KvmSec: A Security Extension for Linux Kernel Virtual Machines

Flavio Lombardi
Consiglio Nazionale delle Ricerche
Ufficio Sistemi Informativi
Piazzale Aldo Moro 7, 00185 Rome, Italy
flavio.lombardi@cnr.it

Roberto Di Pietro*
UNESCO Chair in Data Privacy
Universitat Rovira i Virgili
Tarragona, Spain
roberto.dipietro@urv.cat

ABSTRACT

Virtualization is increasingly being used in regular desktop PCs, data centers and server farms. One of the advantages of introducing this additional architectural layer is to increase overall system security.
In this paper we propose an architecture (*KvmSec*) that is an extension to the Linux Kernel Virtual Machine aimed at increasing the security of guest virtual machines. KvmSec can protect guest virtual machines against attacks such as viruses and kernel rootkits. KvmSec enjoys the following features: it is transparent to guest machines; it is hard to access even from a compromised virtual machine; it can collect data, analyze them, and act consequently on guest machines; it can provide secure communication between each of the guests and the host; and, it can be deployed on Linux hosts and at present supports Linux guest machines. These features are leveraged to implement a real-time monitoring and security management system. Further, differences and advantages over previous solutions are highlighted, as well as a concrete roadmap for further development.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems —*Security and Protection*; K.6.5 [**Security and Protection**]: Unauthorized access —*hacking*

General Terms

Virtual Machine, Security

Keywords

Hypervisor, Kernel Virtual Machine, Integrity, Real-Time Monitoring.

*Also with Dipartimento di Matematica, Università di Roma Tre. E-mail: dipietro@mat.uniroma3.it

1. INTRODUCTION

Virtualization is an old idea that is living a new golden era. It is becoming widely used in regular desktop PCs and increasingly in data centers and server farms, where server consolidation is a real need. So far, the most widely adopted x86 virtualization solutions are: VMware, Xen [4], User Mode Linux, Qemu [5] and KVM [10]. Even though the reliability of virtualization solutions has increased in recent years, the level of robustness to attacks compromising the security of services and operating systems inside virtual machines is a vexed issue. Malicious intruders often succeed in attacking a system and gaining administrator privileges. In this way, malware such as trojans and backdoors can be inserted and important or private file content can be accessed. Since most kind of attacks imply changes to some part of the filesystem, intrusion detection tools usually check against file modification, especially for those files in the critical paths.

The integrity tool and data it collects are usually placed on the same system that is being monitored. This is a problem in case such a system gets compromised because malware might tamper with the hooks of the monitoring system. However, the additional layer introduced by virtualization can increase the overall system security since a virtual machine (also *VM*, *guest VM* or simply *guest*) can be protected by additional boundaries [3]. In particular, an hypervisor (also Virtual Machine Monitor, *VMM* or *host* in the following) can provide a trusted computing base where the monitoring system can be located, out of reach of malware. In such case, even if an attacker succeeds in compromising a VM, the traces of its attempt cannot be deleted and the external security system cannot be deactivated.

Contributions This paper addresses the problem of secure monitoring of guest machines in a server consolidation scenario, i.e. where multiple guests run on a single host. The primary research contribution we give is the architecture of a system (KvmSec) that can monitor and protect such guests. The presented approach to real-time integrity allows the host to control unauthorized changes inside guest virtual machines. KvmSec can detect and react to such anomalies in guests. It is an extension of the Linux Kernel Virtual Machine [10] minimizing accessibility and visibility of the monitoring system itself.

Roadmap The rest of the paper is organized as follows: Section 2 surveys present technologies and related work and provides background information for our work; Section 3 describes KvmSec requirements and architecture; Section 4 provides some implementation details; Section 5 discusses

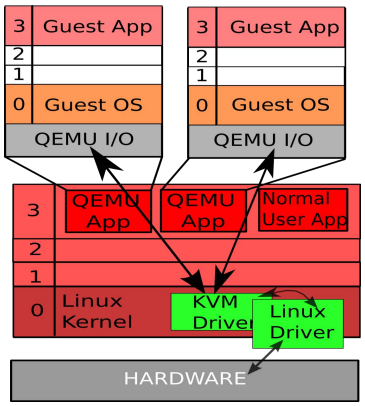

Figure 1: KVM Execution Model

Feature	Kvm	Xen
Device Drivers	standard Linux kernel	derived from Linux kernel
Upgradeability	modules allow possibly easy real-time upgrade	requires restarting host
HW Support	wholly supported by standard kernel	mostly supported by standard kernel
Flexibility	seamlessly hosted on 2.6.x kernels	already packaged in most distributions
Maturity	recently inserted into streamline kernel	mature commercially-supported project

Table 1: KVM vs Xen

KvmSec compared with previous results; finally, in Section 6 conclusions are drawn.

