SICS Technical Report
T2009:14A
ISSN: 1100-3154

Secure Virtualization and Multicore Platforms State-of-the-Art report

by

Heradon Douglas and Christian Gehrmann

Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, SWEDEN

2009-12-08

Secure Virtualization and Multicore Platforms

State-of-the-art report[1]

Heradon Douglas and Christian Gehrmann

SICS

Table of Contents

[1] We gratefully acknowledge the support of the Swedish Governmental Agency for Innovation Systems, VINNOVA, under grant 2009-01753.

Summary

Virtualization, the use of hypervisors or virtual machine monitors to support multiple 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.

This report gives an overview of the state-of-the art regarding virtualization, multi-core systems and *security*. The report is a major deliverable to the SVaMP project pre-study and will serve as a basis for in-depth analysis of a selected set of multicore target systems in the second phase of the project. Starting from the state-of-the art designs described in this report, the second phase of the project will also identify design patterns and derive system models for secure virtualized multicore systems.

Abbreviations

ABI	Application Binary Interface
API	Application Programming Interface
ASID	Address Space Identifier
CPU	Central Processing Unit
DMA	Direct Memory Access
DMAC	DMA Controller
DMR	Dual-Modular Redundancy
DRM	Digital Rights Management
EPT	Extended Page Table
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
NUMA	Non-Uniform Memory Architecture
SPMD	Single-Program, Multiple Data
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TLB	Translation Lookaside Buffer
TPR	Task Priority Register
VBAR	Vector Base Address Register

VM	Virtual Machine
VMCS	Virtual Machine Control Structure
VMI	VM introspection
VMM	Virtual Machine Monitor
VPID	Virtual Process Identifier

1. Introduction

This report gives an overview of virtualization technologies and research recent research results in the area. The purpose with the report is to give the foundation for the SVaMP project platform analysis, requirements and modeling work.

The report is organized as follows. First, in Section 2, we give basic definitions regarding virtualization and the technologies behind virtualization. Section 3 discusses different hypervisor/virtual machine monitor architectures. In Section 4, we explain the major different motivations for introducing virtualization in a system. Section 5 describes important virtualization enabling hardware architectures. In Section 6, we discuss different hypervisor protected software architectures. The focus is well known design and description of hypervisor based platform security services. Finally, in Section 7, an overview of multicore systems and issues are given and in particular we treat virtualization in relation to mutlicore systems.

2. Virtualization technologies

2.1. What is virtualization?

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.

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. 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.

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 the *guest*.

2.2. Virtualization Basics

2.2.1. Interfaces

An excellent overview of virtual machines is found here [79], and in a book by the same authors ([80]). The article discusses, 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 follows.

2.2.1.1. Instruction Set Architecture (ISA)

The 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.

2.2.1.2. 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 [30].

2.2.1.3. Applicatoin Binary Interface (ABI)

The 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.

2.2.1.4. Application Programming Interface (API)

An 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 usually work out so ideally.) As previously mentioned, however, all software is executed through the ISA in the end -- meaning that any API or application will have to be recompiled, even if it doesn't have to be reimplemented, as it moves to a new platform.

2.2.1.5. *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 understood 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 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 [79] -- 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 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.

2.3.Types of virtualization

There are two most prominent basic types of virtualization -- process virtualization and system virtualization [79]. Also noteworthy topics are binary translation, paravirtualization, and previrtualization (approaches to system and process virtualization), as well as containers, a more lightweight relative of system virtualization. These concepts illustrate basic types of virtualization currently in use.

2.3.1. Process virtualization

Process-level virtualization is a fundamental concept in virtually every modern mainstream computer system. In process virtualization, an operating system virtualizes the memory address space, central processing unit (CPU), 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 [79].

The operating system manages the 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 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 pages in processes' virtual address space to actual physical memory pages. 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 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 is thus the outward-facing facade of a complex internal system of technologies.

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.

