Detection of Malware and Kernel-level Rootkits in Cloud Computing Environments

I. INTRODUCTION

Virtualization infrastructure enables the sharing of physical computing resources among multiple users for cloud computing services.

A virtualization environment presents virtual machines that access the shared physical resources of a host server via a virtual machine manager (aka., hypervisor).

Cyberattacks targeted at the virtualization infrastructure becomes sophisticated. Attacks such as VENOM (Virtualized Environment Neglected Operations Manipulation) enable an attacker to break out of a guest virtual machine and access the underlying hypervisor.

Approaches to malware detection in cloud computing environments can be classified into distributed and hypervisorbased malware detection. Distributed malware detection consists of an in-VM agent running within the guest VM, and a remote management server monitoring its behaviour. While this enables a single point of control for attack detection within guests, the need for a signature database makes it vulnerable against zero-day attacks.

Hypervisor-based malware detection, on the other hand, involves the use of the underlying hypervisor to detect malware within the guests. While this protects the integrity of the monitored results, it requires significant modifications to the hypervisor making it infeasible for deployment in a production environment.

In this paper we present a novel virtualization security system. It combines system call monitoring and system call hashing in the guest kernel together with SVM-based external monitoring on the host. By this way, our malware and rootkit detection solution protects the guests against attacks without incurring significant performance overhead.

The remainder of the paper is arranged as follows. Section II presents our virtualization security solution. Section III presents a detailed implementation of the proposed solution.

Section IV presents the performance and evaluation results of the implemented solution, with future work discussed in Section VI.

II. INTEGRATED DETECTION OF MALWARE AND KERNEL-LEVEL ROOTKIT

A. System architecture

The design rationale behind our approach is to implement a security system that detects in real-time the presence of attacks in guests, without the need for a signature database. To achieve this goal, we set the following design guidelines.

• r1 - Transparency: The system should be able to get a comprehensive view of the guest activities.

• r2 - Efficiency: The system should be able to detect attack presence with as little performance overhead as possible.

• r3 - Real-time: The system should be able to detect all possible threats and take proactive action as soon as it happens.

• r4 - Behavior-based: The system should be able to detect attack presence without having to rely on a signature database.

• r5 - Deployability: The system should be readily deployable in production environment with minimal effort.

Our virtualization security solution consists of a protected monitoring module which runs within the guest kernel space. Installed within the guest after its creation, it monitors the guest activities (r1) and passes any suspicious activities to the control monitor running on the host in real time (r3). To ensure that it is protected against removal from the guest, it is password protected and can only be removed by the system administrator.

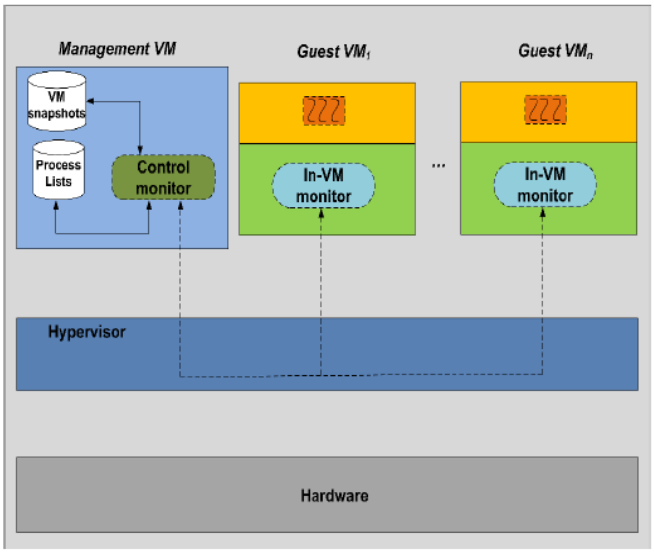

Fig. 1. Integrated malware detection solution

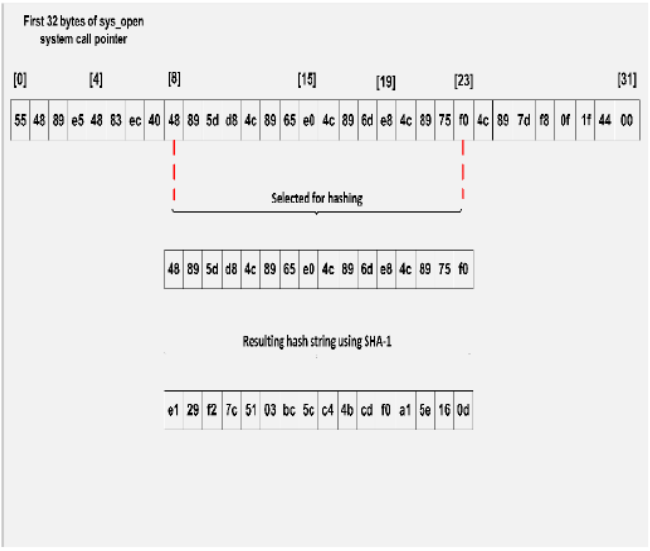

Fig. 2. System call hashing

In addition, we determine the presence of an attack by monitoring the guest behaviour rather than using a signature database. This is done through selective system call interception, the details of which are discussed in the next subsection. This enables us to obtain an accurate view of the guest behaviour, without having to intercept all possible system call functions (r2).