2. BACKGROUND

2.1 Virtualization Architectures

In the following, we analyze the most relevant open source virtualization architectures: *Xen* and *KVM*, in order to justify why we chose the latter. *Full-virt* is a technology that exploits CPU virtualization support (AMD-V and Intel-VT). CPUs supporting such a technology feature a new ringlevel that allows operating systems in guest VMs to run unmodified and unaware of the existence of an additional higher privileged ringlevel. The other approach to virtualization is *para-virt*. Such approach does not exploit CPU virtualization support and usually requires changes to guest operating systems.

Xen, the most widely adopted virtualization solution in research, is composed of The *Xen* hypervisor (or VMM), a privileged VM (Dom0) and a common VM (DomU) where the guest OS is executed. *Xen* features: 1) Shared Memory: a communication channel between VMs; 2) Ether Channel: a signaling channel between VMs; 3) Shared Memory Access Control: an access matrix describing which VM can access shared memory. *Xen* drivers are confined in the privileged Dom0 domain. All I/O requests by the various domains are performed by Dom0 and it is the hypervisor responsibility to switch from DomU to Dom0 when a I/O operation is requested.

KVM is the new mainstream Linux virtualization solution, being part of the official Linux kernel source since version 2.6.20. *KVM* consists (see Fig. 1 for the execution model) of an hypervisor (a Linux kernel module) and a modified version of the Qemu [5] emulation software. *KVM* is a standard kernel module, as a consequence it exploits the standard, reliable and frequently updated Linux device drivers. On one hand, this is one of the reasons why *KVM* is less vulnerable to attacks than *Xen*, whose driver development is slower than standard Linux. On the other hand, kernel codebase is overall larger than *Xen* and can potentially contain more vulnerabilities and bugs. A comparison of the features provided by *Xen* and *KVM* is summarized in Table 1. It appears that *KVM*, while not being fully mature yet, offers clear advantages over *Xen*, in particular, regarding wider hardware support and increased flexibility (e.g. the redeployment of a newer *KVM* version does not require a host reboot). Furthermore, large efforts are being devoted to put both KVM and Qemu in production.

2.2 Related Work

In the following we briefly survey the most relevant solutions to the problem of integrity monitoring, that can be reported to the broader area of intrusion detection.

The IBM Integrity Management Architecture (*IMA*)[13] [8] offers integrity measurement via SHA checksums. Executables, library and kernel module checksums are calculated and stored by the modified Linux kernel itself on the fly at load-time. In addition, *IMA* hashes can be stored in a protected hardware Platform Configuration Register (PCR) of a Trusted Platform Module (TPM) security chip attached to the system and a remote party can validate the system integrity by comparing stored and remotely calculated hashes. Although the impact of checksumming on performance can be significant and CPU-based virtualization support is not exploited, IMA offers a complete working reference architecture.

The vast majority of present approaches leverage the *Xen* hypervisor isolation capabilities: The *sHype* [7] system for Xen offers the isolation of VMs with MAC enforcement and proposes an approach to manage covert channels whereas Yang [16] modifies Xen to protect user application data privacy by removing the operating system from the trusted base. *XenFIT* [11] is a real-time Filesystem Integrity Tool protecting a database and the FIT system from the attacker by exploiting the isolation given by the Xen hypervisor. In fact, both FIT and the database are deployed onto a separate VM. By the same authors, *XenRIM* is composed of a module (XenRIMU) in DomU, collecting information about the client, and a daemon process (Xenrimd) in Dom0 that checks the information collected by XenRIMU and reports any violation of the security policy.

