2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing

A Survey of Virtualization Technologies Focusing on Untrusted Code Execution

Yan Wen[1,2] Jinjing Zhao[1,2] Gang Zhao[1,2] Hua Chen [1,2] Dongxia Wang
[1]Beijing Institute of System Engineering, Beijing, China
[2]National Key Laboratory of Science and Technology on Information System Security
Beijing, China
celestialwy@126.com, misszhaojinjing@sina.com, zg@public.bise.ac.cn, chenhua8011@gmail.com, dongxiawang@126.com

Abstract—**In response to a continually advancing threat incurred by untrusted codes from Internet, various virtualization-based technologies have been proposed. Such technologies utilize a software layer, a virtual machine monitor or hypervisor, to achieve the highest privilege in a computer system. Generally, they construct isolated execution environments to run the untrusted code while shielding the other parts of the system from the potential security issues. In this paper, we survey a number of virtualization-based technologies with the goal of finding an appropriate candidate to serve as an untrusted code execution solution on PC platforms. Contenders are reviewed with a number of desirable properties, especially security, transparency portability and performance.**

Keywords- Untrusted code execution; virtualization.

I. INTRODUCTION

With the popularization of Internet, the means o releasing software and getting software are also increasingly dependent on the Internet. In everyday use, a growing number of individual users become accustomed to download software from the Internet in order to take full advantage of plenty of software resources. Therefore, intermingled software quality and untrusted software result in a large number of potentially unsafe and unstable factors.

In addition, a large number of untrusted Web sites that may contain malicious code constitute a serious threat to the security of the PC platforms. Such malicious code makes use of the vulnerabilities of Web browsers or OSes to attack the computer system. In 2008, after the study lasted 10 months, Google's researchers found that the Google database had processed billions of Web site addresses during this period, wherein there were over three million sites containing malicious code. In addition, they also found that there were approximately 1.3% of the searching requests results containing one malicious site. Some of these malicious sites were built to deliberately attack the visitors, but most of them were legitimate sites which were compromized by attackers. As to PC users, such security threats incurred by browsing these untrusted sites cannot be ignored.

Some host-based mechanisms are introduced to enhance the host security, i.e., access control, virus detection, and so on. But access control can be fooled by authorized but malicious users, masqueraders, and misfeasors. Although virus detection and similar technologies can be deployed to detect widely prevalent malicious codes, they are limited not only in theory but in practice, because in theory it is undecidable whether an arbitrary program contains a computer virus, and in practice it is also very difficult to accurately analyze the polymorphic or metamorphic virus code.

Sandbox is a more promising approach for defending against potential malicious code. However, the main drawback of sandbox based approaches is the difficulty of policy selection. Too often, sandbox tools incorporate highly restrictive policies that preclude the execution of many useful applications. The net result is that users end up choosing functionality over security.

As a supplementation, isolated execution mechanisms based on various virtualization technologies are proposed to bound the damage caused by undetected or detected intrusions during their latencies without negating the functionality benefits of untrusted code.

In this paper, we survey a number of virtualization-based technologies in order to evaluate the approach which is an appropriate candidate to serve as isolated execution environment for untrusted code, especially on PC platforms. This paper reviews the candidates with a number of desirable properties, especially security, transparency portability and performance.

The rest of the paper is organized as follows. In Section 2, we take an overview of existing various virtualization technologies. Section 3 provides a discussion on the sandbox-based untrusted code execution. Section 4 covers the approaches based on API layer virtualization. And Section 5 illustrates the approaches based on OS layer virtualization. In Section 6, the approaches based on hardware-abstraction layer virtualization are analyzed in details. Last, we compare all the above approaches and present our conclusion.

II. OVERVIEW OF VIRTUALIZATION TECHNOLOGIES

In recent years, the virtualization technology has gotten more and more attention and concern, e.g., Vmware's virtualization productions have been deployed in more than 80 percent of the global top-100 enterprises. With in-depth study over the past ten years, the virtualization technology has been widely applied in various important fields, exploring enterprise computing, disaster recovery, distributed computing, system security and so on.

