew escape system demonstrator. (2003, December 22). Retrieved from http://www.spaceref.com/news/viewpr.html?pid=13275

Green Hills software INTEGRITY-178B separation kernel security target. (2008, May 30). Retrieved from http://www.niap-ccevs.org/st/st_vid10119-st.pdf

Greve, D., & VanderLeest, S. H. (2013). Data flow analysis of a Xen-based separation kernel. In *7th Layered Assurance Workshop* (pp. 1–34).

Haas, J. (n.d.). RTLinux HOWTO, 4.2 creating RTLinux threads. Retrieved November 10, 2014 from http://linux.about.com/od/howtos/a/rtlinuxhowto4b.htm

Habib, I. (2008, February). Virtualization with KVM. Retrieved from http://www.linuxjournal.com/article/9764

Han, S., & Jin, H. (2011, October). Full virtualization based ARINC 653 partitioning. *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th* (pp. 7E1–1).

Heiser, G., & Leslie, B. (2010, August). The OKL4 microvisor: Convergence point of microkernels and hypervisors. *Proceedings of the First ACM Asia-Pacific Workshop on Workshop on Systems*. pp. 19–24.

Holmstrøm, D. E. (2012). *Software and software architecture for a student satellite*. Trondheim, Norway: Norwegian University of Science and Technology.

Home page. (n.d.). Retrieved January 1, 2015, from http://ecos.sourceware.org/

Howard, C. (2007, April 4). LynuxWorks provides safety-critical RTOS for European Space Agency's Galileo satellite navigation system. *Military and Aerospace Electronics*. Retrieved from http://www.militaryaerospace.com/articles/2007/04/lynuxworks-provides-safety-critical-rtos-for-european-space-agencys-galileo-satellite-navigation-system.html

Howard, C. (2011, March 1). RTOS for a software driven world. *Military and Aerospace Electronics*. Retrieved from http://www.militaryaerospace.com/articles/print/volume-22/issue-30/technology-focus/rtos-for-a-software-driven-world.html

Huffine, C. (2005, March 1). Linux on a small satellite. Retrieved from http://www.linuxjournal.com/article/7767

Hussein, S. (2009, May). Containing Linux instances with OpenVZ. Retrieved from http://www.opensourceforu.com/2009/05/containing-linux-instances-with-openvz/

Intelligent automation. (n.d.). Retrieved January 30, 2015, from http://www.i-a-i.com/?News/2013/darpa-awards-iai-a-new-contract-to-develop-a-light-and-secure-satellite-hypervisor

Integrity multivisor. (n.d.). Retrieved December 20, 2014 from http://www.ghs.com/products/rtos/integrity_virtualization.html

Integrity multivisor datasheets. (n.d.). Retrieved from http://www.ghs.com/download/datasheets/INTEGRITY_Multivisor.pdf

Introduction to linux for real-time control, introductory guidelines and references for control engineers and manager. (2002). Retrieved from http://www.aeolean.com/html/RealTimeLinux/RealTimeLinuxReport-2.0.0.pdf

Iqbal, A., Sadeque, N., & Mutia, R. I. (2009). An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems. *Report, Department of Electrical and Information Technology, Lund University, Sweden, 2110*.

Jacobson, V. (1992). TCP extensions for high performance. Retrieved from https://www.ietf.org/rfc/rfc1323.txt

Jaekel, S., Stelzer, M., & Herpel, H. J. (2014, March). Robust and modular on-board architecture for future robotic spacecraft. In *Aerospace Conference, 2014 IEEE* (pp. 1–11).

Jeong, S. (2013). In-depth overview of x86 server virtualization technology. Retrieved from http://www.cubrid.org/blog/dev-platform/x86-server-virtualization-technology/

Joe, H., Jeong, H., Yoon, Y., Kim, H., Han, S., & Jin, H. W. (2012, October). Full virtualizing micro hypervisor for spacecraft flight computer. In *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st* (pp. 6C5-1–6C5-9).