2.3.2. System virtualization

In contrast to process virtualization, in system virtualization an entire system is virtualized, enabling multiple virtual systems to run isolated alongside each other [79]. 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 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).

Note here that the VMM may divide the system resources in different ways. 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. (This 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 [79].

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 [89], as well as due to commodity hardware support discussed in section 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 [54].

2.3.3. 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 translation is sometimes known as a ``Whole System" VMM [79].

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 ([18]), 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. However, as already mentioned, ISA translation will likely degrade performance.

A virtualization system may translate or optimize the guest ISA in different ways [79]. 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 ([17]).

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.

2.3.4. Paravirtualization

In relation to ISA translation, paravirtualization represents a different, possibly complementary approach to virtualization. In paravirtualization, the 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 [89]. The Denali system uses paravirtualization in support of a lightweight, multi-VM environment suited for networked application servers [100].

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 software 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.

2.3.5. 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). 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 [92], 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 [58].

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.

2.3.6. 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 [89]. An example is the OpenVZ project for Linux [65], or the system proposed in [81].

Applications running in the containers must run natively on the underlying OS -- containers do not promote 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.

2.4.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 they may be not so amenable to virtualization. For instance, exokernels [37] 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.

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 as illustrated in Figure 1.

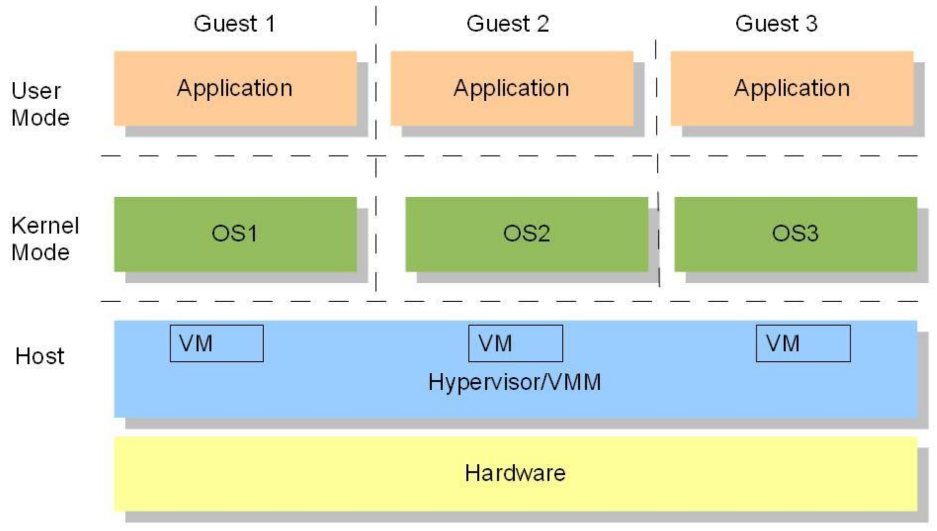

Figure 1: Typical VM software architecture

3.1. Traditional hypervisors

Traditional hypervisors, such as Xen [19] and VMWare ESX [93], run on the bare metal and support multiple virtual machines. This is the classic type of hypervisor, dating back to the 1970s [41], 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 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 [29] (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.

3.1.1. 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 [75]. 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 [101]. 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 5 on virtualization hardware support.

3.2. Hosted hypervisors

A hosted hypervisor, such as VirtualBox[95] or VMWare Workstation [83][94], 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 [79]. 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.

3.3.Microkernels

Microkernels such as L4 [88] 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 [45]. (However, not everyone agrees on that last point [16][42].) 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 [62]. The virtual machine runs atop the provisioning layer. Alternatively, an OS can be paravirtualized to run directly atop the microkernel, as in L4Linux [57].

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 [53][63]. This may be of special interest to parties building systems for certification by the Common Criteria [24], 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 ([64]) 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. (This is similar to the device driver approach in Xen.)

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 [93], 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'' [53], 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 [77] -- 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 and possibly ISA virtualization layer between a guest OS and the hardware.