Differing in accordance with the upper application interface (illustrated in Fig. 1), the virtualization technologies achieve multiple virtualized levels of computer system, including hardware-abstraction layer VM, OS layer VM, API layer VM, as well as programming language layer VM.

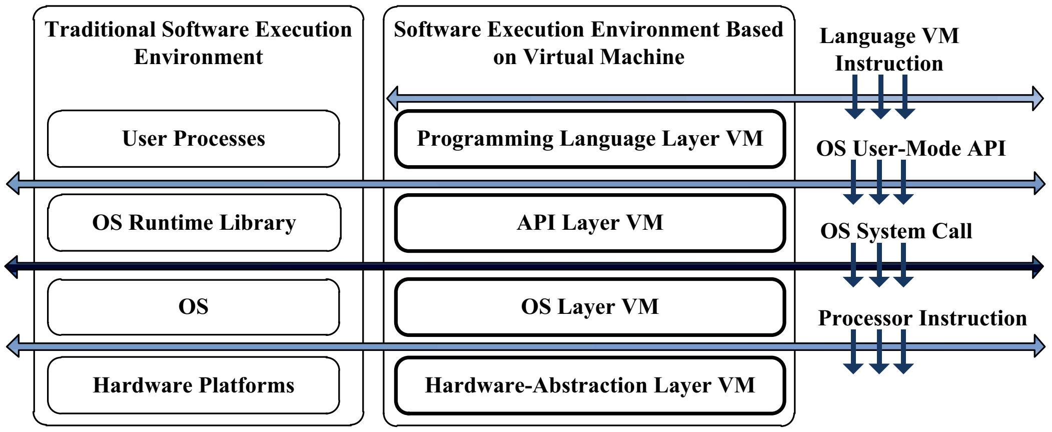

Traditional Software Execution Environment
Software Execution Environment Based on Virtual Machine
Language VM Instruction
User Processes
Programming Language Layer VM
OS User-Mode API
OS Runtime Library
API Layer VM
OS System Call
OS
OS Layer VM
Processor Instruction
Hardware Platforms
Hardware-Abstraction Layer VM

Fig.1 Hierarchical Virtual Machine Classification.

Hardware-abstraction layer VM. In the point of view of the upper level software (i.e., guest OS), a hardware-abstraction layer VM constructs a complete computer hardware platform. As illustrated in Fig. 1, the interface between this VM and the guest OS just is the complete set of processor instructions.

OS layer VM. By creating a clone of an OS' execution environment dynamically, an OS layer VM can build multiple virtualized containers. In the point of view of a program running in one container, such VM provides a complete OS software environment. The interface between it and the upper layer software (user mode application or kernel module) is system calls.

API layer VM. Such VM simulate the execution environment of one specific OS for the upper layer software (user mode applications). What it virtualizes is just OS user-mode API, instead of the processor instructions or system calls.

Programming language layer virtual machine. Such VM executes the programing language VM instructions in with JIT technology (Just-In-Time), so it can achieve the cross-platform feature.

III. UNTRUSTED CODE EXECUTION BASED ON SANDBOX

A sandbox is an environment in which the behavior of processes is restricted according to a serious of security policies. Sandbox-based approaches monitor a program's behavior and block the actions that may compromise the system's security.

The sandbox-based isolated execution technologies have previously been studied by researchers in the context of Java applets execution [1, 2]. Compared with general applications, such applets do not make much access to system resources. So, such sandbox approaches often relied on executing these untrusted applets on a "remote playground", i.e., an isolated computer. However, most of the desktop applications will usually require access to more resources, such as the file system on the user's computer. To run such applications on a remote playground, the complete execution environment on the user's computer, especially the entire file system contents, should be duplicated to the remote playground.

Janus [3] and Chakravyuha [4] implement sandbox using the kernel interposition mechanism. MAPbox [5] introduces a sandbox mechanism with the aim at making the sandbox more configurable and usable via providing a template for sandbox policies based on a classification of application behavior. Safe Virtual Execution (SVE) [6] implements a sandbox using *software dynamic translation*, a technology for modifying program binaries at runtime. Systrace [7] proposes a sandbox system that notifies the user about all the system calls that a program tries to invoke and then generates a policy for the program according to the response from the user.