Our virtualization security solution features the use of three unique techniques, namely system call monitoring, system call hashing and SVM-based attack classification.

B. System call monitoring

Used by userspace processes to request privileged instructions from the kernel, system calls provide an insight into the guest behavior and are monitored in existing malware detection approaches.

Monitoring system calls typically involves intercepting every system call triggered from the guest userspace and verifying their parameters before execution. While monitoring system calls provides a comprehensive view of the guest behavior, one of the limitations of existing approaches is that they tend to monitor all possible system call functions which are triggered. Given that the system call monitoring is done from the guest kernel space, that tends to incur significant performance overhead due to the amount of context switching involved in intercepting them.

Therefore, we selectively monitor three main system call functions which are namely sys_open, sys_execve, and sys_connect. We select these three system calls based on the observation that presence of potential malware and rootkit attacks can be identified by monitoring the file access, program executions as well as network port accesses within the guest.

We monitor the sys_open system call function in order to identify file accesses within the guest. Specifically we monitor the file path contained in its system call parameter to determine any system file access.

By the same token, we monitor the file path to the binary program which is executed by intercepting the sys_execve system call function and extracting its system call parameter. This enables us to identify the program executed as well as other additional information such as execution privileges. In order to monitor suspicious network connections, we monitor the sys_connect system call function. This enables us to extract the port number which is opened from the sockaddr structure.

Given the information from each of these system calls, we conclude that we can monitor the guest state effectively without the need for additional system call monitoring.

C. System call hashing

While userspace malware can be identified by system call monitoring, this is limited in protecting the guest against kernel-level rootkits. This is due to their ability to manipulate the system call table and change its entries to point to their crafted system call functions instead of the legitimate functions while hiding within the system.

System call hashing operates by first obtaining from the system call table a copy of the monitored system call body upon installation. However, rather than copying the entire length of the system call body, only 8 bytes from an initial starting offset of the 9th byte are copied. This stems from the observation that, while the first 8 bytes tend to be the same across system call bodies, the bytes thereafter tend to vary significantly. This enables us to identify any discrepancies between the malicious system call and the original system call bodies. Figure 2 shows how system call hashing works for the sys_open system call entry.

During the lifetime of the guest VM, the in-VM monitor scans the system calls to determine if suspicious file and port accesses are made. Any anormalies within the guest are sent to the control monitor as a binary vector for malware classification using Support Vector Machines (SVM).

D. SVM-based attack classification

While monitoring file and executable access can be used to identify potential signs of malware activity, it alone cannot be relied upon for threat detection. We therefore employ Support Vector Machines (SVM) to automate rootkit and malware classification based on guest behavior (r4). Implemented as part of the control monitor is the SVM classifier which is designed to automate the malware classification process. This is based on the observation that the behaviour of the monitored system calls can be classified as either malicious (i.e., attack) or benign (i.e., non-attack). In addition given that we monitor the states of only three system calls, the training set describing all their possible states (2_3 = 8 states) is small enough to obtain fast convergence. This makes SVM a suitable candidate to be used for our implemented system.

In order to train our SVM classifier, we first determine all possible states of the monitored system calls and represent them in the form of binary vectors. For each of the monitored system call state we set each to 1 if we identify potential malware attack presence based on their system call parameters and 0 otherwise. We then represent their combined states as a 3-tuple binary vector of the form [sys_open, sys_execve, sys_connect] as shown in Table I.

TABLE I. CLASSIFICATION SCHEME FOR SVM

sys_open	sys_execve	sys_connect	Classification
0	0	0	NO_ATTACK
0	0	1	ATTACK
0	1	0	NO_ATTACK
0	1	1	ATTACK
1	0	0	NO_ATTACK
1	0	1	ATTACK
1	1	0	ATTACK
1	1	1	ATTACK

III. IMPLEMENTATION

We implement our malware and rootkit detection system on our HPC computing nodes. We use these nodes to create a virtualization testbed by running Kernel-based Virtual Machine (KVM) on them. Our implemented malware and rootkit detection features two main components, namely the protected in-VM monitor and the control monitor.

A. Protected in-VM monitor

We implement the protected in-VM monitor as a loadable kernel module (LKM) using KProbes together with GNU C. Figure 3 shows a conceptual diagram of the in-VM monitor.