There are numerous thin hypervisor architectures in the literature, including the aforementioned SecVisor [77] and BitVisor [78]. 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 security architecture.

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." [79]. Virtualization yields numerous advantages, some of which are discussed in the following sections.

4.1. Isolation

The fundamental 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 capture any isolated domain, such as a microkernel address space.) This leads to robustness and security.

It is worth mentioning nowadays that, 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.

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 addresses this problem by placing a trustworthy hypervisor at the highest privilege on the system and running virtual machines at reduced privilege. 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.

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.

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.

4.5. Management

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

4.5.1. Consolidation/Resource sharing

Virtualization can increase efficiency in resource utilization via consolidation [44][54]. 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 [41].

4.5.2. 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) [44][54].

4.5.3. Migration

Virtual machines can be migrated live (that is, in the middle of execution) between systems. Research has been done to support virtualization-based migration even on mobile platforms [84]. In theory, computing context could be migrated between any compatible device capable of virtualization. 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 further enabling 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.

4.6. Security

Last but definitely not least, virtualization can provide security advantages, and is moving more and more in this direction [54]. 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.

4.7. Typical Virtualization Scenarios

4.7.1. 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.

4.7.2. 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 [61]. 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.

4.7.3. 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.

4.7.4. 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 [85], such as an employee smartphone containing isolated home and work environments [54]. 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 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 [2], VirtualLogix [3], and now VMWare [4]. It has been found to be not unduly onerous to port virtualization architectures to mobile platforms [25], and open systems such as the L4 microkernel [88] and Xen on ARM [46][103] afford open, low-cost solutions.

Therefore, the benefits of virtualization already discussed can be brought to mobile systems, in addition to 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 [44]. Isolated virtual machines can serve as mobile testbed components or nodes in opportunistic mobile sensor networks [32], and support heterogeneous application

[2]http://www.ok-labs.com/

[3]http://www.virtuallogix.com/

[4]http://www.vmware.com/technology/mobile/

environments [44]. Modularity and live system migration is of special interest in the mobile environment. 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 [27], 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.

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 privileged 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, so called *shadow page tables* as illustrated in Figure 2. In another example, regarding device I/O where devices use 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.

