Thin Hypervisor-Based Security Architectures for Embedded Platforms

Heradon Douglas
The Royal Institute of Technology, Stockholm, Sweden

Advisor: Christian Gehrmann
Swedish Institute of Computer Science, Stockholm, Sweden

February 26, 2010

To my wife, Guiniwere, who is everything to me.
Till min hustru, Guiniwere, som är allt för mig.

I would also like to thank my advisor, Christian Gehrmann, for his support, guidance and collaboration; Louise Yngström, Alan Davidson, Stewart Kowalski and my other teachers and colleagues at DSV for their generosity and tutelage; and my friends and family for their love.

ABSTRACT

Virtualization has grown increasingly popular, thanks to its benefits of isolation, management, and utilization, supported by hardware advances. It is also receiving attention for its potential to support security, through hypervisor-based services and advanced protections supplied to guests. Today, virtualization is even making inroads in the embedded space, and embedded systems, with their security needs, have already started to benefit from virtualization's security potential. In this thesis, we investigate the possibilities for thin hypervisor-based security on embedded platforms. In addition to significant background study, we present implementation of a low-footprint, thin hypervisor capable of provding security protections to a single FreeRTOS guest kernel on ARM. Backed by performance test results, our hypervisor provides security to a formerly unsecured kernel with minimal performance overhead, and represents a first step in a greater research effort into the security advantages and possibilities of embedded thin hypervisors. Our results show that thin hypervisors are both possible and beneficial even on limited embedded systems, and sets the stage for more advanced investigations, implementations, and security applications in the future.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ABI	Application Binary Interface
API	Application Programming Interface
ASID	Address Space Identifier
CPSR	current program status register
CPU	central processing unit
DMA	direct memory access
DMAC	DMA controller
DMR	dual-modular redundancy
DRM	Digital Rights Management
EPT	Extended Page Table
FCSE PID	Fast Context-Switch Extension Process ID
I/O	Input/Output
IOMMU	I/O Memory Management Unit
IPC	interprocess communication
ISA	Instruction Set Architecture
MAC	Mandatory Access Control
MMM	Mixed-Mode Multicore reliability
MMU	Memory Management Unit
MPU	Memory Protection Unit
MVA	modified virtual address
NUMA	Non-Uniform Memory Architecture
OMTP	Open Mobile Terminal Platform
OSTI	Open and Secure Terminal Initiative
OVP	Open Virtual Platforms
SPMD	single-program, multiple data

SPSR	saved program status register
TCB	trusted computing base
TCG	Trusted Computing Group
TLB	translation lookaside buffer
TPM	Trusted Platform Module
TPR	Task Priority Register
VBAR	Vector Base Address Register
VM	virtual machine
VMCB	Virtual Machine Control Block
VMCS	Virtual Machine Control Structure
VMI	VM introspection
VMM	virtual machine monitor
VPID	Virtual Process Identifier

1. INTRODUCTION

1.1 Security and Virtualization on Embedded Systems

Virtualization, the use of hypervisors or virtual machine monitors to support one or more *virtual machines* on a single real machine, is quickly becoming more and more popular today due to its benefits of increased hardware utilization and system management flexibility, and because of increasing hardware and software support for virtualization in commodity platforms. With the hypervisor providing an abstraction layer separating virtual machines from the real hardware, and isolating virtual machines from each other, many useful architectural possibilities arise.

In addition to hardware utilization and system management, virtualization has been shown to be a strong enabler for security – both as a result of the isolation enforced by the hypervisor between virtual machines, and due to the hypervisor's high-privilege suitability as a strong base for security services provided for the virtual machines.

Additionally, multicore is quickly gaining prevalence, with all manner of systems shifting to multicore hardware. Virtualization presents both opportunities and challenges with multicore hardware – while the layer of abstraction provided by the hypervisor affords a unique opportunity to manage multicore complexity and heterogeneity beneath the virtual machines, supporting multicore in the hypervisor in a robust and secure way is not a trivial task.

These issues become especially interesting and relevant in embedded scenarios. Both virtualization and multicore are enjoying quickly increasing prominence in the embedded world. Embedded system software is growing in complexity, and embedded systems are being used in more and more mission-crtical, security-focused situations. Virtualization can answer many security challenges in the embedded world (via hypervisor supported isolation and security services), as well as practical challenges such as abstracting varied or quickly changing hardware and managing power usage, in addition to inspiring new applications such as flexible system composition where virtual machines can be combined in novel ways on a single platform. Since virtualization enables security services to be implemented outside a virtual machine, the implementation can be decoupled from the considerable heterogeneity in embedded systems software (including proprietary system stacks). And, embedded virtualization also presents the opportunity to abstract hardware heterogeneity and multicore complexity. Virtualization thus offers profound opportunities and challenges for embedded systems.

1.2 Thesis Organization

This thesis is organized as follows. The *Introduction* chapter defines the general problem, motivations, goals, and methods of the research. Due to the thorough and detailed background required to set the stage for later research phases, background material is withheld from the introduction and instead included in Chapters 2 and 3. Chapter 2, *Virtualization Technologies*, gives an extensive overview of virtualization systems in use today (including software and hardware aspects), as well as an examination of virtualization as an enabler for security architectures and services and an overview of numerous security services presented in current research. Chapter 3, *Multicore and Embedded Systems*, gives an overview of embedded systems, multicore hardware, and their relation to virtualization and virtualization-based security.

Chapter 4, *Thin Hypervisors on ARM Architecture*, presents the ARM architecture as a platform for thin hypervisors, describing the basics of ARMv5 architecture and suggesting approaches and challenges for implementing a thin hypervisor upon it. This chapter also includes suggestions on how to implement selected security services from section 2.7, and furthermore incorporates commentary on how ARM hardware support for virtualization ("TrustZone") could help or hinder thin hypervisors. Chapter 5 describes our implementation of a thin hypervisor, including an analysis of its security, as well as design recommendations for future implementation. Chapter 6 describes test procedures conducted on our implementation, and results. Chapter 7 presents conclusions, including recommendations for future work.

1.3 Problem Definition

1.3.1 Impetus

Motivated by concerns briefly outlined in section 1.1, we are interested in exploring the possibilities of thin hypervisor-based architectures as a way of providing security services to and possibly managing multicore hardware for an embedded system. Such a thin hypervisor is intended to be an extremely small footprint, dedicated functionality hypervisor, inexpensive to run and typically only supporting one virtual machine for simplicity, but still capable of providing important security services. Due to their small size and light overhead, such thin hypervisors should be extremely appropriate for constrained embedded platforms. They can provide an avenue for implementing relevant security functionality (including memory protection, isolation of security applications, and system monitoring services), and may provide an avenue for managing and leveraging multicore hardware.

1.3.2 Originality

While there is substantial work being done in the area of virtualization, and even a good amount in the area of embedded virtualization, the body of work thins out when it comes to *multicore* virtualization and in the area of ultra-thin hypervisors. Furthermore, within embedded virtualization, there has been little work done on support for security services beyond virtual machine isolation.

And, virtually no work has been done in the area of ultra-thin hypervisors for embedded systems.

1.3.3 Feasibility

By focusing on research into thin hypervisors with minimal complexity, we ensure that implementation is still feasible for the time and resource constraints of a master's thesis project. Via freely available embedded hardware emulators, it is possible to implement and test implemetations efficiently. Furthermore, even if only limited implementation is possible, it is still quite feasible to assess the current state of the art, and thereupon suggest and motivate designs and recommendations for future research.

1.4 Purpose and Goals

The principal purpose of this research is to facilitate greater security for embedded systems through use of thin hypervisor-based security protections. A secondary purpose is to set the stage for facilitating secure, robust support for multicore and heterogeneous hardware in embedded virtualization, in service of system robustness and performance.

The individual goals we intend to accomplish in this research to support these overall purposes include:

1. A thorough investigation into current virtualization technologies, security architectures, and multicore and embedded systems, and how virtualization can apply to multicore and embedded scenarios.

2. Implementation of a basic thin hypervisor running on a simulated embedded hardware platform, capable of providing security to a guest OS. The simulated platform will be single core and as simple as possible to facilitate development.

3. Offering of considerations based on the research for how implementation could be extended to support additional security services and heterogeneous/multicore embedded hardware.

4. Conducting of performance tests on the simulated platform.

1.5 Method

Logical approach

The logical approach in our research will comprise a blend of induction and deduction. Background study of existing research will lead to theoretical approaches and motivation for new solutions. Both background study and subsequently formulated design approaches will guide implementation efforts. Implementation and background research experience will feed back into suggestions for improved designs, new solutions and future work. Empirical test results will assess the effectiveness of our implementation.

Data collection approach

Data collection will begin with extensive study and analysis of exsiting work and technology. It will continue with empirical testing of our implementated solutions. Note that specific test procedures will be described in Chapter 6.

2. VIRTUALIZATION TECHNOLOGIES

2.1 What is virtualization?

An excellent overview of virtual machines is found here [95], and in a book by the same authors[96]. Virtualization is a computer system abstraction, in which a layer of virtualization logic manages and provides "virtualized" resources to a client layer running above it. The client accesses resources using standard interfaces, but the interfaces do not communicate with the resources directly; instead, the virtualization layer manages the real resources and possibly multiplexes them among more than one client. See figure 2.1.

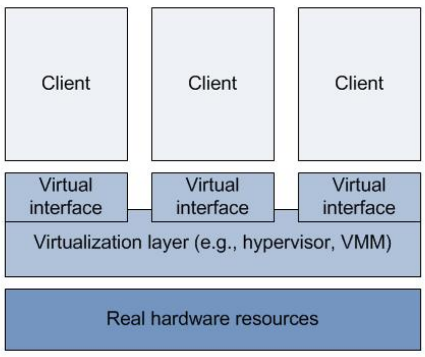

Fig. 2.1: Virtualization in a nutshell

The virtualization layer resides at a higher privilege level than the clients, and can interpose between the clients and the hardware. This means that it can intercept important instructions and events and handle them specially before they are executed or handled by the hardware. For example, if a client attempts to execute an instruction on a virtual device, the virtualization layer may have to intercept that instruction and implement it in a different way on the real resources in its control. This interposition behavior is illustrated in figure 2.2.

Each client is presented with the illusion of having sole access to its resources, thanks to the management performed by the virtualization layer. The virtualization layer is responsible for maintaining this illusion and ensuring correctness in the resource multiplexing. Virtualization therefore promotes efficient resource utilization via sharing among clients, and furthermore maintains isolation between clients, who need not know of each other's existence. Virtualization also serves to abstract the real resources to the client, which decouples the client from the real resources, facilitating greater architectural flexibility and mobility in system design.

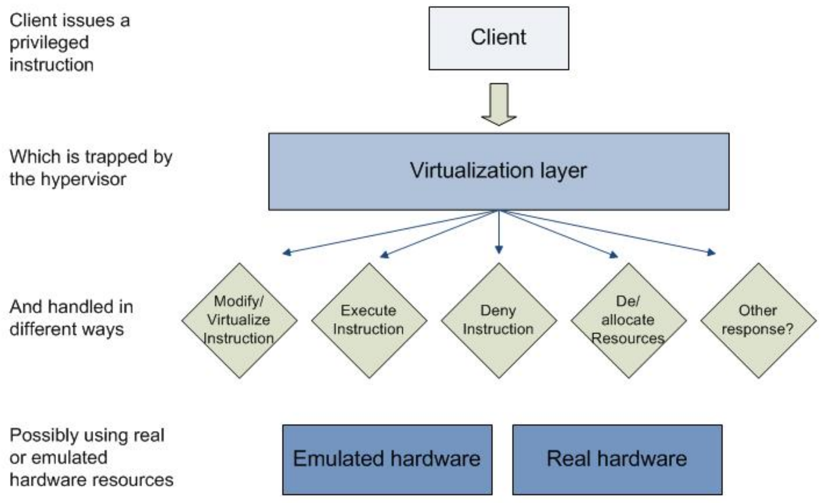
Client issues a privileged instruction
Client
Which is trapped by the hypervisor
Virtualization layer
And handled in different ways
Modify/ Virtualize Instruction
Execute Instruction
Deny Instruction
De/ allocate Resources
Other response?
Possibly using real or emulated hardware resources
Emulated hardware
Real hardware

Fig. 2.2: Interposition

For these reasons, virtualization technology has become more prominent, and its viable uses have expanded. Today virtualization is used in enterprise systems, service providers, home desktops, mobile devices, and production systems, among other venues.

Oftentimes, the client in a virtualization system is known as a *guest*.

2.2 Virtualization Basics

2.2.1 Interfaces

The article and book cited above ([95, 96]) discuss, in part, how virtualization can be understood in terms of the interfaces present at different levels of a typical computer system. Interfaces offer different levels of abstraction which clients use to access resources. Virtualization technology exposes an expected interface, but behind the scenes is virtualizing resources accessed by the interface – for example, in the case of a disk input/output interface, the “disk” that the interface provides access to may actually be a file on a real disk when implemented by a virtualization layer. A discussion of important interfaces in a typical computer system follow, as seen in [95].

ISA

The *Instruction Set Architecture* (ISA) is the lowest level instruction interface that communicates directly with hardware. Software may be interpreted by intermediaries, for example a Java Virtual Machine or .NET runtime, or a script interpreter for scripting languages like Perl or Python, or it may be compiled from a high-level programming language like C, and the software may utilize system calls that execute code found in the operating system kernel, but in the end all software is executed through the ISA. In a typical system, some of the ISA can be used directly by applications, but another part of the ISA (usually that dealing with critical system resources) is only available to the higher-privileged operating system. If unprivileged software attempts to use a restricted portion of the ISA, the instruction will "trap" to the privileged operating system.

Device drivers

Device drivers are a software interface provided by device vendors to enable the operating system to control devices (hard drives, graphics cards, etc.). Device drivers often reside in the operating system kernel and run at high privilege, and are hence part of the trusted computing base in traditional systems – but as they are not always written with ideal security or robustness, they constitute a dominant source of operating system errors [36].

ABI

The *Application Binary Interface* (ABI) is the abstracted interface to system resources that the operating system exposes to clients (applications). The ABI typically consists of system calls. Through system calls, applications can obtain access to system resources mediated by the operating system. The operating system ensures the access is permitted and grants it in a safe manner. The ABI can remain consistent across different hardware platforms since the operating system handles the particularities of the underlying hardware, thus exposing a common interface regardless of platform differences.

API

An *Application Programming Interface* (API) provides a higher level of abstraction than the ABI. Functionality is provided to applications in the form of external code "libraries" that are accessed using a function call interface. This abstraction can facilitate a common interface for applications not only across different hardware platforms (as with the ABI), but also across different operating systems, since the API can be reimplemented as necessary for each ABI. Furthermore, APIs can be built on top of other APIs, making it at least possible that only the lower-level APIs will have to be reimplemented to be used on a new operating system. (In reality, however, depending on the language used to implement the library, it doesn't always work out so ideally.) As previously mentioned, however, all software is executed through the ISA in the end – meaning that an API or application may have to be recompiled, even if it doesn't have to be reimplemented, as it moves to a new platform.

Interfaces, abstraction, and virtualization

Each of these interface levels represents an opportunity for virtualization, since clients of an interface depend only on the structure and behavior of the interface (also known as its *contract*), and not its implementation. Here we see the idea of *abstraction*. Abstraction concerns providing a convenient interface to clients, and can be illustrated as follows – an application asking an operating system for a TCP/IP network connection most likely does not care if the connection is formed over a wireless link, a cellular radio, or an ethernet cable, or if TCP semantics are achieved using other protocols, and it does not care about the network card model or the exact hardware instructions needed to set up and tear down the connection. The operating system deals with all these issues, and presents the application with a handle to a convenient TCP/IP connection that adheres to the high-level interface contract, but may be implemented under the surface in numerous ways. Abstraction enables clients to use resources in a safe and easy manner, saving time and effort for common tasks. Virtualization, however, usually means more than just abstraction; it implies more about the nature of what lies behind the abstraction. A virtualization layer not only preserves abstraction for its clients, but may also use intermediate structures and abstractions between the real resources and the virtual resources it presents to clients[95] – such as using files on a real disk to simulate virtual disks, or using various resources and techniques above the physical memory to simulate private address spaces. And it may multiplex resources, such as the central processing unit (CPU), among multiple clients, presenting each client with a picture of the resource corresponding to the client's own context, creating in effect more instances of the resource then exist in actuality.

2.2.2 Types of virtualization

There are two most prominent basic types of virtualization – process virtualization and system virtualization[95]. Also noteworthy topics are binary translation, paravirtualization, and previrtualization, which are approaches to system and/or process virtualization, as well as containers, a more lightweight relative of system virtualization. These concepts illustrate basic types of virtualization currently in use.

Process virtualization

Process-level virtualization[96, ch. 3] is a fundamental concept in virtually every modern mainstream computer system. In process virtualization, an operating system virtualizes the memory address space, CPU registers, and other system resources for each running process. Each process interacts with the operating system using a virtual ABI or API, unaware of the activities of other processes[95].

Processes, OSs, and memory hierarchy are discussed at length in [93]. The operating system manages process virtualization and maintains the context for each process. For instance, in a context switch, the operating system must swap in the register values for the newly scheduled process, so that the process can begin executing where it left off. The operating system typically has a scheduling algorithm to ensure that every process gets a fair share of CPU time, thereby maintaining the illusion of sole access to the CPU. Through virtual memory, each process has the illusion of its own independent address space, in which its own data and code as well as system and application libraries are accessible. A process typically can't access the address space of another process. The operating system achieves virtualization of memory through the use of page tables, which translate the virtual memory page addresses in processes' virtual address space to actual physical memory page addresses. To map a virtual address to a physical address, the operating system conducts a "page table walk" and finds the physical page corresponding to the virtual page in question. In this way, different processes can even access the same system libraries in the same physical locations, but in possibly different virtual pages in their own address spaces. A process simply sees a long array of bytes, whereas underneath, some or all of those bytes may be loaded into different physical memory pages or stored in the backing store (usually on a hard drive). Furthermore, a modern processor typically has multiple cache levels (termed the L1 cache, L2 cache, and so on) where recently or frequently used memory pages can be stored to enhance retrieval performance – the higher the level, the smaller the cache size but the greater the speed. (A computer system memory hierarchy can often be visualized as a pyramid, with slower, lower cost, higher capacity storage media at the bottom, and faster, higher cost, lesser capacity media at the top.) And, a CPU typically also uses other specialized caches and chips, such as a *translation lookaside buffer* (TLB) that caches translations from virtual page numbers to physical page numbers (that is, the results of page table walks). Virtual memory, depicted in figure 2.3, is thus the outward-facing facade of a complex internal system of technologies.

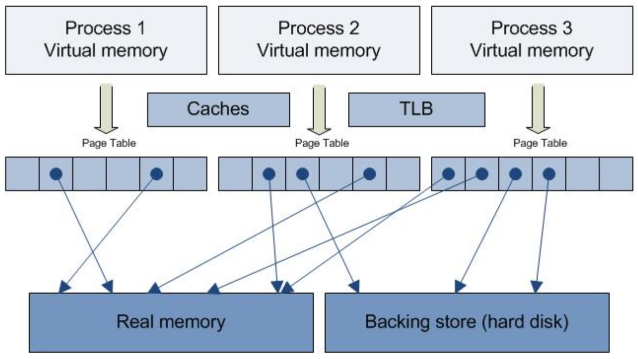

Fig. 2.3: Virtual memory for processes

In short, processes interact obliviously with virtual memory and other resources through standard ABI and APIs, while the operating system manages the virtualization and multiplexing of resources under the hood.

System virtualization

In contrast to process virtualization, in system virtualization[96, ch. 8] an *entire hardware system* is virtualized, enabling multiple virtual systems to run isolated alongside each other [95]. A *hypervisor* or *virtual machine monitor* (VMM) virtualizes all the resources of a real machine, including CPU, devices, memory, and processes, creating a virtual environment known as a *virtual machine* (VM). Software running in the virtual machine has the illusion of running in a real machine, and has access to all the resources of a real machine through a virtualized ISA. The hypervisor manages the real resources, and provides them safely to the virtual machines. The hypervisor may support one or more virtual machines, and thus is responsible for making sure all real machine resources are properly managed and shared, and for maintaining the illusion of the virtual resources presented to each virtual machine (so that each virtual machine "thinks" it has its own real machine). This type of virtualization is depicted in figure 2.4.

Note here that the VMM may divide the system resources in different ways.

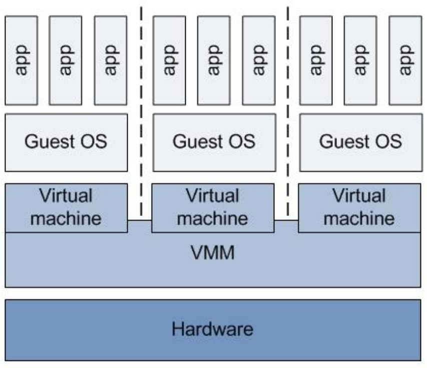

Fig. 2.4: System virtualization

For instance, if there are multiple CPU cores, it may allocate specific cores to specific VMs in a fixed manner, or it may adopt a dynamic scheme where cores are assigned and unassigned to VMs flexibly, as needed. (The latter is similar to how an operating system allocates the CPU to its processes via its scheduling algorithm.) The same goes for memory usage – portions of memory may be statically allocated to VMs, or memory may be kept in a "pool" that is dynamically allocated to and deallocated from VMs. Static allocation of cores and memory is simpler, and results in stronger isolation, but dynamic allocation may result in better utilization and performance[95].

Virtualization of this standard type has been around for decades, and is increasing quickly in popularity today, thanks to the flexibility and cost-saving benefits it confers on organizations[105], as well as due to commodity hardware support discussed in section 2.5. Note as well that it is expanding from its traditional ground (the data center) and into newer areas such as security and mobile/embedded applications[64].

ISA translation

If the guest and virtualization host utilize the same ISA, then no ISA translation is necessary. Clearly, running the host and guest with the same ISA and thus not requiring translation is simpler, and better for performance. Scenarios do arise, however, in which the guest uses a different ISA than the host. In these cases, the host must translate the guest's ISA. Both process and system virtualization layers can translate the ISA; a VMM supporting ISA or *binary* translation[96, ch. 2] is sometimes known as a "Whole System" VMM[95].

ISA translation can enable operating systems compiled for one type of hardware to run on a different type of hardware. Therefore, it enables a software stack for one platform to be completely transitioned to a new type of hardware. This may be quite useful. For example, if a company requires a large legacy application but lacks the resources to port it to new hardware, they can use a whole system VMM. Another example of the benefits of ISA translation might be if an ISA has evolved in a new or branching CPU line, but older software should still be supported – systems such as the IA32 Execution Layer, or IA32-EL[22], which supports execution of Intel IA-32 compatible software on Itanium processors, can be used. Alternatively, if a company develops for multiple hardware platforms, whole-system VMMs can facilitate multiple-ISA development environments consolidated on a single workstation.

A virtualization system may translate or optimize the guest ISA in different ways[95]. Through *interpretation*, an emulator runs a binary compiled for one ISA by reading the instructions one by one and translating them to a different ISA compatible with the underlying system. Through *dynamic binary translation*, blocks of instructions are translated at once and cached for later, resulting in higher performance than interpretation. Even if the guest and host run the same ISA, the virtualization layer may also seek to dynamically optimize the binary code, as in the case of the HP Dyanmo system[21].

Binary translation may also be needed in systems where the hardware is not virtualization-friendly; in these cases, the VMM can translate unsafe instructions from a VM into safe instructions.

Paravirtualization

In relation to ISA translation, *paravirtualization* represents a different, possibly complementary approach to virtualization. In paravirtualization, the actual guest code is modified to use a different interface that is either safer or easier to virtualize, improves performance, or both. The interface used by the modified guest will either access the hardware directly or use virtual resources under the control of the VMM, depending on the situation, facilitating performance and reliability[105]. Sometimes portions of the interface that call into a hypervisor are known as *hypercalls*. The Denali system first coined the term paravirtualization, utilizing the strategy in support of a lightweight, multi-VM environment suited for networked application servers[118]. Other systems, such as Xen[23], also use paravirtualization.

Paravirtualization comes, of course, at the cost of modifying the guest software, which may be impossible or difficult to achieve and maintain. But in cases of well-maintained, open software (such as Linux), paravirtualized distributions may be conveniently available.

Like binary translation, paravirtualization can also serve in situations where underlying hardware is not supportive of virtualization. The paravirtualization of the guest gives the VMM control over all sensitive operations that must be virtualized and managed.

Pre-virtualization

Pre-virtualization, or *transparent paravirtualization*, as it is sometimes called, attempts to bring the benefits of both binary translation (which offers flexibility) and paravirtualization (which brings performance)[68]. Pre-virtualization is achieved via an intermediary between the guest code and the VMM – this intermediary can come in the form of either a standard, neutral interface agreed on by VMM and guest OS developers, or an automated offline translation process such as using a special compiler. Both are offered by the L4Ka implementation of the L4 microkernel – L4Ka supports the generic Virtual Machine Interface proposed by VMWare [109], and also provides their Afterburner tool that compiles unmodified guest OS code with special notations that enable it to run on a special, guest-neutral VMM layer[68].

Pre-virtualization aims to decouple the authoring of guest OS code from the usage of a VMM platform, and thereby retain the security and performance enhancements of paravirtualization without the ususal development overhead – a neutral interface or offline compilation process facilitate this decoupling. Pre-virtualization is a newer technique that bears watching.

Containers

Containers are an approach to virtualization that runs above a standard operating system but provides a complete, lightweight, isolated virtual environment for collections of processes [105]. An example is the OpenVZ project for Linux[80], or the system proposed in [97].

Applications running in the containers must run natively on the underlying OS – containers do not enable heterogeneous OS environments. But in such situations, containers can pose a less-resource intensive path to system isolation than traditional virtualization.

One must, however, observe that a container system is not a minimal trusted hypervisor, but instead running as a part of what may be a monolithic OS; hence, any security ramifications in the container system architecture and the isolation mechanisms must be considered.

2.2.3 Non-standard systems

The above discussion on the basics of virtualization has concerned itself with typical system types, where layers of abstraction are used to expose higher and higher level interfaces to clients, promoting portability and ease-of-use, and creating a hierarchy of responsibility based on interface contracts. This common sort of architecture lends itself to virtualization. But it is worth mentioning that there are other types of computer systems in existence that may be not so amenable to virtualization. For instance, exokernels[43] take a totally different approach – instead of trying to abstract and “baby-proof” a system with higher and higher level interfaces, exokernels provide unfettered access to resources and allow applications to work out the details of resource saftey and management for themselves. This yields much more control and power to the application developer, but is more difficult and dangerous to deal with – similar to the difference between programming in C and Java.

2.3 Hypervisors

The hypervisor or VMM is the layer of software that performs system virtualization, facilitating the use of the virtual machine as a system abstraction.

2.3.1 Traditional hypervisors

Traditional hypervisors, such as Xen[23] and VMWare ESX[110], run on the bare metal and support multiple virtual machines. This is the classic type of hypervisor, dating back to the 1970s[48], when they commonly ran on mainframes. A traditional hypervisor must provide device drivers and any other components or services necessary to support a complete virtual system and ISA for its virtual machines.

To virtualize a complete ISA and system environment, traditional hypervisors may use paravirtualization, as Xen does, or binary translation, as VMWare ESX does, or a combination of both, or neither, depending on such aspects as system requirements and available hardware support.

The Xen hypervisor originally required paravirtualization, but can now support full virtualization if the system offers modern virtualization hardware support (see section 2.5). Additionally, Xen deals with device drivers in an interesting way. Instead of having all the device drivers included in the hypervisor itself, it instead uses the device drivers running in the OS found in the special high-privilege Xen administrative domain, sometimes known as Dom0[35, ch. 6]. Dom0 runs an OS with all necessary device drivers. The other guests have been modified, as part of the necessary paravirtualization, to use simple abstract device interfaces that the hypervisor then implements through request and response communication with Dom0 and its actual device drivers.

Protection rings and modes

In traditional hypervisor architecture, the hypervisor leverages a hardware-enforced security mechanism known as *privilege rings* or *protection rings*, or the closely related *processor mode* mechanism, to protect itself from guest VMs and to protect VMs from each other. The protection ring concept was introduced in the Multics operating system in the 1970s[90]. With protection rings, different types of code execute in different rings, with higher privilege code running in higher rings (ring 0 being the highest), with only specific predefined gateway mechanisms able to transfer execution from one ring to another. Processor modes function in a similar way. The current mode is stored as a hardware flag, and only when in certain modes can particular instructions execute. Transition between modes is a protected operation. For example, Linux and Windows typically use two modes – supervisor and user – and only the supervisor mode can execute hardware-critical instructions such as disabling interrupts, with the system call interface enabling transition from user to supervisor mode [119]. Memory pages associated with different rings or modes are protected from access by lower privilege rings or modes. Rings and modes can be orthogonal concepts, coexisting to form a lattice of privilege state.