Jones, K. H., & Gross, J. (2014). Reducing size, weight, and power (SWaP) of perception systems in small autonomous aerial systems. Retrieved from http://arc.aiaa.org/doi/abs/10.2514/6.2014-2705

Jones, M. T. (2011, January 25). Platform emulation with bochs. Retrieved from http://www.ibm.com/developerworks/library/l-bochs/

Jones, T. (2008, April 15). Anatomy of real-time Linux architectures. Retrieved from http://www.ibm.com/developerworks/library/l-real-time-linux/

Jones, T. (2010, May 25). Virtualization. Retrieved from http://www.datamation.com/netsys/article.php/3884091/Virtualization.htm

Kaiser, R. (2007). *Scheduling virtual machines in real-time embedded systems*. Klein-Winternheim, Germany: SYSGO AG.

Kaiser, R. (2009). *Bringing together real-time and virtualization*. Klein-Winternheim, Germany: SYSGO AG.

Kang, S., & Kim, H. (2014, March). The study of the virtual machine for space real-time embedded systems. In *Aerospace Conference, 2014 IEEE* (pp. 1–7).

Katz, D. S., & Some, R. R. (2003). Advances robotic space exploration. Retrieved from http://web.ci.uchicago.edu/~dsk/papers/computer2003.pdf

Keesee, J. (2003, October). Satellite system software. Presented at MIT Department of Aero/Astro, Cambridge, MA.

Kenyon, S., Bridges, C. P., Liddle, D., Dyer, R., Parsons, J., Feltham, D., Taylor, R. Mellor, D., Schofield, A., & Linehan, R. (2011, October). STRaND-1: Use of a $500 Smartphone as the Central Avionics of a Nanosatellite. In *Proceedings of the 2nd International Astronautical Congress 2011, (IAC'11)*. p. 1–19.

Kirch, J. (2007, September). Virtual machine security guidelines, version 1.0. Retrieved from http://benchmarks.cisecurity.org/tools2/vm/CIS_VM_Benchmark_v1.0.pdf

Kohno, T., Broido, A., & Claffy, K. C. (2005). Remote physical device fingerprinting. *Dependable and Secure Computing, IEEE Transactions, 2*(2), 93–108.

Komolafe, O., & Sventek, J. (2006/07). Information for practical sessions. Retrieved from http://www.dcs.gla.ac.uk/~joe/Teaching/ESW1/Session4/esw1-practicalsinfo.pdf

Kovacs, E. (2014, October 2). Xen Hypervisor vulnerability exposed virtualized servers. Retrieved from http://www.securityweek.com/xen-hypervisor-vulnerability-exposed-virtualized-servers

Krüger, T., Schiele, A., & Hambuchen, K. (2013, May). Exoskeleton control of the robonaut through rapid and ros. In *Proceedings of the 12th Symposium on Advanced Space Technologies in Robotics and Automation, Noordwijk, Netherlands*.

Landley, R. (2009, September). Developing for non-x86 targets using QEMU. Retrieved from http://landley.net/aboriginal/presentation.html

Lee, M. (2012, January 24). Space qualified RTEMS. Retrieved from http://comments.gmane.org/gmane.os.rtems.user/18900

Lehrbaum, R. (2013, April 11). Android apps taps secure resources via ARIM TrustZone. Retrieved from http://linuxgizmos.com/android-app-taps-secure-resources-via-arm-trustzone/

Leiner, B., Schlager, M., Obermaisser, R., & Huber, B. (2007). A comparison of partitioning operating systems for integrated systems. In *Computer Safety, Reliability, and Security* (pp. 342–355). Springer, Berlin Heidelberg.

Leroux, P. (2005). RTOS vs. GPOS: What is best for embedded development. *Embedded Computing Design*.

LeVasseur, J., Uhlig, V., Chapman, M., Chubb, P., Leslie, B., & Heiser, G. (2005). *Pre-virtualization: Slashing the cost of virtualization*. Karlsruhe, Germany: Universität Karlsruhe, Fakultät für Informatik

LithOS. (n.d.). Retrieved October 6, 2014, from http://www.fentiss.com/en/products/lithos.html

Lynx software technologies patented technology speeds handling of hardware events. (n.d.). Retrieved May 12, 2014, from http://www.lynx.com/whitepaper/lynx-software-technologies-patented-technology-speeds-handling-of-hardware-events/

LynxOS-178. (n.d.). Retrieved December 28, 2014, from http://www.lynx.com/pdf/LynxOS-178DatasheetFinal.pdf

Mars reconnaissance orbiter. (n.d.). Retrieved October 11, 2014, from http://mars.jpl.nasa.gov/mro/

Masmano, M., Ripoll, I., Peiró, S., & Crespo, A. (2010, May). Xtratum for leon3: An open source hypervisor for high integrity systems. In *European Conference on Embedded Real Time Software and Systems. ERTS2* (Vol. 2010).

