This is a pre-print version of our paper published at Elsevier Computer Networks journal, in may 2007, available at doi:10.1016/j.comnet.2006.09.007

Protecting Host-Based Intrusion Detectors through Virtual Machines

M. Laureano, C. Maziero[1], E. Jamhour
Graduate Program in Applied Computer Science
Pontifical Catholic University of Paraná – Brazil
lastname@ppgia.pucpr.br

Abstract: *Intrusion detection systems continuously watch the activity of a network or computer, looking for attack or intrusion evidences. However, host-based intrusion detectors are particularly vulnerable, as they can be disabled or tampered by successful intruders. This work proposes and implements an architecture model aimed at protecting host-based intrusion detectors, through the application of the virtual machine concept. Virtual machine environments are becoming an interesting alternative for several computing systems, because of their advantages in terms of cost and portability. The architecture proposal presented here makes use of the execution spaces separation provided by a virtual machine monitor, in order to separate the intrusion detection system from the system under monitoring. In consequence, the intrusion detector becomes invisible and inaccessible to intruders. The architecture implementation and the tests performed show the viability of this solution.*

Keywords: *security, intrusion detection, virtual machines.*

[1] Contact author.

1. Introduction

Several tools contribute to improve the security of a computing system. Among them, intrusion detection systems (IDS) stand out. Such systems continuously watch the system activity, looking for attacks or intrusion evidences. Network-based intrusion detectors scans data collected from the network to detect malicious activity, and thus can be installed on dedicated, well protected machines. On the other hand, host-based intrusion detectors analyze local data collected from computing hosts. Running as processes in the monitored system, they are particularly vulnerable to successful intruders. Once an intruder enters the system, it is able to defeat or modify the intrusion detector, in order to hide his/her presence.

Virtual machines can be used to improve the security of a computing system against attacks to its services [Chen 2001]. The virtual machine concept was defined in the 1960s: in the IBM VM/370 environment, a virtual machine created an exclusive environment for each user [Goldberg 1973]. The use of virtual machines is becoming interesting also in modern computing systems, because of their advantages in terms of cost and portability [Blunden 2002]. Examples of currently used virtual machines environments are VMWare [VMWare 1999] and UML – User-Mode Linux [Dike 2000]. A frequent use of virtual machine –based systems is the so-called server consolidation: instead of using several physical equipments, one uses a single (and more powerful) hardware equipment, in which several distinct, isolated virtual machines host distinct operating systems, applications, and services.

This work proposes and implements an architecture model aimed at protecting host-based intrusion detectors, through the application of the virtual machine concept. The architecture proposal presented here makes use of the execution spaces separation provided by a virtual machine monitor, in order to separate the intrusion detection system from the system under monitoring. This separation protects the intrusion detector, as it becomes invisible and inaccessible to guest processes (and to eventual intruders). Through modifications on the virtual machine monitor, it is possible to transparently collect information about the guest operating system activity, including users and processes. This data is then sent to an external intrusion detector, running in the host operating system. Using a previous behavior database for

comparison (created from previous executions), the intrusion detector can look for behavior deviations in guest users and/or processes. If an intrusion is suspected, a response system can act in order to prevent or defeat it. This feature is easily implemented by intercepting system calls issued by guest processes.

This article is structured as follows: section 2 recalls some virtual machine concepts used in this work; section 3 introduces intrusion detection techniques; section 4 details the proposal, section 5 describes the current implementation, section 6 presents experimental results, and section 7 discusses related work.

2. Virtual Machines

A virtual machine (VM) is defined in [Popek 1974] as an efficient and isolated duplicate of a real machine. Typical uses for virtual machine systems include the development and testing of new operating systems, simultaneously running distinct operating systems on the same hardware, and server consolidation [Sugerman 2001].