Following this pattern, the hypervisor commonly runs in the highest privilege ring or mode (possibly a new mode above supervisor mode, such as a *hypervisor mode*), enabling it to oversee the guest VMs and intercept and handle all important instructions affecting the hardware resources that it must manage. This subject will be further discussed in section 2.5 on virtualization hardware support.

2.3.2 Hosted hypervisors

A hosted hypervisor, such as VirtualBox[113] or VMWare Workstation[99, 111], runs atop a standard operating system and supports multiple virtual machines. The hypervisor runs as a user application, and therefore so do all the virtual machines. Performance is preserved by having as many VM instructions as possible run natively on the processor. Privileged instructions issued by the VMs (for example, those that would normally run in ring 0) must be caught and virtualized by the hypervisor, so that VMs don't interfere with each other or with the host. One potential advantage of the hosted approach is that existing device drivers and other services in the host operating system can be used by the hypervisor and virtualized for its virtual machines (as opposed to the hypervisor containing its own device drivers), reducing hypervisor size and complexity[95]. Additionally, hosted hypervisors often support useful networking configurations (such as bridged networking, where each VM can in effect obtain its own IP address and thereby network with each other and the host), as well as sharing of resources with the host (such as shared disks). Hosted hypervisors provide a convenient avenue for desktop users to take advantage of virtualization.

2.3.3 Microkernels

Microkernels such as L4[104] offer a minimal layer over the hardware to provide basic system services, such as interprocess communication (IPC) and processes or threads with isolated address spaces, and can serve as an apt base for virtualization[53]. (However, not everyone agrees on that last point [20, 49].) Microkernels typically do not offer device drivers or other bulkier parts of a traditional hypervisor or operating system. To support virtualization, such services are often provided by a provisioning application such as Iguana on L4[73]. The virtual machine runs atop the provisioning layer. Alternatively, an OS can be paravirtualized to run directly atop the microkernel, as in L4Linux[67].

Microkernels can be small enough to support formal verification, providing formal assurance for a system's trusted computing base (TCB), as in the recently verified seL4 microkernel [63, 74]. This may be of special interest to parties building systems for certification by the Common Criteria[28], or in any domain where runtime reliability and security are mission-critical objectives.

Microkernels can give rise to interesting architectures. Since other applications can be written to run on the microkernel in addition to provisioned virtual machines, with each application running in its own address space isolated by the trusted microkernel, a system can be built consisting of applications and entire operating systems running side by side and interacting through IPC. Furthermore, the company Open Kernel Labs advertises an L4 microkernel-based architecture where not only applications and operating systems, but also device drivers, file systems, and other components can be run in isolated domains, and where device drivers running in one operating system can be used by other operating systems via the mediation of the microkernel[75]. (This is similar to the device driver approach in Xen.)

2.3.4 Thin hypervisors

There is some debate as to what really constitutes a “thin” hypervisor. How thin does it have to be to be called thin? What functionality should it provide? VMWare ESXi, which installs directly on server hardware and has a 32MB footprint[110], is advertised as an ultra-thin hypervisor. But other hypervisors out there are considerably smaller, and one could argue that 32MB is still quite large enough to harbor bugs and be difficult to verify. The seL4 microkernel has “8,700 lines of C code and 600 lines of assembler”[63], and thus is quite a bit smaller while still providing isolation (although not, in itself, capable of full virtual machine support). SecVisor, a thin hypervisor intended to sit below a single OS and provide kernel integriy protection, is even tinier, coming in at 1112 lines when proper CPU support for memory virtualization is available [91] – but of course, it offers still less functionality than seL4. This also indicates that the term "hypervisor" is a superset of "virtual machine monitor", including as well architectures that provide but a thin monitoring, interposition or translation layer between a guest OS and the hardware.

Thin hypervisors are a subject of interest in this thesis. There are numerous thin hypervisor architectures in the research, including the aforementioned SecVisor[91] and also BitVisor[92]. Like traditional hypervisors and microkernels, thin hypervisors run on the bare metal. We will be most interested in ultra-thin hypervisors that monitor and interpose between the hardware and a single guest OS running above it. This presents the opportunity to implement various services without the guest needing to know, including security services. Since ultra thin hypervisors are intended to be extremely small and efficient, they are thus suitable for low cost, low resource computing environments such as embedded systems.

The issue of hardware support is especially relevant for ultra-thin hypervisors, since any activities that can be handled by hardware relieve the hypervisor of extra code and complexity. Since an ultra-thin hypervisor runs with such a bare-bones codebase, hardware support will be instrumental in determining what it can do.

One interesting question is if it is possible to create an ultra-thin hypervisor that will run beneath a traditional hypervisor/VMM, instead of beneath a typical guest OS, and thereby effectively provide security services for multiple VMs but still with an extremely tiny footprint. It is also interesting to consider the possibility of multicore support in a thin hypervisor, given the added complexity yet increasing relevance and prevalence of multicore hardware.

Thin hypervisors will be discussed more later in the context of implementation and security architecture.

2.4 Advantages of System Virtualization

Traditional system virtualization, by enabling entire virtual machines to be logically separated by the hypervisor from the hardware they run on, creates compelling possibilities for system design. Put another way, "by freeing developers and users from traditional interface and resource constraints, VMs enhance software interoperability, system impregnability, and platform versatility." [95] Virtualization yields numerous easily discernible advantages, some of which are discussed in the following sections.

2.4.1 Isolation

A fundamental and manifest advantage of virtualization is isolation between the virtual machines, or domains, enforced by the hypervisor. (Domain is a more generic term than virtual machine, and can denote any isolated domain, such as a microkernel address space.) This leads to robustness and security.

It is worth mentioning that nowadays, instead of traditional pure isolation, virtualization is used in architectures where virtual machines are intended to cooperate in some way – especially in mobile and embedded platforms, discussed in a later section. Therefore it may be important for the hypervisor to provide secure services for inter-VM communication, such as microkernel IPC, while still preserving isolation.

2.4.2 Minimized trusted computing base

A user application depends on, or trusts, all the software running beneath it. A compromise in any software beneath it on the stack, or in any other software that can compromise or control any software on the stack, can compromise the application itself. In modern operating systems, where software often runs with administrative privileges, a compromise of any piece of software can result in total machine compromise and therefore be devastating to any other software running on the machine. Such an architecture presents an immense *attack surface* – the entire exposed facade through which the attacker can approach the system. It could include user applications, operating system interfaces, network services, devices and device drivers, etc.

Virtualization can address this problem by placing a trustworthy hypervisor at the highest privilege on the system and running virtual machines at reduced privilege. Guest software can be partitioned into virtual machines that are trusted and untrusted, and a compromise of an untrusted VM will have no effect on a trusted VM, since the hypervisor guards the gates, so to speak. Total machine compromise now requires compromise of the hypervisor, which typically presents a much slimmer attack surface than mainstream operating systems (although of course that varies in practice). A slimmer attack surface means, in principle, that it is easier to protect correctly. We have already seen in this chapter that very thin hypervisor layers and microkernels have been developed, and even formally verified.

2.4.3 Architectural flexibility

The decoupling of virtual and real renders a great deal of architectural flexibility. VMs can be combined on a single platform arbitrarily to meet particular needs. In the case of whole-system VMMs that translate the ISA, the flexibility even extends to running VMs on more than one type of hardware, and combining VMs meant for more than one type of hardware on a single platform.

2.4.4 Simplified development

Virtualization can lead to simplified software development and easier porting. As mentioned, instead of porting an application to a new operating system, an entire legacy software stack can simply run in a virtual machine, alongside other operating systems, on a single platform. In the case of ISA translation, instead of targeting every hardware platform, a developer can write for one platform, and rely on virtualization to extend support to other platforms.

In addition to reducing the need for porting and developing across platforms, virtualization can also facilitate more productive development environments, for instance by enabling a development or testing workstation to run instances of all target operating systems.

Another example is that when developing a system typically comprised of multiple separate machines, system virtualization can be used to virtualize all these machines on a single machine and connect them with a virtual network.

This approach can also be used to facilitate product demos of such systems – instead of bringing all the separate machines to a customer, a laptop hosting all the necessary virtual machines can be used to portably demonstrate system functionality.

2.4.5 Management

The properties of virtualization result in many interesting benefits when it comes to system management.

Consolidation/resource sharing

Virtualization can increase efficiency in resource utilization via consolidation[51, 64]. Systems with lower needs can be run together on single machines. More can be done with less hardware. Virtualization's effectiveness in reducing costs has been known for decades[48].

Load balancing and power management

In the same vein as consolidation, virtualization can be used to balance CPU load by moving VMs off of heavily loaded platforms (load balancing), and can also be used to combine VMs from lightly loaded machines onto fewer machines in order to power down unneeded hardware (power management)[51, 64].

Migration

Virtual machines can be migrated live (that is, in the middle of execution) between systems, an increasingly useful capability[96, ch.10]. Research has been done to support virtualization-based migration even on mobile platforms [100]. In theory, computing context could be migrated to any compatible platform. Challenges include ensuring that a fully compatible environment is provided for virtual machines in each system they migrate to (including a consistent ISA), so that execution can be safely resumed. Besides facilitating the above-mentioned management applications of consolidation and load balancing, migration supports new scenarios where working context is seamlessly transitioned between environments, such as for employees working in multiple corporate offices, client sites, and travel in between.

2.4.6 Security

Last but definitely not least, virtualization can provide security advantages, and is moving more and more in this direction[64][96, ch. 10]. Of course, these advantages are founded on the minimized TCB and VM/VMM isolation mentioned earlier, the basic properties that make virtualization attractive in secure system design. But building upon these foundational properties can lead to substantial additional security benefit.

A hypervisor has great visibility into and control over its virtual machines, yet is isolated from them, and thus forms an apt base for security services of many and varied persuasions. An interesting aspect of virtualization-based security architecture is that it can bring security services to unmodified guest systems, including commodity platforms.

By using virtualization in the creation of secure systems, designers can reap not only the bounty of isolated domains, but additionally the harvest of whatever security services the hypervisor can support. A later section will discuss virtualization-based security services in greater detail.

2.5 Hardware Support for Virtualization

Virtualization benefits from support in the underlying hardware architecture. If hardware is not built with system virtualization in mind, then it can become difficult or impossible to implement virtualization correctly and efficiently. Challenges can include virtualization of the CPU, memory, and device input/output. For example, if a non-privileged CPU instruction (that is, a portion of the ISA that non-privileged user code is still permitted to execute) can modify some piece of hardware state for the entire machine, then one virtual machine is effectively able to modify the system state of another virtual machine. The VMM must prevent this breach of consistency. In another common example relating to memory virtualization, standard page tables are designed for one level of virtualized memory, but virtualization requires two – one layer for the VMM to virtualize the physical memory for the guest VMs, and one layer for the guest VMs to virtualize memory for their own processes. Lacking hardware support for this second level of paging can incur performance penalties. (Software mechanisms for implementing two-level paging are sometimes known as *shadow page tables.*) In another example, regarding device Input/Output (I/O) where devices use direct memory access (DMA) to write directly to memory pages, a VMM must ensure that devices being used by one VM are not allowed to write to memory used by another VM. If the VMM must validate every I/O operation in software, it can be expensive. There are many other potential issues with hardware and virtualization, mostly centering around the cost and difficulty of trapping/intercepting and emulating instructions and dealing with overhead from frequent context switches in and out of the hypervisor and VMs whenever privileged state is accessed. It is important that hardware contain mechanims for dealing with virtualization issues if virtualization is to be effectively and reasonabley supported.

Without hardware support, VMMs can also rely on the aforementioned *paravirtualization*, in which the source code of an operating system is modified to use a different interface to the VMM that the VMM can virtualize safely and efficiently, or the already described binary translation [72], in which the VMM translates unsafe instructions at runtime. Neither of these solutions is ideal, since paravirtualization, while effective and often resulting in performance enhancements, requires source-code level modification of an operating system (something not always easy or possible), and translation, as stated earlier, can be resource intensive and complicated. (Pre-virtualization could offer a better solution here.) Specifically regarding I/O virtualization without hardware support, a VMM can emulate actual devices (so that device instructions from VMs are intercepted and emulated by the VM, analagous to binary translation), supporting existing interfaces, or it can provide specially crafted new device interfaces to its VMs[57]. Emulating devices in a VM can be slow, and difficult to implement correctly, while providing a new interface requires modification to a VM's device drivers and/or OS, which may be inconvenient. Besides sidestepping these troubles, having hardware shoulder more of the burden for virtualization support can simplify a hypervisor's code overall, further minimizing the TCB, easing development, and raising assurace in security[72]. There are other software-based solutions for enabling virtualization without hardware support, such as the "Gandalf" VMM [60] that attempts to implement lightweight shadow paging for memory management, but it is unlikely that a software-based solution will be able to compete with a competent hardware-based solution.

2.5.1 Basic virtualization requirements

Popek and Goldberg outlined basic requirements for a system to support virtual machines in 1974[84]. The three main requirements are summed up in a simple way in [2]:

1. **Fidelity** – Also called *equivalency*, fidelity indicates that running software on a virtual machine should result in identical results or behavior as running it on a real machine (excepting time-related issues).
2. **Performance** – Performance should be reasonably efficient, which is achieved by having as many instructions as possible run natively, direct on the hardware, without trapping to the VMM.
3. **Safety** – The hypervisor or VMM must have total control over the virtualized hardware resources.

Many modern hardware platforms were not designed to support virtualization and did not meet the fidelity requirement out of the box, meaning that VMM software had to do extra work – negatively impacting the efficiency requirement. But today, CPUs are being built with more built-in virtualization support, including chips by Intel and AMD, and are actually able to meet Popek and Goldberg's requirements.

2.5.2 Challenges in $x86$ architecture

Intel $x86$ CPU architecture formerly offered no virtualization support, and indeed included many issues that hindered correct virtualization (necessitating binary translation or paravirtualization). As a common architecture, it is worth taking a closer look at some of its issues. Virtualization challenges in Intel $x86$ architecture include (as described in [72]):

- Certain IA-32 and Itanium instructions can reveal the current protection ring level to the guest OS. Under virtualization, the guest OS will be running in a lower-than-normal privilege ring. Therefore, being able to discern the current ring breaks Popek and Goldberg's fidelity condition, and can reveal to the guest that it is running in a virtual machine.
- In general, if a guest OS is made to run at lower privilege than ring 0, issues may arise if any portion of the OS was written expecting to be run in ring 0.
- Some IA-32 and Itanium non-faulting instructions (that is, non-trapping, non-privileged instructions) modify privileged CPU state. User-level code

can execute such instructions, and they don't trap to the operating system. Therefore, VMs can issue non-trapping instructions that modify state affecting other VMs.

- IA-32 SYSENTER and SYSEXIT instructions, typically used to start and end system calls, cause a trap to and exit from ring 0, respectively. If SYSEXIT is called outside ring 0, it causes a trap to ring 0. With a VMM running at ring 0, SYSENTER and SYSEXIT will therefore trap to the VMM – on system call entry (when the user application calls SYSENTER, trapping to ring 0) and exit (when the guest OS not at ring 0 calls SYSEXIT, resulting in a trap to ring 0). This creates additional overhead and complication for the VMM.
- Activating and deactivating interrupt masking (for blocking of external interrupts from devices) by the guest OS is a privileged action and may be a frequent activity. Without hardware support, it could be costly for a VMM to virtualize this functionality. This concern also applies to any privileged CPU state that may be accessed frequently.
- Also relating to interrupt masking, the VMM may have to deliver virtual interrupts to a VM, but the guest OS may have masked interrupts. Some mechanism is required to ensure prompt delivery of virtual interrupts from the VMM when the guest deactivates masking.
- Some aspects of IA-32 and Itanium CPU state are hidden – meaning they are inaccessible for reading and/or writing by software – and it is therefore impossible for a context switch between VMs to properly transition that state.
- Intel CPUs typically contain four protection rings. The hypervisor runs at ring 0. In 64-bit mode, the paging-based memory protection mechanism doesn't distinguish between rings 0-2; therefore, the guest OS must run at ring 3, putting it at the same privilege level as user applications (and therefore leaving the guest OS less protected from the applications running on it). This phenomenon is known as *ring compression*.

Modern Intel and AMD CPUs offer hardware support to deal with these challenges. Prominent aspects of hardware virtualization support include support for virtualization of CPU, memory, and device I/O, as well as support for guest migration.

2.5.3 Intel VT

Intel Virtualization Technology (VT) is a family of technologies supporting virtualization on Intel IA-32, Xeon, and Itanium platforms. It includes elements of support for CPU, memory, and I/O virtualization, and guest migration.

Intel VT on IA-32 and Xeon is known as VT-x, whereas Intel VT for Itanium is known as VT-i. Of those two, this document will focus on VT-x. Intel VT also includes a component known as VT-d for I/O virtualization, discussed in later this section, and VT-c for enhancing virtual machine networking, which is not discussed.

VT-x

Technologies under the VT-x heading include support for CPU and memory virtualization, as well as guest migration.

A foundational element of Intel VT-x's CPU virtualization support is the addition of a new bit of CPU state, orthogonal to protection ring, known as *VMX root operation mode*[72]. (Intel VT-i has a similar new bit – the "vm" bit in the processor status register, or PSR.) The hypervisor runs in VMX root mode, whereas virtual machines do not. When executed outside VMX root mode, certain privileged instructions will invariably trap to VMX root mode (and hence the VMM), and other instructions and events (such as different exceptions) can also be configured to trap to VMX root mode. Exit from VMX root mode is called a *VM entry* and entry to this mode is called a *VM exit.* VM entries and exits are managed in hardware via a structure known as the Virtual Machine Control Structure (VMCS). The VMCS stores virtualization-critical CPU state for VMs and the VMM so that it can be correctly swapped in and out by hardware during VM entries and exits, freeing VMM software from this burden. Note also that the VMCS contains and provides access to formerly hidden CPU state, so that the entire CPU state can be virtualized.

The VMCS stores the configuration for which optional instructions and events will trap to VMX root mode. This enables the VMM to "protect" appropriate registers, handle certain instructions and exceptions, handle activity on certain input/output ports, and other conditions. A set of CPU instructions provides the VMM with configuration access to the VMCS.

Regarding interrupt masking and virtualization, the interrupt masking state of each VM is virtualized and maintained in the VMCS. Further, VT-x provides a control feature whereby a VMM can force traps on all external interrupts and prevent a VM from modifying the interrupt masking state (and attempts by the VM to modify the state won't trap to the VMM). There is also a feature whereby a VMM can request a trap if the VM deactivates masking [72]. Therefore, if masking is active, the VMM can request a trap when masking is again deactivated – and then deliver a virtual interrupt.

Additionally, it is important to observe that since VMX root mode is orthogonal to protection ring, a guest OS can still run at ring 0 – just not in VMX root mode. This alleviates any problems arising from a guest OS running at lower privilege but expecting to run at ring 0 (or from a guest OS being able to detect that it isn't running in ring 0). It also solves the problem of SYSENTER and SYSEXIT always faulting to the VMM and thus impacting system call performance – now, they will behave as expected, since the guest OS will run in ring 0.

Another salient element of VT-x's CPU virtualization support is hardware support for virtualizing the Task Priority Register (TPR)[72]. The TPR resides in the Advanced Programmable Interrupt Controller (APIC), and tracks the current task priority – only interrupts of higher priority priority will be delivered. An OS may require frequent access to the TPR to manage task priority (and therefore interrupt delivery and performance), but a guest OS must not modify the state for any other guest OSes, and trapping frequent TPR access in the VMM could be expensive. Under VT-x, a virtualized copy of the TPR for each VM can be kept in the VMCS, enabling the guest to manage its own task priority state – and a VM exit will only occur when the guest attempts to drop its own TPR value below a threshold value also set in the VMCS [72]. The VM can therefore modify, within set bounds, its TPR – without trapping to the VMM. (This technology is advertised as Intel VT FlexPriority.)

Moving on from virtualization of the CPU, Intel VT-x also now contains a feature called Extended Page Tables (EPTs)[56], which support virtualization memory management. Standard hardware page tables translate from virtual page numbers to physical page numbers. In virtualization scenarios, use of these basic page tables requires frequent synchronization effort for the VMM, since (as described in the beginning of section 2.5) the VMM needs to virtualize the physical page numbers for each guest. The VMM must somehow maintain the physical mappings for each guest VM. With EPTs, there are now two levels of page tables – one page tabe translates from "guest virtual" to "guest physical" page numbers for each VM, and a second page table translates from "guest physical" to the "host physical" page numbers that correspond to actual physical memory. In this way, a VM is free to access and use its own page tables, mapping between the VM's own virtual and "guest physical" addresses, in a normal way, without needing to trap to the VMM – resulting in performance savings.

However, EPTs do result in a longer page table "walk" (a page table walk is the process of "walking" though the page tables to find the physical address corresponding to a virtual address), due to the second page table level. Therefore, if a process incurs many TLB misses, necessitating many page table walks, performance could suffer. One possible solution to this problem is to increase page size, which could reduce the number of TLB misses (depending on the process's memory layout).

Another VT-x feature supporting memory virtualization is Virtual Process Identifiers (VPIDs), which enable a VMM to maintain a unique ID for each process running within the VMs (and for its own process). TLB entries can then be tagged with a VPID, and therefore the TLB won't have to be flushed (which is expensive) in VM entries and exits ([72]), since entries for different VMs are distinguishable.

Finally, VT-x includes a component dubbed "FlexMigration" that facilitates migration of guest VMs among supporting Intel CPUs. Migration of guest VMs in a varied host pool can be challenging, since guest VMs may query the CPU for its ID and thereafter expect the presence of a certain instruction set, but then may be migrated to another system supporting slightly different instructions. FlexMigration helps possibly heterogeneous systems in the pool to expose consistent instruction sets to all VMs, thus enabling live guest migration.

VT-d

Device I/O uses DMA, enabling devices to write directly to memory pages without going through the operating system kernel. (DMA for devices has been a source of security issues in the past, with devices such as Firewire devices being able to write to kernel memory, even if accessed by an unprivileged user. Attacks on the system via DMA are sometimes called "attacks from below".) The problem with DMA for devices on virtualization platforms is that devices being used by a guest shouldn't be allowed to access memory pages on the system belonging to other guests or the VMM – therefore, on traditional systems, all device I/O operations must be checked with or virtualized by the VMM, thereby reducing performance. Hardware support can enable guest associations and memory access permissions to be established for devices and automatically checked for any I/O operation.

Intel VT for Directed I/O (also known as Intel VT-d) offers hardware support for device I/O on virtualization platforms[57]. It provides several key features (as described in [57]):

- *Device assignment* – The hardware enables specification of numerous isolated domains (which might correspond to virtual machines on a virtualization platform). Devices can be assigned to one or more domains, so that they can only be used by those domains. In particular, this allows a VM domain to use the device without trapping to the VMM.
- *DMA remapping* – through use of I/O page tables, the pages included in each I/O domain and the pages that can be accessed by each device can be restricted. Furthermore, pages that devices write to can be logically remapped to other physical pages. In I/O operations, the page tables are consulted to check if the page in question may be accessed by the device in question on behalf of the current domain. Different I/O domains are effectively isolated from each other. Note that this feature is necessary to make device assignment safely usable – since it prevents a device assigned to one domain from accessing pages belonging to another domain.
- *Interrupt remapping* – Device interrupts can be restricted to particular domains, so that devices only issue interrupts to the domains that are expecting them.

DMA remapping offers a plethora of potential uses, both for standard systems with a single OS and for VMMs with multiple VMs[57]. For standard systems, DMA remapping can be used to protect the operating system from devices (by prohibiting device access to kernel memory pages), and to partition system memory into different I/O domains to isolate the activity of different devices. It can also be used on 64-bit systems to support legacy 32-bit devices that are only equipped to write to a 4GB physical address space; the addresses the device writes to can be remapped to higher addresses in the larger system address space (which would otherwise require expensive OS-managed bounce buffers).

A VMM, on the other hand, might simply assign devices to domains (which will most likely correspond to VMs), and devices will thereby be restricted to operating on any memory owned by that domain (VM). As mentioned, this will also enable guest VMs (and their device drivers) to interact with their assigned I/O devices without trapping to the VMM. Furthermore, the VMM can assign devices to multiple domains to facilitate I/O sharing or communication. Finally, if the VMM virtualizes the DMA remapping instructions for its VMs, then the guest VMs can use the remapping support in a similar way to an OS on a standard system – protecting the OS, limiting and partitioning the memory regions that a device can write to, and remapping regions for legacy devices. To virtualize the remapping instructions and state, the VMM could maintain this state (in an eagerly updated "shadow copy"[57]) for each VM, by intercepting VM modification of its I/O page tables and VM usage of the registers controlling the remapping. (Perhaps a future hardware revision could provide built-in hardware support for virtualization of the remapping facilities.)

The interrupt remapping component of VT-d can also be put to multiple uses by a VMM[57]. A VMM can ensure that device-generated interrupts are routed only to the domains that the devices are assigned to. It can also use the remapping hardware as a kind of “interrupt firewall” to ensure that external interrupts do not have characteristics that would cause them to be confused with internal VMM interrupts. Finally, the interrupt remapping can be used to enable safe migration of interrupts (the transfer of interrupts to the correct processor) when the associated domain/workload has moved to another processor – useful in load balancing situations.

2.5.4 AMD-V

AMD’s version of virtualization support is entitled AMD-V[8], and offers comparable support for CPU, memory, and I/O virtualization, and migration.

CPU

AMD-V incorporates a new bit of CPU state entitled “guest mode” [3] that is analagous to non-VMX root mode in Intel VT-x. Guest mode is entered via the VMRUN instruction. Whenever VMRUN is called for a specific VM, the hardware accesses a structure called a *Virtual Machine Control Block* (VMCB) for that VM. The VMCB stores configuration information on what events and interrupts should be intercepted by the VMM for that guest, as well as CPU state for that VM, and bits to indicate additional special instructions for preparing the VM’s execution environment. On VMRUN, the VMCB is used to swap in the VM CPU state, and VMM state is saved to memory for later.

AMD-V also offers similar support to Intel for interrupt virtualization[3]. First, it has a master bit in the VMCB that activates or deactivates interrupt virtualization – if active, then the guest interrupt masking bit only controls virtual interrupts, and the VMM’s interrupt masking bit controls physical (external) interrupts. (If interrupts aren’t virtualized, the guest controls both physical and virtual interrupt masking.) If interrupts are virtualized, then the TPR value for each guest is also virtualized. The VMM can choose to intercept all physical interrupts, deliver virtual interrupts to guests, and also force a trap when a VM with interrupts masked enables them once again. There are additionally mechanisms for the VMM to clear out the pending interrupt queue in an arbitrary manner or disregard certain interrupt vectors when determining the highest priority pending external interrupt – this can help in the case of a VM that is blocking other VMs by not processing its own external interrupts.

Memory

Rapid Virtual Indexing, also known as Nested Paging, is AMD’s version of hardware support for virtualization memory management[4]. Like Intel’s EPTs, it incorporates a second level of hardware page tables, eliminating the need for shadow paging. It functions similarly to EPTs, and has been shown to yield dramatic performance increases (but, likewise, potentially suffers from the problem of slower page table walks)[108].

Address Space Identifiers (ASIDs) are used to eliminate the need for TLB flushes when switching to a new VM[4]. An ASID is a unique ID assigned to each guest by the hypervisor, and is used to tag TLB entries, so that TLB entries for different VMs can be distinguished. It is similar to Intel's VPID feature, and basically updates the TLB along with the page tables to support a two-level virtual memory scheme.

Migration

AMD-V Extended Migration also provides hardware support for live migration of VMs between AMD Opteron processors in a pool of systems[5]. This support includes features to facilitate backward compatibility (by limiting the instruction set features exposed to guests to the lowest common denominator of all systems in the pool) and forward compatibility (by allowing VMMs to disable instructions found on newer processors that guests expect to *not* be functioning). In other words, similar to Intel's FlexMigration, it helps ensure that a guest will never find an unexpected instruction environment, no matter where it migrates to in the pool.

I/O

Similar to Intel VT-d, AMD-V contains a component termed an I/O Memory Management Unit (IOMMU) (previously DEV) that provides support for I/O virtualization[7]. It uses similar components – through I/O page tables, I/O memory accesses are checked for permissibility and remapped. Through a device table, devices can be assigned to certain domains, which correspond to a particular portion of the I/O page tables (and therefore memory regions and remappings). And, through an interrupt mapping table, interrupts are checked for permissibility and routed to the appropriate domains.