McKenney, Paul. (2005, August 10). A realtime preemption overview. Retrieved from http://lwn.net/Articles/146861/

Messenger. (n.d.). Retrieved December 28, 2014, from http://messenger.jhuapl.edu/the_mission/

Microkernel architecture. (n.d.). Retrieved November 4, 2014, from http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.sys_arch%2Ftopic%2Fintro_MICROKERNELARCH.html

MILS platform. (n.d.). Retrieved April 30, 2014, from http://www.windriver.com/products/platforms/vxworks-mils/MILS-3_PN.pdf

Mishchenko, D. (2010). *VMware ESXi: Planning, implementation, and security*. Boston, MA: Cengage Learning.

Moore, J. W. (1998, October). IEEE/EIA 12207 as the foundation for enterprise software processes. *Sixteenth Annual Pacific Northwest Software Quality Conference*.

Moore, R. (2005). Mutex tech note, Mutexes provide a level of safety for mutual exclusion, not possible with counting or binary semaphores. Retrieved from http://www.smxrtos.com/articles/techppr/mutex.htm

Müller, K., Paulitsch, M., Tverdyshev, S., & Blasum, H. (2012). MILS-related information flow control in the avionic domain: A view on security-enhancing software architectures. In *DSN Workshops* (pp. 1–6).

Munro, J. (2001). Virtual machines and VMWare part 1. Retrieved from http://www.extremetech.com/computing/72186-virtual-machines-vmware-part-i/6

Murphy, A. (n.d.). Virtualization defined-eight different ways. Retrieved from April 3, 2014. http://www.meritalk.com/uploads_legacy/whitepapers/Virtualization%20Defined%20-%20Eight%20Different%20Ways.pdf

Murray, D., Milos, G., & Hand, S. (2008). Improving Xen Security through Dissagregation. Retrieved from https://www.cl.cam.ac.uk/research/srg/netos/papers/2008-murray2008improving.pdf.

NanoMind computers. (n.d.). Retrieved December 15, 2014, from http://gomspace.com/index.php?p=products-a712c

NASA reference documents. (2013). Retrieved from http://snebulos.mit.edu/projects/reference/NASA-Generic/

NASA software guidelines. (n.d.). Matrix of NASA and IEEE software standards and guides. Retrieved September 10, 2014, from LaRC_Local_Version_of_SWG_Matrix.doc

NASA. (2004a). NASA software safety guidebook. Retrieved from http://www.hq.nasa.gov/office/codeq/doctree/871913.pdf

NASA. (2004b). Software safety standard NASA technical standard. Retrieved from http://www.system-safety.org/Documents/NASA-STD-8719.13B.pdf

NASA. (n.d.). *Technical support package*. (GSC-16856-1). Washington, DC: Author.

NASA's Mars rover curiosity powered by wind river. (n.d.). Retrieved February 15, 2015, from http://www.windriver.com/announces/curiosity/Wind-River_NASA_0812.pdf

NASA's Orion crew exploration vehicle built with INTEGRITY-178B. New generation of space exploration utilizes green hills software. (2008, September 8). Retrieved from http://www.ghs.com/news/20080908_integrity178b_nasa.html

Nelson, S. (2003, June). Certification processes for safety-critical and mission-critical aerospace software. Retrieved from http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20040014965.pdf

New-generation aircraft offer key to slimmer, smarter satellites. (n.d.). Retrieved October 15, 2014, from http://www.esa.int/Our_Activities/Technology/New-generation_aircraft_offer_key_to_slimmer_smarter_satellites

NOVA virtualization. (n.d.). Retrieved February 11, 2015, from http://hypervisor.org/poster-osdi.png