A virtual machine environment is created by a *Virtual Machine Monitor* (VMM), also called an "operating system for operating systems" [Kelem 1991]. The monitor creates one or more virtual machines on a single real machine. Each VM provides facilities for an application or a "*guest system*" that believes to be executing on a standard hardware environment. VM monitors build some properties that are useful in system security, like *isolation* (a software running in a VM cannot access or modify the monitor or other VM), *inspection* (the monitor can access the entire VM state), and *interposition* (the monitor can intercept and modify operations issued by a VM) [Kelem 1991, Garfinkel 2003].

There are two classical approaches to organize virtual machine systems: *type I*, in which the virtual machine monitor is implemented between the hardware and the guest system(s), and *type II*, in which the monitor is implemented as a normal process of an underlying real operating system, called the *host system* [Chen 2001]. This article considers the application of type II virtual machine environments in system security.

Standard PC processors provide no adequate support for virtualization [Robin 2000]. Consequently, virtualization overhead can be as high as 50% of total computing time [Blunden 2002, Dike 2000, VMWare 1999]. However, recent research significantly reduced such costs under 10%, as shown in [King 2002, King 2003, Whitaker 2002]. Using advanced techniques like on-the-fly code rewriting and host system fine-tuning, the *Xen project* [Barham 2003] obtained average computing costs under 3% for virtualizing Linux, FreeBSD, and Windows XP. These works open many perspectives on the use of virtual machines in production environments.

3. Intrusion detection

An Intrusion Detection System (IDS) continuously collects and analyzes data from a computing system, aiming to detect intrusive actions. With respect to the origin of analyzed data, there are two main approaches for intrusion detection [Allen 1999]: *network-based IDS* (NIDS), which are based on watching the network traffic flowing through the systems to monitor, and *host-based IDS* (HIDS), which are based on watching local activity on a host, like processes, network connections, system calls, log files, etc. The main weakness of host-based intrusion detection is its relative fragility: in order to collect system activity data, the HIDS software (or an agent on its behalf) should be installed in the machine to monitor. This agent can be deactivated or tampered by a successful intruder, in order to mask his/her presence.

Techniques used to analyze collected data in order to detect intrusions can be classified in: *signature detection*, when collected data are compared to a base of known attack patterns (or *signatures*), and *anomaly detection*, when collected data are compared to previously stored data representing the normal activity of the system. Normality deviations are then signaled as threats.

4. Protecting Intrusion Detectors through Virtual Machines

As previously shown, host-based IDS are vulnerable to local attacks, because the intruder can disable or tamper them. The use of virtual machines provides a solution to this problem. The proposal presented here allows building more reliable host-based intrusion detection systems.

The proposal's main idea is to encapsulate the system to monitor inside a virtual machine, which is monitored from outside (the host system). The intrusion detection and response mechanisms are implemented outside the virtual machine, i.e. out of reach of intruders. This proposal considers a type II virtual machine monitor, so the detection and response system can be implemented as normal processes on the host system. Fig. 1 illustrates the main components of the proposed architecture.