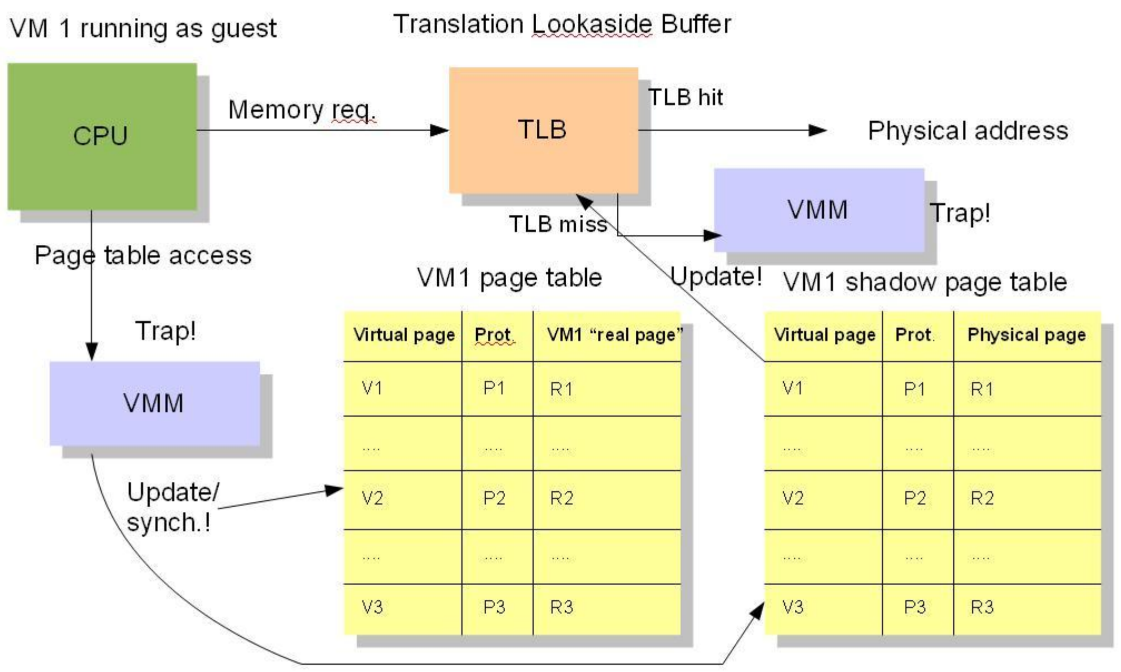

Figure 2: Usage of shadow page tables

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 [61], 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 [49]. 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 [61]. There are other software-based solutions for enabling virtualization without hardware support, such as the ``Gandalf'' VMM [50] 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.

5.1.Basic virtualization requirements

Popek and Goldberg outlined basic requirements for a system to support virtual machines in 1974 [69]. 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.

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 *x*86 architecture include (as described in [61]):

- 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 -- bon 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.

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.

5.3.1. 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* [61]. (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 guest 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 [61]. 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) [61]. 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 shadow value below a threshold value also set in the VMCS [61].

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) [44], 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 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 Identifier (VPID), 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 ([61]), 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.

5.3.2. 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 [49]. It provides several key features (as described in [49]):

- *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 [49]. 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'' [49]) 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 [49]. 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.

5.4.AMD-V

AMD's version of virtualization support is entitled AMD-V [6], and offers comparable support for CPU, memory, and I/O virtualization, and migration.

5.4.1. CPU

AMD-V incorporates a new bit of CPU state entitled ``guest mode'' [7] 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 [7]. 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.

5.4.2. Memory

Rapid Virtual Indexing, also known as Nested Paging, is AMD's version of hardware support for virtualization memory management [5]. 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) [91].

Address Space Identifiers (ASIDs) are used to eliminate the need for TLB flushes when switching to a new VM [5]. 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.

5.4.3. Migration

AMD-V Extended Migration also provides hardware support for live migration of VMs between AMD Opteron processors in a pool of systems [8]. 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.

5.4.4. 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 [4]. 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 [4].

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 [13]. 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 *x*86 virtualization support packages. It is described in detail in [13], and also in [11].

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 ([12][15]), 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 insecure 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 in normal operating systems. 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 always traps to Monitor mode. External interrupts and aborts can also be made to trap to Monitor mode, but system calls, MMU memory faults, and misuse of undefined or privileged instructions can't be configured to trap to Monitor mode. The Secure and Normal worlds and Monitor mode have their own exception handlers. Monitor mode is responsible for swapping in and out CPU state (i.e., registers) when switching from one world to another, enabling execution to begin where it left off in whichever world is being switched to.

TrustZone supplies two virtual MMUs for 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. 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 masked).

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 some kind of TPM 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 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 [14]:

- **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 SRAM. 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 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'' ([14]) 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.´

Tier Three can therefore also be used to support system 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.

6. Hypervisor-based security architectures

6.1. 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) [39].

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.

6.2. 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 [22]. 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 [40]:

- **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 [90]. 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.

6.3. Architectural limitations

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

6.3.1. 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 [39], 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 [66], as well as the as yet unreleased VIX toolkit (also for Xen) [43], 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.

6.3.2. 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 [67] 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.

6.4. 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.

6.4.1. 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.

6.4.2. 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).

6.4.3. 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.

6.4.4. 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. The 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.

6.5. 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.