On-Line applications research corporation. (2013, February). RTEMS C user guide. Retrieved from https://docs.rtems.org/doc-current/share/rtems/pdf/c_user.pdf

One-stop-shop for all your CubeSat and nanosat systems, The. (n.d.). Retrieved December 16, 2014, from http://www.cubesatshop.com/

Opdenacker, M. (2004). Realtime in embedded Linux systems. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.172.5175&rep=rep1&type=pdf

Operating systems. (2008, July 23). Retrieved from http://www.esa.int/TEC/Software_engineering_and_standardisation/TECLUMKNUQE_2.html

Pad abort demonstrator to test crew escape technologies. (2003, September). Retrieved from http://www.nasa.gov/centers/marshall/pdf/104862main_padabort.pdf

Para virtualized quests for Xhyp. (n.d.). Retrieved January 3, 2015, from http://x-hyp.org/products/guests/

Parkinson, P. (2011). Safety, security and multicore. In C. Dale & T. Anderso (Eds.), *Advances in systems safety* (pp. 215–232). London: Springer.

Parkinson, P. (n.d.). Case study: European geostationary navigation overlay system. Retrieved May 30, 2014, from http://www.windriver.com/customers/customer-success/documents/CS_EGNOS_v2_0610.pdf

Parkinson, P., & Kinnan, L. (n.d.). Safety-critical software development for integrated modular avionics [white paper]. Retrieved May 30, 2014, from http://www.element14.com/community/servlet/JiveServlet/previewBody/19565-102-1-59593/Safety-Critical%20Software%20Development%20for.pdf

Pettersson, M., & Svensson, M. (2006, November 17). Memory management in VxWorks compared to RTLinux. Retrieved from http://home.mit.bme.hu/~meszaros/edu/embedded_systems/literature_files/3787.pdf

Prieto, S. S., Tejedor, I. G., Meziat, D., & Sánchez, A. V. (2004). Is Linux ready for space applications? Madrid, Spain: Computer Engineering Department (University of Alcala).

Prisaznuk, P. J. (2008, October). ARINC 653 role in integrated modular avionics (IMA). *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th* (pp. 1–E).

Products PikeOS hypervisor. (n.d.). Retrieved January 30, 2015, from http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/

Profiles. (n.d.). Retrieved July 10, 2014, from http://www.windriver.com/products/vxworks/

Publishable Summary. (2012, October 1). EURO-MILS: Secure European virtualisation for trustworthy applications in critical domains. Retrieved from http://www.euromils.eu/downloads/Deliverables/Y2/02_EURO-MILS-318353-D42.3-2nd-periodic-report-publishable-summary.pdf

Qemu. (n.d.). KVM. Retrieved April 13, 2014, from http://wiki.qemu.org/KVM

QNX. (n.d.). Retrieved November 4, 2014, from http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.sys_arch%2Ftopic%2Fintro_MICROKERNELARCH.html

Raines, D. (2012). From satellite to human-rated spaceflight: Adapting executive software to meet the requirements of manned missions. In *AIAA SPACE 2012 Conference & Exposition* (pp. 1–4).

Rajulu, B., Dasiga, S., & Iyer, N. R. (2014, March). Open source RTOS implementation for on-board computer (OBC) in STUDSAT-2. In *Aerospace Conference, 2014 IEEE* (pp. 1–13).

Ramsey, J. (2007, February). Integrated modular avionics: Less is more. Retrieved from http://www.aviationtoday.com/av/commercial/Integrated-Modular-Avionics-Less-is-More_8420.html#.VOKaFvnF-Sp

Real Time Engineers, Ltd. (2014). The FreeRTOS reference manual for FreeRTOS version 8.2.0. Bristol, United Kingdom: Texas Instruments.

Real-Time Linux wiki. (n.d.). Retrieved March 21, 2015, from https://rt.wiki.kernel.org/index.php/Main_Page

Report of the National Commission for the Review of the National Reconnaissance Office. (2000). [Executive Summary]. Retrieved from http://fas.org/irp/nro/commission/nro.pdf

Ricklefs, R. (n.d.). Real-time Linux at MLRS. Retrieved November 18, 2014, from http://cddis.gsfc.nasa.gov/lw18/docs/posters/13-Po25-Ricklefs.pdf

Rosa, J. L. S. R. C. (2011). *Development and update of aerospace applications in partitioned architectures*. PhD. Dissertation, Universidade de Lisboa, Lisbon, Portugal.