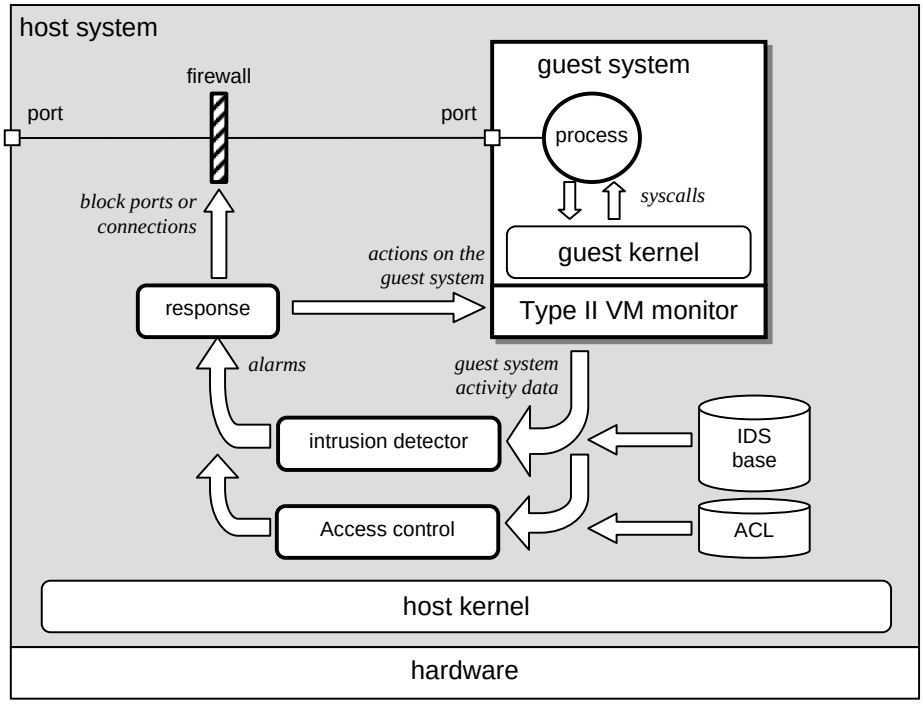

Figure 1. Proposed architecture

The interaction of guest system processes with the outside world is done only through the network, using a software firewall managed by the host kernel (like Linux *iptables*, for instance). Under the guest system's viewpoint, it is an external firewall, therefore inaccessible to intruders.

The main architecture modules are the *intrusion detector* module, which compares data collected from the guest system against a previously stored *IDS database*, the *access control* module, which checks if processes and users are known and respect a previously built *access control list*, and the *response* module, which receives *alarms* issued by the intrusion detector or

the access control module and transforms them in *actions* on the guest system and/or the host firewall.

The interactions between the guest system and the intrusion detection and response modules are carried out through the virtual machine monitor. Two types of interaction are defined: *monitoring*, in which guest data is supplied by the virtual machine monitor for external analysis and storing, and *response*, as the response module can act on the guest system in response to intrusions. Beyond actions on the guest system, the response module can also interact with the host firewall, blocking ports and connections to the guest system as needed.

4.1 Detecting intrusions on the guest system

The system calls issued by a process constitute a rich source of information about its activity. Several papers describe techniques for anomaly-based intrusion detection using such data. In the proposal presented in [Forrest 1996, Hofmeyr 1998], system calls issued by a process are sequentially recorded, discarding their parameters. This execution history is then transformed in sets of sequences of length *k*. The collection of all possible sequences of length *k* defines the normal behavior of that process. Any sequence of *k* system calls issued by that process and not present in its normal behavior (previously stored sequences) is considered an anomaly. To illustrate that technique, let us consider a UNIX process which issued the following system calls during its execution:

```
[ open  read  mmap  mmap open  read  mmap ]
```

Adopting *k*=3, the following set of sequences is obtained:

```
[ open  read  mmap ]
[ read  mmap  mmap ]
[ mmap  mmap  open ]
[ mmap  open  read ]
[ open  read  mmap ]
```

If the process issues a different sequence, like [`open open read`], it should be placed under suspicion. Despite the set of system calls to be system-dependant and the capture of the complete behavior of a process to be potentially laborious, this method presents a good efficiency, as shown by their authors [Hofmeyr 1998].

Although our current implementation adopted this anomaly-based approach for intrusion detection, using the system call sequence analysis algorithm, the architecture presented in figure 1 is generic enough to easily accept other common approaches.

4.2 Access Control

Beyond anomaly-based intrusion detection, guest data provided by the virtual machine monitor can be used to carry out other analysis. One interesting possibility is to compare guest system activity against a previously stored access-control list (ACL) which defines which users are allowed to run which executables. Users and/or executables not in the ACL should have their processes labeled as suspect. This facility is provided by the *access control module* in our architecture. As the architecture does not impose a specific access-control model, more complex models can be used as well.

4.3 Learning and monitoring

The system has two operation modes: a *learning mode* and a *monitoring mode*. When in the learning mode, the system stores the sequences of system calls for guest processes. Also, all the processes executing in the guest system and their respective users are recorded as *authorized* processes and users, thus automatically generating an access-control list (ACL). Therefore, the learning mode allows recording the “normal behavior” of the system, collecting essential data for further intrusion detection and ACL violations.