IV. UNTRUSTED CODE EXECUTION BASED ON API LAYER VIRTUALIZATION

Peng Liu, et al., [8] propose the concept of *one-way isolation* as an effective means to isolate the effects of running processes from the point they are compromised. They develop one-way isolation protocols for databases and file systems. However, they did not present the implementation of their approach. As a result, they do not consider the research challenges that arise due to the nature of COTS applications and commodity OSes.

Alcatraz [9] and its improved version *Security Execution Environment (SEE)* [10] reproduce the behavior of applications as if they were running natively on the underlying host OS. However, this approach does not achieve OS isolation, so such protection mechanism can be bypassed by kernel-mode malicious code. Besides, in SEE, a number of privileged operations, such as mounting file systems, and loading/unloading modules are not permitted.

V. UNTRUSTED CODE EXECUTION BASED ON OS LAYER VIRTUALIZATION

A typical environment for running programs should include OS, user libraries, file systems, and environment settings. If a runtime environment contains all of these key components, an application itself cannot distinguish between running on a physical system, or running in a virtualized system. The main idea of the OS layer virtualization technology is dynamically replicating the software environment of the host OS in order to create multiple isolated virtualized systems.

Jail [11] of the FreeBSD is an OS layer virtualization technology. FreeBSD is divided into multiple independent environments called Jails. Every Jail can independently manage OS resources, such as processes, file systems, and network resources. The OS resources that one user can access are bounded into this user's Jail. A Jail is created via so-called Jail's system calls. The first process of one Jail and all its sub-processes belong to this Jail. No process can simultaneously belong to multiple Jails. Jails is valuable to isolate a certain type of applications.

Zone of Solaris [12, 13] also provides a similar mechanism. A Zone refers to an a virtualized OS environment in Solaris. It is a partitioning technology used to virtualize OS services, and its purpose is to provide a safe isolated environment to host and run a variety of applications. There are two types of Zone: global zone and non-global zone. Booted from the computer hardware platform is the global zone. A Solaris can run only one global zone. The administrator of global zone can create non-global zone using *zonecfg* and *zoneadm*. The global zone holds the control of installation, maintenance, operation and damage of all non-global zones. Zone can provide the virtualized services and isolated namespaces to the processes running in non-global zones. Consequently, it can isolate the processes running a non-global zone from the progresses running in other zones. This isolation prevents the processes running in a non-global zone from monitoring or affecting the processes running in other zones. The processes running in non-global zone, even with superuser privileges, cannot view or affect the activities in other zones. Zone also provides an abstraction layer which is capable of separating the applications from the physical properties of the hardware wherein these applications are deployed.

Virtual Private Server (VPS) [14] divides the server OS environment into several isolated virtualized execution container, called VPS. The administrator can configure a specified number of memory, disk, network bandwidth and other resources for each VPS

UML (User Mode Linux) [15-17] is an open source project which makes a Linux run as a separate process on another Linux, namely host Linux). UML does not require additional virtualization software. It only needs to apply a patch to the Linux kernel source code. UML's patch converts the standard Linux kernel into an OS that can run as an independent process. When running a UML, a complete file system should be assigned to it. The architecture of UML is illustrated in Fig. 2. The UML kernel receives the system call requests from the isolated untrusted application, and then sent them to the host Linux kernel to process.

VI. UNTRUSTED CODE EXECUTION BASED ON HARDWARE-ABSTRACTION LAYER VIRTUALIZATION

As early as the 1970s, the IBM System 360/370/CP-40/CP-67 [18-21] systems have achieved the hardware-abstraction layer virtualization technology, which was originally developed in order to create multiple isolated user OS environments on one mainframe computer.