Rosenblum, M., & Garfinkel, T. (2005). Virtual machine monitors: Current technology and future trends. *Computer*, *38*(5), 39–47.

Rostedt, S., & Hart, D. V. (2007, June). Internals of the RT patch. In *Proceedings of the Linux Symposium*.

RT-Xen project. (2013, July 16). RT-Xen project receives grant from Office of Naval Research. Retrieved from http://cse.wustl.edu/Research/Pages/news-story.aspx?news=476

RTEMS 4.10.99.0 on-line library. (2014, November 10). http://docs.rtems.org/doc-current/share/rtems/html/posix_users/Memory-Management-Manager-mprotect-_002d-Change-Memory-Protection.html

RTEMS Architecture. (n.d.). Retrieved March 1, 2015, from http://rtemscentre.edisoft.pt/index.php?module=ContentExpress&file=index&func=display&ceid=21&meid=37

RTEMS community. (n.d.). Retrieved November 1, 2014, from http://www.rtems.com/community

RTEMS. (n.d.). Retrieved October 11, 2014, from http://www.fentiss.com/en/products/rtems.html

RTLinux. (n.d.). Retrieved December 16, 2014, from http://www.cs.fsu.edu/~baker/realtime/restricted/notes/rtlinux.html

RTOS 101. (n.d.). Retrieved January 29, 2015, from http://www.nasa.gov/sites/default/files/482489main_4100_-_RTOS_101.pdf

Rufino, J., & Craveiro, J. (2008, July). Robust partitioning and composability in ARINC 653 conformant real-time operating systems. In *1st INTERAC Research Network Plenary Workshop, Braga, Portugal.*

Rufino, J., Craveiro, J., Schoofs, T., Tatibana, C., & Windsor, J. (2009, May). AIR Technology: A step towards ARINC 653 in space. In *Proceedings DASIA.*

Rufino J., & Filipe, S. (2007, December). AIR project final report. Technical report TR 07–35. Retrieved from http://air.di.fc.ul.pt/air/downloads/07-35.pdf

Rushby, J. (2000). *Partitioning in avionics architectures: Requirements, mechanisms, and assurance*. Menlo Park, CA: SRI International Computer Science Lab.

Rushby, J. (2011). New challenges in certification for aircraft software. Retrieved from http://www.csl.sri.com/users/rushby/papers/emsoft11.pdf

Rutkowska, J., & Tereshkin, A. (2008). Bluepilling the xen hypervisor. In *Black Hat USA, 2008*.

Safety critical products: Integrity®-178B RTOS. (n.d.). Retrieved January 11, 2015, from http://www.ghs.com/products/safety_critical/integrity-do-178b.html

Safety-critical RTOS support extended for Microsemi's smartfusion2 SoC FPGAs. (2013, July 29). Retrieved from http://www.highintegritysystems.com/wittenstein-high-integrity-systems-extends-safety-critical-rtos-support-microsemis-smartfusion2-soc-fpgas/

Sahoo, J., Mohapatra, S., & Lath, R. (2010, April). Virtualization: A survey on concepts, taxonomy and associated security issues. In *Computer and Network Technology (ICCNT), 2010 Second International Conference* (pp. 222–226).

Samolej, S. (2011). ARINC specification 653 based real-time software engineering. *e-Informatica*, *5*(1), 39–49.

Santangelo, A. D. (2013). An open source space hypervisor for small satellites. In *AIAA SPACE 2013 Conference and Exposition* (pp. 1–10).

Scharpf, K. (2013, December 11). The last cathedral-democratizing flight software. Retrieved from http://flightsoftware.jhuapl.edu/files/2013/talks/FSW-13-TALKS/KS_FSW2013.pdf

Schoofs, T. (2011, December 21). AIR-overview. Retrieved from http://www.gmv.com/export/sites/gmv/DocumentosPDF/air/Presentation_GMV-AIR-1.1.pdf

Schoofs, T., Santos, S., Tatibana, C., & Anjos, J. (2009, October). An integrated modular avionics development environment. *Digital Avionics Systems Conference, 2009. DASC'09. IEEE/AIAA 28th* (pp. 1–A).