When in monitoring mode, the intrusion detection module receives data from the virtual machine monitor and compares it to the “normal” data stored previously, during the learning phase. The current prototype analyzes sequences of system calls issued by guest processes, using

the algorithm presented in [Hofmeyr 1998]. If a system call sequence issued by a given process is not found in the stored data, an anomalous situation is signaled and that process is declared suspect. Also, processes not respecting the previously generated ACL are declared suspect by the access control module.

4.4 Restricting suspect processes

Suspect processes are to be restricted in their access to the guest system, to prevent harmful actions. Such restriction is currently implemented as denying suspect processes access to some system calls. The papers [Bernaschi 2000, Bernaschi 2002] classify the UNIX system calls in functionality groups (communication, file system and memory management are some examples) and *levels of threat*. According to them, system calls classified in threat level 1 can be used to get privileged access to the operating system; the level 2 contains system calls that can be used for denial of service attacks; system calls able to compromise processes are classed in threat level 3; finally, system calls in level 4 are harmless for system security.

This classification is being used here as follows: all the system calls which can be used to gain privileged access to the guest operating system (classified as threat level 1 in [Bernaschi 2002] and shown in table 1) are denied for suspect processes. This mechanism is implemented by the virtual machine monitor, which can intercept system calls issued by guest processes. Using this approach, the guest operating system can isolate a suspect process without causing severe impact on other guest processes.

Table 1: System Calls denied to suspect processes

Group	System Calls
File system and devices	`open link unlink chmod lchown rename fchown chown mknod mount symlink fchmod`
Process management	`execve setgid setreuid setregid setgroups setfsuid setfsgid setresuid setresgid setuid`
Module management	`init_module`

The architecture presented here keeps the detection and response system out of reach of intruders. However, to guarantee the system security it is important to observe that interactions with the guest system always must be done through the virtual machine monitor. Also, the virtual machine monitor must be inaccessible to guest system processes (this is a conceptual property of virtual machine monitors). Finally, all network services must be provided by guest system processes; network access to the underlying host system should be carefully controlled.

5. Current implementation

A prototype was implemented in a Linux platform, using the virtual *User-Mode Linux* (UML) monitor [Dike 2000]. UML implements a type II monitor, which allows running Linux guest systems on top of a Linux host. It should be noticed that UML performance is fair under commercial products like VMWare [VMWare 1999], but it is open source. UML code was modified to allow extracting detailed data from the guest system, like the system calls issued by guest processes. The communication between the UML monitor and the monitoring process was done through named pipes (this way, the host operating system synchronizes the data flow between them).

Two different implementations were built: a *synchronous* and an *asynchronous* one. In the synchronous implementation, each system call issued by a guest process is sent by the monitor to the external IDS; the guest process pauses until the system call is validated. This approach is simpler to implement, but imposes a high performance cost on guest processes. On the other hand, the asynchronous implementation is more complex but offers better performance. In such approach, the monitor sends each system call issued by guest processes to the external IDS; guest processes are not imposed to wait for system call validations. If the IDS detects suspect actions coming from a guest process, it will warn the monitor through an UNIX signal. This approach leads to a small time gap between a (possible) malicious action performed by a guest process and its countermeasures (classification of such process as suspect).

The current ACL implementation consists simply on a table containing pairs [`uid`, `path`] of authorized users and executables (the table supports wildcards on both fields). Any process not matching an ACL entry will be labeled as suspect.

6. Experimental results

Using the prototype, some time measures were carried out on the execution of basic user commands, in order to evaluate the performance impact of the proposal. The utilities `ps`, `find`, `ls`, and `who` were selected because they are UNIX tools frequently tampered by intruder root kits, and because they can generate a large number of system calls during their execution.