6.5.1. Isolation architectures

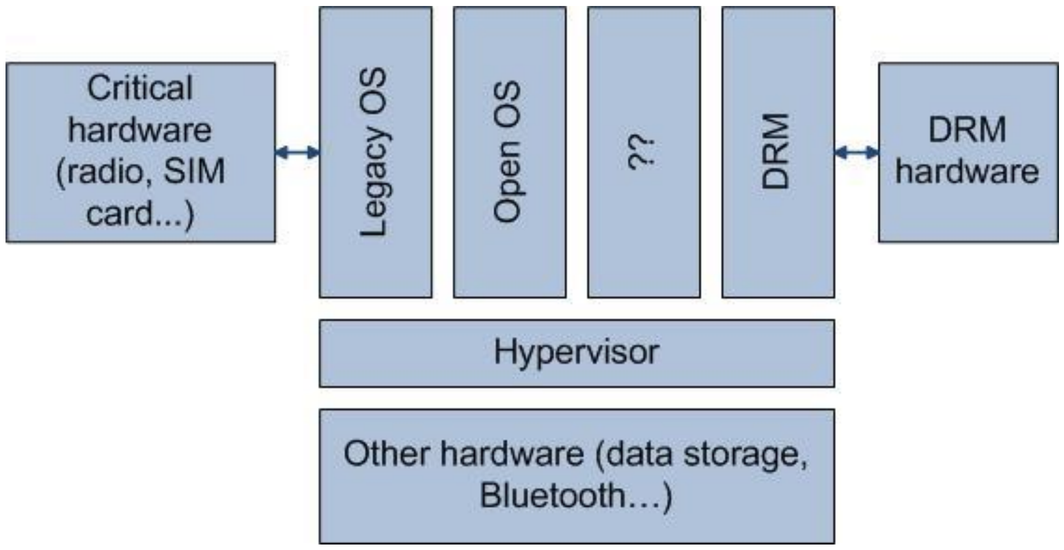
Critical hardware (radio, SIM card...)
Legacy OS
Open OS
??
DRM
DRM hardware
Hypervisor
Other hardware (data storage, Bluetooth…)

Figure 3: 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 [25] (see Figure 3). 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 [32]-- if the device's Bluetooth implementation is compromised (Bluetooth has been known to have security vulnerabilities [74]), 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.

6.5.2. Kernel code integrity

There are multiple research systems supporting kernel code integrity.

First off, we have SecVisor [77], 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 user-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 in 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 [104] 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.

6.5.3. Memory multi-shadowing

The Overshadow system [28] 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}) [105] (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.

6.5.4. Protecting against a malicious OS

In a follow up [70] 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 nevarious 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.

6.5.5. I/O Security

BitVisor [78] 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.

6.5.6. Componentization

The Nizza system [47] 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 system call 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.

6.5.7. Mandatory Access Control (MAC)

MAC policies such as Bell LaPadula, the Biba integrity model, and the Chinese Wall model [10] 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 [73] 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 [31] 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 [82] modules.

6.5.8. Instruction set virtualization

The *Secure Virtual Architecture* (SVA) system [33] 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 allcation, 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 assembly). 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.

6.6.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.

6.6.1. 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 [87], and virtualization may possibly be used to fulfill this specification. While SELinux has already been used to do so [1][106], to our knowledge virtualization has not. Discussion on utilizing ARM TrustZone technology to facilitate Trusted Computing is found in [102].

An early VMM-based system for attestation was Terra [38]. 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.

6.6.2. Malware analysis

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

Firstly, the Patagonix system [59] 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 arguably much easier to acquire system binaries than to implement system-dependent logic), the system's generic 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 described here [51] 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 supports multiple VMMs, including Xen, VMware, UserMode Linux and QEMU.

6.6.3. Intrusion detection

Another natural virtualization monitoring service is intrusion detection. The previously introduced Livewire system [39] 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 [52], 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.

6.6.4. 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 [43], 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.

6.6.5. Execution logging and replay

Another apt and canonical use of virtualization's monitoring possibilities is to log and replay VM execution. The ReVirt system [35] 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 [36] 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.

6.7. 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][106]), although we also saw with Xen Security Modules [31] that Flask may *complement* rather than supplant virtualization.