Seagrave, G. (2008). SpaceCube: A reconfigurable processing platform for space. Retrieved from https://nepp.nasa.gov/mapld_2008/presentations/i/08%20-%20Godfrey_John_mapld08_pres_1.pdf

Secure separation architecture white paper PDF download form. (n.d.). Retrieved June 10, 2014, from http://www.ghs.com/articles/index.php?wp=secure_separation

SBIR safehype. (n.d.). Retrieved January 30, 2015, from https://www.sbir.gov/sbirsearch/detail/666406

Silva, H., Sousa, J., Freitas, D., Faustino, S., Constantino, A., & Coutinho, M. (2009). RTEMS improvement-space qualification of RTEMS executive. In *1st Simpósio de Informática-INFORUM, University of Lisbon*.

Smith, J. E., & Nair, R. (2005). The architecture of virtual machines. *Computer*, *38*(5), 32–38.

SpaceX. (n.d.). Retrieved December 26, 2014, from http://www.spacex.com/sites/spacex/files/pdf/DragonLabFactSheet.pdf

Steinberg, U., & Kauer, B. (2010, April). NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems* (pp. 209–222).

Studer, N. (2014). Xen and the art of certification. Xen developer summit 2014. Retrieved from http://www.xenproject.org/presentations-and-videos/video/xpds14v-certification.html

SYSGO's safe and secure virtualization PikeOS now available for LEON and RTEMS. (2010, August 26). Retrieved from http://www.sysgo.com/news-events/press/press/details/article/sysgos-safe-and-secure-virtualization-pikeos-now-available-for-leon-and-rtems/

Tavares, A., Carvalho, A., Rodrigues, P., Garcia, P., Gomes, T., Cabral, J., & Ekpanyapong, M. (2012, March). A customizable and ARINC 653 quasi-compliant hypervisor. *Industrial Technology (ICIT), 2012 IEEE International Conference* (pp. 140–147).

Terrasa, A., Garcia-Fornes, A., & Espinosa, A. (2002, September 10). RTL POSIX trace 1.0 (a POSIX trace system in RT-Linux. Retrieved from http://www.gti-ia.upv.es/sma/tools/rtl-ptm/archivos/documentation/rtl-posixtrace.pdf

Teston, F., Vuilleumier, P., Hardy, D., & Bernaerts, D. (2004, October). The PROBA-1 microsatellite. In *Proc. of SPIE Vol. 5546*, pp. 132–140).

Threadx. (n.d.). Retrieved January 15, 2015, from http://rtos.com/products/threadx/

Tverdyshev, S. (2011). Extending the GWV security policy and its modular application to a separation kernel. In *NASA Formal Methods* (pp. 391–405). Springer Verlag Berlin, Heidelberg.

Understanding full virtualization, paravirtualization, and hardware assist. (2007). Retrieved April 13, 2014, from http://www.vmware.com/files/pdf/VMware_ paravirtualization.pdf

USENIX. (2001). 2001 proceedings of the 2001 USENIX annual technical conference. Retrieved from http://www.vmware.com/pdf/usenix_io_devices.pdf

VanderLeest, S. H. (2010, October). ARINC 653 hypervisor. *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th* (pp. 5–E).

VanderLeest, S. H., Greve, D., & Skentzos, P. (2013, October). A safe & secure arinc 653 hypervisor. In *Digital Avionics Systems Conference (DASC), 2013 IEEE/AIAA 32nd* (pp. 7B4–1).