The command execution times were measured in five situations: a) in the host system, b) in the original guest system, c) in the guest system on learning mode, d) in the guest system on monitoring mode, and e) in the guest system on monitoring mode, but using an asynchronous implementation. Observed variances were under 5% in all time measurements. The hardware used in the experiments was a dual-processor server (Dual P3 1130 MHz, 2 GBytes RAM). The host system was running a 2.6.9 SMP Linux kernel, and the guest systems used single-CPU 2.6.9 Linux kernels.

Table 2 presents the average execution times for each command and their relative overheads. The number of syscalls issued by each command execution is also presented. Execution times observed in the guest system (b) are far superior to those observed in the host system (a); this is due to the high virtualization overhead presented by UML. Also, for the synchronous implementation, the overheads imposed by modifications in the virtual machine monitor to interact with the external learning, detection, and response mechanisms are quite high, in both modes (c and d). This cost is due to the non-optimized implementation of the learning and monitoring routines and of their interaction with the UML monitor.

Table 2: Average execution times (milisseconds)

Command		`ps –ef`	`find / > /dev/null`	`ls -laR / > /dev/null`	`who –b`
# of system calls		536	10055	17225	96
(a) host	Time	25	125	802	5
(b) guest	Time	68	484	1160	29
	overhead relative to (a)	172%	287%	44%	480%
(c) learning mode	time	81	812	1784	32
	overhead relative to (b)	19%	67%	53%	10%
(d) synchronous monitoring mode	Time	107	857	1790	33
	overhead relative to (b)	57%	77%	54%	13%
(e) asynchronous monitoring mode	time	68	532	1232	30
	overhead relative to (b)	0%	10%	6%	3%

In order to evaluate the impact of our proposal on guest processes using the network, some tests were carried out using the `wget` tool (a command-line HTTP/FTP client). The tests consisted on downloading 100Kb and 1Mb remote files. Table 3 summarizes the results, which show overheads under 10% when using the asynchronous implementation.

Table 3: Average download times (milisseconds)

Test		100Kb remote file	1 Mb remote file
# of system calls		394	1737
(a) host	Time	28	154
(b) guest	Time	68	212
	overhead relative to (a)	143%	37%
(c) learning mode	time	81	432
	overhead relative to (b)	19%	103%
(d) synchronous monitoring mode	Time	117	481
	overhead relative to (b)	72%	126
(e) asynchronous monitoring mode	time	71	229
	overhead relative to (b)	4%	8%

Additionally, in order to evaluate the effectiveness of the architecture in detecting and defeating intrusions, some tests have been carried out using popular *rootkits* (described in table 3 and available at http://www.antiserver.it/Backdoor-Rootkit/).

Table 4: Rootkits used to test the prototype

Name	Description
FK 0.4	Linux Kernel Module rootkit and Trojan SSH.
Adore	Hides files, directories, processes, network traffic. It installs a backdoor and a control program.
ARK 1.0	Ambient's Rootkit for Linux . Includes backdoor versions of commands syslogd , login , sshd , ls , du , ps, pstree , killall, and netstat .
Knark v.2.4.3	Hides files, network traffic, processes and redirects program execution.
hhp-trosniff	Complete set of modifications of ssh , ssh2m sshd2, and openssh , to extract and to register origin, destination, host name, user name, and password.
ulogin.c	Universal login Trojan - Used to record login names and passwords.

These *rootkits* modify commands of the original operating system to prevent their detection (hiding the intruder's processes, files, network connections and so) and to steal typed information like logins and passwords (through modifications in commands like `telnet`, `sshd` and `login`). All tools available in those rootkits were executed with standard parameters, and all the modifications inserted by them were detected in all the executions.

The tests evidenced the effectiveness and complementarity of both mechanisms implemented in the system: the intrusion detection mechanism detects and hinders the execution of known but tampered binary files, while the access control hinders the execution of unknown binary files, or processes launched by unknown or unauthorized users.

