Secure & Resilient Cyber Architectures Conference
MITRE, McLean, VA, October 29, 2010

Security Implications of Different Virtualization Approaches for Secure Cyber Architectures

Sanaz Rahimi and Mehdi Zargham
Department of Computer Science
Southern Illinois University
Carbondale, IL
Email: {srahimi, mehdi}@cs.siu.edu

***Abstract*—Virtualization is increasingly being used as a component in designing secure cyber architectures. The proposed applications include strong isolation, monitoring, fault tolerance, execution replay, etc. However, there are various virtualization approaches which differ in their security implications, proper applications, overheads, requirements, and most importantly threat models. Firstly, virtualization solutions range from hardware assisted to completely software emulated. They can be further categorized by their abstraction level from hardware virtualization, to operating system level virtualization (containers), to application virtualization. Finally, they differ widely in the way they handle I/O; from having a separate I/O virtual machine (privileged partition) to hardware assisted I/O virtualization (IOMMU). In order to deploy the most effective virtualization approach when designing a secure cyber system, it is imperative to fully understand the benefits offered and the trade-offs involved in each method. In this work, we study the spectrum of virtualization modes and discuss their strengths, weaknesses, requirements, and threat models. For instance, lightweight paravirtualization with IOMMU can reduce the size of the trusted computing base (TCB) significantly which makes it a proper candidate for high assurance isolation and thin client applications. On the other hand, containers have the similar TCB size as a full blown operating system, but they preserve the semantic information lost in the lower level virtualizations which makes them suitable for replay and monitoring applications. Moreover, we identify several gaps combinations that have not been implemented yet, to the best of our knowledge which, if available, can be valuable for specific applications and assurance requirements.**

I. Introduction

Virtualization is increasingly being used as a component in designing secure cyber architectures. Although it was originally developed decades ago in the early days of computing as a means of resource sharing, virtualization has reemerged as a viable solution for secure and high performance computing in recent years. With the advent of fast and virtualization extended hardware, the performance penalty of virtualization has been reduced significantly enabling a personal computer to run multiple virtual machines (VMs) at the same time.

In the realm of security, virtualization has been used in a wide variety of applications including isolation, monitoring, fault tolerance, intrusion tracking, software impact analysis, and execution replay. However different virtualization approaches differ considerably in their security implications, proper applications, overheads, requirements, and threat models. Unfortunately, this simple fact is sometimes ignored when virtualization is used as a security component. Moreover, as a software system, a virtual machine monitor (VMM, a.k.a hypervisor) is prone to software flaws and vulnerabilities. Relying on a large VMM as an always trusted component is no more justified than relying on any other pieces of software. However, what can potentially be different is the size of the trusted computing base (TCB) and the abstraction levels when using virtualization as a security component. In order to utilize virtualization effectively one has to understand the trade-offs and security implications of different virtualization techniques. In this paper, we study each of these techniques, their security implications, and proper applications. We use the size of TCB as a measure of security. Of course rigorous software/hardware verification techniques can result in fewer flaws and vulnerabilities, but given similar techniques and limited verification resources, TCB can be a good indicator of how much trust can be placed on a system. In the rest of the paper, the system hardware is always part of the TCB unless otherwise stated.

The rest of the paper is organized as follows. Section II describes full and paravirtualization techniques. Section III explains virtualization in software and hardware. The abstraction level of each virtualization techniques is described in Section IV. Section V studies important I/O virtualization methods. We describe the entire virtualization spectrum and its applications in Section VI before concluding the paper in Section VIII. The security implications of each approach is discussed in the section describing that element and summarized in Section VI.

II. Full vs. Paravirtualization

Operating system (OS) virtualization techniques can be divided into two large categories: full virtualization and paravirtualization.

In paravirtualization, an OS kernel must be modified and ported to the application programming interface (API) of the VMM. The OS kernel consists of virtualizable and non-virtualizable instructions. The virtualizable instructions (non-privileged) are executed directly on the hardware of the machine. However, the non-virtualizable (privileged) instructions including memory management, interrupt handling, time keeping, I/O handling require more care. They cannot be executed directly on the hardware because they may interfere with the other VMs running on the same machine. In the paravirtualization model, these instructions are replaced with calls into the VMM (*hypercalls*.) In this sense, a paravirtualized operating system is aware of the virtualization layer. As a result, paravirtualization can achieve near hardware performances with minimum overhead. Xen [5] is an example of a paravirtualization VMM. Its downside is that it requires access to the source code of the operating system for porting.