Another possibility is enforcing security via FPGAs. [26] 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 [97]), tamper proof smart cards for a variety of cryptography and authentication applications, or Trusted Platform Modules (TPMs) for sealed storage, attestation, and other uses. Of course, any of these hardware solutions could also be combined with virtualization.

7. Multicore systems

7.1. Why multicore?

An excellent overview of multicore hardware today, including hardware and software concerns and challenges, is found here [23]. Additionally, this article [68] also discusses contemporary multicore developments, and furthermore merges the discussion with description of virtualization concerns and opportunities on multicore.

As noted in [23] [68] 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 [23] [68]:

- 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.

In particular, also noted by [23] [68] 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 [21]. 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.

7.2. 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 [86], a broad spectrum of issues come to light. A range of hardware concerns in multicore systems is illustrated in [23] (Section 2.1), and summarized in the following subsections.

7.2.1. 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.

7.2.2. 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 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, nonparallelizable 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.

7.2.3. 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 as the number of cores increases.

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 [86].

7.2.4. 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.

7.2.5. Extended instruction sets¨'

The x86 instruction set is firmly in place, and isn't going anywhere [68]; hence, though it may not have been intended to support multicore from the beginning, it is necessary 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 [23] (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.

7.2.6. 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 [60].

7.3.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 [23] (Section 2.2), and summarized in the followikng subsections.

7.3.1. 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 [96].

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 [34].

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 [71].

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.

7.3.2. 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 non-determinism 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.

7.3.3. 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 [68].

7.3.4. 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.

7.4. Interesting multicore architectures

7.4.1. The Barrelfish multikernel

The Barrelfish operating system [76][20][21] 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* [20], 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.

7.4.2. 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][98], 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.

7.4.3. Mixed-Mode Multicore (MMM) reliability

For the same reasons, the authors in [98] 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 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.

7.5. Multicore and virtualization

As mentioned, a discussion of multicore and virtualization can be found in [68]. 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 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 for answers.

7.5.1. 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.

7.5.1.1. Managing dynamic heterogeneity

For example, the authors of [99] (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 manage the heterogeneity for virtual machines. The virtualization layer can also support MMM.

7.5.1.2. Sidecore

Another interesting architecture is *Sidecore* [56], 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 offer 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. 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 [55], 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 inter-core communication [72], as in AMD HyperTransport [3] and Intel QuickPath [48], which will further improve the inter-core communication latency of sidecore-inspired architectures.

8. References

[1] Onur Aciicmez and Afshin Latifi and Jean-Pierre Seifert and Xinwen Zhang. A Trusted Mobile Phone Prototype. 5th IEEE Consumer Communications and Networking Conference (CCNC), Las Vegas, NV, USA, 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, pages 2-13, 2006.

[3] Advanced Micro Devices, Inc.. AMD HyperTransport Technology page. http://www.amd.com/us/products/technologies/hypertransport-technology/Pages/hypertransport-technology.aspx (accessed 12 Oct 2009), 2009.

[4] Advanced Micro Devices, Inc.. AMD I/O Virtualization Technology (IOMMU) Specification. http://www.mimuw.edu.pl/ vincent/lecture6/sources/amd-pacifica-specification.pdf (last accessed 24 Sept 2009), 2009.

[5] Advanced Micro Devices, Inc.. AMD-V Nested Paging. available at http://developer.amd.com/assets/NPT-WP-1 1-final-TM.pdf, 2008.

[6] Advanced Micro Devices, Inc.. AMD-V Technology page. http://www.amd.com/us/products/technologies/virtualization/Pages/amd-v.aspx, 2009.

[7] 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), 2005.

[8] Advanced Micro Devices, Inc.. 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, 2008.

[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), 2007.

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

[11] ARM Limited. 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.

[12] 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.

[13] ARM Limited. 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.

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

[15] ARM Limited. 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.

[16] 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, 2009.

[17] Vasanth Bala and 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.