SecVisor [14] exploits CPU-based virtualization support in order to create a minimal hypervisor ensuring Linux kernel code integrity against code injection attacks. SecVisor codebase is very small, which is good as it reduces the TCB size, but unfortunately it supports only a single guest. As a consequence *SecVisor* is not suitable for OS security in a server consolidation scenario. Furthermore, it only succeeds

in protecting kernel code but not kernel data.

Lares [9] is an architecture for active monitoring using virtualization that is placed mostly in a separate Xen VM and installs hooks in the guest VM. In order to ensure such hooks are not overwritten, *Lares* features a memory protection system using per-page write permissions with additional finer checks. *Lares* first prototype seems to perform well, however its monitoring approach makes use of hooks and can potentially be detected by the guest through performance analysis.

XenKimono [12] is an IDS aimed at discovering malicious intrusions by analyzing the guest kernel internal data structures from outside (as later leveraged in a work by Tamberi[15]). All *XenKimono* modules are inside the host machine and analyze the VM raw memory for malware (e.g.. rootkits). The translation of raw VM memory in higher level kernel data structures is done by extracting kernel symbols from the DomU kernel binaries and exploiting the LKCD library [2]. This way XenKimono is able to locate DomU kernel data structures in raw memory.

3. KVMSEC: A SECURE MONITORING ARCHITECTURE

3.1 Threat Model

We suppose the hypervisor and host kernel are part of the trusted computing base, whereas guests are not. Attacks can be performed and malware can be injected inside guest virtual machines at runtime. When running, the guests can be subject to viruses, code injection, buffer overrun and all possible kinds of intrusion. The intruder might exploit zero-day vulnerabilities affecting the kernel and applications and can remotely exploit such vulnerabilities in an attempt to gain root privileges.

3.2 Requirements and caveats

In the following, based on the above introduced threat model, we describe the main requirements of a Security System for virtual machines, some caveats regarding the problems that can be encountered, and a possible way to solve them.

Requirements: A security system for VMs has to leverage core concepts of Intrusion Detection Systems [6]. Such a system must match the following requirements:

RQ1 Transparency: the system should minimize visibility from the VMs; that is, potential intruders should not be able to detect the presence of a monitoring system.

RQ2 Immunity to attacks from the Guest: the host system and sibling guests should be protected from attacks proceedings from a compromised guest; further, features provided by the host system should not be affected.

RQ3 Deployability: the system should be deployable on the vast majority of available hardware.

RQ4 Dynamic reaction: the system should detect an intrusion attempt in a guest and take appropriate actions against it or against a compromised guest.

Caveats:

PR1 A communication channel from guest to host is required for the latter to read useful data/extract information from the former, but it has to be hidden to guest userspace (see RQ1).

PR2 A signaling channel between guest VM and host allows the latter to be notified when interesting events take place in the former, but again it should be as hidden as possible (see RQ1).

PR3 Some kind of access control to the above channels is required in order to ensure consistency and to protect from information leakage; however, such an access control mechanism should not have a negative impact on performance.

4. KVMSEC: IMPLEMENTATION

We implemented a prototype of KvmSec to prove the viability of our approach. The KvmSec architecture is composed of multiple modules living both in the host and (optionally) in the guest kernels that communicate (through secure channels) to provide the host with up to date and accurate guest status. The core modules of the detection system are located on the host machine (so that an attacker confined in a guest hardly can reach them). Data: (a) can be collected by guest VM processes or (b) can be collected exclusively by processes located on the host side.

In particular, (a) allows to collect more accurate and complete guest data, but can be easier to detect. A daemon in the guest is devoted to collecting data of interest and this can help reducing the computational load at the host side. However, notice that there is a tradeoff between monitoring system stealthiness and reduction of computational load on host. As for (b) (no guest component), this technique is theoretically less detectable (see RQ1), but only allows a more limited monitoring. For this reason KvmSec preferred operating mode is (a) but (b) is also supported.
It is worth noticing that in KvmSec each Virtual Machine uses its own private memory area for communicating with the host, so it is totally independent from other VMs (see RQ2).