Full virtualization, on the other hand, virtualizes the entire hardware platform (including BIOS, I/O, etc.). It can virtualize unmodified OS code and it does not require access to the source code. VMWare Workstation [28] is a full virtualization VMM. There are several techniques to deal with non-virtualizable instructions in full virtualization: software emulation, binary translation (code scanning), and hardware assisted. The first technique, emulates the hardware instruction in software, thus providing the desired effect for the OS. Binary translation runs the majority of the OS code on the hardware, but replaces the binary of the non-virtualizable instructions by new code sequences to achieve the intended effect on the virtual hardware. This is done by scanning the OS code pages in the memory (hence the name "code scanning"). Finally the hardware assisted model uses hardware support to trap privileged instructions and call the VMM for proper handling. We explain hardware assisted virtualization in more detail in the next section.

Full virtualization is in general slower than paravirtualization. Among the full virtualization techniques, hardware assisted model has the lowest and software emulation has the highest overhead.

In paravirtualization, the VMM can avoid virtualizing many of the hardware features (e.g. BIOS) simply by changing the OS privileged instructions. Moreover, by placing hypercalls in the OS code, it can also avoid emulation or binary translation. All this means that paravirtualization does not have the complexities of full virtualization, thus its code size can be very small. Since the entire hypervisor often falls into the TCB, smaller code size can result in a simpler and more easily verifiable VMM. Many secure hypervisors such as SecVisor [24] ($\sim$1112 lines-of-code) use paravirtualization to keep the code size tiny (they are also called *thin hypervisors*). Unmodified OS code translates into complex and large full virtualization VMMs. In addition, the code scanning and software emulation logic add to the complexity of the VMM. Among the full virtualization techniques, hardware assisted model can achieve the smallest code size and the highest assurance level.

III. Software vs. Hardware Assisted Virtualization

Virtualization can be performed in software or hardware. The software based model performs all the virtualization tasks in software and it can run on commodity processors.

Software based VMMs perform memory and I/O virtualization in software. Most OSes support paged virtual memory and manage that by directly accessing the processor memory management unit (MMU), but direct access to MMU or physical memory by a VM can result in an interference with other VMs. Since the VMM has to isolate different VMs from each other it has to virtualize the paged memory for the guest OSes. This is done by adding an extra address translation step to virtual memory. The VMM manages a data structure called a *shadow page table* which translates a VM's virtual address (a.k.a guest virtual address) to system physical address. Also software based virtualization has to emulate the I/O devices for the VMs which adds to the complexity of the VMM.

Hardware assisted virtualization simplifies the VMM by supporting many of the tasks in hardware. First, hardware assisted virtualization including Intel VT [2] and AMD SVM [1] support two-step address translation in hardware (called Extended Page Table for Intel and Nested Page Table for AMD.) They also virtualize interrupts and I/O (see Section V) which simplifies full virtualization. Moreover, hardware assisted virtualization provides simple instructions to access hidden processor states (non-software accessible) which facilitates VM entry/exits. This feature makes context switching between VMs very efficient. Finally, registers that are frequently accessed by the guest OS (e.g. task priority register) may also be shadowed for performance reasons.

Hardware assisted virtualization can greatly simplify the VMM. In fact, thin hypervisors often use hardware features to avoid software emulation of virtualization tasks (e.g. SecVisor). Also, features implemented in hardware can be more difficult to modify maliciously which adds to the assurance of hardware assisted VMMs. However, there are two important points to consider here. First, in many cases the hardware provides the appropriate data structures that are managed by the VMMs (e.g. Extended/Nested Page Table), so a flaw or vulnerability in that part of the VMM code can similarly affect both hardware and software based virtualization. For example, a malicious translation of guest physical address to system physical address can give a VM unauthorized access to the memory of another VM. Second, many VMM implementation support both a hardware assisted mode and a software fallback mode which means its code size and the TCB remain large.