It is worth mentioning that, due to AMD64 systems consisting potentially of multiple processors and device nodes that are spread out and connected with AMD "HyperTrasport" links, an IOMMU can only intercept I/O memory accesses if the operation goes through the IOMMU node in the HyperTransport network – therfore, multiple IOMMUs can be necessary to cover all devices[7].

2.5.5 ARM TrustZone

ARM TrustZone technology, for ARM11 and ARM Cortex embedded processors (include ARM Cortex-A9 MPCore multicore processors), offers support for creating two securely isolated virtual cores (or "*worlds*", as they are termed) on a single real core. One world is Secure and one world is Normal, and TrustZone manages transitions between them, preventing state or data from leaking from the Secure world to the Normal world[18]. While overall less developed and more limited in capabilities than Intel VT or AMD-V, and intended more for supporting security architectures in general, it does offers some similar components to those found in $x86$ virtualization support packages. It is described in detail in [18] and [17].

First to mention is that the system bus control signals now contain one extra bit, the NS or "Non-Secure" bit, that functions like a 33rd address bit to differentiate between the two worlds. Each virtual core has its own address space – through special TrustZone memory controllers ([13, 14]), physical memory is statically assigned to the Secure or Normal worlds. Furthermore, TrustZone provides a feature called the Advanced Peripheral Bus (APB) that is connected to the main system bus by a bridge component – this bridge component enforces security for all peripherals on the APB, and can deny unsecure or otherwise problematic transactions from being dispatched to peripherals. Hardware devices can be assigned to the Secure or Normal world. This enables tight control of, for example, the interrupt controller, screen and keyboard.

Both worlds have user and privileged modes as usual. But the Secure world also contains a special mode called "Monitor Mode" that is responsible for context switching between the two worlds. The *secure monitor call* (SMC) instruction can be issued by the Normal world when in privileged mode, and always traps to Monitor mode. The Secure and Normal worlds and Monitor mode each have their own exception handlers and vector tables, so exceptions issued in the Normal world will trap to privileged mode handlers in the Normal world. *External* interrupts and aborts can also be made to trap to Monitor mode, but system calls, Memory Management Unit (MMU) memory faults, and misuse of undefined or privileged instructions can't be configured to trap to Monitor mode. Monitor mode is responsible for swapping in and out CPU state when switching from one world to another, allowing execution to begin where it left off in whichever world is being switched to.

TrustZone supplies a virtual MMU for each of its two worlds, enabling each one to manage its own virtual to physical mappings for greater efficiency and isolation. Note that the Secure world can map in pages from the Normal world, but not the other way around. Important MMU state (such as the location of page tables) is kept independently for each world. Additionally, TLB entries are tagged with the associated world, to prevent the need for TLB flushes in a context switch between worlds. Cache entries are also tagged with the associated world, easily facilitating cache usage by both worlds.

External interrupts generated for either world can be handled efficiently; if they are destined for the currently running world, then they are delivered immediately, whereas if they are intended for the other world, execution can trap to Monitor mode and then the interrupt can be properly routed. The Monitor typically runs in a non-interruptible state (interrupts disabled).

In addition to the above-described mechanisms, one could say that security begins on TrustZone platforms with the secure boot process initiated when the device is powered on. The hardware bootloaders that kick off the process can utilize public key cryptography to verify the integrity of code at each successive step in the process (creating a chain of trust), and can leverage a Trusted Platform Module (TPM)[121] or other tamper-resistant module. The system always boots into the Secure world first, and the Secure world then loads the Normal world – this prevents untrusted code in the Normal world from making unauthorized system changes before the Secure world has properly prepared the system.

There are numerous other features available in the TrustZone hardware "library", including a special DMA controller (DMAC) capable of simultaneously handling channels for the Secure and Normal worlds. As previously covered, taking the I/O memory traffic burden off of the processor and the high-privilege software can offer significant performance savings.

So, while TrustZone doesn't offer support for arbitrarily many virtual machines, it does support two strongly isolated virtual cores with partitioned devices and independent memory management facilities, as well as regulated paths for transition between the two worlds.

Potential TrustZone system designs for secure architectures, using various TrustZone-supportive hardware components, can be broken down into different tiers, as described in [19]:

- **Tier One** – In this basic (and low-cost) mode, intended to support secure PIN entry and payment protocols, the Secure world runs a Secure OS and the Normal world runs an Open OS. The Open OS is running and controlling input peripherals and the screen the majority of the time, but if secure entry of a PIN or other data is required (especially in service of some type of payment transaction), the Secure OS takes control of the input devices and the screen. The Secure OS uses an isolated contiguous block of memory. It is booted with a trusted boot process, whereby a hardware component boots a base OS, then loads the Secure OS, which subsequently loads the Open OS.
- **Tier Two** – A superset of Tier One, Tier Two is intended to support Digital Rights Management (DRM) applications. The Secure OS owns certain protected memory regions used for DRM content, and if the Secure OS itself doesn't perform the decoding, then an external chip or other peripheral can also be used (and access to this component will be restricted to the Secure OS). Tier Two involves more complex control capabilities over devices and I/O than Tier One, to safeguard the protected content.
- **Tier Three** – Tier Three, a superset of Tier Two, is intended to offer full support for "cloud computing" in which secure services run in a protected manner in the Secure OS and untrusted data is received, processed, and distributed by the Open OS. It increases support for device control, and adds the DMA controller as well as additional acceleration mechanisms for securely, efficiently processing DRM on large content files.

It is at least *conceivable* to use TrustZone to support virtualization. The hypervisor runs in the Secure world, and a single guest runs in the Normal world. The DMA controller and device control mechanisms support I/O virtualization, and the TrustZone isolation mechanisms and interrupt handling support isolate the hypervisor from the guest. However, the lack of complete interposition capabilities (for example, on system calls and memory faults) and inability to control the Normal world memory management may create difficulties.

2.6 Typical Virtualization Scenarios

In this section we will discuss virtualization scenarios seen today.

2.6.1 Hosting center

Hosting centers can use virtualization to provide systems for clients. Clients can share time on virtualized systems with quality of service guarantees. Restricted to their own isolated domains, clients are prevented from interfering with each other. This scenario sounds quite familiar to the time-sharing mainframes of yesteryear, and indeed the scenarios bear resemblance. The hosting center is a very typical virtualization use-case, where VMs are purely isolated and share resources according to a local policy.

2.6.2 Desktop

Virtualization on the desktop is becoming much more common nowadays, which has inspired (and is inspired by) progress in virtualization support in commodity desktop hardware [72]. In corporations, especially development houses, virtualization is used to give engineers easy access to multiple target platforms. Another possible corporate scenario is enabling employees to have virtual machines configured for different clients or workplace scenarios on one machine. With VirtualBox freely available, even home users can cheaply leverage virtualization to access multiple operating systems or partition their system into trusted and untrusted domains. Virtualization gives desktop users the freedom to have all the heterogeneous computing environments they need at their fingertips, without absorbing extra hardware cost.

2.6.3 Service provider

A service provider (such as a web service provider) may utilize virtualization to consolidate resources or servers onto fewer hardware platforms. For instance, a web application may have a front end web server and multiple back end tier servers, hosted as virtual machines on a single physical machine. Application servers and/or machines processing business logic and may actually be virtual machines that migrate across banks of real machines as needed.

2.6.4 Mobile/embedded

Lastly, a quickly emerging virtualization scenario is the mobile/embedded arena – it is becoming more and more common now to have mobile devices containing isolated domains entrusted with different purposes[101], such as an employee smartphone containing isolated home and work environments[64]. With processors shrinking in size and increasing in performance, growing numbers of embedded systems have the power to support virtualization and leverage its benefits. Embedded CPUs with multiple cores and/or potential built-in security/virtualization support, as in the already discussed ARM Trustzone, further enhance possibilities.

Multiple companies are working in the mobile virtualization space, including Open Kernel Labs[76], VirtualLogix [107], and now VMWare[112]. It has been found to be not unduly onerous to port virtualization architectures to mobile platforms[29], and open systems such as the L4 microkernel [104] and Xen on ARM [55, 123] afford open, low-cost solutions.

Therefore, the benefits of virtualization already discussed can be brought to mobile systems, additionally enabling applications and benefits specific to the mobile/embeddded environment. For example, due to the high frequency of hardware changes and the wide variety of available platforms in embedded systems, virtualization can provide an especially convenient layer of abstraction to facilitate application development. Applications could be distributed as an entire software stack (including a specific OS) to run in a VM, and therefore not depend on any particular ABI[51]. Isolated virtual machines can serve as mobile testbed components or nodes in opportunistic mobile sensor networks [38], and support heterogeneous application environments[51]. Modularity and live system migration is of interest in the mobile environment[100]. Virtualization can also support mobile payment, banking, ticketing, or other similar applications via isolated trusted components (as in TrustZone design tiers) – for instance, Chaum's vision of a digital wallet, with one domain controlled by the bank and one domain by the user [32], could potentially be implemented with virtualization, enabling people to carry "e-cash" in their PDA or smartphone. And of course, beyond isolation, many aspects of security in embedded scenarios may be served by virtualization, as will be discussed later.

2.7 Hypervisor-based security architectures

2.7.1 Advantages

Virtualization serves as a powerful enabler for security services and security architectures, due to the hypervisor's minimized TCB, the isolation enforced between hypervisors and guests, and the hypervisor's presence in a higher hardware protection zone than the guest(s). Security services based on a hypervisor have excellent visibility into guests, yet are still securely protected from guests – this overcomes the problems inherent in traditional architectures such as intrusion detection systems, where the security service is either remotely located (with greatly reduced visibility) or located on the monitored system itself (with greatly increased vulnerability to attackers)[46].

Due to modern operating systems' bulk and complexity, and abundance of continually unearthed critical security flaws, security services implemented by such OSs may not be trustworthy. In fact, the OSs themselves may not be trustworthy. Hypervisor-based security services can be externally applied, in some cases to totally unmodified guest OSs, and thereby bring more trustworthy security. This can provide protection for the guest OS from its applications, for the guest applications from each other, and even for guest applications from the guest OS.

Implementing secure services through hypervisors and virtualization also benefits from virtualization's inherent modularity. Services can potentially be reused for different guests and on different hardware platforms. This could facilitate, for example, a company enforcing consistent security policies efficiently on a wide variety of systems.

2.7.2 Virtualization security challenges

While offering clear benefits, virtualization also creates security-related challenges that must be considered when implementing hypervisor-based secure architectures.

Virtualization is simpler when it concerns strictly isolated virtual machines – but what about when VMs must cooperate? Bellovin discusses the difficulties in defining the interfaces and interactions between VMs, and how this breaks pure isolation and introduces problems[26]. Indeed, as shall be discussed later, there are many emerging scenarios (particularly in mobile platforms) where isolated domains must cooperate in some fashion, and in such cases some sort of mandatory access control, information flow control, or other mechanisms must ensure the security of the interactions and the protection of important resources in the system.

Garfinkel and Rosenblum enumerate a number of potential security problems introduced by virtualization [47]:

- **Scaling** – Virtualization enables rapid creation and addition of new virtual machines. Without total automation, this dynamic growth capacity can destabilize security management activities such as system configuration and updates, resulting in vulnerability to security incidents.

- **Transience** – Whereas normal computing environments/networks tend to converge on a stable state, with a consistent collection of machines, virtualization environments can have machines that quickly come and go. This can foil attempts at consistent management, and leave, for instance, VMs that come and go and are vulnerable to and/or infected by a worm that goes undetected. Infections can persist within such a fluctuating environment and be difficult to stamp out.

- **Software lifecycle** – Since a VM's state is encapsulated in the VMM software (along with any supporting hardware), snapshots of state can easily be taken. A VM can be instantiated from a prior snapshot, enabling easy state rollback – this can interfere with assumptions about the lifecycle of running software. For example, previously applied patches or updates may be lost, or VMs that accept one-time passwords may be made to re-accept used passwords. If rolled back state causes the reuse of stream cipher keys or repetition of other cryptographic mechanisms that shouldn't be reused in an identical fashion, cryptosystems may be compromised.

- **Diversity** – Increased heterogeneity of operating systems and environments will increase security management difficulties, and present a more varied attack surface.

- **Mobility** – While also cited as an advantage of virtualization, mobility and migration automatically engender more complexity and security issues. Moving a VM across different machines automatically increases that VM's TCB to include each one of those machines – therefore increasing security risk, and in a dynamic environment, potentially making it harder to track which VMs may have been exposed to physical machine compromises. It also poses the danger of moving VMs from an untrusted environment (such as a home machine) to a trusted environment, and makes it easier for a malicious insider to steal a machine (since a machine is simply a file on a disk).

- **Identity** – Static means of identifying machines, such as MAC addresses or owner name, may not function with virtualization. Machine ownership and responsibility is harder to track in a dynamic virtualized environment.

- **Data lifetime** – Guest OSs may have security requirements about data lifetime that are invalidated by a VMM's logging and instruction replay mechanisms; through external logging facilities, combined with VM mobility, it is possible that sensitive data may be left in widely distributed persistent storage.

Nichols echoes the configuration and management difficulties, and highlights other virtualization security issues in [106]. For instance, virtual networks, whose traffic is routed internally within a physical machine, won't be protected by all the usual physical network security mechanisms, allowing attacks to be mounted and spread. Furthermore, attacks on VMMs yield a bigger payoff than traditional OS platforms, since a VMM can control multiple virtual machines (and possibly a varying collection over time), so any hypervisor vulnerability becomes extremely critical. Nichols also mentions how security and management tools supporting virtual environments in general are not yet mature, due to the relatively recent gains in virtualization popularity.

Measures may have to be taken to address these challenges, depending on local requirements. Fortunately, ultra-thin, single guest, monitoring/enforcement-oriented hypervisors are not affected by many of these concerns – their small code size lessens likelihood of hypervisor compromise, with a single guest and no hypervisor network presence there is no virtual network, and they do not support the complex management features (mobility, transience) that result in security difficulties. However, they may create some additional management complexity simply because of the increase in individual system complexity. Also, should such a monitoring hypervisor be made to sit beneath a traditional VMM, some of these issues may of course need to be addressed again.

2.7.3 Architectural limitations

Hypervisor-based security services are not a panacea. There are limitations to what can be accomplished.

The semantic gap

Hypervisor-based services, as running external to and at higher privilege than the guest OSes, have complete access to guest memory, but do not have intimate access to guest OS services and context. They have total visibility into the guest, and have the capacity to see all guest memory pages, but they do not have interactivity with guest ABIs, APIs, and abstractions. To have understanding of guest state, the hypervisor (or the service running on it) must somehow bridge the so-called *semantic gap* – the gap in understanding between the hypervisor's view and the guest OS state. Without additional facilities to bridge this gap, the hypervisor will see guest memory, but it will be a meaningless jumble of values. The hypervisor must be endowed with relevant structural, contextual knowledge of the particular guest OS in question.

This is important for security because many security services must have accurate understanding of relevant guest state to implement meaningful functionality. Without such knowledge, a service won't know what is happening in a guest nor will it be able to make reasonable deductions, decisions, or actions based on guest state. Such a service must have processing facilities capable of mapping in guest pages and then interpreting the pages to divine the current relevant state from raw guest memory. Different services may require knowledge of different aspects of guest state.

Using the hypervisor's view into its VMs coupled with contextual knowledge and processing facilities to interpret guest OS state is known as *VM introspection* (VMI); introduced in the Livewire system[46], it is an established technique, but increases the complexity of the security services code, and furthermore introduces management issues since the knowledge base must remain updated in parallel with any relevant updates to the monitored guest OS.

VMI could be divided into two areas – inspection, and interpretation (or *semantic reconstruction*). Inspection is the process of actually mapping the proper guest pages into hypervisor memory. Interpretation is the process of comprehending those pages. There are VMI frameworks in existence such as the publicly available XenAccess[81], as well as the as yet unreleased VIX toolkit (also for Xen)[50], that attempt to provide extensible foundations and tools for VMI. VIX, for instance, contains a set of Unix-like utilities built over an inspection library that can be used from a Xen administrative domain to examine a running virtual machine – this may reveal relevant forensics data, or discrepancies between the guest OS and VMM views due to malware such as rootkits. XenAccess provides an API for mapping and inspecting guest pages from an observer domain, and some examples of how to use the API and interpret guest memory. More advanced, context-specific modules for interpreting state can be built above XenAccess.

Interposition granularity

For performance reasons, as many guest instructions as possible run directly on the hardware. However, as we know, certain instructions and events must trap to and be handled by the hypervisor so it can enforce virtualization, isolation, and so forth. The granularity of events on which the hypervisor can interpose is limited by the hardware interface. The ability to handle events by immediately trapping to hypervisor control is sometimes called *active monitoring*, since the hypervisor and the security service can guarantee active response to supported events, as opposed to *passive monitoring*, wherein guests are periodically monitored at the discretion of the hypervisor-based monitoring service. Passive monitoring by the hypervisor can't guarantee discovery of problems resident in unmonitored state or conditions that can hide or change between monitoring cycles, and can't support immediate prevention or handling of events or negative conditions as they arise.

The hypervisor can handle any event that can be made to trap to the hypervisor's high privilege mode, possibly including privileged instructions, memory accesses, device operations, exceptions/interrupts, or other conditions. Without special virtualization support in hardware, the range and specification of traps may be more limited. In either case, the hardware-supported granularity may not be sufficient for certain applications. For example, certain security monitoring services may need to guarantee response to fine-grained guest events. This problem can be alleviated by using already discussed techniques such as paravirtualization and previrtualization, where the guest OS is made to use a hypercall interface, or binary translation, where appropriate instructions are translated at runtime. These techniques suffer problems already mentioned. Another method is to dynamically introduce hook code into the guest OS, but this code, as resident on the guest and potentially vulnerable to guest compromise, comes with its own security problems. The Lares system[82] uses carefully placed and VMM-protected hook code injected into the guest OS to increase the active monitoring capabilities of the VMM.

Limitations on interposition granularity and the capacity for VM introspection are critical issues for implementing security services, and any improvement to either area will enhance the possibilities for virtualization-based security architectures.

2.7.4 Architectural patterns

When designing virtualization-based security services (that is, security services that run atop a hypervisor and operate on guest domains), there are basic architectural/design patterns that may be followed.

Augmented traditional hypervisor

One method for implementing security services using traditional system virtualization hypervisors is to implement the services in the hypervisor itself. This may be convenient for development, especially if the code of the hypervisor is readily available and already understood. However, it poses the major disadvantage of adding to the complexity and code size of the hypervisor, which counters one of virtualization's fundamental strong points – the minimized TCB presented by the hypervisor. Therefore, it is most likely advisable to take a different approach.

Security VM

With traditional hypervisors, it is quite common to implement security services in a specially designated security VM, similar to Xen's administrative domain "dom0". Through this approach, the security services run in a special VM granted all the necessary privileges by the hypervisor, presumably runnning a stripped down operating system specially crafted for the security services. The VMM/hypervisor must be modified only to the extent that it can communicate with the security VM and provide it with the privileges and resources it needs to implement the security services. This approach, while probably presenting more development overhead than developing directly in the hypervisor, preserves the hypervisor's minimal TCB, and is furthermore more modular (enabling the security services to be more easily modified, transferred, or recombined in other systems in the future).

Microkernel application

In the case of a microkernel serving as a hypervisor, security services can be implemented in a specially written microkernel application, which will run in its own protected address space. It can connect to the VM provisioning layers using the microkernel's IPC services. The application will have to run with sufficient privileges to implement the desired security services.

Thin hypervisor

Lastly, thin, single-guest hypervisors can be used to provide an ultra-low footprint monitoring and enforcement layer between hardware and OS software for implementing security services. An extremely small code size can lead to easier verification and hopefully therefore stronger security and correctness. It is important to consider what types of services can be implemented on which hardware platforms, and still maintain the ultra-low footprint. In this thesis, we are particularly interested in the possibilities of thin hypervisors on embedded systems.

2.7.5 Isolation-based services

We can now briefly examine some of the potential security services provided by virtualization-based architectures, of which there are many. They can be loosely divided into two categories – monitoring- and isolation-based services. Monitoring-based services focus on observing, interpreting, and possibly responding to VM state, and may make heavy use of VM introspection. Isolation-based services, on the other hand, leverage the hypervisor's high privilege and interposition capability to isolate and protect system components and enforce system security. Note that this distinction is not precise, and other categorizations are possible.

We will describe some isolation-based services first.

Isolation architectures

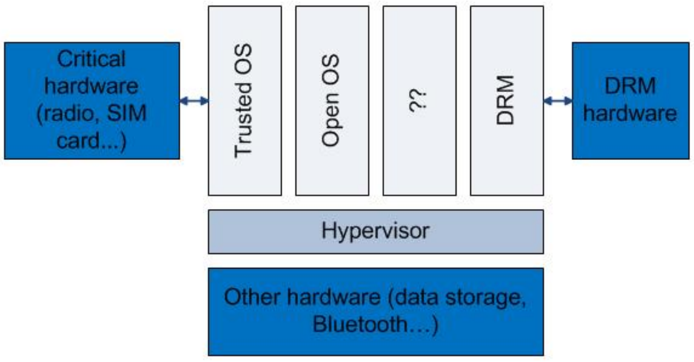
Critical hardware (radio, SIM card...)
Trusted OS
Open OS
??
DRM
DRM hardware
Hypervisor
Other hardware (data storage, Bluetooth...)

Fig. 2.5: Domain isolation in a mobile device

While this section focuses on isolation-based security *services*, it is also worth discussing the interesting possibilities for isolation *architectures* engendered by virtualization. Although isolation of hosted domains is a given security advantage in virtualization, it bears deeper investigation in specific contexts. For example, in a system that contains important components and trusted and untrusted software, virtualization can be used to create a safer environment for the trusted and critical components. Envision a mobile system containing trusted cellular hardware (including the SIM card and cellular radio, which must be safe from compromise), a trusted software stack that controls authentication and the critical hardware, an untrusted software stack running user applications and accessing wireless networks (such as cellular, 802.11 or Bluetooth), a trusted hardware and software component for decoding and protecting DRM content, and possibly other components that must be protected (such as a module for storing private user information). While these components may have been initially contained in a single domain/OS, hence each vulnerable to any compromise of the other, virtualization can support such a scenario by isolating each component in its own domain[29] (see figure 2.5). The hypervisor-enforced isolation protects each domain from the compromise of other components (and potentially protecting each component from even a compromise of itself). The hypervisor must provide secure communication facilities between domains, and possibly limit the communication to only what is needed to support functional requirements. To illustrate the advantages with an example from [38]– if the device's Bluetooth implementation is compromised (Bluetooth has been known to have security vulnerabilities[88]), user applications may be vulnerable, but system authentication and the cellular radio will remain unharmed. Therefore, virtualization can be used to partition a system into various isolated yet cooperative domains and thereby increase security for the system as a whole, also reducing the TCB for the most important components.

Kernel code integrity

There are multiple research systems supporting kernel code integrity.

First off, we have SecVisor[91], an ultra-thin hypervisor (only 1100 lines of code in the presence of Intel-VT or AMD-V) supporting a single guest, ensuring that only approved code is ever executed in kernel mode and that kernel code is not modified (except by SecVisor). The system uses IOMMU support to prevent DMA writes to kernel code, page tables for memory protection, and MMU support to virtualize guest OS memory to protect the page tables. All hardware locations where kernel entry points are specified (such as the interrupt vector table) are virtualized, so that SecVisor can always verify that kernel entries will go to a valid kernel code location. When in kernel mode, user mode pages are marked non-executable, and vice versa. This forces a trap whenever transitioning between modes, enabling SecVisor to switch the set of pages marked non-executable. This trap also enables SecVisor to enforce, in the case of transitioning from kernel code to user code, that the CPU switches to user mode. (Therefore, a buffer overflow in the kernel can't be made to execute shellcode supplied by a malicious user process.) When marking pages executable, SecVisor also marks them read-only, so that code that can be executed can't be modified. Furthermore, when entering kernel mode, SecVisor only marks as executable those pages that are approved by the kernel code policy, so that execution of non-approved code will trap.

A VMM-based kernel protection system, found in [124] and dubbed $UCON_{KI}$ for *usage control framework for kernel integrity*, offers more flexibility. This system provides an access control model based on subjects, objects, attributes, rights, and events. Subjects include processes and loadable kernel modules, objects include kernel memory spaces and registers, attributes describe subjects or objects, rights are actions on objects permissible by subjects, and events are key points at which policy can be enforced by the system. Virtual machine introspection techniques are used to determine subject attributes. Policy includes predicates describing whether rights are to be granted or denied depending on events, subjects, objects, and attributes. One interesting feature of the system is that rights and attributes are dynamic and mutable with continuity – meaning that if an event happens which changes a subject's attributes, its currently granted access rights may be revoked. There may be cascading rights evaluations from a single event. The authors successfully used the system to summarily defeat a large collection of rootkits attempting to modify the kernel. The flexibility of the system indicates it could be adapted and expanded for further uses. In tests it was run on the Bochs emulator, but could be used with other virtualization layers as well.

Memory multi-shadowing

The Overshadow system[33] runs in a VMM and protects applications on a guest OS from each other and from the guest OS itself by using multiple views of guest application memory. To the application, the real view of memory is presented. To other processes (including the OS), an encrypted and integrity-protected view of the memory is presented. The crucial component in this system is a protected *shim* that is inserted into protected applications at load time – this shim is needed to identify and maintain the context of each protected application, and is also used by the Overshadow system to handle complicated operations such as marshalling system call arguments and return values to enable safe transition of data across the application-OS protection boundary. The shim uses a hypercall interface to communicate directly with the VMM. Overshadow uses multiple page tables for an application (one with cleartext pages for the application's own use, one with ciphertext for the use of the rest of the system), and any protected page will only be present in one page table at any given time. Pages can be swapped to disk in encrypted state, and encrypted data can be moved around by untrusted components. In one limited sense, Overshadow is able to remove the OS from the application's TCB, in that the OS can no longer inspect or tamper with application memory pages. The Overshadow system was built on a VMWare binary translation VMM, but it is pointed out that a smaller, higher assurance VMM could have been used as well.

Another system dubbed *Software-Privacy Preserving Platform* (SP^3)[125] (published at the same time as Overshadow, in March 2008) also protects data secrecy for user applications, including memory pages and even registers (the latter during context switches). However, unlike Overshadow, SP^3 instead relies on extensions to the page table and emulation of a modified $x86$ interface in the Xen hypervisor, and requires modification of guest code to utilize new virtual instructions that prompt the hypervisor to invoke operations for creating and managing protection domains. Fortunately, at least in the case of Linux, significant modifications to the guest OS were not required. Protection domains can consist of one or more guest processes, and memory for a domain is encrypted with a domain-specific set of keys. Furthermore, the hypervisor maintains a cache of decrypted pages, to speed up memory accesses in cases when the page has been already encrypted. So, while some features of this system are more developed than Overshadow, it does require modification of guest code, which Overshadow managed to avoid.

Protecting against a malicious OS

In a follow up[85] to Overshadow, it is pointed out that many virtualization-based security architectures have focused on isolating applications and domains, protecting memory, and other such services, but have not addressed "OS semantics". The authors highlight that in spite of Overshadow's memory protection, it won't safeguard an application against its own vulnerabilities, nor can it prevent a compromised and malicious OS from posing a serious threat. For example, a malicious OS could grant multiple mutexes simultaneously, or simply refuse to schedule a process, or carry out other nefarious activities that render applications useless. Therefore, the authors suggest and motivate more developed system components that expand Overshadow's model and take more aspects of security-critical functionality out of the hands of the OS, protecting applications at the level of OS semantics.

I/O Security

BitVisor[92] is a thin hypervisor system that provides I/O security for a single guest OS. It relies on modern virtualization hardware support (Intel VT or AMD-V). For example, it uses IOMMU functionality to protect against DMA attacks, and I/O instruction trapping bitmaps to configure which devices' instructions will trap to the hypervisor. It implements its services via what it terms *parapass-through* drivers – drivers that can be substantially smaller than usual device drivers, since they only need to handle a small subset of normal driver functionality, namely the control and data instructions. Handling the control instructions enables BitVisor to observe device state, and handling the data instructions enables it to perform security operations on the data. Such a parapass-through driver resides in the hypervisor layer. Most I/O instructions pass through the driver directly to the hardware, but the control and data instructions are specially handled. A test system was implemented using an ATA parapass-through driver to perform encryption of stored data, a service that could be provided regardless of the guest OS.