7. Related work

The paper [Chen 2001] cited some benefits the use of virtual machines can bring to the security and compatibility of systems, as the capture and processing of log messages, intrusion detection

through the control of virtual machine internal state) or system migration easiness. However, the article does not demonstrate how these proposals should be structured and implemented, nor analyzes their impact on system performance.

The reference [Dunlap 2002] describes an experience of use of virtual machines for the security of systems. The proposal defines an intermediate layer between the monitor and the host system, called *Revirt*. This layer captures the data sent through the *syslog* process (the standard UNIX logging daemon) of the virtual machine and sends it to the host system for saving and later analysis. However, if the virtual system is compromised, the *syslog* daemon can be terminated and/or the log messages can be manipulated by the invader, and consequently are no more reliable.

The work described in [Garfinkel 2003] is the closest to our approach. It defines an architecture for intrusion detection in virtual machines called VMI-IDS (*Virtual Machine Introspection Intrusion Detection System*). Their approach considers the use of a type I monitor, executing directly on top of the hardware. The IDS executes in a privileged virtual machine and scans data extracted from the other VMs, searching for intrusion evidences. Only the low-level internal state of each virtual machine is analyzed, without taking in account the activities carried out by its guest processes. Also, the system response ability is limited: in case of intrusion suspicion, the suspect virtual machine is suspended for deeper analysis; if the intrusion is confirmed, the virtual machine is restarted from a safe state.

That approach differs from our proposal in several aspects, like the nature of collected data, the intrusion detection methods, the access control feature, and more specific intrusion response. Our proposal allows analyzing processes separately, detecting anomalous activities and hindering intrusions from compromised processes. This way, perturbations on valid guest processes are minimized. Moreover, there is no need to suspend the entire virtual machine for intrusion confirmation. Another unique feature in our proposal is the use of an authorization model (ACL) for users and processes, automatically generated during the learning phase.

An alternative approach to protect intrusion detectors from local attacks could be carried out through the use of multiple user-contexts. Some recent operating system kernels [Pfitzmann

2001, Embry 2001, VServer 2004, Tucker 2004] can define several autonomous and isolated user contexts. In such approach, the intrusion detector and the response system would be installed on a more privileged context, from which they could monitor and act on processes running in the other contexts. This approach can achieve good performance results, but imposes the same operating system to all user contexts.

8. Conclusion

This paper describes a proposal to increase the security of computing systems using virtual machines. The basis of the proposal is to monitor guest processes' actions through an intrusion detection system, external to the virtual machine. The data used in intrusion detection is obtained from the virtual machine monitor and analyzed by an IDS process in the underlying real machine. The detection system is inaccessible to virtual machine processes and cannot be subverted by intruders. Also, the intrusion detection module is able to track the activity of isolated processes, and the response module can restrict their execution without disturbing other non-related guest processes.

The main objective of the project, to hinder the execution of suspect process in the virtual machine and consequently avoid the system compromise, was reached with the current prototype. However, complementary work must be done to improve the performance of the current intrusion detection and response mechanism and thus to minimize its overhead. We are currently investigating to UML, and improving the current prototype implementation.

Another aspect to be refined is to define more flexible ways to interact with the guest kernel, allowing killing or suspending suspect guest processes. Also, the interactions between the response module and the host system firewall, to block suspect network traffic, need to be detailed and implemented.

In order to ease the use of the system, next prototype will allow both monitoring and learning modes to occur simultaneously, for distinct processes. This would allow the system to "learn" about a recently installed application, while monitoring the other guest processes.

Other questions to be studied include implementing detection mechanisms based on other relevant data, like the network traffic generated by the virtual machine, and the behavior of guest users. Maybe faster and more sophisticated algorithms for intrusion detection can be implemented based of such information, helping to reduce the occurrence of false results (positive and negative).

Publication history for this research:

- 08/04: Euromicro 2004: short version, without asynchronous implementation and ACL.
- 04/04: WSeg 2004 (Brazilian workshop on security): idem, in Portuguese.
- 10/03: SSI 2003 (Brazilian conference on security): first version, in Portuguese.