KvmSec implementation (see Fig. 2) is divided in two major sections: host and guest. Both have a similar structure, that is: 1) a kernel daemon that manages and shares a communication channel; 2) a module that dynamically receives messages, analyzes them and then acts (e.g. generating responses).

In the following we briefly outline the solutions we adopted for KvmSec in response to the technical difficulties we faced, while still complying the stated requirements:

SL1 Since a host-guest communication system is not available in KVM, we had to devise a simple communication system using shared memory (see PR1).

SL2 Lack of a guest to host signaling channel in KVM lead us to design a simple signaling mechanism also using shared memory (see PR2). We chose not to implement such a system as the *event channel* for Xen also because the implementation of a signaling channel would have made the whole KvmSec more visible to an attacker in the guest (see RQ1 and RQ2).

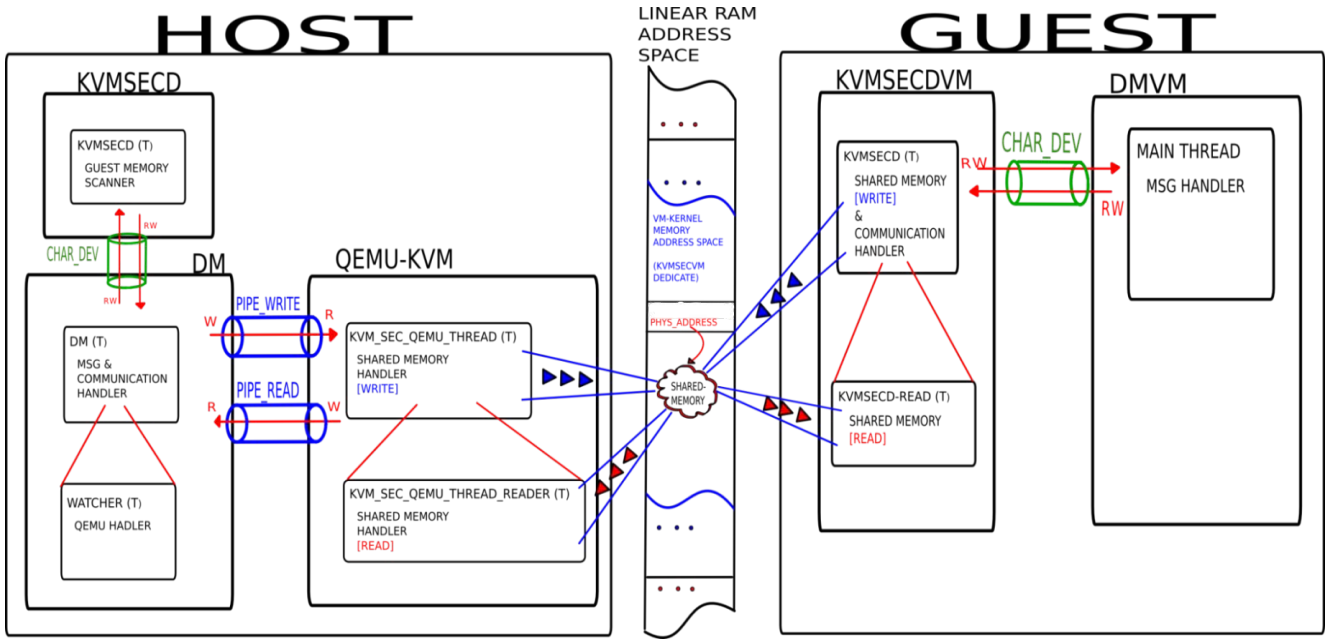

Figure 2: KvmSec Architecture: modules, channels and threads

SL3 Lack of access control to shared memory in KVM lead us to synchronize host-guest accesses to shared memory. In order to simplify access control management, each KvmSec VM has been provided its own shared memory area for communications with the host. Furthermore, a simple lock mechanism is implemented for each of the two unidirectional channels in order to synchronize access to the shared memory area where messages transit (see PR3).

In KVM, unlike Xen, shared memory is not directly managed by the hypervisor but by the main emulation process, that is Qemu-KVM. The choice of a communication channel using shared memory is in compliance with RQ1. In fact, using a virtual network socket between host and guest would have resulted in a visible and vulnerable communication channel (as happens in AIDE[1]). In addition, the message handler is contained in a guest kernel module in order to make it as much secure as possible, in accordance with RQ2. On the host side the message handler is implemented inside the shared memory management module within Qemu-KVM. KVM shared memory (see Fig. 3) is composed of two data buffers and two locks that protect the corresponding critical sections.