[18] Leonid Baraz and Tevi Devor and Orna Etzion and Shalom Goldenberg and Alex Skaletsky and Yun Wang and Yigal Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, pages 191-201, 2003.

[19] Paul Barham and Boris Dragovic and Keir Fraser and Steven Hand and Tim Harris and Alex Ho and Rolf Neugebauer and Ian Pratt and Andrew Warfield. Xen and the Art of Virtualization. Proceedings of the nineteenth ACM symposium on Operating systems principles in Operating Systems Review, pages 164--177, New York, 2003. ACM Press.

[20] Andrew Baumann and Paul Barham and Pierre-Evariste Dagand and Tim Harris and Rebecca Isaacs and Simon Peter and Timothy Roscoe and Adrian Schüpbach and 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, 2009.

[21] Andrew Baumann and Simon Peter and Adrian Schüpbach and Akhilesh Singhania and Timothy Roscoe and Paul Barham and Rebecca Isaacs. Your computer is already a distributed system. Why isn't your OS?. Proceedings of the 12th USENIX Workshop on Hot Topics in Operating Systems, Monte Verità, Switzerland, 2009.

[22] Steven M. Bellovin. Virtual Machines, Virtual Security?. CACM: Communications of the ACM, 49(10):104, 2006.

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

[24] 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, 2009. available at http://www.commoncriteriaportal.org/thecc.html (last accessed 19 September 2009).

[25] Jörg Brakensiek and Axel Dröge and Martin Botteck and 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, 2008.

[26] 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, 2008.

[27] David Chaum and Torben Pryds Pederson. Wallet Databases with Observers. Advances in Cryptology – CRYPTO 1992, LNCS 740, pages 89 - 105, 1993.

[28] Xiaoxin Chen and Tal Garfinkel and E. Christopher Lewis and Pratap Subrahmanyam and Carl A. Waldspurger and Dan Boneh and 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), 2008.

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

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

[31] 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), 2007.

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

[33] John Criswell and Andrew Lenharth and 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), pages 351-366, 2007.

[34] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering, 5(1):46-55, 1998.

[35] George W. Dunlap and Samuel T. King and Sukru Cinar and 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).

[36] George W. Dunlap and Dominic G. Lucchetti and 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, 2009.

[37] Dawson R. Engler and 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), 1995.

[38] Tal Garfinkel and Ben Pfaff and Jim Chow and 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), 2003.

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

[40] Tal Garfinkel and Mendel Rosenblum. 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), 2005.

[41] Robert P. Goldberg. Survey of Virtual Machine Research. IEEE Computer, :34--45, 1974. available at https://agora.cs.illinois.edu/download/attachments/10454931/goldberg74.pdf.

[42] Steven Hand and Andrew Warfield and Keir Fraser and 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, 2005.

[43] Brian Hay and Kara Nance. Forensics Examination of Volatile System Data Using Virtual Introspection. ACM SIGOPS Operating Systems Review, 42(3):74--82, 2008.

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

[45] Gernot Heiser and Volkmar Uhlig and Joshua LeVasseur. Are Virtual-Machine Monitors Microkernels Done Right?. Operating Systems Review, 40(1):95--99, 2006.

[46] Joo-Young Hwang and Sang-Bum Suh and Sung-Kwan Heo and Chan-Ju Park and Jae-Min Ryu and 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, 2008.

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

[48] Intel Corporation. Intel QuickPath Technology page. http://www.intel.com/technology/quickpath/ (accessed 12 October, 2009), 2009.

[49] Intel Corporation. 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), 2008.

[50] 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), pages 508-515, Orlando, FL, USA, 2008.

[51] 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, 2007.

[52] Ashlesha Joshi and Samuel T. King and 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), pages 91--104, Brighton, UK, 2005.

[53] Gerwin Klein and Kevin Elphinstone and Gernot Heiser and June Andronick and David Cock and Philip Derrin and Dhammika Elkaduwe and Kai Engelhardt and Rafal Kolanski

and Michael Norrish and Thomas Sewell and 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, 2009. available at http://ertos.nicta.com.au/publications/papers/Klein_EHACDEEKNSTW_09.pdf (accessed 26 September, 2009).