All above papers are available online at http://www.ppgia.pucpr.br/pesquisa/sisdist/publica.htm.

References

J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, E. Stoner. "*State of the Practice of Intrusion Detection Technologies*", Technical Report CMU/SEI-99-TR028. Carnegie Mellon University, 1999.

P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. "*Xen and the Art of Virtualization*", Proceedings of the ACM Symposium on Operating Systems Principles – SOSP, 2003.

M. Bernaschi, E. Grabrielli, L. Mancini. "*Operating System Enhancements to Prevent the Misuse of System Calls*", Proceedings of the ACM Conference on Computer and Communications Security, 2000.

M. Bernaschi, E. Grabrielli, L. Mancini. "*REMUS: A Security-Enhaced Operating System*", ACM Transactions on Information and System Security. Vol 5, number 1, 2002.

B. Blunden. "*Virtual Machine Design and Implementation in C/C++*", Wordware Publ. Plano, Texas – USA, 2002.

P. Chen, B. Noble. "*When Virtual Is Better Than Real*", Proceedings of the Workshop on Hot Topics in Operating Systems – HotOS, 2001.

J. Dike. "*A User-mode port of the Linux Kernel*", Proceedings of the 4th Annual Linux Showcase & Conference. Atlanta – USA, 2000.

G. Dunlap, S. King, S. Cinar, M. Basrai, P. Chen. "*ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay*", Proceedings of the Symposium on Operating Systems Design and Implementation – OSDI, 2002.

R. Embry. *FreeVSD Enables Safe Experimentation*. Linux Journal, Issue 87, July 2001.

S. Forrest, S. Hofmeyr, A. Somayaji. "*A sense of self for Unix processes*", Proceedings of the IEEE Symposium on Research in Security and Privacy, 1996.

T. Garfinkel, M. Rosenblum. "*A Virtual Machine Introspection Based Architecture for Intrusion Detection*", Proceedings of the Network and Distributed System Security Symposium – NDSS, 2003.

R. Goldberg. "*Architecture of Virtual Machines*", AFIPS National Computer Conference. New York – NY – USA, 1973.

S. Hofmeyr, S. Forrest, A. Somayaji. "*Intrusion Detection using Sequences of System Calls*", Journal of Computer Security, 6:151–180, 1998.

N. Kelem, R. Feiertag. "*A Separation Model for Virtual Machine Monitors*", Research in Security and Privacy. Proceedings of the IEEE Computer Society Symposium, pages 78-86, 1991.

S. King, P. Chen. "*Operating System Extensions to Support Host Based Virtual Machines*", Technical Report CSE-TR-465-02, University of Michigan, 2002.

S. King, G. Dunlap, P. Chen. "*Operating System Support for Virtual Machines*", Proceedings of the USENIX Technical Conference, 2003.

Linux VServer Project. http://www.linux-vserver.org, 2004.

B. Pfitzmann, J. Riordan, C. Stüble, M. Waidner, A. Weber. *The PERSEUS System Architecture*. Research Report RZ 3335 04/09/01, IBM Research Division, Zurich, April 2001.

G. Popek, R. Goldberg. "*Formal Requirements for Virtualizable Third Generation Architectures*", Communications of the ACM. Volume 17, number 7, pages 412-421, 1974.

J. Robin, C. Irvine. "*Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor*". Usenix Security Symposium, 2000.

J. Sugerman, V. Ganesh, L. Beng-Hong. *Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor*. Proceedings of the USENIX Annual Technical Conference, 2001.

A. Tucker, D. Comay. *Solaris Zones: Operating System Support for Server Consolidation*. 3rd Usenix Virtual Machine Research & Technology Symposium, 2004.

VMware Inc. "*VMware Technical White Paper*", Palo Alto – CA – USA, 1999.

A. Whitaker, M. Shaw, S. Gribble. "*Denali: A Scalable Isolation Kernel*", Proceedings of the 10th ACM SIGOPS European Workshop, Saint-Emilion – France, 2002.