To meet RQ4 KvmSec should also be able to actively monitor critical processes running inside guest VMs. At present, such functionality is not fully implemented; however KvmSec can periodically check the existence and liveness of a number of daemon processes inside guests. If one of these processes is (abnormally) terminated, appropriate countermeasures are taken by the host, including collecting data for forensic analysis, or even freezing the guest or possibly restarting it (using a clean disk image if available). KvmSec can create a host-side database containing computed checksums for selected critical path files of VMs. A runtime daemon can then recompute the hash values for monitored files. If a mismatch is found, countermeasures such as those just depicted above, can be taken.

The communication protocol (see Fig. 4) between the various modules is similar. Salient differences between host and guest are:

1. Management and allocation of shared memory: In the guest the shared memory is allocated and managed by the kernel module, whereas in the host the shared memory must be already allocated (in the VM) and its management is delegated to the Qemu-KVM process.

2. The number of modules: In the VM we only need a pair of modules because shared memory management is delegated to the kernel whereas in the host we need three modules, as explained in the following subsection.

4.1 KvmSec Host side

The host part is composed of three modules: KvmSecD, DM and Qemu-KVM.

1) KvmSecD: It is a kernel daemon and has access to all VM address space. In addition, this module is aware of the other two host daemons (Qemu-KVM and DM) since they register their pids with KvmSecD.

Communication channel towards DM: The communication between KvmSecD and DM is managed through a combination of character devices (called *char_dev*) controlled by the DM through the IOCTL interface and POSIX signals. The capabilities of the character devices are extended using the IOCTL interface that allows defining a series of macros that can be used to interact with the kernel module and to define access policies to such devices.

In every communication phase the core elements (that is, the buffers) are protected by locks inside the kernel module (for that purpose our code makes use of the following

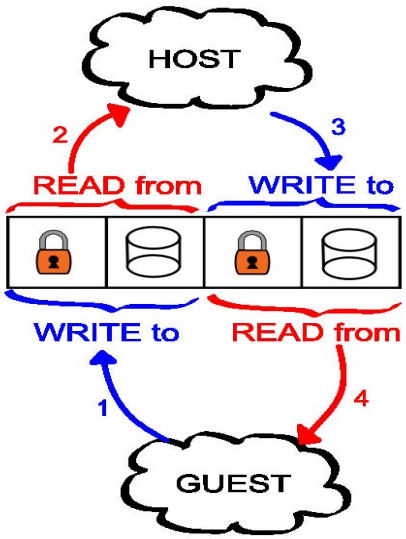

Figure 3: KvmSec Shared Memory Access Model

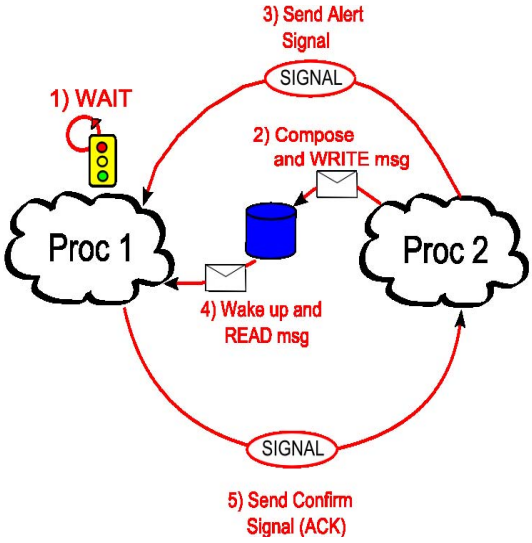

Figure 4: KvmSec Communication Protocol

primitives: semaphores (*sem_wait*), mutexes (*mux*), atomic variables (*reg* and *qemu_alive*). Access to the buffer by a process (the DM) is done through macros since the critical section has to be protected. In this way the system is scalable to more than one userspace process, since all of them access the buffer through these macros.