Componentization

The Nizza system[54] is based on a L4-microkernel variant and provides a way to decompose an operating system and its applications into critical/secure and non-critical components, reducing the TCB for applications and even removing the OS from the TCB. Security critical components, such as for sealed storage, cryptography, and ensuring application isolation within a GUI, are run as microkernel applications. These components are loaded by an additional "loader" microkernel application. The guest OS may be paravirtualized to run on the microkernel, or may run above a VM provisioning layer. Applications and the guest OS then rely on the isolated, minimized microkernel components for secure functionality. They connect to these services using the IPC interface exposed by the microkernel.

The Nizza system does require potentially extensive modification to guest software, but presents a compelling method of drastically reducing application TCB. In comparison to the Nizza system, Overshadow attempts a similar (albeit lesser) goal without requiring guest modification, which leads to more performance and implementation challenges on the VMM side.

Mandatory Access Control

Mandatory Access Control (MAC) policies such as Bell LaPadula, the Biba integrity model, and the Chinese Wall model[12] can offer stronger security for critical applications. With hypervisor-based MAC, the benefits of MAC can be brought to existing systems and architectures, and enable greater security for virtual domains. The sHype system[87] brings MAC to the Xen hypervisor. Its granularity operates at the level of VMs and the shared VM resources (event channels and shared memory) used by Xen guest device drivers, enabling the mentioned MAC policies and others to be applied to domains and their interactions. This can facilitate a secure VM coalition as earlier described, where domains cooperate securely to achieve the system goals.

The Xen Security Modules project[37] is still developing, and attempts to modularize the application of MAC and other services for Xen. It provides a common framework whereby different security services and models can be used depending on the situation. For instance, it supports both sHype and Flask[98] modules.

Instruction set virtualization

The *Secure Virtual Architecture* (SVA) system[39] presents an interesting design where a hypervisor layer exports a type safe instruction set interface for carrying out all the activities in the system. The interface is divided into SVA-Core (which includes all instructions for typical computation, including logic, arithmetic, memory allocation, function calls, branching, and other instructions) and SVA-OS (consisting of privileged OS-only operations such as I/O and MMU configuration that are typically implemented in assembler). All virtual machines must use this interface. Operating systems that run in a virtual machine will have to be ported in three steps:

1. Port the platform-dependent portions of the kernel, including all assembly code, to use the SVA interface. The authors argue that this is acceptable as a typical step for porting an OS, and furthermore may be easier for SVA, since SVA's interface is higher level and more abstract than typical ISAs.

2. Make certain documented, specific changes to kernel memory allocators.

3. Optional modifications to the kernel to improve SVA performance.

Applications, on the other hand, typically need only be recompiled to take advantage of the secure SVA-Core interface.

The system uses a "safety checking compiler" to compile guest code to produce SVA *bytecode*, whose safety properties are then checked at load time by a Java-reminiscent "bytecode verifier". This process can occur offline, combined with digital signatures to authenticate the verification. A runtime translator converts the bytecode into native machine instructions.

Since SVA can manage all critical system operations via its type safe interface, it can provide security guarantees for the guest systems (even though the guest kernel is probably written in C), including control flow integrity, type safety for certain types of objects, array bounds safety, no dereferences of uninitialized pointers, and no double frees, among others.

In a sense, SVA is like "Java for operating systems" in that safety guarantees are enforced and software isolated by a virtualization layer – but it is quite interesting to consider how this system facilitates bringing such guarantees to legacy systems implemented in unsafe languages with arguably reasonable porting cost. It in effect creates a new interface layer between the ISA and the ABI.

2.7.6 Monitoring-based services

Now we shall discuss some monitoring-based services presented in research. Monitoring services also leverage the hypervisor's high privilege, but focus more on observing, interpreting, and possibly responding to guest state. Monitoring services may operate at a higher level of abstraction than isolation services, and require knowledge and interpretation of higher level guest OS abstractions.

Attestation

Hypervisors, in their high-privilege position, can be used to attest to guest code integrity and state. This, of course, aligns with the Trusted Computing Group (TCG) and their architectures for remote attestation. An emerging potential area for attestation is on mobile devices; the TCG has relased a mobile platform specification[103], and virtualization may possibly be used to fulfill this specification. While SELinux has already been used to do so[1, 126], to our knowledge virtualization has not. Discussion on utilizing ARM TrustZone technology to facilitate Trusted Computing is found in [122].

An early VMM-based system for attestation was Terra[45]. Terra, using a "Trusted VMM" coupled with a management VM, supports open and closed box domains, sealed storage, and remote code attestation for domains. If a domain is designated as closed-box, Terra gives it stronger isolation – in addition to standard memory isolation, it will provide privacy and integrity protection for stored data, thus sealing it off from observers. Closed-box domains can't even be examined by the system owner. A closed box domain can approximate a proprietary closed box system such as a hardware component or custom embedded system. Suggested examples of such systems are game consoles, ATMs and mobile phones. Terra was implemented using VMWare GSX Server, with a management VM that is charged with allocating resources (memory, disk, devices) as well as setting up connections between VMs. It is remarked that, as with Overshadow, a higher assurance VMM could be used in production environments. Due to Terra's support for closed-box domains and sealed storage, it could be argued that it also provides isolation-based services, but it was placed in the monitoring section due to its attestation and trusted computing emphasis.

Malware analysis

Numerous virtualization-based systems for malware analysis have been presented. Two examples will be discussed.

Firstly, the Patagonix system[70] is interesting because it attempts to dispense with the semantic gap in a unique way. It tracks code execution by using generic hardware mechanisms that remain consistent independent of any OS differences. By setting the non-executable (NX) bit on all pages, any code execution traps to the hypervisor, whereupon the page can be inspected. (Code need only be inspected when it first runs, or after it is modified.) Hardware-stored data such as addresses of page tables themselves is used to differentiate between execution contexts. The system uses a database of known good binaries (including Windows and Linux kernel binaries) to check the identity of executing code. This database is the only aspect of the system that is OS-dependent, and since it is decoupled from the implementation of the system (and it is arguably much easier to acquire system binaries than to implement system-dependent logic), the system's genericity and convenience is maintained. The results of the identity checking are sent to the user, who can compare Patagonix's report on currently executing code with the report issued by the OS itself, and thereby detect covert executions like rootkits. The system successfully detected all rootkits tested on it. So long as a sufficient database of known-good binaries for the guest in question is available, the system can support any guest.

Another system[61] offers broader malware detection support, but is more heavily dependent on VM introspection. It uses VM introspection and semantic reconstruction to capture the relevant state of an observed system (files, processes, etc.). This state can be compared with the state reported by the operating system to detect discrepancies. The semantic reconstruction facilities also enable the system to run existing malware detection utilities *externally* on a VM, potentially even facilitating the use of utilities written for one platform to scan a different platform. The system can be run on multiple VMMs, including Xen, VMware, UserMode Linux and QEMU.

Intrusion detection

Another natural virtualization monitoring service is intrusion detection. The previously introduced Livewire system[46] was the seminal use of VM introspection. It consists of a management VM, running both a policy engine and a semantic reconstruction component that used standard crash dump utilities on guest pages to analyze system state. A later system, Introvirt[62], supports an interesting feature whereby exploit-specific predicates (possibly written by a software patch author) can be used to provide perfect detection of the occurrence of the exploit. To bridge the semantic gap between predicates and guest software, and enable predicates to be highly expressive, the system can execute existing guest code (such as system calls or application functions) in the guest address space. To prevent modification to guest state as a result of executing the guest code, the system supports rollback functionality.

Forensics

Virtualization-based forensics services enable new possibilities for live forensics analysis. While offline analysis can accommodate many forensics applications, volatile and dynamic system state can only be obtained via live analysis of a running system under attack. Traditionally however, live analysis presents difficulties since the presence of the forensics investigator might be easily discerned by an attacker, and other aspects of system state may be affected by the investigator's presence. In the previously cited system using the VIX toolkit[50], safe live analysis is enabled via virtual machine isolation and introspection. The system runs in a Xen administrative domain, and data is therefore gathered externally to the monitored user VM. While the authors hope the system is undetectable, they acknowledge that using timing/performance analysis or other similar circumstantial techniques an attacker *may* be able to conclude that the system is being monitored. It has been suggested that running such a forensics system on its own core in a multicore system might lessen the potential for timing analysis, but it may still be necessary to "freeze" the monitored system in certain moments to gather state information.

Execution logging and replay

Another apt and canonical use of virtualization's monitoring possibilities is to log and replay VM execution. The ReVirt system[41] enables complete logging and replay of VM execution, and since it is VMM-based, the logging will persist in periods before, during, and after guest attacks. Then, if an attack is discovered, the incident can be replayed in exactitude to ascertain its source, cause, effects, and so on. It can also be used to generally audit system activities. ReVirt can naturally enhance or be combined with intrusion detection, malware analysis and forensics services.

To reconstruct execution completely, instruction by instruction, ReVirt must log all non-deterministic events and data, and it does so with reasonable performance. Non-deterministic events that must be logged include device input and system interrupts – fortunately, such events can be handled by the VMM.

SMP-ReVirt[42] brings the same complete logging and replay functionality to multiprocessor systems, and must deal with such challenges as shared memory (since the order of operations on such memory by different cores must be preserved), which can introduce significant performance overhead over single-processor ReVirt.

2.7.7 Alternatives

What other alternatives are out there for implementing security services in a way that is isolated from yet with high-privilege visibility into the monitored system? We have already mentioned Flask/SELinux as possible alternatives ([1, 126]), although we also saw with Xen Security Modules[37] that Flask may *complement* rather than supplant virtualization.

Another possibility is enforcing security via FPGAs. [30] proposes a solution where a FPGA is used to enforce a configurable security policy in a high-performance hardware-based manner. Other dedicated hardware security modules may be able to offer specific high-assurance security services, such as storage or I/O encryption modules (as in the venerable BLACKER [115]), tamperproof smart cards for a variety of cryptography and authentication applications, or TPMs[121] for sealed storage, attestation, and other uses. Of course, any of these hardware solutions could also be combined with virtualization.

2.8 Summary

This chapter has introduced virtualization, covered in depth some important principles behind virtualization, and shown different virtualization trends and types. It has examined different types of hypervisors, typical virtualization use cases, and general advantages that can be obtained with virtualization. The chapter also takes a good look at modern virtualization hardware support, and a quite thorough examination of security architectures that leverage virtualization – including investigating the benefits, challenges, and limitations of virtualization for security, and a study of security services implemented in research. The chapter shows foremost that virtualization is a powerful and relevant technology, and a viable means by which to enhance system security in a variety of ways. It also demonstrates that *hardware support* is truly critical and a great aid for virtualization and security.

3. MULTICORE AND EMBEDDED SYSTEMS

In this chapter, we will discuss the characteristics of embedded systems and multicore systems, and their relation to virtualization.

3.1 Embedded systems

An overview of embedded systems in relation to virtualization, including a discussion of traditional traits and incoming trends in embedded systems, is found in [51]. Some of the main points will be summed up in the following two subsections 3.1.1 and 3.1.2.

3.1.1 Traditional characteristics

Embedded systems are traditionally known by a number of traits. Firstly, they are resource constrained, having limits on power, CPU performance, memory, and/or any other such limitations. They may have real-time needs, in which certain performance guarantees must be met to fulfill the responsibilities of the system. They have typically been hardware-centric, with more functionality in hardware than in software, and whatever software is present has often been of minimal complexity. Embedded systems are traditionally closed and proprietary, perhaps based on a custom hardware and software setup catered specifically to the system at hand. The software portions are often static, unlike frequently updated desktop computer systems. And another ruling principle is heterogeneity – due to the broad number of tailored embedded applications and systems, and their proprietary, custom nature, the embedded world is populated with a great variety of hardware architectures and systems overall.

Embedded systems can clearly have very different requirements and attributes than desktop or other systems. They may also have special robustness, security, and tamperproofing needs, and may require special considerations in their communications and interfaces with other systems.

3.1.2 Emerging trends

While some of their traditional characteristics may still hold, embedded systems are changing rapidly in many ways. For instance, their software is increasing in size and complexity, with millions of lines of code running in a smartphone and gigabytes of software running in a car. Embedded systems are supporting higher level development and traditional APIs, with developers with no previous embedded experience authoring for embedded platforms (such as the iPhone). From closed, proprietary systems, there is a growing trend to support open and/or mainstream application-friendly OSes. The static nature of the software stack is being replaced by a highly dynamic and complex picture. And, the ubiquity of embedded systems and their use in critical situations are increasing. Therefore, the picture of embedded systems is changing, introducing new challenges (including security challenges).

3.2 Virtualization and embedded systems

Virtualization, with its isolation and security benefits as well as its management-oriented advantages, promises to bring great dividends to embedded systems, addressing traditional and emerging needs.

3.2.1 Existing platforms

Embedded platforms are still lagging behind $x86$ platforms when it comes to in-built virtualization support. The TrustZone architecture described earlier is the only embedded system we know of that approaches a level of virtualization support. However, this dearth of support has not stopped development of embedded virtualization platforms.

Xen-ARM

The Xen-ARM project[123] brings the Xen hypervisor to the ARM architecture. Initially described in [55], the implementation had to deal with several challenges and limitations of the ARM architecture. For instance, the ARM architecture only supported two privilege levels, so the lower user level was logically separated into two privilege levels for the guest OS and applications. Memory protection for the 3 required privilege modes (hypervisor, kernel, user) was achieved by using 3 of ARM's 16 available memory protection domains. Since the ARM TLB didn't support tagging when switching between address spaces, the system was specially made to at least allow context switches vertically in the stack (from guest application to guest kernel to hypervisor) without necessitating a TLB flush. Additionally, the implementers attempted to make intelligent and resourceful use of TLB lockdown entries that persist across a flush, allowing oft-used hypervisor pages to stay in the TLB. Finally, to support virtualization requirements and maintain performance, the system follows in traditional Xen footsteps by requiring paravirtualization of the guest OS, which can simplify many aspects including memory management.

L4 Microkernel

The L4 microkernel family[104] has also been ported to embedded platforms such as ARM. As discussed, it can provide a base for virtualization support.

3.2.2 Companies

As further evidence that embedded virtualization is taking off, several companies are now competing in the space.

First off, VMWare is planning offerings in the mobile virtualization area[112], fueled by their purchase of Trango[10]. However, no concrete information is yet available.

VirtualLogix[107] offers a traditional hypervisor-based virtualization platform for embedded and mobile systems. They apparently aim to support real-time embedded OS requirements, and their website advertises their virtualization platform as a strong way to secure a trusted system OS from an open application OS. Detailed information on their closed platform is, however, hard to come by.

Open Kernel Labs[76], on the other hand, base their products on open source L4 microkernels, and encourage open development. Via the L4 microkernel, they support virtualization, and also aim for interesting Nizza-like architectures based on "hypercells" that enable the partitioning of many different system resources into separate domains[75].

In a repeat of the microkernel vs. VMM debate waged in [49, 53], Open Kernel Labs and VirtualLogix have argued over the merit of their respective approaches to virtualization. A paper by VirtualLogix associates[20] criticized microkernels and presented negative performance test results, while in a multi-part blog response Gernot Heiser of Open Kernel Labs decried the paper's conclusions and methods[52]. While clearly about angling for competitive advantage, these exchanges emphasize that there are differences between microkernels and hypervisors, with different benefits and drawbacks, that must be considered in context. For instance, microkernels may be much smaller than a hypervisor, and thus higher assurance, but using a microkernel may require more extensive paravirtualization of guest software, and leveraging more potential microkernel benefits such as finely partitioning system functionality may require even more software modification. Additionally, hypervisors are closely tied to the hardware, and present a hardware-like interface to guests, whereas microkernels present a small number of low level OS abstractions. This means that microkernels may duplicate some functionality that will be implemented again by guest OSs, and that a hypervisor may present a more familiar interface to guests, but it also means that architectures built atop microkernels might have the potential to be more platform or hardware independent.

3.2.3 Applications

Numerous applications for embedded virtualization have been suggested and/or implemented.

In [51], Heiser suggests the following potential uses:

- Supporting heterogeneous operating system environments. For instance, a device might combine a proprietary, legacy, real-time software stack of millions of lines of code with a similarly large application OS like Windows, Linux, or Symbian. The VMM can ensure real-time priority to the legacy stack, and lesser priority to the application OS.
- In the fast-changing world of embedded hardware, virtualization can abstract hardware architecture, and also enable software to move between multicore and single core platforms.
- Of special note is virtualization's potential to abstract and manage forthcoming many-core hardware platforms, such as Tilera's Tile64 [102].
- Of course, virtualization can support security in embedded systems as it can on other systems, and can especially apply in the previously given

example of partitioning embedded/mobile systems into domains to protect the various system stakeholders (operator, user, protected content owner). With these new scenarios where components cooperate but must protect the interests of various parties, virtualization can provide such protection.

- Supporting new methods of application distribution. Via virtualization, an entire specialized software stack can be distributed as one package, and run in either a virtual machine or as a microkernel application. For instance, a multitasking real time application using a minimal FreeRTOS scheduler[44] or something similar can be distributed as a self-contained binary to be run in a VM.

In [38], the authors likewise suggest a number of potential applications:

- System security, as already discussed.
- Security services, including for instance the isolation and monitoring services described earlier.
- Mobile testbeds, where a VM conducting experimentation or otherwise processing test data is run in the background on a virtualization-capable device. This allows researchers to take advantage of mobile networks and environments in their testing, and the research VM will be isolated from other device domains, but it still requires some thought as to protecting the privacy of the device owner should the research VM process data such as phone location or other revealing information.
- Opportunistic sensor networks, where a sensor VM migrates to supporting nearby mobile/embedded devices. This again requires thought as to how to protect the user's privacy, and also to prevent the opportunistic migration path from becoming a malware or otherwise malicious vector.

In [29], the authors discuss similar potential applications, notably including system security and the potential to safely support new and popular open mobile operating systems alongside other closed stacks and modules. They also highlight the potential for virtualization to ease embedded development processes, quickening time to market by abstracting the heterogeneity of embedded platforms and enabling developers to focus on a reduced common environment.

Recommendations from the Open and Secure Terminal Initiative (OSTI)[59] and Open Mobile Terminal Platform (OMTP) project[78, 77] also delve into the system security potential for embedded virtualization, focusing on isolation, recommending architectures where the legacy/operator OS is isolated from the open/application OS. The fact that large industry players, such as Intel behind OSTI and AT&T, Vodafone, and Orange as members of OMTP, are supporting embedded virtualization is further evidence of its arrival in the market and the industry.

Finally, as an example of a specific security service, a multilevel MAC-based architecture for embedded virtualization is described in [69]. The system is built on Xen-ARM, and is based on the aforementioned sHype system. Furthermore, it attempted to deal with specific embedded considerations, such as the need for minimal performance overhead, the potential for a malicious domain to cause denial of service by draining device power and resources, the potential for malicious DRM tampering, and the need for protected mobile financial services.

There are undoubtedly more potential applications waiting to be explored. Virtualization can provide solutions to deal with traditional embedded concerns, such as need for securely protected systems, different proprietary software stacks, heterogeneous hardare, and real time processing, and can also deal with emerging embedded systems needs, such as newer security concerns, needs for multi-stakeholder systems, securing systems that require open OSs, enabling dynamic and flexible software distribution, and abstracting the complex and evolving hardware scene (especially in the context of multicore and many-core). Questions include how virtualization hardware support can and should be expanded for embedded systems, how virtualization platforms can maintain adequate performance on embedded systems, what virtualization approaches (paravirtualization, binary translation, previrtualization, hardware support) and design patterns are most apt, how virtualization can best support embedded multicore, and what virtualization-based security services can and should be supported on embedded systems.

3.3 Multicore systems

3.3.1 Why multicore?

An excellent overview of multicore hardware today, including hardware and software concerns and challenges, is found in [27]. Additionally, [83] also discusses contemporary multicore developments, and furthermore merges the discussion with a description of virtualization concerns and opportunities on multicore.

As noted in [27, 83] and elsewhere, the advent of ubiquitous multicore is due to the megahertz plateau in CPU development. Heat and power consumption curves increase beyond tractable levels when CPU clock speeds are pushed beyond their current, leveling-off capabilities. New methods were needed to increase performance, among them the following (as described in [27, 83]:

- Increase the L2 cache size. The benefits of this strategy can only be as great as the losses due to L2 cache misses, which vary from context to context. Note that increasing L1 cache size is not recommended, since making the L1 cache too large would have a negative impact on clock frequency.

- Exploit *instruction level parallelism* (ILP) by having the CPU execute parallelizable instructions simultaneously. The benefits of this technique are limited by the inherenet parallelism in the instruction set and the executing program, and it must be balanced with the resultant complexity in hardware needed to detect and exploit ILP.

- Increase use of pipelining, wherein multiple instructions are piped through the different stages of an execution cycle one after the other, so that overall throughput is increased. Multiple instructions can be active in different stages of processing, instead of the processor having to complete the execution of an instruction before starting a new one. However, this approach can increase processor complexity, as well as increase the time for a single instruction to be processed.

- *Simultaneous multithreading* (SMT), also called *Hyperthreading* on Intel platforms, where a single core with multiple functional units can execute multiple threads simultaneously.
- Multicore CPUs, where multiple cores are located on a single chip.

All these techniques have of course been used. This thesis focuses on the issues and opportunities arising from multicore hardware (which often includes use of SMT, ILP, pipelining, and increased cache sizes).

In particular, also noted by [27, 83] and elsewhere, multicore CPUs create challenges for both software and hardware. The dominant Von Neumann hardware architecture, with a uniform memory space accompanied by input, output, and a sequential processing unit, lends itself to single processor systems. The creators of the Barrelfish multicore operating system agree that OS designers, in spite of the considerable differences between multicore and single core hardware, still think of systems in a Von Neumann way (in part due to the continuance of laborious cache coherence mechanisms) – continuing to see a system with a uniform computation and memory architecture[25]. There are many new hardware-related questions that must be addressed in order to create efficient and suitable multicore systems. In addition, common software development models, as an outgrowth of the sequential Von Neumann instruction architecture, are not well suited to parallel programming. Software developers in general do not have the tools or knowledge to leverage parallelism in most types of software, presenting a formidable obstacle to the fruitful use of multicore hardware. The following subsections will discuss these issues.

3.3.2 Hardware considerations

In some situations, multicore hardware might seem to be a simple extension of single core. For example, in a basic dual-core situation, the two cores might have private L1 caches, but share the L2 cache and the communication interfaces. The rest of the system might be the same. However, as system complexity increases, such as in a many-core hardware platform like Tilera's Tile*Pro*64[102], a broad spectrum of issues come to light. A range of hardware concerns in multicore systems is illustrated in [27, section 2.1], and summarized in the following subsections.

Core count and complexity

The number of cores that a system should have is directly related to the parallelism in the expected workload. If performance gain for adding cores is not linear, it is most likely better to focus on increasing the performance capacity of each of a few cores. If on the other hand performance gain for adding cores *is* expected to be linear, then more cores are most welcome. However, here an interesting phenomenon takes hold, where the spatial area of the chip must be considered – performance gains resultant from adding any complexity to the chip must be proportional to the increase in chip spatial area (that is, the gain should be at least as substantial as the area increase), or else the better path is to simply add additional chips. This concern is only the first way in which we will see that physical size and layout affect multicore systems.

Core heterogeneity

Cores in a multicore system may be homogeneous (identical) or heterogeneous by design. There may also be a distinction where cores implement the same instruction set, but have differing assemblies of functional units or other components. For generic, non-specialized workloads, fully homogeneous cores (as found in Intel or Tilera processors) are advisable. However, in specialized cases where the workload is expected to have characteristics appropriate to multiple architectures, heterogeneity may be beneficial. The Cell processor[120] is an example in which some cores use different instruction sets than others. A common pattern in large heterogeneous core systems is to have a small number of high performance cores that execute generic, non-parallelizable workloads, and a large number of small cores usable for highly parallel workloads. It must be noted that core heterogeneity can greatly complicate software development, and taking full advantage of the available core palette can be challenging.

Core heterogeneity in general is more common in embedded systems than desktop systems.

Memory hierarchy

Memory hierarchy becomes considerably more complicated in a multicore scenario. Cores may have internal memory for their own use, and they typically still have a private L1 cache. But should cores share an L2 cache, or have private L2 caches as well? Should they share an L3 cache? How many cores should share each cache? Shared caches can result in better utilization of hardware, and may create performance gains in situations where cores are sharing loads or in other such circumstances, but sharing requires more costly external (off-core) communication, and may hurt performance in other scenarios. It also decreases inherent isolation between cores (which may be a security or reliability concern). Additionally, the less cache sharing, the more complex the coherency maintenance mechanisms must be – if each core has a fully private, multi-megabyte L2 cache, and the system has many cores, maintaining coherency can be daunting. On the other hand, sharing a cache between too many cores also becomes complicated and costly. The problems will only increase with the number of cores.

The Tile64 is an example of a multicore CPU where each CPU in an eight by eight mesh has its own L2 cache, and the chip even supports an additional “dynamic distributed cache” (DDC) comprising the caches of a core’s neighbors [102].

Interconnects (core communication)

Cores in a multicore system need to communicate with each other. A primary reason is to support cache coherency. We will not go in depth into the various possibilities for core interconnection (such as crossbars, rings, meshes, and hierarchies) here, but as with other aspects, the challenges of core interconnection increase with the number of cores, and physical layout of the cores can become an important consideration.

Extended instruction sets

The $x86$ instruction set is firmly in place, and isn't going anywhere[83]; hence, though $x86$ may not have been intended to support multicore from the beginning, it is necessary now to use expanded instructions that *can* support multicore. In general, if an ISA must continue to be used on multicore hardware, it may be necessary to upgrade it with special instructions to support multicore operations, especially specific instructions relevant to implementing shared/transactional memory[27, section 2.3.1] or low-latency message passing. For instance, memory shuffling instructions that can atomically read a location value and set a new value based on a test predicate can be useful for synchronization.

Other concerns

Other issues, including how the main system memory will be laid out and interface with the cores, and maintain sufficient bandwidth to the cores, as well as how many simultaneous threads to support on each core, are other important concerns with their own tradeoffs. For instance, supporting more simultaneous threads on a core can increase the number of cache misses (since multiple threads compete for the cache), but can overall increase performance and utilization since the core's processing components will be used by other threads when a thread must wait for a cache miss to be filled. Regarding memory interfaces, in some cases with large numbers of cores, it may even become beneficial to forego the traditional strategy of having external interfaces along the periphery of the chip and instead stack chips in a 3D manner[71].

3.3.3 Software considerations

Some would say that software is at the heart of the multicore problem, since all the advanced hardware in the world isn't going to help if software isn't written to utilize multicore capabilities. Software concerns in multicore systems are discussed in [27, section 2.2], and summarized in the followikng subsections.

Programming models

The dominant imperative programming model, where instruction after instruction, function after function are executed in sequence without easy support for concurrent programming and synchronization and safe sharing of data, must be evolved to suport multicore. But concurrency and synchronization are not simple tasks – for instance, concurrency vulnerabilities have been discovered in system call wrappers (system call interposition layers/reference monitors intended to support security) due to improper synchronization between the wrappers and the system calls, among other causes[114].

A general strategy for how to handle interprocess (inter-core) cooperation and concurrent programming must be settled on. The fundamental mechanism can be something along the lines of shared memory, where cores synchronize and share access to regions of memory, or message passing. Message passing may be more useful in situations where cores are more widely distributed and do not have easy access to shared physical memory. If software such as the OS kernel is to run on multiple cores, special care must be taken when synchronizing its data.

Firstly, though, one must note that programming may indeed proceed using the standard sequential model, should a compiler be available that can automatically extract parallelism. However, the parallelism to be found in common programs may be quite minimal, not to mention difficult for a compiler to discover and articulate. Therefore, it is most likely needed to proceed with other approaches.

There are many programming models available to support concurrency and parallelism. The dominant model is kernel threads (such as *pthreads* on Unix). Kernel threads are supported by the OS, and hence are expensive to create and destroy. They may need to synchronize with other threads using mutexes or other synchronization primitives or strategies. Kernel threads are a low level primitive, and thus suitable for expert implmentation – including implementation of additional higher-level programming models.

User-level threads, as opposed to kernel threads, are created and managed by user-level processes. This can make them less expensive than kernel threads. However, they are far less common than kernel threads.

In the *single-program, multiple data* (SPMD) model, the program is meant to be run identically in multiple threads on multiple collections of data. This model could be seen as a master with worker threads, where the master sends data to a force of identical workers who operate on the data in parallel. It may be that the parallel workers collectively contribute to a greater result, requiring concurrent operation. OpenMP is a programming language extension that was originally implemented to support this model[40].