IV. Abstraction Level

So far we have discussed virtualization at the system level, but virtualization at other abstraction levels have also been used for security application. The other two major virtualizations are OS level and application virtualization.

A. OS Level Virtualization

OS level virtualization works one abstraction level above system virtualization. It uses the OS kernel to provide isolation and containment for multiple user level virtual environments. Each virtual environment is also called a container or jail. Each virtual environment has its own filesystem, devices (`/dev` in Linux), and network interface. These resources are fairly shared between the virtual environment by the kernel. Note that system virtualization works at the abstraction level of memory pages, disk blocks, and I/O devices while OS level virtualization operates on memory regions, files, `/dev` filesystem, and network sockets. Examples of OS level virtualization include OpenVZ [14], VServer [25], BSD Jails [23], and Virttuozzo [3].

OS level virtualization gives the hypervisor (kernel) access to OS internal data structures and abstractions which are lost at the system virtualization level. In order to recover this information, some system level VMMs use virtual machine introspection (VMI) [18].

OS level virtualization provide valuable information about the virtual environment at the expense of increasing the TCB size. In this model, the entire OS kernel code is part of the TCB. Although the applications inside the virtual environment may not be trusted, the OS kernel must be trusted and free of vulnerabilities that may cause privilege escalation for the applications.

B. Application Virtualization

A higher abstraction level is application virtualization. In this approach, the application is provided a sandbox with limited and sometimes controlled set of instructions (a.k.a virtual instruction set). The application is not allowed to execute any instruction other than those provided in the sandbox. Application virtualization gives a very fine-grained control over the code actions. Java Virtual Machine (JVM) [8] is an example of application virtualization. LLVA [4] is another virtual instruction set that works with C and C++ source codes.

Application virtualization can tightly control a program at the level of instructions, but it relies on the OS and the virtual environment (JVM or runtime environment). The OS and virtual environment (analogous to the hypervisor in system level virtualization) are in the TCB. Although it is possible to provide some code guarantees such as type safety for the OS itself by running it on top of the virtual instruction set (see SVA [7]), the OS is still trusted to behave correctly. As a result, application virtualization provides the finest-grained information about the virtual machine (e.g. a Java applet) at the expense of having the largest TCB compared to other virtualization approaches.

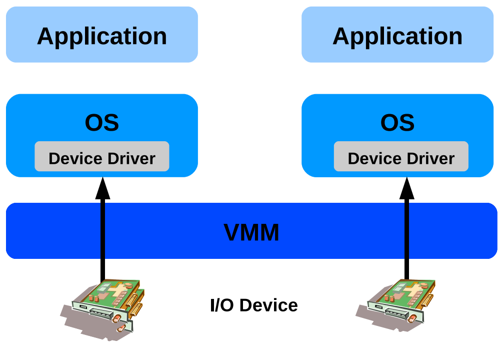

Fig. 1. Pure isolation I/O model

V. I/O Virtualization

Virtualization techniques also differ in the way they virtualize I/O devices. This is specifically relevant to system level virtualization, so we present it here in that context.

A. Pure Isolation

The simplest I/O virtualization approach is to have a separate I/O device for each VM running in the machine (see Figure 1). In this approach, there is no need to share an I/O device between multiple VMs. The device drivers are placed inside the VM's guest OS which makes the VMM very simple and small. This approach has the obvious disadvantage of the need for multiple hardware devices which adds to the implementation and maintenance cost.

From the security perspective, the pure isolation model provides the best assurance. Since each VM can only operate with its dedicated I/O device, the risk of malicious interference is dramatically reduced. Pure isolation also have a small TCB. If the overall goal is the isolation of the VMs, only the hypervisor (minus the device drivers) is part of the TCB. Note that the device driver may be malicious and can malfunction, but pure isolation prevents the corrupted VM from interfering with other VMs.