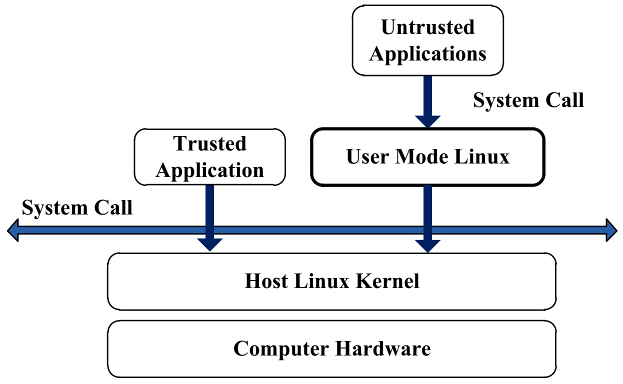
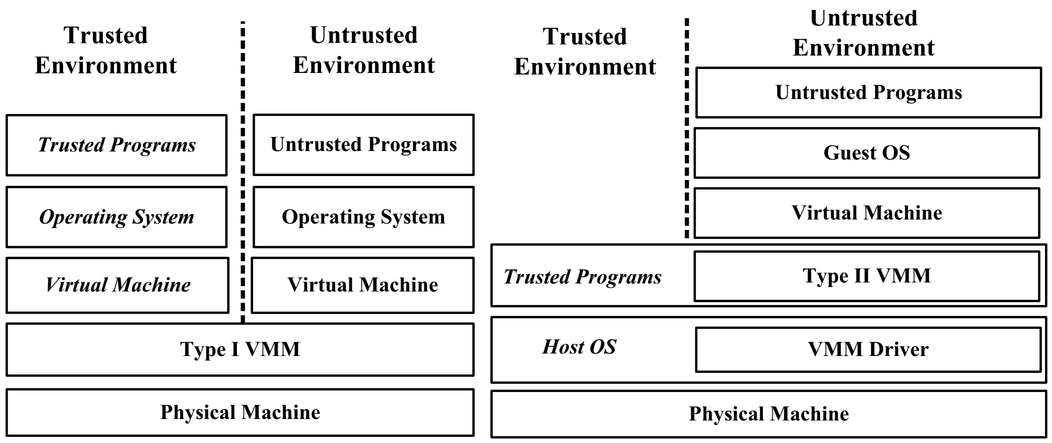


Fig.2 The Architecture of User Mode Linux.

The software running on the hardware-abstraction layer VM is named as guest OS. This virtualization technology utilized the similarity between the guest system and the host platform to reduce the execution delay of guest instructions. At present, most commercial virtualization products are just using this technology to achieve efficient and practical virtualized computer systems.

Hardware-abstraction layer virtualization technology utilizes the Virtual Machine Monitor (VMM) to serve as the a middle layer to isolate multiple execution environments. VMM, running as the abstraction of a physical computer system, provides the hardware device mapping to guest OS. In accordance with the definition of Goldberg [22], VMM is the software which could create efficient and isolated copies for the computer system. These copies are named with Virtual Machine (VM). In a VM, a subset of the virtual processor instruction set should be able to directly be executed by the physical processor. According to the differences of locations and implementation methods, Goldberg defines two types of VMMs, namely Type I VMM and Type II VMM. As shown in Figure 3,Type I VMM runs directly on a bare computer system. So, it must be firstly deployed, all the guest OSes have to be installed on top of it. With the hardware support, Type I VMM can achieve the best performance. IBM VM/370, VMware ESX Server [23], Xen [24-26], Denali [27-29] all belong to Type I VMM. Type II VMM should be installed on an existing OS (host OS). It uses the host OS to manage and access various system resources, such as files, various kinds of the I/O devices, etc. VMware Workstation [30], Parallel Workstation [31], VitualBox are widely used Type II VMM.