The *task* programming model is slightly different, in that a task is an independent unit of work that may be executed in parallel, but doesn't have to be. Cilk is a task-oriented extension to the C programming language[11].

Domain-specific languages, as opposed to generic languages like C, C++, and Java, may provide a deft approach to extracting parallelism from a workload, in that the specific parallelizable qualities of the workload can be brought out and facilitated by the language.

Although there are clearly many alternatives for paralell programming, most development uses kernel threads (and that only minimally), and the overall mentality of most software development is, understandbly, grounded in the sequential model.

Programming tools

Programming tools, including languages and debugging support, must meet the challenge of multicore, multithreaded development.

Debugging of course becomes instantly more complex if there are multiple execution contexts in a program. Concurrent programming gives rise to nondeterminism as well as new error classes such as deadlock. Debuggers meant for single-threaded development may be insufficient to deal with such complexity, and programmers used to single-threaded development may not know how to debug multithreaded programs.

Programming languages need to evolve to support multithreaded, multicore-friendly development. As mentioned, extensions such as OpenMP and Cilk provide high-level mechanisms for leveraging multicore parallelization. A difficulty here is that fundamental change of programming languages and models takes significant time, and multicore hardware, unfortunately, is being introduced into a legacy world filled with single-threaded code and mentality.

Locality

The easy accessibility of memory to cores, whether from caches or system memory, is essential for performance. The more that cores can be made to reference locally accessible memory, the higher performance that can be attained. Different strategies can increase locality within a specific core, or within an entire chip, with different tradeoffs. For example, if memory locality can be increased within a chip, this might result in more communication between cores within the chip as they share their caches, but less communication off the chip, the latter type of communication being more expensive. Multicore systems in the future seem to be heading towards more Non-Uniform Memory Architecture (NUMA)-like architectures, where physical memory is more closely associated with individual multicore CPUs, in an effort to enhance locality[83].

Load-balancing and scheduling

Scheduling of threads (including when and how often they are scheduled and how they are distributed among cores) is clearly an important challenge in multicore. Different scheduling policies can greatly influence system performance and properties, including (of course) locality. Scheduling must also be considered in higher level models like tasks, where threads are but an underlying entity.

3.3.4 Interesting multicore architectures

The Barrelfish multikernel

The Barrelfish operating system[89, 25, 24] is intended to deal with both increasing system heterogeneity and the distributed nature of multicore hardware. The authors argue that OSs are still being developed as if they are to be run on uniform CPU and memory architectures, but they need to be rewritten to function well on, take advantage of and scale on new multicore hardware. Additionally, with continual rapid, dynamic shifts in hardware technologies, increasing core counts, and massive amounts of variety present in cores, devices, memory hierarchies, core interconnects, and other hardware aspects, it is difficult for designers to optimize for certain system configurations. Greater flexibility and management of diversity is required. To achieve this, Barrelfish acknowledges modern computer systems as networked enviornments in their own right and attempts to integrate distributed systems lessons in supporting dynamic, diverse, adaptable, scalable systems. Introducing the concept of a *multikernel*[24], Barrelfish treats cores as indepedent, isolated, distributed entities, capable of running independent software stacks and communicating with each other via message passing/IPC. It is capable of managing heterogeneous cores. The authors argue that handling shared state via message passing is less expensive than using shared memory, and that by making the OS implementation as independent as possible from the specifics of hardware implementation, it can remain easily adaptable and scalable to new architectures. (Only the message passing mechanisms and the device- and CPU- specific interfaces are tailored to specific hardware.)

In a multikernel model, each core is intended to be a truly independent entity. In the Barrelfish multikernel, each core has its own independent *CPU driver* running in privileged mode. CPU drivers share no state with each other. This minimal driver is non-preemptible, and processes traps and interrupts in serial. It does not perform inter-core communication. A user-level *monitor* process also runs on each core, and is responsible for communicating with other cores and the system and maintaining its own copies of any global system state. Processes in Barrelfish are unconventionally implemented as a collection of "dispatcher" objects. A process has dispatchers situated on each core upon which it might execute. The CPU drivers schedule the dispatchers, and then a dispatcher runs its own user-level thread scheduling on its own core.

Configurable isolation

With cores sharing components such as caches, core interconnects, and external communication interfaces, multicore hardware presents the potential for isolation problems. This may result in security issues, where state leaks between execution contexts. It may also result in reliability issues, since a failure in one core (or its components) may cascade into a failure in another core. Furthermore, as hardware feature size and the space between components decreases, the likelihood for hardware failure increases[9, 116], meaning that in today's chips there is more risk for such hardware failures. Therefore, in [9], the authors propose a *configurable isolation* model where cores can be configured as fully isolated from each other and not sharing any unnecessary components in critical scenarios, or can be allowed to share resources in the usual way in common scenarios. The authors point out the need for fault isolation, detection, and repair, and argue that such a configurable isolation system would support scenarios where either speed or reliability are important concerns.

Mixed-Mode Multicore reliability

For the same reasons, the authors in [116] provide a system for making flexible use of *dual-modular redundancy* (DMR), in which a process is run simultaenously on multiple cores in order to achieve greater robustness. The contribution of the research is Mixed-Mode Multicore reliability (MMM). On traditional DMR systems, everything runs in DMR mode, which can be expensive. Under MMM, only critical processes are run in DMR mode, while non-critical processes can execute normally. As with the configurable isolation proposal, this system enables users to take advantage of robustness or performance, depending on the needs of the situation.

3.4 Multicore and virtualization

As mentioned, a discussion of multicore and virtualization can be found in [83]. The article highlights how virtualization provides a promising path for scalablitiy – indeed, if software cannot generally be adapted to parallel models, then at least utilization of multicore hardware can be achieved by housing multiple independent systems on it. It also emphasizes that virtualization is good for locality – if a virtual machine is assigned to a single chip or core, then its locality to that chip or core will increase. Finally, it mentions that I/O requirements will be more complex and critical in a multicore virtualization scenario, requiring channel and device assignments for cores and VMs. Fortunately, I/O hardware support (Intel VT-d and AMD-IOMMU) seems to be rising to the challenge.

Overall, virtualization seems like an apt way to leverage multicore hardware. Multiple VMs can be hosted on a system, and users may benefit from adopting a new architectural perspective, where they divide their system into categorized and trusted/untrusted domains (for example, one domain for financial applications, one for games, one for office work, etc.). It has already been mentioned how virtualization can abstract hardware differences, and thus facilitiate smoother transitions between platforms as hardware evolves. It appears a big win, so to speak.

However, before jumping ahead, we must consider some important issues. First, how does this affect the security of the system? With more complex hardware and software, is it more likely that the system will host critical vulnerabilities? Will the potential isolation and reliability difficulties in multicore engender security liabilities? Furthermore, hypervisors must be explicitly designed (and made more complex) to support multicore. These changes may make the hypervisor – the base of the system TCB – harder to verify. For example, the seL4 microkernel, as we saw, is formally verified, but only for a single core environment! What kind of multicore support does a particular hypervisor offer? How good is it at promoting fair and efficient scheduling, and locality? One can't sidestep these issues when looking to multicore virtualization.

3.4.1 Multicore virtualization architectures

There are a number of architectures in research that are specifically intended to enhance system potential via multicore and virtualization. In this section we wil discuss two of them.

Managing dynamic heterogeneity

For example, the authors of [117] (who were also behind the MMM system above) propose a system addressing an interesting problem. They point out that even if a multicore system has physically homogeneous cores, those cores can exhibit widely varying runtime characteristics, making them in effect heterogeneous. These characteristics can include thermal state, other hardware strain, cache and TLB contents, and potentially other aspects, and altogether this runtime heterogeneity can have a sizble impact on performance. The proposed system is a thin hypervisor meant to run directly on the hardware and abstract and manage this multicore *dynamic heterogeneity*, and thereby increase overall system performance. The hypervisor can support different nummbers of virtual cores than there are real cores, and can be run below a guest OS or a traditional hypervisor and thereby manage the heterogeneity for virtual machines. The virtualization layer can also support MMM.

Sidecore

Another interesting architecture is *Sidecore*[66], whose authors make the observation that VM entries and exits are expensive even with hardware support, and offer a solution to this problem that leverages multicore hardware. They describe a system whereby the VMM functionality is partitioned and partially assigned to specific cores in a multicore system. Then, those cores (termed *sidecores*) will always run in VMM mode, thereby removing the need for VM entries and exits for those cores. By using *sidecalls* for certain tasks, guest VMs or system devices can communicate with the sidecores, rather than perform costly VM entries and exits to enter VMM mode themselves. The paper includes experimental results highlighting the performance advantages of implementing an operation via sidecalls instead of typical VM entries and exits.

The authors also cite many other supporting influences that facilitate or justify this sort of architecture. For instance, it is reasonably argued that having cores specialize on portions of the VMM will increase locality. They also suggest that, as in [65], assigning certain functionality to specialized heterogeneous cores can increase performance, and that assigning cores will simplify and enhance scalability for I/O in multicore virtualization systems. Finally, they cite evidence that multicore architecture is moving towards high-performance intercore communication[86], as in AMD HyperTransport[6] and Intel QuickPath[58], which will further improve the inter-core communication latency of sidecore-inspired architectures.

3.5 Embedded multicore systems

As one might expect, the multicore embedded market is set for large growth, but embedded software must be rewritten to take advantage of multicore hardware[34]. There are many multicore embedded architectures available, including ARM[1], MIPS[2], and Freescale (Power)[3], in addition to the previsouly mentioned Tilera.

3.5.1 Virtualization and embedded multicore

Given that embedded systems are heading to multicore, virtualization can help. As already mentioned in this thesis (and discussed in [51]), virtualization can abstract hardware hetereogeneity and changes, thus enabling embedded software to more easily transition to new multicore platforms. Virtualization can also potentially help embedded systems make intelligent use of multicore hardware, assigning system components and balancing load across cores and/or minimizing power usage as dictated by system constraints. In this way legacy embedded software that is ignorant of multicore can still benefit from multicore hardware.

In fact, from a security standpoint, it is critical that embedded hypervisors support multicore and are able to intelligently handle its complexity, so that isolative security architectures and other valuable services can still be supported. While challenging enough to support multicore complexity on desktop systems, hypervisors and security services must deal with embedded system attributes – lack of hardware support for virtualization, limited system resources, wider variation in hardware, and other such issues that complicate implementation.

Regarding embedded multicore hardware support for virtualization, the only player is, again, TrustZone – for example, the ARM Cortex-A9 MPCore can support up to 4 cores, each offering TrustZone features. Some question remains,

[1] `http://www.arm.com/products/CPUs/ARMCortex-A9_MPCore.html`
[2] `http://www.mips.com/products/processors/32-64-bit-cores/mips32-1004k/`
[3] `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4040&fsrch=1`

however, as to how effective TrustZone is for supporting full virtualization with multiple virtual machines. In general, more virtualization hardware support could be helpful for embedded scenarios, at least in generic cases where open or application based OSs and/or non-specific workloads are expected – in extremely particular scenarios, with proprietary software stacks, it may be that para- or pre-virtualization or binary translation offer implementers better solutions than generic hardware support.

An embedded hypervisor stands at a unique position to both facilitate multicore hardware utilization and enforce security, but in so doing it may be forced to bear the brunt of the mutlicore burden, having to support and abstract widely varying and fast changing embedded multicore hardware (potentially without strong virtualization hardware support).

3.6 Summary

This chapter has performed a study of embedded and multicore systems, and their relation to virtualization and security. It has demonstrated that heterogeneous embedded systems are increasingly in need of security, and that virtualization is quickly gaining prominence in the embedded world and is being used to provide embedded security. It has also demonstrated that multicore is on the rise, even in the embedded space, and brings with it a number of interesting challenges and opportunities, especially in the context of virtualization. The chapter shows that when developing on multicore hardware, it is critical to remain cognizant of sundry multicore issues, especially when developing a low level system like a hypervisor. Together with Chapter 2, this chapter motivates the need for further harnessing the use of virtualization in embedded systems – primarily for security, but also as a means to abstract and manage hardware complexity and heterogeneity, and to potentially facilitate utilization of multicore.

4. THIN HYPERVISORS ON ARM ARCHITECTURE

In this thesis, we have expressed an intent to explore and develop possibilities for thin hypervisor-based security architectures on embedded systems. This is clearly an immense research topic that extends far beyond the scope of a master's thesis. However, to provide a good start and foundation for futher research, we intend to implement a basic thin hypervisor on a single core embedded hardware platform, to host a guest operating system above it, and to experiment with different possible security protections provided by the hypervisor. This basic thin hypervisor will provide a platform for implementation and testing, evaluation of constraints and possibilities for embedded thin hypervisors, and assessment of challenges and opportunities for future work. Therefore, this thesis represents a first step in a larger effort.

To facilitate this research, we will ground our efforts in a specific embedded architecture – the ARMv5 architecture – and even a specific CPU, the ARM926EJ-S. While not the newest variant, ARMv5 architecture is still quite common. Moreover, since hardware support for virtualization in embedded systems is scarce (again, to our knowledge, only even partially available in ARM TrustZone processors), it makes sense to focus initial efforts on a platform without such support. Furthermore, the cost of ARM TrustZone CPUs is prohibitive, while ARMv5 architecture CPUs (including the ARM926EJ-S and others) can be easily and flexibly simulated using free Open Virtual Platforms (OVP) simulation tools[79], which enable the construction of simulated platforms containing various CPUs, devices, memory, and busses arranged in flexible configurations. Therefore, we will focus on the ARMv5 architecture, but also offer comparisons with TrustZone to facilitate transition or later applications. Additionally, while in this thesis we will focus on a single processor with a single core, we will offer suggestions based on our experience for expanding thin hypervisor support to multiple processors/cores.

Additionally, for a guest OS, we will use FreeRTOS[44], an extremely small, task-based, widely ported, real-time multitasking kernel for embedded systems. We chose FreeRTOS because of its simplicity and popularity, as well as for its preemptive multitasking scheduler. Using a small, manageable kernel that is less burdensome and complex than a mainstream OS kernel will increase the likelihood of achieving interesting results appropriate for a first-step project.

This chapter will set the stage for implementation by discussing essential foundational issues, and outlining implementation concerns and approaches. To begin with, section 4.1 offers a short description of key points of the ARMv5 architecture. We will also provide a brief comparison with more advanced ARM TrustZone CPUs. Next, we will describe the FreeRTOS kernel in section 4.2. A discussion of concerns and strategies for implementing thin hypervisors in general on the ARMv5 architecture (as well as specifically for FreeRTOS) follows in section 4.3. As in section 4.1, we will briefly cover salient differences in possible approach with a TrustZone CPU. Finally, in section 4.4 we will examine a few potential security services to be supported by embedded thin hypervisors, discussing how such existing virtualization-based security services could be ported and adapted to a thin hypervisor on ARMv5. Likewise, we will identify opportunities and challenges for porting said services to TrustZone CPUs.

4.1 ARMv5 Architecture

The ARMv5 architecture, and specifically the ARM926EJ-S CPU, are described in [15, 94] and [16], respectively. Here we will provide a brief overview of some of the important aspects.

4.1.1 Operating modes

There are 7 operating modes in ARMv5, as described in table 4.1. Each operating mode is entered on different occasions and serves different purposes. User (`usr`) mode is non-privileged, while all other modes are privileged. Access to certain registers and instructions, as well as access to protected memory regions if an MMU is used, requires operation in privileged mode.

Mode	**Mode number**	**Description**
User (`usr`)	`0b10000` (`0x10`)	User execution mode
FIQ (`fiq`)	`0b10001` (`0x11`)	Fast, high priority external interrupt mode
IRQ (`irq`)	`0b10010` (`0x12`)	Normal priority external interrupt mode
Supervisor (`svc`)	`0b10011` (`0x13`)	For software interrupts (system calls) and other OS kernel activities
Abort (`abt`)	`0b10111` (`0x17`)	For memory faults
Undefined (`und`)	`0b11011` (`0x1b`)	For handling undefined instructions (which facilitates emulation, and instruction set expansion)
System (`sys`)	`0b11111` (`0x1f`)	For other privileged OS kernel operations

Tab. 4.1: ARMv5 operating modes

4.1.2 Exceptions and exception handlers

There are also seven types of exceptions, each of which trigger transition to a specific mode. Exception types are described in table 4.2. Reset, FIQ and IRQ interrupts are triggered by signals on external CPU ports. Software interrupts, used for system calls, are trigered by the non-privileged `SWI` instruction. Prefetch and data aborts are triggered by the MMU coprocessor as it checks memory access attempts. Undefined instruction exceptions are triggered by the CPU attempting to execute an unknown or malformed instruction, or a privileged instruction from a user mode.

When an exception occurs, the CPU halts current execution and switches to the appropriate operating mode. The CPU branches to a set location called an "exception vector" that is specific to the exception type. An exception vector is a specific location in memory, part of an "exception vector table", and usually contains an instruction that transfers execution to an exception handler. The exception vector table is usually located at address 0x0 in memory. Each 4 byte entry in the table is an exception vector for one of the seven exception types. The exception vector table should be initialized at system startup. The addresses of the exception vector for each exception type are also shown in table 4.2.

Exception	**Traps to mode**	**Vector address**
Reset (resets the CPU)	`svc`	`0x00000000`
Undefined instruction execution	`und`	`0x00000004`
Software interrupt (via `SWI` instruction)	`svc`	`0x00000008`
Prefetch abort (instruction fetch memory fault)	`abt`	`0x0000000C`
Data abort (data access memory fault)	`abt`	`0x00000010`
IRQ (normal interrupt)	`irq`	`0x00000018`
FIQ (fast, high priority interrupt)	`fiq`	`0x0000001C`

Tab. 4.2: ARMv5 exceptions

4.1.3 Registers

The ARM architecture has 16 general purpose registers, designated `r0 - r15`. However, three of these registers are designated for special usage, as described in table 4.3.

Register	**Special purpose**
`r13`	Stack Pointer (`SP`).
`r14`	Link Register (`LR`). Holds return addresses for branch instructions and exceptions.
`r15`	Accesses the program counter (PC).

Tab. 4.3: ARMv5 "special" general purpose registers.

Some operating modes have their own instances of certain general purpose registers. `fiq` mode has its own `r8 - r14` registers, and `irq`, `svc`, `abt`, and `und` each have their own `r13` and `r14` registers (the `SP` and `LR`). The larger number of private registers for `fiq` mode is to faciliate the processing of extremely fast interrupts that only require the use of `r8` and above. When in any of these modes, only their private versions of the registers are visible. One interesting point to remember is that each mode has its own SP, and should therefore have an independent stack. With these modes "banking" their own private copies of these registers, easier transition and isolation between modes is facilitated. Note that System (`sys`) mode does not have any banked registers; therefore, it always has the same general purpose registers as seen in user mode.

In addition to the general purpose registers, there is a status register known as the `CPSR` or *Current Program Status Register*. In this register are encoded important attributes of system state, including the enabled/disabled status of

IRQ and FIQ interrupts, conditional flags for executing conditional instructions, as well as the current operating mode. Each exception-centric operating mode (that is, `fiq`, `irq`, `svc`, `abt`, and `und`) also has a private `SPSR` (Saved Program Status Register) for saving and restoring the CPSR for the faulting mode.

Even though some registers are banked, thus preserving state for individual modes, it is likely that an exception handler needs to use registers that aren't banked – in which case the non-banked registers need to be saved (typically on the stack). Furthermore, an additional exception triggered while already in the associated operating mode will result in loss of banked state. For example, if the CPU is in IRQ mode and receives another IRQ exception, then the IRQ-specific private registers may be overwritten with new values. One way to deal with this problem is to ensure interrupts are always disabled in the operating mode in question. If that is unacceptable, another way to solve this problem is to write "reentrant" exception handlers, which safely store all state so that interrupts can "pile up."

4.1.4 MMU

The ARMv5 memory management unit in the ARM926EJ-S processor can manage a single address space via page tables and provides simple access control configuration based on domains and permission bits. The MMU contains a TLB capable of caching 64 virtual-physical translations to reduce the need for page table walks.

When the MMU is enabled, the processor interprets each address it sees as a *modified virtual address* (MVA). A MVA is passed to the MMU, which first looks to see if a MVA-to-physical address translation is present in the TLB. If not, the MMU walks the page table (which is stored in system memory) and translates the MVA to the appropriate physical address. The MVA's physical page translation is then stored as an entry in the TLB, possibly evicting another TLB entry.

MMU registers

The MMU, part of the "system control coprocessor" in the ARM926EJ-S CPU, is controlled by a set of coprocessor registers. Coprocessor registers are read from and written to via the `MRC` and `MCR` instructions, respectively. Access to these registers is a privileged operation, and will trigger an Undefined exception if attempted from `usr` mode. A description of the important registers exposed by the coprocessor is found in table 4.4.

Page table structure

The page table in system memory has a simple structure. First, the base table (pointed to by the TTB register) has 4096 entries, each of which is either a *section descriptor* or a *coarse page table descriptor*. A section descriptor describes a simple 1MB region of physical memory, whereas a coarse page table descriptor points to another table containing descriptors for either large (64KB) or small (4KB) physical pages that total 1MB of physical space.

A section descriptor contains the base address for the physical memory section (at 1MB granularity), a 4-bit domain specifier associating the section with

Register	Name	Description
`c1`	Control	Contains bits for enabling the MMU (the M bit), enabling address alignment faults (the A bit), and the S (system) and R (ROM) protection bits which affect MMU access control checks.
`c2`	TTB	The translation table base register stores the base address of the translation table (page table).
`c3`	Domain AC	Contains two bits of access control information for each of the 16 domains supported by the MMU. By these two bits, domains are designated as Manager domains, Client domains, or No Access domains.
`c5`	FSR	When a memory abort occurs, the fault status register indicates the domain from which the fault originated and a status code depicting the nature of the fault.
`c6`	FAR	The fault address register holds the MVA whose attempted access yielded a Data Abort exception. (In the case of a prefetch abort, the faulting MVA is stored in general purpose register `r14`.)
`c8`	TLB operations	For invalidating specific TLB entries, or flushing the entire TLB.
`c10`	TLB lockdown	For locking down specific TLB entries, so they always remain in the TLB, even after a flush.
`c13`	FCSE PID	The Fast Context-Switch Extension Process ID (FCSE PID) register facilitates context switching between minimal address spaces without requiring a TLB flush.

Tab. 4.4: ARMv5 system control coprocessor registers

one of the 16 MMU domains, and two access permission bits. It also contains bits indicating whether the memory region is bufferable and cacheable.

A coarse page table descriptor contains the base address for the coarse page table, and a domain specifier. The large and small page descriptors contained in the coarse page table contain the base address for the physical memory they describe, 4 sets of 2-bit access permissions (one for each quarter of the page), and bits indicating whether the region is cacheable and bufferable. When small pages are used, a coarse page table contains 256 entries, addressing a total of 1MB of virtual address space.

Note that this setup means that domains, which are specified in the first level table, can only apply at 1MB granularity, to sections or entire coarse page tables – individual pages can't be assigned to specific domains. It is still possible to operate with less physical memory however, and simply not map all the pages in a domain to real memory.

A MVA contains all the information necessary to walk the page table. The high 12 bits in a MVA always point to an entry in the base table. If that entry is a section descriptor, then the remaining 20 bits point to an address within the 1MB of physical memory indicated by the section descriptor. If that entry is a coarse page table descriptor, then the next 4 or 9 bits (depending on if the coarse table contains large or small page descriptors) index the appropriate coarse page table entry, and the final MVA bits point to an address within the large or small physical page.

Each running process in a typical OS will probably need its own page table, so that each process can have an independent virtual address space. When an OS switches to another running process, it changes the `TTB` register to point to the appropriate page table.

Also note that there is additional support for *tiny* (1KB) pages via *fine page tables* (as opposed to coarse page tables) and *tiny page descriptors* that we won't discuss here.

Access control

As we have seen so far, memory regions (coarse page tables or sections) are assigned to a domain. Each of the 16 domains has two bits in the domain access control register *c3* that designate it as either Client, Manager, or No Access. Memory regions at the page or section level also have access permission (AP) bits. There are also S (system protection) and R (ROM protection) bits in the CPSR register. The AP, S, and R bits, together with the current operating mode (privileged or non-privileged), determine whether a memory access attempt will succeed or fail.

First, if the domain in question is a Manager domain, then access is unconditionally granted, regardless of any other bits. If the domain is a No Access domain, then access is unconditionally denied. If the domain is a Client domain, then the 2 access permission bits and the S and R bits are evaluated. These 4 bits, in conjunction with the current privileged/non-privileged operating mode, determine if read-only, read/write, or no access is allowed. Note that an access decision is limited to these three varieties – read-only, read/write, and no access. This means that it is impossible to mark a page as non-executable, a limitation that will create some difficulties later in implementing thin hypervisor-based security services. For instance, kernel code integrity services (see section 4.4.1) depend on a NX bit to ensure non-approved pages aren't executed while in the kernel.

Table 4.5 describes access control decisions based on the AP and S and R bits and operating mode, as seen in [15].

S bit	**R bit**	**AP bits**	**Privileged Mode**	**Non-Privileged Mode**
0	0	00	No Access	No Access
ignored	ignored	01	Read/Write	No Access
ignored	ignored	10	Read/Write	Read-only
ignored	ignored	11	Read/Write	Read/Write
0	1	00	Read only	Read only
1	0	00	Read only	No access
1	1	00	Unpredictable	Unpredictable

Tab. 4.5: ARMv5 MMU access control

The S and R bits create more opportunities for privileged mode access control – the AP bits alone either grant full access or no access to privileged mode, but the S and R bits can grant read-only privileged access. However, because the S and R bits are stored in an MMU register, they are a global setting, and hence offer less flexibility than per-page settings.

If an access attempt is denied, a page table walk discovers an invalid descriptor, or some other problem arises, an abort exception is issued.

Domains and the TLB

One interesting feature of domains is that domain membership is part of a TLB and cache entry. This means that changing the MMU's domain access control bits in the `c3` register doesn't require a TLB flush. The end result is that via domains, large regions of memory can be enabled and disabled without changing page tables or flushing the TLB. We will leverage this mechanism in our implementation.

Fast Context-Switch Extension

As mentioned briefly earlier, the system control coprocessor contains a register holding the FCSE PID, which can enable support for fast context switching. In a typical context switch (where execution changes to a different user process), the TLB must be flushed, since typical operating systems provide an identical virtual address space for each running process. This results in expensive page table walks.

When used, the FCSE inserts a step in the calculation of the MVA. Ordinarily, addresses seen by the CPU are treated as MVAs and forwarded directly to the MMU. With FCSE, if an address A seen by the CPU points to a location within 0 - 32MB, then the MVA is calculated as:

$$MVA = A + (FCSEPID * 32MB)$$

This facilitates systems where all guest processes can run with an independent 32MB address space, and context switches between them will not require a TLB flush – as described here[31].

This feature is an interesting innovation, and might work well for embedded scenarios where processes may not need large quantities of virtual memory, but a 32MB address space certainly won't suffice for all applications. It also creates potential difficulties when trying to protect the address spaces from each other, since it is possible to "manually" access any address space by simply specifying addresses above 32MB.

4.1.5 Key differences with ARM TrustZone processors

ARM processors that implement TrustZone (a.k.a., "security extensions") have some key differences from ARMv5 CPUs. First of all, a TrustZone-enabled processor will implement either ARMv6 or ARMv7 architecture, which differ from ARMv5. One example is that ARMv6 and v7 MMUs support a NX (also known as XN, or *execute never*) bit in their page tables, which creates new possibilities. Furthermore, they support two page tables simultaneously – one for memory that is more static (kernel memory), one for memory that will be swapped in and out (application memory) – which can lead to performance benefits and simplified management. Additionally, TLB entries can be tagged with an *Address Space Identifier* (ASID), which can further help in reducing the need for TLB flushes.

And of course, TrustZone offers the security features described earlier in section 2.5.5, including two isolated virtual worlds with the possibility to partition hardware and intercept external interrupts from a Monitor mode. TrustZone CPUs also enable the Secure and Normal worlds, as well as the Monitor mode, to locate their exception vector table in custom locations, via indepedent Vector Base Address Registers (VBARs).

And, there are other "cosmetic" differences to be aware of – for instance, the `swi` or "software interrupt" instruction that traps to Supervisor (`svc`) mode to allow system calls has been replaced by a `svc` instruction ("supervisor call") that also traps to Supervisor mode.

We will see that these differences altogether (excepting the cosmetic ones) present a quite different environment for hypervisors and security services.