2) DM: DM is the first of two userspace daemons and is composed of two threads:

1. DM - This is the main thread that manages: a) The communication among DM and KvmSecD, DM, and Qemu; b) The creation and reception of messages to and from the shared memory throughout Qemu-KVM; c) The registration of the DM pid into KvmSecD;

2. WATCHER - This is a secondary thread that manages: a) The Qemu-KVM startup; b) The registration of the Qemu-KVM pid into KvmSec; c) The abnormal termination of Qemu-KVM;

Communication channel with Qemu: Since both the DM and the Qemu processes are executed in userspace, we can use any of the Linux System V IPC facilities for Inter-Process Communication. In particular, we chose to make use of a named PIPE (or FIFO).

3) Qemu-KVM: Qemu-KVM is a modified version of Qemu that incorporates the communication mechanism with the kernel module KVM.

Communication protocol between Qemu and VM (HOST and VM) : The communication protocol between VM and host relies on the synchronized access to the shared memory area. Qemu provides the *cpu_physical_memory_rw* function that allows to write in the VM memory. The host-VM synchronization is built onto this function. In this way multiple accesses to the two read and write buffers are synchronized, protected and OS-independent.

4.2 KvmSec Guest side

KvmSec guest consists of two modules: KvmSecDVM and DMVM.

1) KvmSecDVM: It is a Linux kernel daemon that manages the VM communications with the host. This daemon has access rights to kernel memory. The shared memory for communication is allocated by this module. In addition, we allocate a buffer that contains the PHYSICAL address of the shared memory. The communication protocol is identical to the one seen above.

2) DMVM: It is a daemon dealing with the monitoring, analysis and creation of responses tasks. This module manages messages to and from the shared memory. The communication channel with KvmSecDVM is a character device (*char_dev*) as in the host. The communication protocol is the same as the one between DM and KvmSecD. The module checks for critical path file access and modification. Future implementations of this module will provide room for other integrity checking plugins.

Environment	Kernel unbz2	Kernel build
Kvm guest	214	35607
KvmSec guest	243	41503

Table 2: Kvm-KvmSec performance in seconds

5. DISCUSSION

The goal of KvmSec is to provide a potentially undetectable and unsubvertible system that can catch integrity violations of virtual machines. The matching of the features provided by the KvmSec against previously highlighted requirements are reported in the following.

Performance: We have carried out basic performance tests [14] to compare KvmSec against standard Kvm. In particular, we have measured the time required: to compile a standard 2.6 kernel with standard configuration (Kernel build) and to decompress its source code tarball (Kernel unbz2). Tests have been conducted on a Vaio laptop, with 2GB of RAM and 2.1 GHz processor using a single core, running Fedora 9 x86 (both guest and host). Preliminary results are reported in Table 2, and show that KvmSec requires a slight additional overhead with respect to Kvm.

Transparency: KvmSec guest-host messages are not han-

dled via the standard network stack (as happens in most integrity architectures, see Section 2.2); so they are virtually undetectable (RQ1); KvmSec relies entirely on its own internal communication protocol (see Section 4.1), which makes it independent from the adopted virtualization system, unlike most proposals based on the Xen hypervisor. Furthermore, in KvmSec each VM has its own reserved shared memory area for guest-host communications; this makes each communication channel independently managed and isolated from other channels (RQ2).

Signalling: KvmSec is potentially less detectable than other systems in literature (e.g. XenRim) since there is no explicit signalling channel between host and guest (RQ1) (see Section 4).

Daemon Process: KvmSec can share the monitoring task between host and guest. This should offer performance benefits and increase quality of monitoring in normal conditions. The tradeoff is that a monitoring guest module might render the whole system more detectable. Anyway such guest module is not strictly required in KvmSec architecture (RQ1).

Resistance to compromising: Note that the core detection system is located on the host side, thus making the system harder to compromise. In addition, the module that manages host-guest communication resides within the kernel of the guest (see Section 4) (RQ2).

Deployment: KvmSec can be deployed on any recent standard Linux kernel whereas most other proposals (e.g. XenKimono) require the *Xen* virtualization system to be installed and to run on the host machine. As a consequence, the number of supported host platforms is much larger for KvmSec than for Xen-based solutions (see Section 2.1) (RQ3).