From the implementation point of view, VMM is responsible for mapping the virtual resources to physical resources, and use the local physical hardware to perform actual computing tasks. When the guest OS tries to access some system resources with the sensitive instructions, VMM will take over the request and perform corresponding emulations. In order to make this mechanism to work effectively, the execution of every sensitive instruction should trigger traps to make VMM be able to capture the event, and then emulate the caught sensitive instructions. Through emulating the sensitive instructions and returning the results to the guest OS, VMM achieves the



(a) Native Architecture based on Type I VMM (b) Hosted Architecture based on Type II VMM
Fig.3 The Untrusted Code Execution Architecture Based on Hardware-Abstraction Layer Virtualization.

protection and switching of different VMMs' contexts, and consequently builds multiple virtualized computer systems on the same physical hardware platform.

However, Intel x86 architecture does not fully support the virtualization, because some x86 sensitive instructions are executed on lower privilege, and cannot trigger traps. This limitation results in VMM cannot directly capture such sensitive instructions. At present, there are three technologies that can solve this problem: full-virtualization based on dynamic instruction translation, para-virtualization and hardware-assisted virtualization.

Compared to other virtualization technologies, the hardware-abstraction layer virtualization effectively implements the isolation of OS. Researchers have proposed a variety untrusted code isolated execution technologies based on the hardware-abstraction layer virtualization.

In 1995, Thomas C. Bressoud, et al., presented a prototype of a virtual machine-based fault-tolerant system [32]. They argue that the reliability and credibility of VMM are much higher than the normal OS because VMM is usually a lightweight system software,. At the same time, due to VMM runs at the highest privilege level, its security features only depend on the hardware instead of the OS.

Around 1997, Stanford University developed Disco [33, 34], it supports running multiple IRIX OSes on a large-scale cc-NUMA computer. Disco VMM shields the complexity of the underlying hardware and significantly reduces the development complexity of system software.

Covirt [35] proposes that most of the application should be executed in a VM rather than just directly running on the host environment. Renato J. Figueiredo [36], Sriya Santhanam [37], Ivan Krsul [38], etc., also propose various isolated execution mechanisms based on the hardware-abstraction layer virtualization technologies in the Grid environment.

University of Washington proposes a lightweight VMM, namely Denali [27-29]. The system design goal is to deploy a virtualized network development and testing environment on a x86 physical host. Denali introduces the so-called para-virtualization technology, i.e., reducing the complexity of the VMM itself and achieving better performance at the cost of modifying the guest OS' code. Inspired by the Denali's para-virtualization technology, University of Cambridge releases a well-known open source projects-Xen. By modifying the Linux kernel source code, Xen converts the sensitive instructions the calls to the virtual layer interface. SVGrid [39] implements a Xen-based virtual execution environment to run the untrusted Grid programs, in order to resist the potentially malicious code.

VMware ESX Server [23, 40] can run multiple copies of the same OS on the same computer system. As a Type I VMM. ESX Server must run directly on top of the hardware system. Consequently, it cannot be deployed on the PC platforms because it is incompatible with the preinstalled OSes.

In addition, the iKernel [41] utilizes the hardware-assisted virtualization to isolate the drivers that there may be bugs or malicious. The Vault [42] proposes a isolated execution environment to process and store all the sensitive data.

VII. Comparision and Conclusions

The sandbox-based approaches suffer the difficulty of security policy selection, i.e., determining the resource access policies that would allow the untrusted code to execute successfully without compromising system security. In consequence, existing sandbox-based approaches often adopt highly restrictive policies that preclude many useful applications. So, the users of PC platforms usually choose functionality over security, i.e., executing untrusted code outside such sandbox tools and exposing themselves to unbounded damage if this code turns out to be malicious.

All the untrusted code execution technologies based on API layer and OS layer virtualization discussed above suffer the same problem: they can be easily bypassed if intruders compromise the OSes and gain system privileges.

Within the native architecture, shown in Fig. 3 (a), all OSes run above the VMs, including the one acting as the trusted environment, cannot but suffer the performance penalty introduced by VMM. However, within the hosted architecture, the trusted environment, i.e., the host OS, suffers no performance degradation introduced by virtualization. In this regard, hosted architecture based on Type II VMM has the advantage of native architecture. On PC platforms, it would be unacceptable for a PC user to completely replace an existing OS with a VMM. In contrast, Type II VMM allows co-existing with a pre-existing host OS.