4.2 The FreeRTOS Kernel

The FreeRTOS kernel[44] is an extremely tiny multitasking real-time kernel ported to a wide collection of embedded sytems. It is based on tasks, which basically consist of a looping function, and are assigned a priority. The kernel implements preemptive, priority-aware scheduling via a periodic timer interrupt. Scheduling can also be non-preemptive (cooperative), in which case tasks must explicitly yield execution. Tasks can communicate with each other via synchronized message queues.

The primary, portable kernel consists of only 3 C source files – `tasks.c`, which contains the task and scheduling logic; `queue.c`, which contains queue structures used by the scheduler and for message passing; and `list.c`, which implements a linked list used in the queue structure. Our interest is mostly in the tasks and scheduling.

Porting to a platform requires additional platform-dependent code, including:

- Low level boot and initialization code to set up hardware, establish stacks in memory, and populate exception vectors, as well as code to set up a timer interrupt at proper frequency and register the kernel's timer tick handler
- Assembler code for basic operations such as saving and restoring CPU context, and enabling and disabling interrupts.
- Interrupt service routines for the timer interrupt or any other interrupts.
- Memory management code for allocating stack space to tasks (the FreeRTOS distribution provides 3 alternative C files containing such code, and at least one suffices for all ported platforms).
- Any device driver code needed by the actual tasks.

The timer interrupt is the heartbeat of the FreeRTOS kernel; at each timer tick, the kernel chooses the highest priority unblocked task for execution.

The kernel provides a collection of header files which expose the task creation, control and communication interfaces. To use the kernel, one writes an application broken down into individual tasks, includes the kernel source when compiling, and simply calls kernel functions from `main` to instantiate and manage tasks and start the real-time scheduler. Therefore, tasks and the kernel reside in the same binary to be executed directly by the target hardware platform.

Both tasks and the kernel run in privileged mode, with tasks in system mode and the kernel in supervisor mode. The kernel allocates a requested amount of memory for the stack of each task, using a malloc-style interface. There is no strict protection between tasks and the kernel or each other.

Because of these last points, FreeRTOS makes for an interesting guest OS to run above a thin hypervisor. Firstly, since the kernel and the tasks don't expect to be at a different privilege level from each other, it ought to be possible to have them both run at user level. Secondly, the hypervisor can provide memory protection for the kernel and the tasks, at least protecting the kernel from the tasks, and possibly protecting the tasks from each other. Furthermore, because FreeRTOS is a widely used embedded real-time OS, it comprises a relevant and useful research platform.

4.3 Thin Hypervisor Approach

When implementing an ultra-thin hypervisor for this thesis, we support a single guest domain, so we don't care about things like VM scheduling. But we do have to address other issues in order to isolate the hypervisor, protect the guest kernel from its applications, and provide the guest kernel with its expected operating environment.

We will begin by discussing potential issues relating to thin hypervisor implementation on ARMv5 in a general way in section 4.3.1. We will then continue with a discussion of how implementation would differ on a more advanced TrustZone platform in section 4.3.2. Finally, we will address implementation of a thin hypervisor specifically for a FreeRTOS guest kernel in section 4.3.3.

4.3.1 General concerns

This section discusses thin hypervisor development concerns in a general way, and is divided into material on operating mode, interposition on guest kernel entries and exits, memory virtualization, use of FCSE in a thin hypervisor, and CPU virtualization. Note that in this section we are not yet considering use of paravirtualization or binary translation, which could alleviate many of the concerns.

Operating mode

A hypervisor must run in a higher privilege mode than the guest system. In the ARM case, there is only one non-privileged mode (`usr` mode), so both the guest kernel and guest applications must run in the same mode (as in Xen-ARM). It must be assessed how this will affect guest OS operation, and what further measures must be taken to protect the guest kernel from its applications. At minimum, this means that all privileged operations attempted by the guest kernel must be virtualized by the hypervisor, including access to privileged registers and instructions. Such operations will be intercepted by the hypervisor's Undefined (`und`) exception handler. Note that without paravirtualization or binary translation, this could quickly become prohibitive, for instance if the guest kernel frequently attempts to enable and disable interrupts.

Protecting the guest kernel from its applications also becomes challenging; a hypervisor must have tight control over memory management and context switches, and ensure that, in spite of each running in User mode, the guest kernel and guest applications each run with expected memory protections.

Interposing on all guest kernel entries and exits

Especially due to the challenges introduced by having the guest kernel and applications run in the same mode, an important condition that a thin hypervisor might have to uphold for successful virtualization of a guest domain is that it interpose on or mediate all guest kernel entries and exits.

To interpose on all kernel entries, it might be sufficient for the hypervisor to interecept and handle all exceptions – including external interrupts (IRQs and FIQs such as timer or I/O interrupts), memory aborts, undefined or privileged instructions, and software interrupts (system calls). This is easily achievable; however, note that exceptions may originate from the guest kernel, in which case they do not signify a kernel entry. In any case, fulfilling this level of interposition will of course cover interception of all attempts to execute privileged instructions (as mentioned above), as such attempts (such as accessing the MMU) generate an undefined instruction exception.

When intercepting any would-be kernel entry, the hypervisor must evaluate the circumstances and take appropriate action. Such action may involve virtualizing or assigning resources, evaluating the security of a memory access or modifying memory access permissions, emulating undefined or unsafe instructions, and forwarding operations to guest kernel handlers.

To interpose on kernel exits, one possible method is that the hypervisor can specially handle attempts by the guest kernel to write to the CPSR and change operating mode. When the guest kernel attempts to change to user mode, a kernel exit is occurring. There are other possibilities as well. For instance, the FreeRTOS distribution has optional support for a wrapper mechanism in which all kernel functions are accessed through controlled wrapper functions that can lift and restore kernel access controls appropriately.

By interposing on all guest kernel entries and exits, we have a foundation that can support other important elements of guest virtualization, including memory and CPU virtualization.

Memory virtualization

To promote Popek and Goldberg's fidelity requirement, the guest domain should run in an address space that is consistent with what it would run in in normal circumstances. Further, to protect the hypervisor memory space, hypervisor pages must be inaccessible by (and ideally, invisible to) the guest domain. To protect the guest kernel, guest kernel pages must be protected from guest applications. As noted previously, on ARM both guest kernel and guest application must run in `usr` mode which means the hypervisor must manage guest kernel memory protection.

As in the case of Xen-ARM, ARM's MMU domains can be used to protect the guest kernel. Xen-ARM uses three domains – one for the hypervisor, one for guest kernels, and one for guest applications[123]. In such a system, protecting the kernel might be managed as follows. Hypervisor pages will be in a domain that will always be designated as Client (called the *hypervisor memory domain*, or *domH*), and marked as only accessible by privileged mode. Guest kernel pages will be in a separate domain (the *kernel memory domain*, or *domK*). On a guest kernel entry, *domK* can be made a Client domain (so that areas of kernel memory can be protected by the hypervisor in different ways), but on an exit to guest application mode, *domK* will be changed to a No Access domain. Finally, the guest application domain (the *application memory domain*, or *domA*) can be set to Client at all times. The three memory domains are listed in table 4.6. Since changes to domain access control take immediate effect without a TLB flush, the mechanism should demonstrate reasonable performance.

Domain	**Short Name**	**Number**	**Contents**
Hypervisor memory domain	domH	0x0	Hypervisor pages
Kernel memory domain	domK	0x1	Guest kernel pages
Application memory domain	domA	0x2	Guest application pages

Tab. 4.6: Memory domains

To facilitate fast context switches (without TLB flushing) vertically in the stack – between hypervisor, guest kernel, and guest application – barring use of FCSE, all three would have to occupy the same address space. This may or may not be possible, depending on the guest OS and its expectations about its address space.

Finally, in order to support virtual address spaces for the guest domain, the hypervisor may need to maintain shadow page tables for the guest kernel. The hypervisor can watch guest OS page tables (which map from “guest virtual” to “guest physical” addresses) and translate them to page tables usable by the real MMU (so they map from “guest virtual” to “machine physical” addresses). One possible method for doing so is to handle the instruction wherein the guest kernel attempts to change the `TTB` register – then the hypervisor can locate the new guest page table in memory, update the corresponding shadow page table, and write-protect the memory page(s) on which the guest page table resides. Any subsequent access to the page table will generate a fault.

The hypervisor will store the shadow page tables in its own protected memory space. In the case of using FCSE, there will be only one page table, but 32MB virtual segments of it will shadow the individual page tables kept by the guest kernel.

Use of FCSE

If we can use FCSE, fast context switching may be simplified. We have not seen the use of FCSE in a hypervisor before. If the guest kernel together with a single guest application can be restricted to a 32MB address space (for example, with the guest kernel using the low 4MB), then using FCSE in a thin hypervisor may be possible. Whenever the guest kernel attempts to change the `TTB` register, indicating a change to a different process’ address space, the hypervisor will intercept it and can then change the FCSE PID. This will cause the MVAs fed to the MMU to point to a different 32MB region of virtual memory. The guest kernel portion of that 32MB will most likely map to the same physical memory for each user application. The hypervisor’s 32MB segment (at 0-32MB) will include physical memory inaccessible to the guest. The hypervisor can also access virtual addresses outside its 32MB by simply using virtual addresses above 32MB. However, as yet it is not clear whether there is an acceptable way to prevent guest processes from doing the same (and thereby accessing other guest application memory or even guest kernel memory) without modifying page tables and flushing the TLB on a context switch, which would negate FCSE's fast context-switching benefit.

One possibility to address this problem would be to again use memory domains – reserve 14 of the 16 memory domains for user applications. Then, as long as there are 14 or fewer user processes, each user process can have its own domain. When switching to a particular user process, the memory domains for all other processes can be marked as No Access (which doesn't require a TLB flush). If there are more than 14 processes, then when switching to a process all other processes that share the same memory domain must be evicted from the TLB, and their page permissions modified to prevent access. Alternatively, after being evicted from the TLB, the pages could be assigned to another memory domain. This solution might work, but it would require performance testing to demonstrate its feasibility – that is, to see if the performance cost of managing and protecting domains in FCSE is outweighed by the benefits of avoiding TLB flushes.

CPU Virtualization

Also in support of the fidelity requirement, the hypervisor must virtualize the CPU so that the guest system's expectations are met. This may mean that on guest kernel entries and exits the hypervisor will have to manually restore and bank the registers that the CPU normally would when switching between privileged modes and user mode. Fortunately, certain ARM instructions can be used to read and write user mode registers even from privileged mode. When the guest leaves the kernel and executes user code again, the hypervisor will likewise save/restore banked state. It is important to be sure, that *all* the registers (including the `CPSR`) are properly handled.

4.3.2 A thin hypervisor on TrustZone

TrustZone, with its virtualization-related features (discussed in section 2.5.5), might seem like an ideal platform for implementing a thin hypervisor, by having the hypervisor run isolated and protected in the Secure world and the guest OS run in the Normal world. This has many advantages. Significantly, the guest OS can now run in privileged mode above its applications. Moreover, the guest OS can perform its own memory management with its own MMU, and the hypervisor can manage its own memory as well as map in guest OS pages. The guest OS can perform its own exception handling with its own independent exception handlers and vector table. The hypervisor can control system boot and assign hardware resources to the Secure and Normal worlds, and can intercept external interrupts (`FIQ` and `IRQ` exceptions and external, non-MMU generated aborts). It can provide a hypercall interface to the guest OS through the agency of the `SMC` instruction, which traps to Monitor mode. Such an interface may in turn be exposed to guest applications via system calls if necessary. ARM TrustZone is clearly capable of supporting two isolated domains, with one having certain powers over the other, but each having independent control of its own basic facilities. As a result, the hypervisor can dispense with memory management and other traditional issues, and focus solely on security and other meaningful functionality.

When it comes to implementing security services in a hypervisor, however, this side-by-side isolation may not be enough. One of the major shortcomings of TrustZone is that it does not enable interposition on *all* exception types – as described earlier, system calls, undefined instruction exceptions, and MMU (both data and prefetch) aborts cannot be configured to trap to Monitor mode. Of course, it could be possible to have privileged mode exception handlers in the Normal world that forward exceptions to the Secure world via a monitor call, but this may not be feasible, reasonable, or trustworthy, depending on the situation. Another potential problem is the independent MMU and other state management in each world – while this does simplify isolation, preventing the Secure world from controlling or protecting important aspects of the Normal world state (such as memory permissions, and page table and exception vector table locations) can impede security-oriented functionality. A hypervisor in the secure world thus becomes a sort of "co-visor" – at higher privilege, but off to the side of the guest OS. We shall see these issues reflected in specific situations in section 4.4.

4.3.3 A thin hypervisor for FreeRTOS

In this section we will describe our approach for implementing a thin hypervisor specifically for a FreeRTOS guest kernel on ARMv5.

Interposition

The hypervisor must run in a privileged mode, with the FreeRTOS kernel and its tasks running in User mode. The hypervisor will have to interpose between FreeRTOS and the hardware, and virtualize whatever privileged operations are required by the kernel and the tasks.

The hypervisor can contain its own platform-dependent code for facilitating critical basic setup, including the establishment of a periodic timer interrupt. Any other platform-dependent code required by the kernel itself can also be located within the hypervisor. For instance, the platform-dependent code for enabling/disabling interrupts can be implemented via a hypercall.

Memory protection

As mentioned in section 4.2, the FreeRTOS kernel does not employ virtual memory address spaces for its tasks. Indeed, each task runs in privileged mode with a dynamically allocated block of memory for its stack. The kernel maintains the heap from which these blocks are allocated. There is no built-in memory protection. Tasks can access any memory in the system.

As thin hypervisor implementers, this state of affairs is interesting to us for three reasons. One, the thin hypervisor does not need to support shadow page tables, since the guest kernel has no such table to shadow. Two, the hypervisor can provide memory protection where there is none – supporting greater security and robustness for FreeRTOS operation. Three, if we can locate the hypervisor in the same 32-bit address space as the kernel and tasks, we need never flush the TLB as part of a context switch.

By using the ARMv5 MMU, we can partition the space into, again, *domH*, *domK*, and *domA* (where *domA* now contains task memory, rather than application memory). *domH* is a Client domain, only accessible in privileged mode.

The separation between *domK* and *domA*, on the other hand, becomes a little more interesting. The kernel contains functions used by the tasks to manage themselves and communicate with each other. In a normal FreeRTOS setup, the kernel and tasks run at the same privilege level, so none of these functions are implemented as system calls – they are more like library calls. With our hypervisor, the tasks need to be able to call these functions, but shouldn't be able to tamper with kernel memory. Ideally, the tasks shouldn't be able to even *read* kernel memory.

We intend to handle this problem by using the wrapper mechanism used already by a few ports of FreeRTOS to embedded systems with a Memory Protection Unit (MPU) – a less capable version of a MMU. When using this mechanism, the kernel is inaccessible to tasks. To access the kernel, tasks call a controlled wrapper function, which can enable access to the kernel, call the desired kernel function, and disable access to the kernel in a controlled fashion.

When it comes to protecting tasks from each other, the situation becomes more complex. A meaningful way to protect tasks would be to separate different tasks' stack memory. One potential way to achieve this would be to modify the memory allocation code to be able to allocate memory from spaces in different domains. Then, when executing a particular task, the domain for its stack would be enabled, but other stack domains would be disabled. If there are not enough memory domains to assign one to each task stack, then cooperating tasks could have their stacks in the same domain. Alternatively, individual page permissions could be managed, which would incur a greater performance hit. Whatever the case, for this to work it must be possible to associate user-mode execution context with a stack domain, so that the hypervisor can update MMU settings appropriately when a context switch occurs.

4.4 Analysis and Transitioning of Current Systems

In this section we will discuss a few security services that are candidates for being implemented in an ARMv5 thin hypevisor. As with section 4, we will discuss for each service general concerns, TrustZone-specific opportunties and challenges, and approaches for implementing the service for a FreeRTOS guest kernel.

4.4.1 Kernel code interity

Implementing a kernel code integrity service is a natural first foray, as such a service has already been run in a thin hypervisor (SecVisor [91]). However, the lack of a NX bit in ARMv5 memory protection will prove to be a serious obstacle.

Protecting kernel code integrity means we must ensure that approved kernel code and *only* approved kernel code is executed when in the guest kernel. This means that memory in all non-kernel pages and even kernel data pages must never be executed when in the guest kernel (even in the presence of kernel vulnerabilities), kernel code must not be modified, and only kernel code approved by a policy is ever executed. In certain situations, it might even be necessary to support dynamic modification of the approved set of kernel code.

To fulfill the above conditions, actions must be taken at key points. Upon entering guest kernel mode, the hypervisor must ensure that only approved code is executed until exiting the guest kernel. Without a NX bit, this unfortunately means that *all* pages that aren't approved kernel code, including kernel data pages and all application pages, must be rendered inaccessible. Otherwise, a code execution vulnerability could be used to execute non-approved code. However, this condition of coure means that any legitimate read-only access by the kernel to any of those pages will generate Data Aborts. Since a kernel must often access its own data pages, and likely application data pages, the cost and complexity of handling these aborts and allowing the valid accesses will likely become prohibitive.

The best possible approach seems to be to use ARM MMU domains to partition the kernel into approved code and data, and to also use one or more domains for application pages. Then, when in kernel mode all non-approved pages can at least be disabled without flushing the TLB or changing page permissions. However, this doesn't get around the excessive abort problem. It would seem that kernel integrity protection service in this context is at an impasse.

Other issues

In order to ensure that only *approved* code is executed, we need to be sure that kernel code pages aren't modified – therefore, all code pages in *domK* should be marked as read-only, even when in the guest kernel. Then, even a kernel vulnerability can't be exploited to modify kernel code.

If the policy doesn't need to be runtime-flexible, then kernel pages can be evaluated against the policy at system startup.

In order to have a runtime-flexible policy defining allowable code, we can begin by evaluating kernel pages against the current policy at startup and marking approved pages as read-only. But from that point, we would need to interpose on policy changes, and update page permissions as the policy changes. An interface for modifying such a policy is a security challenge in its own right, and not discussed here. Likewise, the possible formats of the policy are not discussed here.

Kernel integrity on TrustZone

With a hypervisor in the TrustZone Secure world, how could kernel integrity protection be implemented?

In considering this possibility, we immediately run into the key limitation mentioned earlier in section 4.3.2 – that is, TrustZone offers no built-in way to intercept all Normal world exception types, and therefore no way to interpose on all guest kernel entries and exits. However, there is a way around this issue.

The problem can be remedied by paravirtualizing the guest kernel. This effort could be rather minimal, potentially involving only the insertion of a `SMC` instruction at the start of each exception handler in the guest OS, or in the exception vector table. From here, kernel integrity protection mechanisms can proceed as described above. At this point, the protection mechanisms become simplified in comparison to ARMv5, since any TrustZone processor will have a NX bit (as described in section 4.3.2). The kernel integrity protection mechanisms will protect the paravirtualized kernel code; the kernel must be booted safely by the Secure world to complete the “chain of trust”.

However, there is a potentially weak link in this chain. As also described in section 4.3.2, the hypervisor in the Secure world has no way to control certain aspects of Normal world state or interpose on Normal world privileged attempts to modify this state. This includes the Normal world MMU registers that control the location of the exception vector table and the translation table base addresses. What this means is that if there is some sort of kernel vulnerability whereby the kernel can be wrongfully induced to modify the location of the exception vector table, which wouldn’t violate kernel code integrity, an attacker may be able to circumvent the paravirtualized exception handlers that trap to Monitor mode. Furthermore, if the kernel contains even a small amount of approved malicious code (potentially inserted by an insider, or via a device driver or kernel module), then this approved code could change the exception vector table or the handler code and thereby unravel the mechanism. Basically, it is difficult to protect the integrity of an entity that has ample permissions and opportunity to violate its integrity itself!

In spite of these issues, using TrustZone is still compelling, since it so drastically reduces the complexity of the thin hypervisor. Depending on the complexity of the kernel and other context, TrustZone might possibly make for an attractive kernel code integrity platform.

Kernel code integrity for FreeRTOS

Implementing kernel code integrity for a FreeRTOS guest kernel on ARMv5 is simpler than doing so for a Linux or other mainstream OS kernel. For instance, there is no need with FreeRTOS to load kernel modules or change the kernel code at runtime, so we don’t need to support a dynamically configurable policy. With a simple unified address space, page table management becomes easier.

However, there is still the “missing NX bit” problem described earlier, which appears difficult to overcome.

4.4.2 Application protection

Application memory protection, as offered by Overshadow and SP^3, provides compelling benefits but in both of these cases involves significant complexity. We will look at possible simpler approaches on a thin ARMv5 hypervisor. By application memory protection, we mean protecting memory used by individual applications from being viewed or modified by other applications *or* the OS. There are several possible avenues for achieving this protection.

Via encryption

One possible approach is via encryption, which both Overshadow and SP^3 selected. It might be implemented on a thin hypervisor as follows:

- First, the hypervisor must be able to differentiate between the guest kernel and each application. If the hypervisor is already interposing on guest kernel entries and exits, it thereby distinguishes the kernel from applications. With an OS that manages a page table for each process, applications can be distinguished from each other via the base address for their page table, stored for the current process in the `TTB` register. Note that it will thusly be necessary for the hypervisor to protect guest OS page table memory (as already necessitated for maintaining shadow page tables), to prevent the guest OS from discreetly swapping page tables for different processes. With an OS that keeps all application memory in the same address space, some other method of differentiation will be required, perhaps facilitated by use of ARM memory domains.

- The hypervisor must provide encryption facilities. For simplicity's sake, the hypervisor could keep a single secret key with a single algorithm (e.g., AES-128) that it uses to encrypt all applications, although it wouldn't be difficult to expand this setup to use indendent keys for different processes and multiple algorithms.

- If application protection is to be configurable by the guest applications (at least, capable of being activated and deactivated), the hypervisor must expose some sort of interface. One way to do this is extend the instruction set with a new emulated instruction that will activate and deactivate protection. This instruction would be intercepted and emulated by the hypervisor's undefined instruction handler. On receiving the activation instruction, the hypervisor will inspect the `LR` register, the TTB register, or other registers to see which application or domain the instruction came from. If the instruction came from an application, then the hypervisor will activate protection for that application's memory regions. For this to work, the hypervisor must also be able to tell when the same process exits, so that it can cease protecting those memory regions.

- The hypervisor needs to manage the memory protection, so that an application works with decrypted memory, but the OS sees encrypted memory (for example, when swapping encrypted pages to disk).

  One approach for implementing this is to keep an extra "protection table" alongside each shadow page table that denotes whether each page is currently in an encrypted or decrypted state. Via a shadow page table together with this additional structure, a hypervisor could track if any given page is in physical memory or swapped out, and encrypted or decrypted. It is important that any swapped-out page always be encrypted. On a guest kernel exit, we can mark all encrypted pages for the target application as inaccessible (but cache their actual permissions in the protection table). Then, if the guest application tries to access an encrypted page, it will fault to the hypervisor. The hypervisor can evaluate the access attempt against the cached permissions. If the access is invalid, then execution will be aborted. If the access is valid, the hypervisor must address two alternatives. One, if the page is already in memory, the hypervisor can simply decrypt the page and allow the access to continue. Two, if the page isn't in memory, the hypervisor can swap in the page itself and decrypt, or attempt to delegate to the guest kernel to swap in the page. In the former case, the hypervisor must ensure the guest OS page tables reflect the change; in the latter case, the hypervisor has to somehow ensure that the page is safely decrypted after being swapped in and never visible in the clear by the kernel.

On a kernel entry, all decrypted guest application pages in memory will be marked as inaccessible. If the kernel attempts a valid access to such a page, the hypervisor will encrypt the page in memory. In this way, any page the kernel swaps to disk is sure to be encrypted. To improve efficiency, the hypervisor can cache encrypted/decrypted versions of a page in memory. If a user application accesses an encrypted page (causing it to be decrypted), but doesn't modify it, then a cached encrypted version of the page can be mapped back to guest kernel memory on a kernel entry. Likewise, if the guest kernel accesses user memory pages (hence causing them to be encrypted), after returning to the guest application, cached decrypted versions of those pages can be mapped to application memory when the application accesses them again.

For this scheme to work, cache and TLB entries for decrypted application pages must be invalidated on a kernel entry, just as for encrypted pages belonging to the target application on a kernel exit. This of course may cause performance issues.

- Finally, the hypervisor must specially handle any operations that cross the kernel-application boundary. For instance, when it intercepts a system call, the hypervisor must copy the contents of any pointer arguments to a memory region accessible by the guest OS kernel, then dispatch the system call with the new arguments to the guest OS.

Via memory protection

One other extremely simple possibility to protect guest applications *from only the guest kernel* is to simply mark application domains or pages as inaccessible whenever entering the guest kernel. This would require the hypervisor to copy system call arguments to a neutral buffer space, as described in the last bullet in the above list. This dovetails with previously discussed possible approaches for kernel integrity protection. However, this approach precludes swapping pages to disk, foremost since the guest kernel will have no read access to any guest application pages, and also because such pages would be decrypted and therefore visible to anyone who can read the disk.

Via memory protection and virtualized swapping

Another possibility, that we have not seen in prior work, would be to protect guest application memory by having the hypervisor completely virtualize the swapping of memory pages. To achieve this safely, the hypervisor would need access to a private disk, and most likely a device driver for the disk. It might also be possible to manage a private space on an existing system disk by using a guest OS driver and by intercepting and carefully examining all I/O interrupts. Alternatively, the hypervisor could of course encrypt every page that it swaps to disk, and then any disk could be used.

In this approach, whenever the hypervisor intercepts a memory fault due to a valid access to a page not present in physical memory, it will swap out a page itself and swap in the requested page. In this way, when in guest kernel mode, application domains or pages can simply be marked as No Access, assuming system call parameters can be copied by the hypervisor to a neutral memory space as already described. To protect applications from each other, a pool of separate user application memory domains could be used, as in our suggestion for using FCSE – all domains except the target application's would be marked No Access when entering the target application. This approach circumvents the need for invalidating cache or TLB entries, since memory domain state is evaluated at each access.

In any of the above three strategies, it is clear that performance testing is necessary to ensure the feasibility of any implemented solutions.

Guest application protection on TrustZone

Use of TrustZone for protecting application memory is predicated on the use of paravirtualization, as with kernel integrity protection, to interpose on all kernel entries and exits.

When using TrustZone CPUs that support ASIDs, it becomes simpler to partition the memory space into hardware-enforced components. If the hypervisor can differentiate between guest application processes, then it can assign an address space ID to the guest kernel and each application. When the Monitor mode intercepts kernel entries and exits, it will maintain the current ASID field.

However, since the need for application protection is premised on a malicious OS, the possibilities for TrustZone founder – it is impossible to ensure that the Normal world OS doesn't move around process page tables in memory or conduct other trickery to subvert the system, even if the Secure world completely manages swapping pages itself.

Task protection for FreeRTOS

If on a kernel entry we mark the application domain (and any stack domains) as inaccessible, and are able to copy over parameters to kernel functions so that the kernel need not access the application domain for any legitimate reason, it should be possible to protect the tasks from the kernel. Since pages are never swapped to disk, we don't need to consider anything further.

It should also be possible to protect task stack memory from other tasks by using memory domains and/or FCSE in a previously suggested manner. However, fully protecting tasks from each other is more complicated, since normal FreeRTOS tasks may share a great deal of writable static data, kernel message queues, and other data. Since tasks do not have their own address spaces, it becomes difficult to isolate them completely.

4.4.3 Identifying covert binaries

By monitoring all kernel entries and exits, a thin hypervisor on ARMv5 stands in a position to implement an approach similar to that in [70] used to monitor covertly executing binaries.

To implement this well, it is necessary to mark all code pages as inaccessible. If it is not possible to distinguish code pages from data pages, then we run into the same problem encountered before – the lack of a NX bit in ARMv5. Getting around this problem requires marking all pages inaccessible, which may be prohibitive for a running application.

At each memory fault in a code page, the current execution context can be ascertained via the `TTB` register. The accessed page can be hashed and compared with a database of known-good hashed binaries to see if known-good code is being executed. (This is especially important for kernel code.) After this check, the page can be marked read-only in the current page table (and the page table memory protected by the hypervisor), so that accesses to the same page by the same process in the future can proceed freely.

The hypervisor can also intercept each attempt by the guest to change the `TTB` register; by doing so it should be able to see all processes running in the system.

In this way, the hypervisor should be able to get a picture of what code is running in which processes in the system. The hypevisor then needs some mechanism by which to report feedback to the user. Such a mechanism will not be discussed here, although it could involve emulating new instructions usable by guest applications. A user can compare feedback from the hypervisor with information provided by the guest system itself as to which processes are running, and thereby determine via examination of any mismatch if any processes are running covertly.