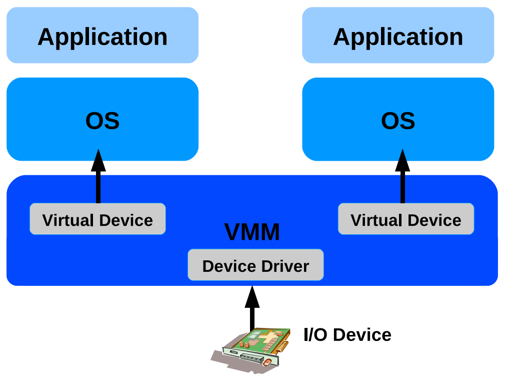

Fig. 2. Shared device driver in VMM I/O model

B. Shared Device Drivers in VMM

The second approach is to share the I/O devices and place the device drivers inside the VMM. In this model, the system can have only one copy of each I/O device which the VMM shares between the VMs. Each VM is provided a virtual I/O device which is then multiplexed into the actual physical device by the VMMs. For example, a write to a virtual hard drive is forwarded to the physical hard disk drive by the VMM to a dedicated location, isolated from other VMs. Figure 2 illustrates this I/O model. KVM [10] uses the shared device drivers model for I/O virtualization. The cost of each I/O operation is a system call to the guest OS, a hypercall to the VMM, a hardware instruction to the physical I/O device, an interrupt by the device to the VMM sending the results back, and a virtual interrupt to the guest OS by the VMM.

The disadvantage of this approach is the complexity and size of the VMM. All codes necessary to emulate a virtual device and the actual device drivers must reside inside the VMM. The TCB in this approach includes the VMM and the device drivers. Note that the VMM may be significantly larger than pure isolation because of the virtual devices.

C. Shared Device Drivers in I/O Partition

The third approach adopted by a number of commercial VMMs is to delicate all the I/O operations to a dedicated VM called the I/O partition (a.k.a privileged partition or dom0). In this model, the device drivers are placed inside the I/O partition and any I/O request by a VM is forwarded to it by the VMM (see Figure 3). The VM is still presented with a virtual device driver. Xen [5] takes this approach.

The I/O partition model keeps the VMM design simple and small because it can rely on commodity OS and device drivers for the I/O partition, but it has the highest performance cost especially for I/O-bound applications. The cost of each I/O operation is a system call to the guest OS, a hypercall to the VMM, a virtual interrupt by the VMM to the I/O partition, a hardware instruction to the physical I/O, an interrupt by the device to the VMM sending the results back, a virtual interrupt by the VMM to the I/O partition, a hypercall from the I/O partition to the VMM, and a virtual interrupt from the VMM to the guest OS. Notice that the cost of a single I/O operation is much higher than that of the previous model (shared device driver in VMM.)

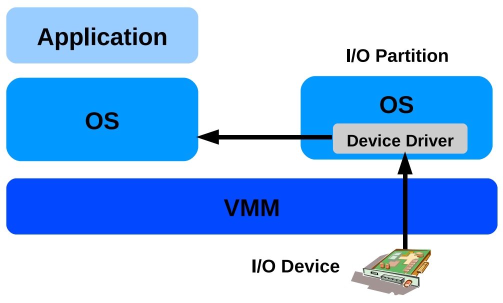

Fig. 3. I/O partition model

Although the VMM itself is small, the TCB in the I/O partition model includes the VMM, the guest OS in the I/O partition, and all the device drivers. As a result, this approach has the largest TCB size compared to other I/O virtualization models.

Sharing the I/O devices in the latter two approaches reduces the assurance of the system. Malicious VMs may use the shared drivers and controllers to interfere with other VMs and violate the security policy. Hardware assisted I/O virtualization (Section V-D) prevents I/O devices from malicious interference in the memory, but higher level interference through software controller is still a threat. This is especially an issue for complex I/O controllers such as the graphics system. Trusted and I/O specific controllers are necessary to preserve the isolation at the device software level (e.g. TrustGraph [20] for secure graphics.)