6. CONCLUSIONS

In this paper we propose an architecture that extends the Linux Kernel Virtual Machine. In particular, we extend KVM focusing on security issues, providing a solution (KvmSec) for real-time integrity monitoring of virtual machines.

To the best of our knowledge this is the first work addressing security issues inside the Linux KVM. The KvmSec prototype we developed enjoys the following features: it is transparent to guest machines (even malicious guest machines); it supports *full-virt*, which renders the system less detectable on guest side; it can collect data and react on guest machines but its core resides in the protected host machine; it provides secure two way communications between the hypervisor and guest machine; it can be deployed on most x86 and x86_64 machines.

As for further research directions, we are striving to ease the way IDS modules and Filesystem Integrity Tools can be plugged in to extend the functionality of the system, fully leveraging the KvmSec features mentioned above. Further investigation will also regard performance issues and the usage of Windows-family guest OSes.

7. ACKNOWLEDGMENTS

This work was partly supported by the Spanish Ministry of Science and Education through projects TSI2007-65406-C03-01 E-AEGIS and CONSOLIDER CSD2007- 00004 ARES, and by the Government of Catalonia (grant 2005 SGR 00446). We would like to acknowledge Francesco Cipollone for his contribution to some practical experiments on kvmsec.

8. REFERENCES

[1] Advanced intrusion detection environment. http://sourceforge.net/projects/aide, 2005.

[2] Sgi inc. lkcd - Linux kernel crash dump. http://lkcd.sf.net, April 2006.

[3] M. Aiken, M. Fähndrich, C. Hawblitzel, and al. Deconstructing process isolation. In *MSPC '06: Proc. of the 2006 workshop on Memory system perf. and correctness*, pages 1–10, New York, NY, USA, 2006.

[4] P. Barham, B. Dragovic, K. Fraser, and al. Xen and the art of virtualization. *SIGOPS Operating Syst. Review*, 37(5):164–177, 2003.

[5] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proc. of USENIX Annual Tech. conf.*, pages 41–41, Berkeley, CA, USA, 2005.

[6] R. Di Pietro and L. V. Mancini. *Intrusion Detection Systems*, volume 38 of *Advances in Information Security*. Springer-Verlag, 2008.

[7] T. Jaeger, R. Sailer, and Y. Sreenivasan. Managing the risk of covert information flows in virtual machine systems. In *SACMAT '07: Proc. of the 12th ACM symposium on Access control models and technologies*, pages 81–90, New York, NY, USA, 2007.

[8] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and al. Linux kernel integrity measurement using contextual inspection. In *STC '07: Proc. of the 2007 ACM workshop on Scalable trusted computing*, pages 21–29, New York, NY, USA, 2007.

[9] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *SP '08: Proc. of the 2008 IEEE symposium on Security and Privacy)*, pages 233–247, Washington DC, USA, 2008.

[10] Qumranet. Linux kernel virtual machine. http://kvm.qumranet.com.

[11] N. A. Quynh and Y. Takefuji. A novel approach for a file-system integrity monitor tool of xen virtual machine. In *ASIACCS '07: Proc. of the 2nd ACM symp. on Information, computer and comm. security*, pages 194–202, New York, NY, USA, 2007.

[12] N. A. Quynh and Y. Takefuji. Towards a tamper-resistant kernel rootkit detector. In *SAC '07: Proc. of the 2007 ACM symposium on Applied computing*, pages 276–283, New York, NY, USA, 2007.

[13] R. Sailer, X. Zhang, T. Jaeger, and al. Design and implementation of a TCG-based integrity measurement architecture. In *SSYM'04: Proc. of the 13th conf. on USENIX Security symposium*, pages 16–16, Berkeley, CA, USA, 2004.

[14] 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: Proc. of ACM SIGOPS symposium on Operating systems principles*, pages 335–350, New York, NY, USA, 2007.

[15] F. Tamberi, D.Maggiari, D. Sgandurra, and al. Semantics-driven introspection in a virtual environment. In *Proc. ISIAS 08*, pages 299–302, 2008.

[16] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proc. of the ACM intl. VEE conf.*, pages 71–80, New York, NY, USA, 2008.