Identifying covert binaries on TrustZone

On TrustZone, once again the primary difficulties are the lack of complete interposition, as well as the independent memory control structures present in each world. Therefore, *actively* identifying covert binaries from the security world would require some level of paravirtualization. However, *passively* monitoring for covert binaries from the Secure world (at the discretion of the “co-visor”) would still be possible, and assuming the monitoring was done frequently enough (and possibly with an unpredictable pattern), then TrustZone easily supplies all the needed components. The Monitor mode is capable of examining Normal world CPU and MMU registers, and the Secure world can map in Normal world memory pages, which furnishes the hypervisor all the tools required to identify covert binaries as in [70].

Identifying covert binaries for FreeRTOS

Identifying covert binaries running on a FreeRTOS kernel using an ARMv5 thin hypervisor should be fairly straightforward, though of questionable usefulness. By watching kernel entries and exits, and observing the memory regions in which each task operates, our hypervisor could track task identity.

A covert binary for FreeRTOS amounts to a covert task. A covert task might be initiated by a malicious or hijacked task. If a task can be started at arbitrary priority and possibly hidden from kernel API functions due to a vulnerability, then a covert task might be possible. The likelihood of this situation seems quite lower than that of a mainstream OS kernel rootkit.

However, it may still be useful for a FreeRTOS hypervisor to keep close watch of executing tasks. Monitoring task execution could support other services, such as preventing tasks from performing malicious operations or otherwise harming other tasks (flooding message queues, attempting to deny service to other tasks, etc.). The potential of such a task execution monitoring service for FreeRTOS can be explored in the future.

4.5 Summary

This chapter introduces our main technologies in detail, the ARMv5 architecture and the FreeRTOS kernel. It outlines general concerns for various aspects of thin hypervisor development on ARMv5, and makes specific observations on supporting FreeRTOS as a guest. It also moves on to suggest specific security services inspired by results from the background study, and potential approaches for implementing them on ARMv5. Throughout, comparison of ARMv5 architecture and approaches with more advanced TrustZone architecture is provided. This chapter provides a foundation for implementation, gathering relevant information, isolating important concerns and providing guidance and strategies.

5. IMPLEMENTATION OF A THIN HYPERVISOR

With the previous chapter laying the groundwork for implementation, in this chapter, we describe our actual implementation. We have implemented a thin hypervisor running on ARM and supporting the FreeRTOS kernel[44] as a single guest. We have simulated an ARM926EJ-S CPU (which supports the ARMv5 architecture) and peripherals using the Open Virtual Platforms (OVP)[79] simulation environment.

Section 5.1 describes our use of OVP. Section 5.2 summarizes the overall structure of our hypervisor, while section 5.3 continues with a closer look at specific important aspects of our implementation. Section 5.4 offers an analysis of the security properties of our implementation, and finally, section 5.5 explores and motivates ideas for expansions to the hypervisor.

5.1 Use of OVP

We used Open Virtual Platforms free simulation tools to build and emulate our ARM platform. We selected OVP because it is flexible and powerful, enabling precise construction of custom hardware simulations with a wide variety of CPUs and peripherals, and facilitating efficient development and testing. Another advantage is that the source code of much of the simulated hardware is available from the OVP website[79]. Thus, one can examine, for instance, the code that models a particular peripheral, which can facilitate implementation and debugging.

To simulate a platform on OVP, one writes a C program that calls OVP APIs to instantiate CPUs, peripherals, buses, connections, and memory, possibly loads a binary file (such as an ELF file) into platform memory, and then starts the simulation. From that point, the simulated hardware platform is running, and the OVP APIs also enable the simulation program to interact with the simulated platform, for example by reading and writing memory and registers, simulating interrupts, or other such operations.

By using platform-specific toolchains (such as the ARM GNU compiler toolset, which can be acquired from the OVP website and elsewhere), one can compile binaries that will run on the simulated platform.

Additionally, while OVP tools include many prebuilt CPU and peripheral models, one can write additional models for peripherals and CPUs and use those in simulated platforms.

We have created and experimented with different platform configurations with various ARM CPUs and peripheral devices. For our FreeRTOS hypervisor, we have adapted an OVP example program that simulates an Atmel ARM board complete with a variety of standard peripherals (including memory, interrupt controller, timer, and others). We selected this particular platform because the FreeRTOS kernel has already been ported to a similar Atmel platform with identical peripherals, and therefore running FreeRTOS on that simulated platform was expected to be relatively simple. One key difference between the original simulated platform and our adapted version is that we "upgraded" the CPU from an ARM7TDMI to an ARM926EJ-S to acquire the MMU in the latter.

In using a simulated platform, one encounters the issue of consistency and compatibility – does the simulated platform correctly duplicate the hardware? OVP only attempts to duplicate the interfaces and behavior, and for the most part we found it to be correct. In some instances, however, we noticed discrepancies between the vendor-documented interface for certain peripherals and the behavior implemented by OVP's simulated peripherals. This development slightly dashed our hopes that running FreeRTOS on our simulated platform would be straightforward, but we were able to work around these issues. The OVP team has been notified of our discoveries.

5.2 Overall structure

This section gives an overview of the structure of our FreeRTOS hypervisor system, which is depicted in figure 5.1.

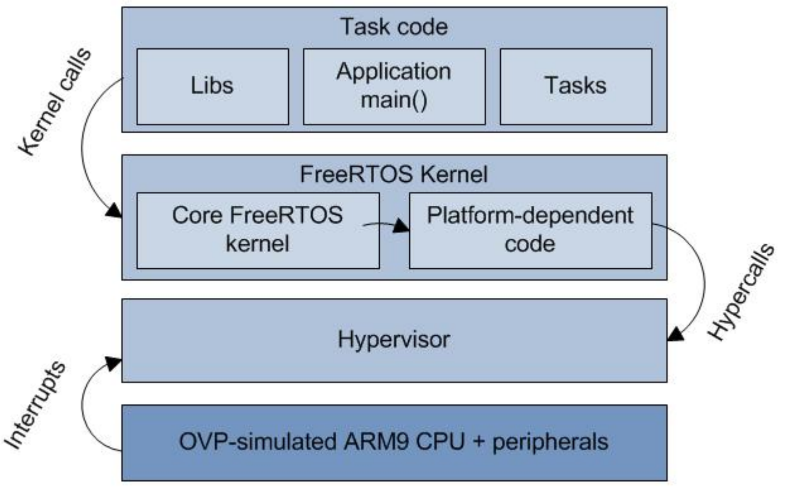

Fig. 5.1: Overall implementation structure

Our software has three central components – one, the core FreeRTOS kernel; two, platform-dependent code outside both the hypervisor and the core kernel that is used by the core kernel; three, the hypervisor layer itself, which contains minor boot and hardware setup code, exposes hypercalls used by the platform-dependent code, and manages the MMU.

The hypervisor runs in privileged mode, whereas all other code runs in user mode. Within user mode, kernel functions and data receive special protection and a virtual mode of their own via use of MMU domains, which will be described in section 5.3.2. The FreeRTOS main and tasks run atop the kernel. All components are compiled into a single binary, which we have elected to do for simplicity, as our current implementation is purposed for performance testing and ascertaining basic security potential – future efforts will look into separating the hypervisor and FreeRTOS binaries and having the hypervisor load the FreeRTOS binary in a trustworthy manner.

The three major components are described here in subsections.

5.2.1 The core kernel

The core FreeRTOS kernel has remained almost completely unchanged. Previously, the kernel allocated memory for all task stacks and for task control blocks (the OS data structure for managing a task) from the same heap. In order to allocate the task control blocks from kernel memory, and to allocate the task stacks from a pool of separated heaps (discussed more in section 5.3.2), we had to modify 9 lines of one of the core kernel files. These lines mostly concerned function calls to platform-dependent code, including memory allocation functions.

5.2.2 Platform-dependent code

In a FreeRTOS port, the platform-depdent code is responsible for carrying out critical, low level actions, as described in section 4.2. This functionality often requires privileged instructions. The core kernel and tasks depend on these operations. In our port of FreeRTOS running on ARMv5, we consider the platform-dependent code to be part of the FreeRTOS kernel, and it is protected as such.

When running our FreeRTOS port above our hypervisor, all the privileged operations must be implemented in the hypervisor, including operations on protected peripherals. This means that, except for task memory allocation, the platform-dependent portion of the FreeRTOS code had to be completely *paravirtualized*, using a simple interface of 10 hypercalls exposed by the hypervisor. Since the platform-dependent code comprises only a small set of operations, transforming it to use the hypercall interface was reasonable.

Additionally, the memory allocation platform-dependent code had to be modified to support allocating from multiple heaps located in different memory regions, as mentioned.

5.2.3 The hypervisor

The hypervisor layer is intended to be generic and not tied to FreeRTOS (although it is of course tied to the underlying ARM hardware). The hypervisor layer, therefore, has no particular knowledge of FreeRTOS code or structures. It contains boot code adapted from boot code in the FreeRTOS distribution, exception handlers, and some MMU setup code, and it exposes the above mentioned hypercall interface to allow safe implementation of critical, platform-dependent functionality. The actual implementations of the hypercalls have been partially adapted from platform-dependent code found already in the FreeRTOS distribution, but were mostly written from scratch. In addition to supporting the platform dependent code, note that two of the hypercalls faciliate entering and exiting the guest kernel, which is discussed more in section 5.3.2. While the generic functionality supported by the hypervisor layer may not be enough to support other OSs, it could be expanded to do so.

However, the major challenge when evolving the hypervisor to support larger, more complex, mainstream guest OSs is to either paravirtualize an entire kernel (as the Xen hypervisor used to require), which could be a significant and non-portable undertaking, or to switch to a binary translation model wherein unsafe instructions are translated live by the hypervisor to safe instructions. The latter has the advantage that the guest OS requires no modification, but requires a significant engineering effort. Both options are decidedly non-trivial. However, hardware advances in embedded virtualization support could make supporting mainstream OSs much easier overall, by offering hardware-based virtualization of privileged operations and state as in Intel VT and AMD-V, so that a guest OS can run with fewer changes. The less a guest OS need be translated or modified, the better.

5.3 Key system aspects

In this section, we will examine more closely some central aspects of the hypervisor and the platform-dependent code.

5.3.1 Hypercall interface

The hypercall interface exposed by the hypervisor is summarized in table 5.1.

The hypercall interface provides safe access to privileged functionality required by the platform-dependent code, tasks, and the MMU wrappers. It is extremely simple, consisting of only 10 calls. A hypercall is triggered by the `SWI` instruction, and therefore handled in `SVC` mode. An identifier and description for each hypercall is found in table 5.1.

ID	Description	Origin restriction
EIN	Enable user mode interrupts	kernel
DIN	Disable user mode interrupts	kernel
ENC	Enter user mode critical section	-
EXC	Exit user mode critical section	-
STI	Set up timer interrupt	kernel
ENK	Enter guest kernel mode	wrappers
EXK	Exit guest kernel mode	wrappers
YLD	Yield execution	-
SYH	Set yield handler	kernel
REC	Restore execution context	kernel

Tab. 5.1: Hypercall interface

The STI call, while currently specific to the timer interrupt, is designed to support future genericity, and could easily be made to allow setting up of different types of interrupts and to take additional parameters. For instance, it could be possible to merge the STI and SYH calls into a single “configure interrupt” call. As of now the only parameter to these hypercalls is a pointer to a handler function that must be located in the kernel.

The interrupt/yielding process, which involves the STI and SYH calls as well as the YLD and REC calls, is described in section 5.3.3). Use of MMU wrappers to protect the kernel, which involves the ENK and EXK calls, is described in section 5.3.2.

The ”Origin restriction” column in table 5.1 refers to where the hypervisor restricts the origin of the hypercall. Most calls must originate in the FreeRTOS kernel. The ENK call must originate in the MMU wrappers. Some calls, such as yielding and entering and exiting critical sections, have no origin restriction, and can be issued directly by tasks. An interface for issuing such unrestricted calls is exposed to tasks via unprotected functions or macros included in the platform-dependent code.

5.3.2 Memory protection

Our thin hypervisor is intended to improve security for our guest kernel and tasks, and a keystone of that security in our current system is the ARMv5 MMU. Using the MMU allows us to protect critical areas in memory while maintaining reasonable performance. The MMU furnishes protection for the critical elements in our TCB, namely the hypervisor and the system-critical hardware, and additionally offers protections for the guest kernel, and the tasks. Only privileged code – the hypervisor – can modify the MMU settings.

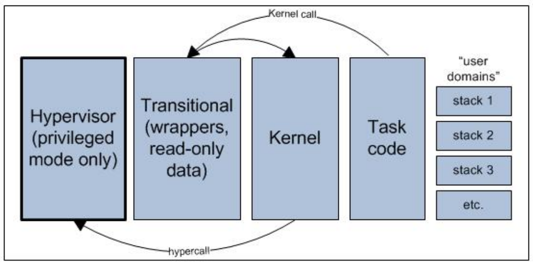

Fig. 5.2: MMU domains

We use several memory domains (depicted in figure 5.2). Firstly, the hypervisor uses the MMU to protect itself and critical device memory (such as the timer and interrupt controller) in the hypervisor domain. The hypervisor domain is only accessible in privileged mode. The system starts in privileged mode, at which point the hypervisor configures the MMU, after which execution transitions to user mode to start the FreeRTOS kernel application. Transition back to privileged mode only occurs as a result of hardware exceptions or hypercalls (the latter are implemented as a system call would normally be), ensuring the MMU isn’t tampered with by unauthorized code.

Secondly, the hypervisor uses the MMU to protect the kernel, including all kernel code and data, in the kernel domain. The FreeRTOS kernel API, which is nomally freely available to task code, is now hidden behind a collection of wrapper functions. Observe that this wrapper mechanism has been previously used by FreeRTOS ports on platforms with a memory protection unit (MPU) – a less flexible version of a MMU. In our system, the wrapper functions reside in a special “transitional” MMU domain that is read-only to user mode and can enable and disable access to the kernel domain. The kernel domain is inaccessible to task code, but the transitional domain can "switch to kernel mode" by enabling the kernel domain with a hypercall. It calls a kernel function, and then "exits kernel mode" by disabling the kernel domain with another hypercall. Thus, through these wrappers and domains, we have created a virtual kernel mode. The Xen-ARM project has similarly used ARM MMU domains to create a virtual kernel mode[55].

Note that as read-only for user mode, the transitional domain can also serve as an area for the hypervisor to store read-only data that the kernel must be able to view. Furthermore, the wrappers provide a means to restrict the kernel interface, by either barring functions outright, or only allowing application code to call functions before the scheduler is initiated. For instance, we have prevented all application code from being able to allocate memory, and prevented the creation of new tasks once the scheduler is started.

Finally, the hypervisor uses the MMU to protect individual task stacks, as follows. While the task domain contains task code and static data, the hypervisor supports 10 additional separate memory domains termed "user domains". A non-privileged execution context can be associated with such a user domain. When restoring execution context, the hypervisor enables access to the associated user domain (if any), in additional to the standard task domain. The FreeRTOS kernel can take advantage of this feature through the modifications to the memory allocation code referenced earlier, which enable the kernel to allocate memory from heaps located in these user domains. The kernel uses this functionality to allocate task stacks in separate user domains.

Using this system, if each task is allocated in a separate domain, it will be impossible for any task to access another task's stack. This approach does limit an application to 10 tasks. However, if feasible, tasks that are known to be trustworthy and mutually trusting can be located in the same domain together.

The effect of this task stack protection is important. Without stack protection, tasks can tamper with each other's stacks, which can result in inteference with task activities, and even arbitrary code execution by overwriting a return address saved in a stack frame. In fact, through overwriting a saved return address, it would even be possible to break the virtual kernel mode by causing a task in kernel mode to return to code outside the kernel without exiting kernel mode. Protecting task stacks from other tasks resolves this problem. Additionally, it should be noted that while an individual task can of course write to its own stack, it can't break the virtual kernel mode, since once in kernel mode all execution is entirely in the hands of the wrappers and the kernel.

Since the hypervisor controls access to the ENK hypercall as described earlier, barring other vulnerabilities, a malicious task can't tamper with the kernel except by using the wrapper interface. Likewise, neither can tasks nor the kernel tamper with the hypervisor except through the hypercall interface.

Since ARM domains are specified at 1MB granularity, even with only 1MB per domain using 14 domains requires at least 14MB of address space. However, it is possible to use far less physical memory by not mapping all pages in a 1MB domain – our entire system, with hypervisor, FreeRTOS kernel, stacks, and tasks and libraries, fits easily within 1MB of real memory, using extremely generous stacks that could be significantly reduced, and is thus suitable for quite limited hardware platforms.

5.3.3 Interrupts and yielding

Our hypervisor protects critical hardware, such as the interrupt controller and timer, with the MMU. No task can manipulate the timer interrupt. However, the hypervisor allows the FreeRTOS kernel to designate a timer interrupt handler function via the STI hypercall.

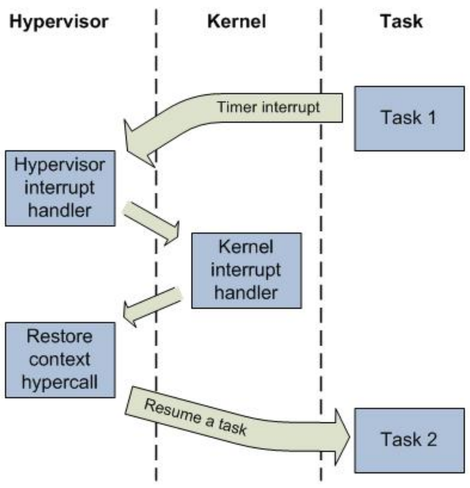

Fig. 5.3: Interrupt sequence

The interrupt sequence is illustrated in figure 5.3. When a timer interrupt occurs, the hypervisor first saves the interrupted task's execution context, which includes registers and CPU state, guest mode (that is, kernel or task, since a task might be interrupted during a kernel call), and other details. The hypervisor then disables user mode interrupts, dismisses the interrupt, and returns to the designated kernel handler function. The function has read-only access to the saved context, which it can store in a task control block. The handler can then perform other activities, such as incrementing a tick counter or scheduling another task for execution, and then it issues a REC hypercall to restore the execution context of a task. Yielding is achieved with a similar mechanism – except, the yield process begins with a YLD hypercall instead of a device interrupt, after which the same process is followed. This mechanism is generic, and would support the addition of other protected interrupts beyond just a timer interrupt.

5.4 Security Analysis

This section presents a brief analysis of the security advantages and shortcomings of our hypervisor.

A normal FreeRTOS application has no security protections. Tasks, hardware, and the kernel are fully exposed. Therefore, our implementation started from zero.

Our thin hypervisor uses the ARM9 MMU to protect itself, critical peripherals, the kernel, and the tasks. Assuming the hypervisor is sound, protections for the hypervisor and the peripherals are unaffected by kernel or task activity.

Furthermore, assuming the kernel is sound, the protections for the kernel and for task stacks will also remain intact, regardless of malicious task activity. Our hypervisor currently cannot protect the kernel against its own vulnerabilities. If a task manages to exploit a kernel vulnerability, it may be able to circumvent kernel memory protections to make unauthorized changes to memory, or break the virtual kernel mode and execute arbitrary code as the kernel. And of course, if the kernel itself is malicious, it can easily break the virtual kernel mode at any time, by causing execution in kernel mode to branch to code outside the kernel – nothing is forcing the kernel to return to the wrapper functions that exit kernel mode. Hypervisor-supported kernel integrity protection as we intend to implement in the future would at least prevent unauthorized code execution in kernel mode, which a normal kernel itself could not guarantee.

As described, task protection is limited to task stacks. While this amounts to a great improvement, there is still globally writable static data in shared libraries, and potentially also in task code, and task code is readable by other tasks. The unified address space of FreeRTOS makes it difficult to completely isolate tasks from each other.

Wrappers are known to be a weak point in security[114]. Our system has the benefit that it is single core, and also that the wrappers are extremely simple – enabling kernel mode, calling a kernel function, and disabling. As discussed earlier, task stack protection prevents a malicious task from hijacking the execution of another task in kernel mode, which would defeat the wrapper mechanism. From our analysis, excepting the presence of kernel vulnerabilities, we cannot see another way that a task would be able to break the wrapper mechanism and the virtual kernel mode it supports. Therefore, breaking the wrappers requires a compromised or malicious kernel.

The security of the system is naturally dependent on the security of the hypervisor. Our hypervisor exposes a minimal hypercall interface, and contains less than 500 lines of code, much of that simple assembler. This makes it reasonable to formally verify its security, which we intend to do in the future.

In summation, assuming hypervisor soundness, our system provides protection for the hypervisor itself and critical peripherals, and assuming kernel soundness, our system provides protection for the kernel and the task stacks. The soundness of the hypervisor is easier to assure, since the FreeRTOS kernel exposes a great deal of functionality to the tasks and was not designed for security; however, we have attempted to minimize the kernel interface and thereby increase the level of assurance. Future implementation of kernel integrity protection in the hypervisor via newer hardware will *not* depend on any assumptions of kernel soundness, and will reduce dependence on the wrapper mechanism. It is such services that only depend on the hypervisor that we are most interested in.

This implementation has explored the possibilities for thin hypervisors on ARMv5, and the security properties that can be accomplished with them, driven by a general manifest need for embedded system security. With this basic intent, we did not yet need to identify particular security applications, though lessons learned can be applied to future research with specific applications and possibly different guest OSs.

5.5 Design Recommendations

This section attempts to harvest lessons learned in the implementation, and to make suggestions for moving to more ambitious implementation targets.

5.5.1 Trapping vs. Hypercalls

In our implementation, we took a paravirtualization approach, enabling privileged operations via hypercalls. To minimize modification to the guest OS, it is possible to explore how some of these operations could instead be implemented by trapping privileged instructions. For example, when user-mode execution attempts to enable or disable interrupts, execution would trap to the hypervisor, and the hypervisor could evaluate the safety of the attempt – where did the attempt occur, what is the intended change, any other context – and thereby decide whether or not to carry out the operation. The advantage to be gained is of course less need to port the guest OS, but the disadvantages are equally clear – increased complexity in the hypervisor, and potential performance reduction due to that complexity. The latter point necessitates performance testing. If the performance impact is minimal, then trapping should probably be chosen over hypercalls, due to the reduction in porting requirements for a guest.

The ideal goal would, of course, be to run a completely unmodified guest OS binary above our thin hypervisor. Without implementing binary translation, this seems quite difficult or impossible on ARMv5. However, if hardware support for virtualization on embedded systems evolves along the lines of $x86$ support, a thin embeddedhypervisor stands in a much stronger position to achieve this aim. Just support for a higher hypervisor operating mode (invisible and inaccessible to normal privileged mode), interrupt virtualization, and of course a NX bit would solve most of the challenges of virtualizing FreeRTOS.

5.5.2 Memory protection

The kernel wrappers we used provide the capacity to interpose on kernel entries and exits, and thereby create a virtual kernel mode. It would be much better, however, if a hardware-based solution were found. Wrapper mechanisms are always a potential source of problems, especially if we move to multicore platforms with increased concurrency. Moreover, it was previously noted in our security analysis in section 5.4 that the wrappers are frail in the face of malicious or compromised kernel code. As mentioned numerous times, on newer hardware with a NX bit we will implement kernel integrity protection, which will solve these problems by never allowing non-kernel code to be executed in kernel mode. Both kernel entries and exits can be detected and interposed upon using the NX bit. However, this solution will require eviction of TLB entries, since it involves modifying specific page permissions – thus potentially affecting performance.

It could be advantageous to abandon the wrapper mechanism completely, since it amounts to an unwanted change to the guest OS. However, since the FreeRTOS kernel doesn't use real system calls, the wrapper mechanism is still convenient as a way to restrict which kernel functions are actually accessible to the tasks. Therefore, the wrapper mechanism could be retained only for facilitating secure kernel entry, with kernel exit always being mediated by the hypervisor via a MMU fault when execution branches to non-kernel code.

5.5.3 Security Services

Potential services and approaches were already discussed in section 4.4, and so will not be discussed in great depth here. Our implementation does not yet provide any additional security services beyond memory protection as covered in this chapter. We of course intend to implement kernel integrity protection, we possibly increased task memory protection. Monitoring and other security services can also be implemented, ideally motivated by specific applications and security-critical scenarios.

5.5.4 Multicore

The main challenge for our current thin hypervisor in moving to a multicore platform would be managing and synchronizing state, especially privileged state, for multiple simultaneous execution contexts, and in general dealing with any concurrency issues. This would increase complexity for interrupt handling and interposition in general.

It was already mentioned that the wrapper mechanism is a potential source of problems especially in a concurrent environment, and special care must be taken to ensure that critical points such as the wrappers and the interrupt handling sequence are safe on a multicore system.

Since each core would have its own set of registers and its own CPSR (or equivalent), it is presumable that separate stacks could be used for handling exceptions on the different processors, and that interrupt state would be independent for each core. It is important to ensure that exceptions (such as memory aborts or IRQ/FIQ interrupts) are delivered to the proper core – without being able to ensure proper delivery, exception handling would become much more complicated.

With enough available cores, it could be worth exploring a Sidecore-like architecture, where instead of straight hypercalls, guest code makes sidecalls to a dedicated hypervisor core, to avoid the cost of hypervisor entries and exits. Barring that, it could at least be possible to look into partitioning hypervisor functionality and dedicating it to specific cores, in an effort to increase locality and performance. Any such approach follow the Barrelfish lead of treating a multicore system as a *distributed* system.

5.5.5 Hardware heterogeneity

It would be useful to begin modifying the hypervisor to support multiple hardware architectures in a modular way. Since the hypervisor currently comprises only one C file, this should not be too difficult. A first step would be to isolate the generic portions of the hypervisor from the platform-dependent portions, which may require a study of a range of hardware platforms to assess their commonalities and differences. Then, platform-dependent portions can be logically organized in appropriate components, and abstracted behind a macro or function call interface.

We have made use of the ARM-specific MMU domains feature, and so as part of expanding to other architectures it should be evaluated how equivalent functionality could be achieved with available features.

By exploring other hardware platforms, it might also become evident that the hypercall interface or other facilities need be evolved, and will in general help the hypervisor to become a more useful and versatile entity.

5.5.6 Multiple guests

Our current thin hypervisor supports a single guest, which greatly simplifies requirements. However, it may be useful to investigate support for multiple guests.

The minimum challenges for supporting multiple guests would be to have proper support for scheduling, context switching, and memory protection. If all guests are as simple as FreeRTOS, these challenges are not so burdensome. Guests could even simply run in fixed memory locations, protected by the MMU, still foregoing the need for virtual memory and page swapping if sufficient physical memory were available.

However, a much more useful scenario would be supporting an untrusted open OS alongside a trusted OS, and requirements for virtualizing a mainstream open OS are clearly much greater and well beyond the current capacity of our hypervisor. Here again we note that advances in hardware support could provide the needed aid. Especially if a system were limited to two guests, it is quite conceivable to have one simpler, trusted OS running in a particular memory region and a mainstream OS running atop advanced support for CPU, I/O and state virtualization and two-level page tables.

5.6 Summary

This chapter described our implementation of a thin hypervisor on ARMv5. We successfully implemented a “real” thin hypervisor providing security, albeit for a small guest OS. As the implementation is security-focused, we have assessed the security advantages and weaknesses of our hypervisor, and made security-oriented recommendations for future designs. The implementation provides a clear base for future work.

6. PERFORMANCE TESTS

While our hypervisor is designed to support security for a guest OS, to be usable it must not impact performance unduly. This chapter documents our performance tests.

6.1 Description of Tests

To assess the performance overhead of our hypervisor, we designed four performance tests intended to cover the key performance burdens imposed by the hypervisor: the MMU kernel wrappers, hypercalls, interrupts, and yielding. The hypervisor executes code for each of these occasions. In a MMU kernel wrapper, used when task code calls a kernel API function, hypercalls are issued to enter and exit guest kernel mode, which modifies MMU settings. A normal FreeRTOS distribution would just execute the kernel function. Additionally, a kernel function itself may use hypercalls. In interrupts and yielding, the hypervisor saves context and returns to a kernel handler function, which performs its own activities and then issues a hypercall to restore task context. A normal FreeRTOS interrupt handler or yield call would save context, perform activities, and restore context all in the same function.

We have run each of our tests in preemptive and non-preemptive mode, both on our hypervisor system and a normal FreeRTOS kernel. Naturally, both preemptive and non-preemptive execution, and therefore all the tests, exercise the interrupt handling mechanism. Our timer interrupt mechanism uses an arbitrarily chosen 4ms period.