D. Hardware Assisted I/O Virtualization – IOMMU

DMA-enabled I/O devices can access the entire physical memory of a machine. To prevent memory corruption and interference, VMMs trap and translate memory accesses by the I/O devices (swiotlb and grant tables.) Recent hardware assisted virtualization technologies implement this translation in hardware. Called an IOMMU (a.k.a Intel VT-d or AMD-Vi), this memory management unit translates I/O device addresses to physical addresses much the same way MMUs translate OS virtual addresses. IOMMU is configured by the VMM to enforce I/O memory separation. It handles the remapping in hardware allowing for the device drivers to be moved into the guest OSes. This simplifies the VMM code by

I/O Model	TCB Size	Performance
Pure Isolation	VMM	High
Shared in VMM	VMM + Device Drivers	Medium
I/O partition	VMM + OS + Device Drivers	Low
IOMMU	VMM	High

TABLE I
COMPARISON OF DIFFERENT I/O VIRTUALIZATION APPROACHES

removing the trapping and translation functionalities. It also moves the device drivers outside the TCB. Note that if a device driver is malicious, the isolation remains intact because the IOMMU limits all memory requests to the VM's domain.

Table I compares the TCB size and relative performance of different I/O virtualization models.

VI. VIRTUALIZATION SPACE AND APPLICATIONS

Different virtualization techniques and approaches provide a variety of choices and options for building a secure cyber architecture, each with its own properties, threat model, and proper applications.

The minimum TCB size is achieved with a system level, hardware assisted paravirtualization VMM with dedicated (pure isolation) I/O. This type of VMM can be as small as $\sim$1000 lines-of-code which makes it suitable for high assurance isolation and trusted separation kernels. The tiny code base of the VMM can be verified (semi)formally using rigorous techniques to ensure complete isolation.

The maximum compatibility is achieved with hardware assisted, software fallback full virtualization with I/O partition. The VMM in this case can support hardware features, but it also has software fallback for commodity processors. Full virtualization ensures that closed source OSes can also be virtualized. Moreover, unmodified device drivers can be placed in the I/O partition which maximizes compatibility. This type of VMM is ideal for software testing, patch/update analysis, malware monitoring, out-of-the-box virus scanning, or side effect analysis. Maximum compatibility ensures that no behavior or side effect is missed during the process.

The shared drivers in VMM model is the closest model to a stand alone OS. It allows the kernel of the operating system to be used as the hypervisor (KVM model.) This approach is appropriate for efficient code maintenance because the OS developers update the device drivers/kernel functionalities for the hypervisor at the same time they update the OS code.

OS level virtualization provides the most semantic information about the virtual environment from the OS data structures. It is the best candidate for application containment, semantic monitoring, and system call level anomaly detection. The semantic information available in the OS level virtualization is lost in system level virtualization which can only be recovered partially using introspection.

Application virtualization provides the finest grained information about the application. It is the most appropriate candidate for secure remote execution and limited, low performance applications. Applets (e.g. Java applets) are the important use case of application virtualization. Note that since the virtual instruction set can be very limited and tightly controlled, virtualized applications are limited in their capabilities. For instance, arbitrary system or filesystem operations may not be possible with virtualized applications.

A gap in the large space of virtualization is a multi-VM, tiny hypervisor that deploys hardware assisted features and IOMMU. Some research prototype VMMs (e.g. SecVisor) use IOMMU, but they only support one VM. Other commercial VMMs such as Xen has large code bases for software fallback.

Another missing combination, to the best of our knowledge, is a completely hardware supported virtualization platform which does not use a software based hypervisor component. Having a completely hardware implemented virtualization can significantly increase the assurance level of the system.

Finally, another important gap in the virtualization technologies is an OS or application level virtualization framework with a smaller-than-OS TCB. This would require some support from the underlying hardware or system level VMM. TrustVisor [16] reduces the size of TCB for an application significantly, but the isolation imposed on the code can limit the granularity of monitoring.

VII. RELATED WORK

There have been a number of efforts in the literature for building trusted systems using virtualization. IBM's PR/SM [6] and VMM based security kernel for VAX architecture [13] are two of the early projects. NetTop [17] and IBM's sHype [22] are two of the more recent virtualization based high assurance systems. Terra [9] and SecVisor [24] also build high assurance systems using virtualization.

Various I/O virtualization techniques and their performance and security trade-offs are studied by Karger and Safford [12]. Perez, et al. [21] study the role of virtualization in hardware-based security. Vasudevan, et al. [26] enumerate and discuss the difficulty of building an integrity protected hypervisor for x86 architecture.

Numerous other applications and tools have been implemented using virtualization, including malware analysis [19], process tracking [11], intrusion detection [15], and honeypots [27].