[54] Kirk L. Kroeker. The Evolution of Virtualization. Communications of the ACM, 52(3):18--20, 2009.

[55] Sanjay Kumar and Ada Gavrilovska and 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.

[56] Sanjay Kumar and Himanshu Raj and 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, 2007.

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

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

[59] Lionel Litty and H. Andrés Lagar-Cavilla and David Lie. Hypervisor Support for Identifying Covertly Executing Binaries. Proceedings of the 17th USENIX Security Symposium, pages 243--258, San Jose, CA, USA, 2008.

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

[61] Gil Neiger. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. Intel Technology Journal, 10(3):167--177, 2006.

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

[63] NICTA. 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), 2009.

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

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

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

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

[68] 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.

[69] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. Communications of the ACM, 17(7):412-421, 1974.

[70] Dan Ports and Tal Garfinkel. Towards Application Security On Untrusted Operating Systems. USENIX Workshop on Hot Topics in Security (HOTSEC), 2008.

[71] Robert D. Blumofe and. Cilk: An Efficient Multithreaded Runtime System. ACM SIGPLAN Notices, 30(8):207-216, 1995.

[72] Bratin Saha and others. Enabling Scalability and Performance in a Large Scale CMP Environment. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, pages 73 - 86, Lisbon, Portugal, 2007.

[73] Reiner Sailer and Treng Jaeger and Enriquillo Valdez and Ronald Perez and Stefan Berger and 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), pages 276--285, 2005.

[74] 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), 2008.

[75] 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.

[76] Adrian Schüpbach and Simon Peter and Andrew Baumann and Timothy Roscoe and Paul Barham and 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, 2008.

[77] Arvind Seshadri and Mark Luk and 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, 2007.

[78] Takahiro Shinagawa and others. 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, 2009.

[79] James E. Smith and Ravi Nair. The Architecture of Virtual Machines. IEEE Computer, 38(5):32-38, 2005.

[80] James E. Smith and Ravi Nair. Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann/Elsevier, 2005.

[81] Stephen Soltesz and Herbert Potzl and Marc E. Fiuczynski and 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), pages 275--287, Lisbon, Portugal, 2007.

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

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

[84] 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, 2008.

[85] Sang-Bum Suh and Sung-Min Lee and Sangdok Mo and Bokdeuk Jeong and Joo-Young Hwang and Chan-Ju Park and Sung-Kwan Heo and Junghyun Yoo and Jae-Min Ryu and Chul-Ryun Kim and Seong-Yeol Park and Jae-Ra Lee and Il-Pyung Park and 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, 2008.

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

[87] 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, 2007.

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

[89] Steven J. Vaughan-Nichols. New Approach to Virtualization is Lightweight. IEEE Computer, 39(11):12-14, 2006.

[90] Steven J. Vaughan-Nichols. Virtualization Sparks Security Concerns. IEEE Computer, 41(8):13-15, 2008.

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

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

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

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

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

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

[97] 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, pages 286-292, Oakland, CA, USA, 1992.

[98] Philip M. Wells and Koushik Chakraborty and Gurindar S. Sohi. Dynamic Heterogeneity and the Need for Multicore Virtualization. ACM SIGOPS Operating Systems Review, 43(2):5-14, 2009.

[99] Philip M. Wells and Koushik Chakraborty and Gurindar S. Sohi. Mixed-Mode Multicore Reliability. Proceedings of the 14th Annual International ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09), 2009.

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

[101] 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.

[102] Johannes Winter. Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms. Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing, pages 21--30, Fairfax, VA, USA, 2008.

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

[104] Min Xu and Xuxian Jiang and 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, 2007.

[105] 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), pages 71--80, Seattle, WA, USA, 2008.

[106] Xinwen Zhang and 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, 2007.