Besides, pushing functionality from traditional OS down one layer in the software stack into the VMM, illustrated in Fig. 3 (a), has a prominent disadvantage for development: The diversity of PC's devices would remarkably increase the development complexity of Type I VMM. There is a large diversity of devices that may be found in PCs, which is a result of the PC's "open" architecture. In the implementation of Type I VMM, it would have to manage these devices. This will require a great programming effort to provide device drivers in the VMM for all supported PC devices. But Type II VMM can be simpler to implement by using well-known OS abstractions.

Of course, Type II VMM has its own disadvantages compared to Type I VMM. The most significant trade-off of it is potential I/O performance degradation within guest OS. But such I/O performance gap between them can be closed with the improvement of hardware virtualization technology [44, 45] and the virtualized I/O device optimization technologies [46].

With the above comparisons, we can draw a conclusion that existing virtualization technologies focusing on untrusted code execution are far from feasible and effective, especially for the PC platforms. The hardware-abstraction layer virtualization technology holds the most powerful isolation ability while suffering the difficulty of being compatible with existing OS. Consequently, more additional technologies should be proposed to make a complementarity, such as the local-booted virtualization technology [47, 48].

REFERENCES

[1] T.-c. Chiueh, H. Sankaran, and A. Neogi, "Spout: A Transparent Distributed Execution Engine for Java Applets." pp. 394-401.

[2] D. Malkhi, and M. K. Reiter, "Secure Execution of Java Applets using A Remote Playground," IEEE Transactions on Software Engineering, vol. 26, no. 12, pp. 1197-1209, 2000.

[3] I. Goldberg, D. Wagner, R. Thomas et al., "A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker)," in Proceedings of the Sixth USENIX UNIX Security Symposium, San Jose, California, 1996.

[4] A. Dan, A. Mohindra, R. Ramaswami et al., ChakraVyuha(CV) : A Sandbox Operating System Environment for Controlled Execution of Alien Code, 1997.

[5] A. Acharya, and M. Raje, "Mapbox: Using Parameterized Behavior Classes to Confine Applications." pp. 1-18.

[6] K. Scott, and J. Davidson, "Safe Virtual Execution using Software Dynamic Translation," in Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC'02), 2002, pp. 209-218.

[7] N. Provos, "Improving Host Security with System Call Policies." pp. 257-271.

[8] P. Liu, S. Jajodia, and C. D. McCollum, "Intrusion Confinement by Isolation in Information Systems," Journal of Computer Security, vol. 8, no. 4, pp. 243-279, 2000.

[9] Z. Liang, V. N. Venkatakrishnan, and R. Sekar, "Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs." pp. 182-191.

[10] W. Sun, Z. Liang, R. Sekar et al., "One-way Isolation: An Effective Approach for Realizing Safe Execution Environments," in Proceedings of Network and Distributed Systems Security Symposium (NDSS'05), 2005, pp. 1-18.

[11] P.-H. Kamp, and R. N. M. Watson, "Jails: Confining the omnipotent root."

[12] D. Price, and A. Tucker, "Solaris Zones: Operating System Support for Consolidating Commercial Workloads."

[13] A. Tucker, and D. Comay, "Solaris Zones: Operating System Support for Server Consolidation."

[14] "Linux-VServer, http://linux-vserver.org."

[15] J. Dike, "A User-mode Port of the Linux Kernel."

[16] H.-J. Hoxer, K. Buchacker, and V. Sieh, "Implementing a User-Mode Linux with Minimal Changes from Original Kernel." pp. 72-82.

[17] D. Jeff, User Mode Linux: Prentice Hall, 2006.

[18] M. Schaefer, B. Gold, R. Linde et al., "Program Confinement in KVM/370." pp. 404–410.

[19] B. D. Gold, R. R. Linde, M. Schaefer et al., "VM/370 Security Retrofit Rrogram." pp. 411 - 418.

[20] R. J. Creasy, "the Origin of theVM/370 Time-sharing System," IBM Journal of Research and Development, September 1981.