Virtualization Station. (2008). KVM hypervisor integrated in Linux Kernel 2.6.20. Retrieved from http://virtualization-station.blogspot.com/2008/12/kvm-hypervisor-integrated-in-linux.html

Virtualization: Gaming on Xen. (2013). Retrieved from http://linuxforcynics.com/ hardware/virtualization-gaming-on-xen

VMWare knowledge base. (n.d.). Retrieved March 4, 2014, from http://kb.vmware.com/ selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId= 1019471

Volpe, R., Nesnas, I. A.D., Estlin, T., Mutz, D., Petras, R., & Das, H. (2000). CLARAty: Coupled layer architecture for robotic autonomy. Retrieved from https://www-robotics.jpl.nasa.gov/publications/Issa_Nesnas/CLARAty.pdf

vSphere ESXi. (n.d.). Retrieved January 2, 2014, from http://www.vmware.com/products/vsphere/features-esxi-hypervisor

VxWorks architecture supplemental 6.2. (2011, October, 11). Retrieved from http://www.uio.no/studier/emner/matnat/fys/FYS4220/h11/undervisningsmateriale/laboppgaver-rt/vxworks_architecture_supplement_6.2.pdf

VxWorks on the Mars exploration rovers. (n.d.). Retrieved March 20, 2014, from http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/37779/1/05-0825.pdf

Welcome. (n.d.). Retrieved January 15, 2015, from http://www.pumpkininc.com/

Wind river hypervisor. (n.d.). Retrieved February 11, 2015, from http://www.windriver.com/products/product-notes/wind-river-hypervisor-product-note.pdf

Wind river linux. (n.d.). Retrieved January 15, 2015, from http://www.windriver.com/products/linux/

Wind river linux 4. (n.d.). Retrieved September 9, 2014, from http://windriver.com/products/product-notes/PN_Linux_4_1_0811.pdf

Wind river linux 6. (n.d.). Retrieved September 9, 2014, from http://www.windriver.com/products/product-notes/Wiind_River_Linux_6_Product_Note.pdf

Windsor, J., Deredempt, M. H., & De-Ferluc, R. (2011, October). Integrated modular avionics for spacecraft—User requirements, architecture and role definition. In *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th* (pp. 8A6-1–8A6-16).

Windsor, J., & Hjortnaes, K. (2009, July). Time and space partitioning in spacecraft avionics. In *Space Mission Challenges for Information Technology, 2009. Third IEEE International Conference* (pp. 13–20).

Wojtczuk, R. (2008). Subverting the Xen hypervisor. In *Black Hat USA, 2008*.

Wright, C. W., & Walsh, E. J. (1999, February 1). Hunting hurricanes. Retrieved from http://www.linuxjournal.com/article/3212?page=0,0

X-Hyp paravirtualized. (n.d.). Retrieved November 25, 2015, from http://x-hyp.org/products/guests/

Xen hypercall. (n.d.). Retrieved January 10, 2015, from http://wiki.xen.org/wiki/Hypercall

Xen project: RT-xen. (n.d.). RT-Xen: Real-time virtualization in Xen. Retrieved December 7, 2015, from https://blog.xenproject.org/2013/11/27/rt-xen-real-time-virtualization-in-xen/

Xi, S., Wilson, J., Lu, C., & Gill, C. (2011, October). Rt-xen: Towards real-time hypervisor scheduling in xen. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference* (pp. 39–48).

Xtratum hypervisor. (2011). XtratuM hypervisor for Leon3 volume 2: User manual. Retrieved from http://www.xtratum.org/files/xm-3-usermanual-022c.pdf

Xtratum product. (n.d.). Retrieved November 11, 2014, from http://www.fentiss.com/en/products/xtratum.html

xWorks space. (n.d.). Retrieved January 20, 2015, from http://www.windriver.com/in space/

Yodaiken, V. (1999, March). The RTLinux manifesto. Retrieved from http://www.yodaiken.com/papers/rtlmanifesto.pdf

Yodaiken, V. (2001). Getting started with RTLinux. Retrieved from http://cs.uccs.edu/~cchow/pub/rtl/doc/html/GettingStarted/

Zhang, J., Chen, K., Zuo, B., Ma, R., Dong, Y., & Guan, H. (2010, November). Performance analysis towards a kvm-based embedded real-time virtualization architecture. In *Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference* (pp. 421–426).

Zhou, R. (2009, December). Partitioned system with XtratuM on powerPC. Retrieved from https://riunet.upv.es/bitstream/handle/10251/12738/Tesina_Rui_Zhou.pdf?sequence=1

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California