VIII. Conclusion

In this paper, we studied different virtualization approaches and design decisions. Each technique has its security implications specifically for the TCB size, threat model, and suitable applications. Effective use of virtualization in secure cyber architecture depends on deep understanding of these security implications and the correct choice of the most appropriate technique. We plan to study the combinations that have not been implemented in more detail and investigate the feasibility of implementing them using maximum hardware support wherever possible.

References

[1] Secure virtual machine architecture reference manual. White paper, Advanced Micro Devices, 2005.

[2] Intel virtualization technology for directed I/O. White paper, Intel, 2007.

[3] Clustering in parallels virtuozzo-based systems. White paper, Parallels, 2009.

[4] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. Llva: A low-level virtual instruction set architecture. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 205, Washington, DC, USA, 2003. IEEE Computer Society.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[6] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk. Multiple operating systems on one processor complex. *IBM System Journal*, 28(1):104–123, 1989.

[7] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: a safe execution environment for commodity operating systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 351–366, New York, NY, USA, 2007. ACM.

[8] P. W. L. Fong. Reasoning about safety properties in a jvm-like environment. *Sci. Comput. Program.*, 67(2-3):278–300, 2007.

[9] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 193–206, 2003.

[10] I. Habib. Virtualization with kvm. *Linux J.*, 2008(166):8, 2008.

[11] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 2006. USENIX Association.

[12] P. A. Karger and D. R. Safford. I/o for virtual machine monitors: Security and performance issues. *IEEE Security and Privacy*, 6(5):16–23, 2008.

[13] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the vax vmm security kernel. *IEEE Transaction Software Engineering*, 17(11):1147–1165, 1991.

[14] K. Kolyshkin. Virtualization in linux. White paper, OpenVZ, September 2006.

[15] K. Kourai and S. Chiba. Hyperspector: virtual distributed monitoring environments for secure intrusion detection. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 197–207, New York, NY, USA, 2005. ACM.

[16] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *SP '10: Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.

[17] R. Meushaw and D. Simard. NetTop: Commercial technology in high assurance applications. *National Security Agency Tech Trend Notes*, 9(4):3–10, Fall 2000.

[18] K. Nance, M. Bishop, and B. Hay. Virtual machine introspection: Observation or interference? *IEEE Security and Privacy*, 6(5):32–37, 2008.

[19] A. M. Nguyen, N. Schear, H. Jung, A. Godiyal, S. T. King, and H. D. Nguyen. Mavmm: Lightweight and purpose built vmm for malware analysis. In *ACSAC '09: Proceedings of the 2009 Annual Computer Security Applications Conference*, pages 441–450, Washington, DC, USA, 2009. IEEE Computer Society.

[20] H. Okhravi and D. Nicol. Trustgraph: Trusted graphics subsystem for high assurance systems. In *ACSAC '09: Proceedings of IEEE Annual Computer Security Applications Conference*, pages 254–265, Dec. 2009.

[21] R. Perez, L. van Doorn, and R. Sailer. Virtualization and hardware-based security. *IEEE Security and Privacy*, 6(5):24–31, 2008.

[22] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. Griffin, and L. Doorn. Building a mac-based security architecture for the xen open-source hypervisor. In *Proceedings of 21st Annual Computer Security Applications Conference*, pages 285–295, 2005.

[23] E. Sarmiento. Securing freebsd using jail. *Sys Admin*, 10(5):31–37, 2001.

[24] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP '07: Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 335–350, 2007.

[25] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 275–287, 2007.

[26] A. Vasudevan, J. McCune, N. Qu, L. van Doorn, and A. Perrig. Requirements for an integrity-protected hypervisor on the x86 hardware virtualized architecture. In A. Acquisti, S. Smith, and A.-R. Sadeghi, editors, *Trust and Trustworthy Computing*, volume 6101 of *Lecture Notes in Computer Science*, pages 141–165. Springer Berlin / Heidelberg, 2010.

[27] Y.-M. Wang, D. Beck, X. Jiang, and R. Roussev. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *IN NDSS*, 2006.

[28] S. S. Warren. *The VMware Workstation 5 Handbook (Networking & Security)*. Delmar Thomson Learning, 2005.