[21] L. H. Seawright, and R. A. MacKinnon, "VM/370 - a Study of Multiplicity and Usefulness," IBM Systems Journal, January 1979.

[22] R. P. Goldber, "Architectural Principles for Virtual Computer Systems, Ph.D. Thesis," Ph.D. Thesis, Harvard University, Cambridge, MA, 1972.

[23] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," in Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02), 2002, pp. 181-194.

[24] P. Barham, B. Dragovic, K. Fraser et al., "Xen and the Art of Virtualization," in Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03), 2003, pp. 164-177.

[25] B. Clark, T. Deshane, E. Dow et al., "Xen and the Art of Repeated Research," in Proceedings of the Usenix Annual Technical Conference, July 2004.

[26] X. Haifeng, Q. Sihan, and Z. Huanguo, "XEN Virtual Machine Technology and Its Security Analysis," Wuhan University Journal of Natural Sciences, vol. 12, 2007.

[27] A. Whitaker, M. Shaw, and S. D. Gribble, "Scale and Performance in the Denali Isolation Kernel," ACM SIGOPS Operating Systems Review, OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation, vol. 36, no. SPECIAL ISSUE: Virtual machines, 2002.

[28] A. Whitaker, M. Shaw, and S. D. Gribble, Denali: Lightweight Virtual Machines for Distributed and Networked Applications, University of Washington Technical Report 02-02-01 2002.

[29] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: A Scalable Isolation Kernel." pp. 10-15.

[30] VMWare. "VMware Workstation: Run Multiple OS Including Linux & Windows7, on Virtual Machines. http://www.vmware.com/products/workstation/."

[31] SWsoft. "Parallels Workstation, http://www.parallels.com/en/products/workstation."

[32] T. C. Bressoud, and F. B. Schneider, "Hypervisor-based Fault-tolerance." pp. 1-11.

[33] E. Bugnion, S. Devine, K. Govil et al., "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," ACM Transactions on Computer Systems (TOCS), vol. 15, no. 4, pp. 412 - 447, November 1997, 1997.

[34] K. Govil, D. Teodosiu, and Y. H. Rosenblum, "Cellular Disco: Resource Management using Virtual Clusters on Shared-memory Multiprocessors." pp. 154-169.

[35] P. M. Chen, and B. D. Noble, "When Virtual is Better Than Real." pp. 133-138.

[36] R. J. Figueiredo, P. A. Dinda, and J. A. B. Fortes, "A Case For Grid Computing On Virtual Machines." pp. 550-559.

[37] S. Santhanam, P. Elango, A. Arpaci-Dusseau et al., "Deploying Virtual Machines as Sandboxes for the Grid."

[38] I. Krsul, A. Ganguly, J. Zhang et al., "VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing." p. 7.

[39] X. Zhao, K. Borders, and A. Prakash, "SVGrid: A Secure Virtual Environment for Untrusted Grid Applications."

[40] VMware, VMware ESX Server I/O Adapter Compatibility Guide, January 2003.

[41] L. Tan, E. M. Chan, R. Farivar et al., “iKernel: Isolating Buggy and Malicious Device Drivers Using Hardware Virtualization Support,” in Proceedings of the Third IEEE International Symposium on Dependable, Autonomic and Secure Computing, 2007, pp. 134-144.
[42] P. C. S. Kwan1, and G. Durfee, "Practical Uses of Virtual Machines for Protection of Sensitive User Data."
[43] "Kernel brk() vulnerability."
[44] G. Neiger, A. Santoni, F. Leung et al., “Intel® Virtualization Technology: Hardware Support for Efficient Processor Virtualization,” Intel Technology Journal, vol. 10, no. 3, pp. 167–177, August 2006.
[45] K. Adams, and O. Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization." pp. 2 - 13.
[46] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor."
[47] Y. Wen, and H. Wang, "A Secure Virtual Execution Environment for Untrusted Code." pp. 156-167.
[48] Y. Wen, J. Zhao, and H. Wang, "A Novel Approach for Untrusted Code Execution." pp. 398-411.