Our first three tests each use 5 tasks to perform a parallelizable math workload. The workload consists of 10000 work units (each unit consisting of some simple math operations). Tasks take work units in a loop. To ensure consistency, a task takes a work unit inside a critical section, in which interrupts are disabled. It then leaves the critical section and carries out the work unit. Entering and exiting a critical section utilizes hypercalls. In the first test, called "MathTest", the tasks do not yield, and will continue taking work units until preempted. One effect of this is that when using non-preemptive scheduling, one task will simply work through the entire workload by itself. On the other hand, in the second test known as "YieldingMathTest", the tasks yield after completing 5 work units, which results in significant (possibly exaggerated) exercise of the yield mechanism. In the third test, called "WrapperMathTest", the tasks call an arbitrary kernel function after completing 5 work units, resulting in significant exercise of the wrapper mechanism.

The fourth test, "FlashTest", simulates the flashing of a bank of LED lights by having each of 5 tasks in a loop periodically activate and deactivate their own LED light by writing to a certain memory region, as if the LED bank were a memory mapped device. The 5 LEDs flash with a period of 10, 20, 30, 40, and 50 timer ticks, respectively. The test runs until the fastest LED has flashed 20 times.

To repeatedly flash its LED on and off, a task's loop calls a kernel function to delay execution (that is, to sleep) for half of its flash period, then activates its LED, then calls the same kernel function to delay execution for half its period again, then deactivates its LED. This exercises the kernel wrappers, and the yielding mechanism. When no flash task is ready to execute because their delay/sleep times have not expired, the idle task runs. The FreeRTOS idle task simply loops repeatedly until another task is ready to be scheduled.

6.2 Results

We have recorded results in OVP-simulated instruction counts, and have found that, in spite of significant usage of hypervisor mechanisms in the various tests, use of the hypervisor engenders little overhead. These results are auspicious and encouraging, and provide strong initial evidence that security-supportive thin hypervisors are reasonable to use on embedded platforms. Results are summarized in table 6.1.

Test	**Instruction count with hypervisor**	**Instruction count without hypervisor**	**Overhead %**
MathTest, Preemptive	10,696,343	10,472,960	2.133
MathTest, Non-preemptive	10,692,268	10,469,648	2.126
YieldingMathTest, Preemptive	11,885,640	11,109,395	6.987
YieldingMathTest, Non-preemptive	11,880,823	11,105,748	6.979
WrapperMathTest, Preemptive	11,128,362	10,612,814	4.858
WrapperMathTest, Non-preemptive	11,124,540	10,609,665	4.853
FlashTest, Preemptive	163,411,309	163,406,633	0.003
FlashTest, Non-preemptive	163,411,862	163,406,780	0.003

Tab. 6.1: Test Results, in OVP-simulated instruction counts

Observe that the tests with the most exaggerated usage of hypervisor mechanisms, YieldingMathTest and WrapperMathTest, result in the greatest overhead, but even this overhead is still quite acceptable.

The results don't necessarily offer direct feedback for future design recommendations, but they do provide a benchmark and standard by which future implementations can be measured, and they overall encourage continued progress.

7. CONCLUSIONS AND FUTURE WORK

7.1 Future Work

Future work is actually one of the most important aspects of this thesis, since it was a foundational project intended to set the scene for continued efforts. Many significant directions for future work have already been covered (notably in section 5.5), including:

- Security services, including kernel integrity protection and others
- Enhanced memory protection with less reliance on wrappers
- Investigating support for multiple cores
- Branching out to additional hardware platforms
- Using trapping to reduce the need for hypercalls

Additionally, formal verification of implementations is a desired goal, and more thorough security analysis in general and investigation of more specific security applications is encouraged.

While we have emphasized repeatedly that evolving hardware support would be of great benefit and potentially engender many new applications, research should still proceed on currently available hardware – both because it is not clear if or when more advanced hardware will become available, and innumerable new and old embedded systems will remain or be deployed without such advanced support.

7.2 Conclusions

This thesis has demonstrated that embedded thin hypervisors are a viable, useful platform for embedded system security. With extensive background research into virtualization, embedded, and multicore technology, it was shown that virtualization is a strong security enabler and demonstrates potential to manage hardware complexity, and it was motivated that embedded virtualization is on the rise and is already being used to answer embedded systems' security needs. Our implementation shows that thin embedded hypervisors can be small and practical, bringing security to guests with acceptable performance. Both the background study and implementation have yielded substantial artifacts upon which subsequent work can be built. With recommendations for future work well outlined and lessons learned from our implementation and background research, this thesis has achieved the goals established in section 1.4, and prepared the way for meaningful future efforts.

BIBLIOGRAPHY

[1] Onur Aciicmez, Afshin Latifi, Jean-Pierre Seifert, and Xinwen Zhang, *A Trusted Mobile Phone Prototype*, 5th IEEE Consumer Communications and Networking Conference (CCNC) (Las Vegas, NV, USA), January 2008.

[2] Keith Adams and Ole Agesen, *A Comparison of Software and Hardware Techniques for x86 Virtualization*, The 12th International Conference on Architectural Support For Programming Languages and Operating Systems (ASPLOS), San Jose, CA, USA, October 2006, pp. 2–13.

[3] Advanced Micro Devices, Inc., *AMD64 Virtualization Codenamed Pacifica Technology: Secure Virtual Machine Architecture Reference Manual*, `http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf` (last accessed 24 Sept 2009), May 2005.

[4] ______, *AMD-V Nested Paging*, available at `http://developer.amd.com/assets/NPT-WP-11-final-TM.pdf`, July 2008.

[5] ______, *Live Migration with AMD-V Extended Migration Technology*, available at `http://developer.amd.com/assets/43781-3.00-PUB_Live-Virtual-Machine-Migration-on-AMD-processors.pdf`, April 2008.

[6] ______, *AMD HyperTransport Technology page*, `http://www.amd.com/us/products/technologies/hypertransport-technology/Pages/hypertransport-technology.aspx` (accessed 12 Oct 2009), 2009.

[7] ______, *AMD I/O Virtualization Technology (IOMMU) Specification*, `http://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf` (last accessed 24 Sept 2009), February 2009.

[8] ______, *AMD-V technology page*, `http://www.amd.com/us/products/technologies/virtualization/Pages/amd-v.aspx`, 2009.

[9] Nidhi Aggarwal and Parthasarathy Ranganathan, *Configurable Isolation: Building High Availability Systems with Commodity Multi-Core Processors*, The 34th Annual ACM SIGARCH International Symposium on Computer Architecture (ISCA), June 2007.

[10] Alessandro Perilli, *VMWare acquires Trango, the hypervisor is ready to go mobile*, available at `http://www.virtualization.info/2008/11/vmware-acquires-trango-hypervisor-is.html` (accessed 10 October 2009), November 10 2008.

[11] Robert D. Blumofe and, *Cilk: An Efficient Multithreaded Runtime System*, ACM SIGPLAN Notices **30** (1995), no. 8, 207–216.

[12] Ross Anderson, *Security Policies*, available at `http://www.cl.cam.ac.uk/~rja14/Papers/security-policies.pdf`, accessed 10 October 2009.

[13] ARM Limited, *ARM PrimeCell Infrastructure AMBA 3 TrustZone Protection Controller (BP147) Revision: r0p0*, available at `http://infocenter.arm.com/help/topic/com.arm.doc.dto0015a/DTO0015_primecell_infrastructure_amba3_tzpc_bp147_to.pdf` (accessed 3 November 2009), 2004.

[14] ———, *PrimeCell Infrastructure AMBA 3 AXI TrustZone Memory Adapter (BP141) Revision: r0p0*, available at `http://infocenter.arm.com/help/topic/com.arm.doc.dto0015a/DTO0015_primecell_infrastructure_amba3_tzpc_bp147_to.pdf` (accessed 3 November 2009), 2004.

[15] ———, *ARM Architecture Reference Manual*, available at `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0100i/index.html` (accessed 25 October 2009), 2005.

[16] ———, *ARM926EJ-S Technical Reference Manual, rev r0p5*, available at `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0198e/index.html` (accessed 25 October 2009), 2008.

[17] ———, *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*, placeholder at `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406b/index.html` (accessed 2 November 2009), document must be obtained via an ARM support website account, 2009.

[18] ———, *ARM Security Technology: Building a Secure System using TrustZone Technology*, available at `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf` (last accessed 25 September 2009), 2009.

[19] ———, *ARM TrustZone System Design page*, available at `http://www.arm.com/products/security/trustzone/systemdesign.html` (last accessed 25 September 2009), 2009.

[20] François Armand and Michel Gien, *A Practical Look at Micro-Kernels and Virtual Machine Monitors*, Proceedings of the 6th Consumer Communications and Networking Conference (IEEE CCNC '09) (Las Vegas, NV, USA), January 2009.

[21] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia, *Dynamo: A Transparent Dynamic Optimization System*, Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI) (Vancouver, British Columbia, Canada), 2000.

[22] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigal Zemach, *IA-32 execution layer: a two-phase*

dynamic translator designed to support IA-32 applications on Itaniumő-based systems, Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, December 2003, pp. 191–201.

[23] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield, *Xen and the Art of Virtualization*, Proceedings of the nineteenth ACM symposium on Operating systems principles (New York), Operating Systems Review, vol. 37, 5, ACM Press, October 19–22 2003, pp. 164–177.

[24] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, , and Akhilesh Singhania, *The Multikernel: A new OS architecture for scalable multicore systems*, Proceedings of the 22nd ACM Symposium on OS Principles (SOSP '09) (Big Sky, MT, USA), October 2009.

[25] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs, *Your computer is already a distributed system. Why isnŠt your OS?*, Proceedings of the 12th USENIX Workshop on Hot Topics in Operating Systems (Monte Verità, Switzerland), May 2009.

[26] Steven M. Bellovin, *Virtual Machines, Virtual Security?*, CACM: Communications of the ACM **49** (2006), no. 10, 104.

[27] Christer Bengtsson, Mats Brorsson, Håkan Grahn, Erik Hagersten, Bengt Jonsson, Christoph Kessler, Björn Lisper, Per Stenström, and Bertil Svensson, *Multicore computing Ů the state of the art*, available at `http://eprints.sics.se/3546/01/SMI-MulticoreReport-2008.pdf`, accessed 3 October 2009, December 3 2008.

[28] Common Criteria Development Board and Common Criteria Maintenance Board, *Common Criteria for Information Security Evaluation v3.1 release 3*, Members of the Common Criteria Recognition Agreement, July 2009, available at `http://www.commoncriteriaportal.org/thecc.html` (last accessed 19 September 2009).

[29] Jörg Brakensiek, Axel Dröge, Martin Botteck, Hermann Härtig, and Adam Lackorzynski, *Virtualization as an Enabler for Security in Mobile Devices*, First Workshop on Isolation and Integration in Embedded Systems (IIES '08) (Glasgow, UK), April 2008.

[30] Sergey Bratus and Michael E. Locasto, *Traps, Events, Emulation, and Enforcement: Managing the Yin and Yang of Virtualization-Based Security*, Proceedings of the 1st ACM Workshop on Virtual Machine Security (VMSEC '08) (Fairfax, VA, USA), October 2008.

[31] Gilles Chanteperdrix and Richard Cochran, *The ARM Fast Context Switch Extension for Linux*, available at `http://lwn.net/images/conf/rtlws11/papers/proc/p01.pdf` (accessed 25 October 2009).

[32] David Chaum and Torben Pryds Pederson, *Wallet Databases with Observers*, Advances in Cryptology Ű CRYPTO 1992, LNCS 740, 1993, pp. 89 – 105.

[33] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports, *Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems*, Proceedings of the 13th Annual International ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2008.

[34] Ken Cheung, *Embedded Multi-Core CPU Market (VDC Study)*, `http://edablog.com/2007/11/13/vdc-multi-core/` (accessed 11 October 2009), 2009.

[35] David Chisnall, *The Definitive Guide to the Xen Hypervisor*, Prentice Hall, 2008.

[36] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, , and Dawson Engler, *An Emprical Study of Operating System Errors*, Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP), 2001.

[37] George Coker, *Xen Security Modules (XSM)*, Xen Summit 2007 presentation, `http://www.xen.org/files/xensummit_4/xsm-summit-041707_Coker.pdf` (accessed Oct 4 2009), April 2007.

[38] Landon P. Cox and Peter M. Chen, *Pocket Hypervisors: Opportunities and Challenges*, Eighth IEEE Workshop on Mobile Computing Systems and Applications, March 2007.

[39] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve, *Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems*, 21st ACM Symposium on Operating System Principles (SOSP '07), October 2007, pp. 351–366.

[40] Leonardo Dagum and Ramesh Menon, *OpenMP: An Industry Standard API for Shared-Memory Programming*, IEEE Computational Science and Engineering **5** (1998), no. 1, 46–55.

[41] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen, *ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay*, Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI), 2002, (published in special issue of ACM SIGOPS Operating Systems Review, volume 36, winter 2002).

[42] George W. Dunlap, Dominic G. Lucchetti, Peter M. Chen, and Michael A. Fetterman, *Execution Replay for Multiprocessor Virtual Machines*, Proceedings of the 2008 ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments (VEE '08) (Seattle, WA, USA), March 5-7 2009.

[43] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole, *Exokernel: An Operating System Architecture for Application-Level Resource Management*, Proceedings of the 15th Symposium on Operating System Principles(SOSP 1995), December 1995.

[44] FreeRTOS/Real Time Engineers Ltd., *FreeRTOS web page*, `http://www.freertos.org/`, (accessed 7 Nov 2009), 2009.

[45] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh, *Terra: A Virtual Machine-Based Platform for Trusted Computing*, Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003), October 2003.

[46] Tal Garfinkel and Mendel Rosenblum, *A Virtual Machine Introspection Based Architecture for Intrusion Detection*, Proc. Network and Distributed Systems Security Symposium, February 2003.

[47] ———, *When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments*, Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X), May 2005.

[48] Robert P. Goldberg, *Survey of Virtual Machine Research*, IEEE Computer (1974), 34–45, available at `https://agora.cs.illinois.edu/download/attachments/10454931/goldberg74.pdf`.

[49] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos, and Dan Magenheimer, *Are Virtual Machine Monitors Microkernels Done Right?*, Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems (Santa Fe, NM, USA), June 12 - 15 2005.

[50] Brian Hay and Kara Nance, *Forensics Examination of Volatile System Data Using Virtual Introspection*, ACM SIGOPS Operating Systems Review **42** (2008), no. 3, 74–82.

[51] Gernot Heiser, *The Role of Virtualization in Embedded Systems*, First Workshop on Isolation and Integration in Embedded Systems (IIES '08) (Glasgow, UK), April 2008.

[52] ———, *Multi-part blog response to Armand and Gien 2009 paper*, part 1: `http://www.ok-labs.com/blog/entry/benchmarks-how-not-to-do-them/`, part 2: `http://www.ok-labs.com/blog/entry/living-in-the-past`, part 3: `http://www.ok-labs.com/blog/entry/microkernels-101`, part 4: `http://www.ok-labs.com/blog/entry/some-get-it-some-dont/`, accessed 6 October 2009, 2009.

[53] Gernot Heiser, Volkmar Uhlig, and Joshua LeVasseur, *Are Virtual-Machine Monitors Microkernels Done Right?*, Operating Systems Review **40** (2006), no. 1, 95–99.

[54] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter, *The Nizza Secure-System Architecture*, Proceedings of CollaborateCom '05 (San Jose, CA, USA), December 2005.

[55] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim, *Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones*, 5th IEEE Consumer Communications and Networking Conference (CCNC 2008) (Las Vegas, NV, USA), January 2008.

[56] Intel Corporation, *First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)*, available at `http://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf` (last accessed 19 September, 2009), 2008.

[57] ———, *Intel Virtualization Technology for Directed I/O*, available at `http://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf` (last accessed 24 September, 2009), September 2008.

[58] ———, *Intel QuickPath Technology page*, `http://www.intel.com/technology/quickpath/` (accessed 12 October, 2009), 2009.

[59] Intel Corporation and NTT DoCoMo, *Open and Secure Terminal Initiative, Architecture Specification*, available at `http://www.nttdocomo.co.jp/english/corporate/technology/osti/` (accessed 4 October, 2009), 2006.

[60] Megumi Ito and Shuichi Oikawa, *Lightweight Shadow Paging for Efficient Memory Isolation in Gandalf VMM*, 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC) (Orlando, FL, USA), May 2008, pp. 508–515.

[61] Xuxian Jiang and Xinyuan Wang, *Stealthy Malware Detection Through VMM-Based 'Out of the Box' Semantic View Reconstruction*, Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07) (Alexandria, VA, USA), October 29 – November 2 2007.

[62] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen, *Detecting Past and Present Intrusions through Vulnerability-Specific Predicates*, Proceedings of the 20th Symposium on Operating System Principles(SOSP 2005) (Brighton, UK), October 23–26 2005, pp. 91–104.

[63] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood, *seL4: Formal Verification of an OS Kernel*, Proceedings of the 22nd ACM Symposium on OS Principles (SOSP '09) (Big Sky, MT, USA), October 2009, available at `http://ertos.nicta.com.au/publications/papers/Klein_EHACDEEKNSTW_09.pdf` (accessed 26 September, 2009).

[64] Kirk L. Kroeker, *The Evolution of Virtualization*, Communications of the ACM **52** (2009), no. 3, 18–20.

[65] Sanjay Kumar, Ada Gavrilovska, Karsten Schwan, and Srikanth Sundaragopalan, *C-CORE: Using Communication Cores for High Performance Network Services*, Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (NCA '05), 2005.

[66] Sanjay Kumar, Himanshu Raj, Karsten Schwan, and Ivan Ganev, *Re-architecting VMMs for Multicore Systems: The Sidecore Approach*, Proceedings of the 2007 Workshop on the Interaction between Operating Systems and Computer Architecture, June 2007.

[67] L4HQ.org, *L4Linux info page*, http://l4linux.org/ (accessed 28 Sept 2009), 2009.

[68] L4Ka.org, *L4Ka pre-virtualization page*, http://l4ka.org/projects/virtualization/afterburn/ (accessed 28 Sept 2009), 2009.

[69] Sung-Min Lee, Sang-Bum Suh, Bokdeuk Jeong, and Sangdok Mo, *A Multi-Layer Mandatory Access Control Mechanism for Mobile Devices Based on Virtualization*, 5th IEEE Consumer Communications and Networking Conference (CCNC 2008) (Las Vegas, NV, USA), January 2008, pp. 251–256.

[70] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie, *Hypervisor Support for Identifying Covertly Executing Binaries*, Proceedings of the 17th USENIX Security Symposium (San Jose, CA, USA), July 28 – August 1 2008, pp. 243–258.

[71] Gabriel H. Loh, *3d-stacked Memory Architectures for Multi-core Processors*, Proceedings of the 35th International Symposium on Computer Architecture (ISCA '08) (Washington, DC, USA), 2008, pp. 453–464.

[72] Gil Neiger, *Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization*, Intel Technology Journal **10** (2006), no. 3, 167–177.

[73] NICTA, *Iguana project page*, http://www.ertos.nicta.com.au/software/kenge/iguana-project/latest/ (last accessed 19 September, 2009), 2009.

[74] ———, *NICTA announces world-first research breakthrough*, seL4 verification NICTA press release, available at http://www.nicta.com.au/news/home_page_content_listing/world-first_research_breakthrough_promises_safety-critical_software_of_unprecedented_reliability (last accessed 19 September, 2009), August 2009.

[75] Open Kernel Labs, *Open Kernel Labs Secure Hypercell Technology page*, http://www.ok-labs.com/solutions/secure-hypercell-technology (accessed 28 September, 2009), 2009.

[76] ———, *Open Kernel Labs web page*, http://www.ok-labs.com/ (accessed 10 October, 2009), 2009.

[77] Open Mobile Terminal Platform, *OMTP Advanced Trusted Environment: OMTP TR1 version 1.1*, available at http://www.omtp.org/Publications/Display.aspx?Id=48608b5d-ddeb-4c7e-bb90-a409d119f9a4 (accessed 4 October, 2009), May 28 2009.

[78] ———, *OMTP web page*, www.omtp.org (accessed 4 October, 2009), 2009.

[79] Open Virtual Platforms, *OVP website*, http://www.ovpworld.org/ (accessed 25 December, 2009), 2009.

[80] OpenVZ project, *OpenVZ Wiki Main Page*, http://wiki.openvz.org/Main_Page (accessed 28 Sept 2009), 2009.

[81] Bryan D. Payne, Martim Carbone, and Wenke Lee, *Secure and Flexible Monitoring of Virtual Machines*, Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007), December 2007.

[82] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee, *Lares: An Architecture for Secure Active Monitoring Using Virtualization*, Proceedings of the IEEE Symposium on Security and Privacy, May 2008.

[83] Steven Pope and David Riddoch, *Virtualization and multicore x86 CPUs*, EDN (2008), available at http://www.edn.com/article/CA6584878.html, accessed 11 October 2009.

[84] Gerald J. Popek and Robert P. Goldberg, *Formal Requirements for Virtualizable Third Generation Architectures*, Communications of the ACM **17** (1974), no. 7, 412–421.

[85] Dan Ports and Tal Garfinkel, *Towards Application Security On Untrusted Operating Systems*, USENIX Workshop on Hot Topics in Security (HOT-SEC), August 2008.

[86] Bratin Saha et al., *Enabling Scalability and Performance in a Large Scale CMP Environment*, Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (Lisbon, Portugal), 2007, pp. 73 – 86.

[87] Reiner Sailer, Treng Jaeger, Enriquillo Valdez, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn, *Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor*, Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC '05), December 05–09 2005, pp. 276–285.

[88] Karen Scarfone and John Padgette, *Bluetooth Security Guide: Recommendations of the National Institute of Standards and Technology*, NIST Special Publication 800-121, available at http://csrc.nist.gov/publications/nistpubs/800-121/SP800-121.pdf (last accessed 20 September 2009), September 2008.

[89] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs, *Embracing diversity in the Barrelfish manycore operating system*, Proceedings of the Workshop on Managed Many-Core Systems (MMCS '08) (Boston, MA, USA), June 2008.

[90] Michael D. Schroeder and Jerome H Saltzer, *A hardware architecture for implementing protection rings*, Proceedings of the 3rd ACM Symposium on Operating System Principles (SOSP '71) (Palo Alto, CA, USA), 1971.

[91] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig, *SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes*, Proceedings of the 21st Symposium on Operating System Principles(SOSP 2007) (Stevenson, Washington, USA), October 14–17 2007.

[92] Takahiro Shinagawa et al., *BitVisor: A Thin Hypervisor for Enforcing I/O Device Security*, Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments (VEE '09) (Washington, D.C., USA), March 2009.

[93] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne, *Operating System Concepts, 7th Ed.*, Wiley, 2004.

[94] Andrew N. Sloss, Dominic Symes, and Chris Wright, *ARM System Developer's Guide*, Elsevier/Morgan-Kaufmann, 2004.

[95] James E. Smith and Ravi Nair, *The Architecture of Virtual Machines*, IEEE Computer **38** (2005), no. 5, 32–38.

[96] ———, *Virtual Machines: Versatile Platforms for Systems and Processes*, Morgan Kaufmann/Elsevier, 2005.

[97] Stephen Soltesz, Herbert Potzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson, *Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors*, Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys 2007) (Lisbon, Portugal), March 2007, pp. 275–287.

[98] Ray Spencer, Peter Loscocco, Stephen Smalley, Mike Hibler, David Andersen, and Jay Lepreau, *The flask security architecture: system support for diverse security policies*, Proceedings of the 8th conference on USENIX Security Symposium, 1999, p. 11.

[99] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim, *Virtualizing I/O Devices on VMware WorkstationŠs Hosted Virtual Machine Monitor*, Proceedings of the 2001 USENIX Annual Technical Conference (Boston, MA, USA), June 25–30 2001, available at `http://www.vmware.com/vmtn/resources/530` (last accessed 19 September, 2009).

[100] Sang-Bum Suh and Joo-Young Hwang et al, *Computing State Migration between Mobile Platforms for Seamless Computing Environments*, 5th IEEE Consumer Communications and Networking Conference (CCNC 2008) (Las Vegas, NV, USA), January 2008.

[101] Sang-Bum Suh, Sung-Min Lee, Sangdok Mo, Bokdeuk Jeong, Joo-Young Hwang, Chan-Ju Park, Sung-Kwan Heo, Junghyun Yoo, Jae-Min Ryu, Chul-Ryun Kim, Seong-Yeol Park, Jae-Ra Lee, Il-Pyung Park, , and Hosoo Lee, *Demonstration of the Secure VMM for Beyond 3G Mobile Terminal*, 5th IEEE Consumer Communications and Networking Conference (CCNC 2008) (Las Vegas, NV, USA), January 2008.

[102] Tilera, *Tilera processor page*, `http://www.tilera.com/products/processors.php`, accessed 11 October 2009, 2009.

[103] Trusted Computing Group, *TCG Mobile Reference Architecture*, available at `http://www.trustedcomputinggroup.org/resources/mobile_phone_work_group_mobile_reference_architecture`, accessed 28 Sept 2009, June 12 2007.

[104] TU Dresden, *The L4 µ-Kernel Family*, http://os.inf.tu-dresden.de/L4/ (last accessed 19 September, 2009).

[105] Steven J. Vaughan-Nichols, *New Approach to Virtualization is Lightweight*, IEEE Computer **39** (2006), no. 11, 12–14.

[106] ———, *Virtualization Sparks Security Concerns*, IEEE Computer **41** (2008), no. 8, 13–15.

[107] VirtualLogix, *VirtualLogix web page*, http://www.virtuallogix.com/, 2009.

[108] VMWare, *Performance Evaluation of AMD RVI Hardware Assist*, available at http://www.vmware.com/resources/techresources/1079 (accessed 22 September, 2009).

[109] ———, *Transparent Previrtualization info page*, http://www.vmware.com/interfaces/paravirtualization.html (accessed 28 September, 2009).

[110] ———, *VMWare ESX and ESXi product page*, http://www.vmware.com/products/esx/index.html (accessed 19 September, 2009).

[111] ———, *VMWare Workstation product page*, http://www.vmware.com/products/workstation/index.html (accessed 19 September, 2009).

[112] ———, *VMWare Mobile page*, available at http://www.vmware.com/technology/mobile/ (accessed 10 October, 2009), 2009.

[113] John Watson, *VirtualBox: Bits and Bytes Masquerading as Machines*, Linux Journal (2008), available at http://www.linuxjournal.com/article/9941 (accessed 19 September, 2009).

[114] Robert N. M. Watson, *Exploiting Concurrency Vulnerabilities in System Call Wrappers*, 1st USENIX Workshop on Offensive Technologies, 2007.

[115] Clark Weissman, *BLACKER: security for the DDN examples of A1 security engineering trades*, Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy (Oakland, CA, USA), May 4-6 1992, pp. 286–292.

[116] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi, *Dynamic Heterogeneity and the Need for Multicore Virtualization*, ACM SIGOPS Operating Systems Review **43** (2009), no. 2, 5–14.

[117] ———, *Mixed-Mode Multicore Reliability*, Proceedings of the 14th Annual International ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09), March 2009.

[118] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble, *Denali: Lightweight Virtual Machines for Distributed and Networked Applications*, Proceedings of the USENIX Annual Technical Conference, 2002.

[119] wikipedia, *Ring (computer security)*, available at http://en.wikipedia.org/wiki/Ring_(computer_security), last modified on 21 August 2009 (last accessed 19 September, 2009), 2009.

[120] ———, *Cell (microprocessor)*, available at `http://en.wikipedia.org/wiki/Cell_(microprocessor)`, last modified on 12 February 2010 (last accessed 13 February 2010), 2010.

[121] ———, *Trusted Platform Module*, available at `http://en.wikipedia.org/wiki/Trusted_Platform_Module`, last modified on 13 February 2010 (last accessed 13 February 2010), 2010.

[122] Johannes Winter, *Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms*, Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing (Fairfax, VA, USA), October 2008, pp. 21–30.

[123] Xen ARM Project, *Xen ARM Project page*, `http://wiki.xensource.com/xenwiki/XenARM` (last accessed 20 September 2009), 2009.

[124] Min Xu, Xuxian Jiang, Ravi Sandhu, and Xinwen Zhang, *Towards a VMM-based Usage Control Framework for OS Kernel Integrity Protection*, Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT 2007) (Sophia Antipolis, France), June 20–22 2007.

[125] Jisoo Yang and Kang G. Shin, *Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis*, Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08) (Seattle, WA, USA), March 05-07 2008, pp. 71–80.

[126] Xinwen Zhang, Onur Aciicmez, and Jean-Pierre Seifert, *A Trusted Mobile Phone Reference Architecture via Secure Kernel*, Proceedings of the 2nd ACM Workshop on Scalable Trusted Computing) (Alexandria, VA, USA), November 2007.