Upon the creation of the new guest VM, we install the protected in-VM monitor into the guest kernel space. It first obtains from the underlying system call table (sys_call_table) copies of the targeted system call function bodies, before using KProbes to register pre-handler callback functions.

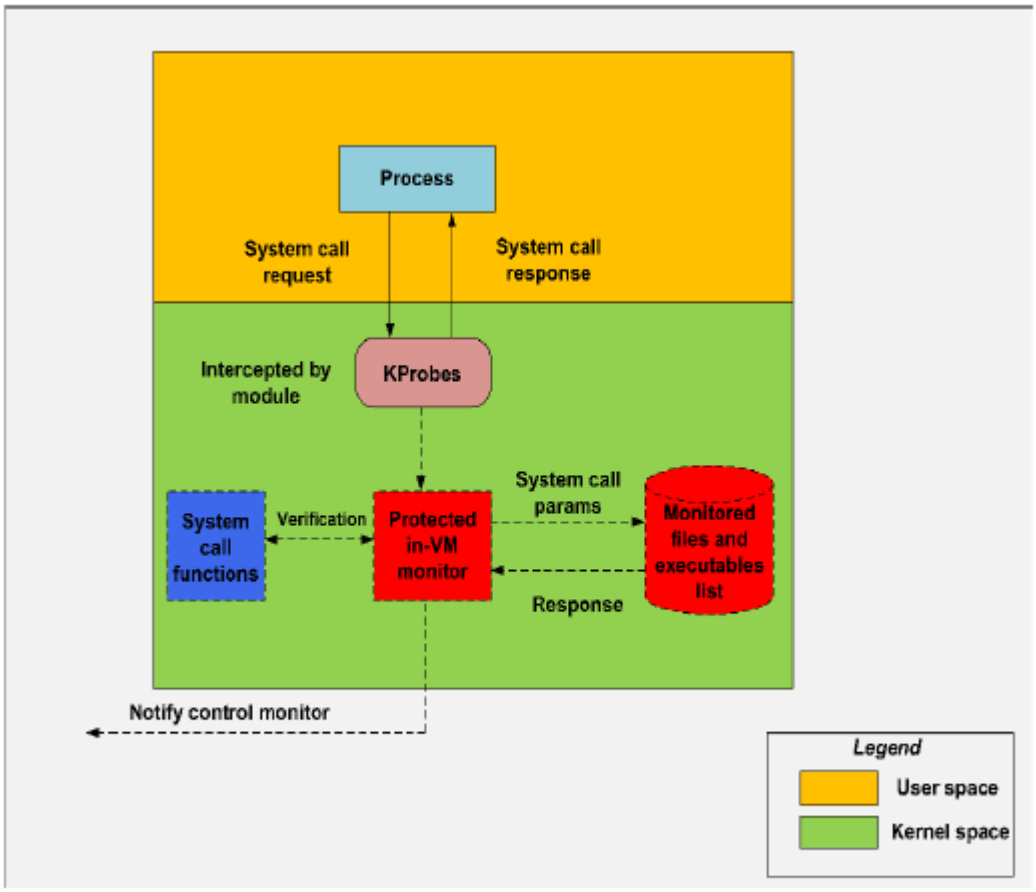

Fig. 3. In-VM monitor

To protect the in-VM monitor against removal by a successful rootkit attack, we incorporate a password mechanism into it. The password is known only to the system administrator and is not allowed access even by the root user. The removal of the in-VM monitor can only be achieved by entering the password, which can be done through the /proc interface.

During the lifetime of the in-VM monitor, the monitor examines the system call parameters and compares them against the monitored file and executable paths which are stored in the system call parameter list. Whenever the monitor identifies the access and execution of the monitored files, it first determines if any of the monitored system call handlers registers a suspicious behaviour before generating a binary vector to notify the control monitor.

B. Control monitor

Running within the HPC host, the control monitor is responsible for monitoring changes within the guest VMs. We implement the control monitor using GNU C together with SVMLight [1] and LibVMI [2]. It consists of three modules, namely the socket listener, SVM attack classification module, and the VMI-based process identification module, as illustrated in Figure 4.

When a guest VM is created, the control monitor takes a snapshot of its memory process table $ProcessTable_{t-1}$ by remotely running a userspace command against the guest (such as ps) and storing the output as a text file. During its lifetime, the socket listener listens for any notifications from the protected in-VM monitor. *SVM attack classification:* When the socket listener receives the binary vector from the guest, it forwards it to the SVM attack classification module. It first extracts the binary values from the received attack vector, before using the offlinetrained SVM classifier to determine attack presence. Once identified, the VMI-based process identification module is then notified to identify the offending user process.

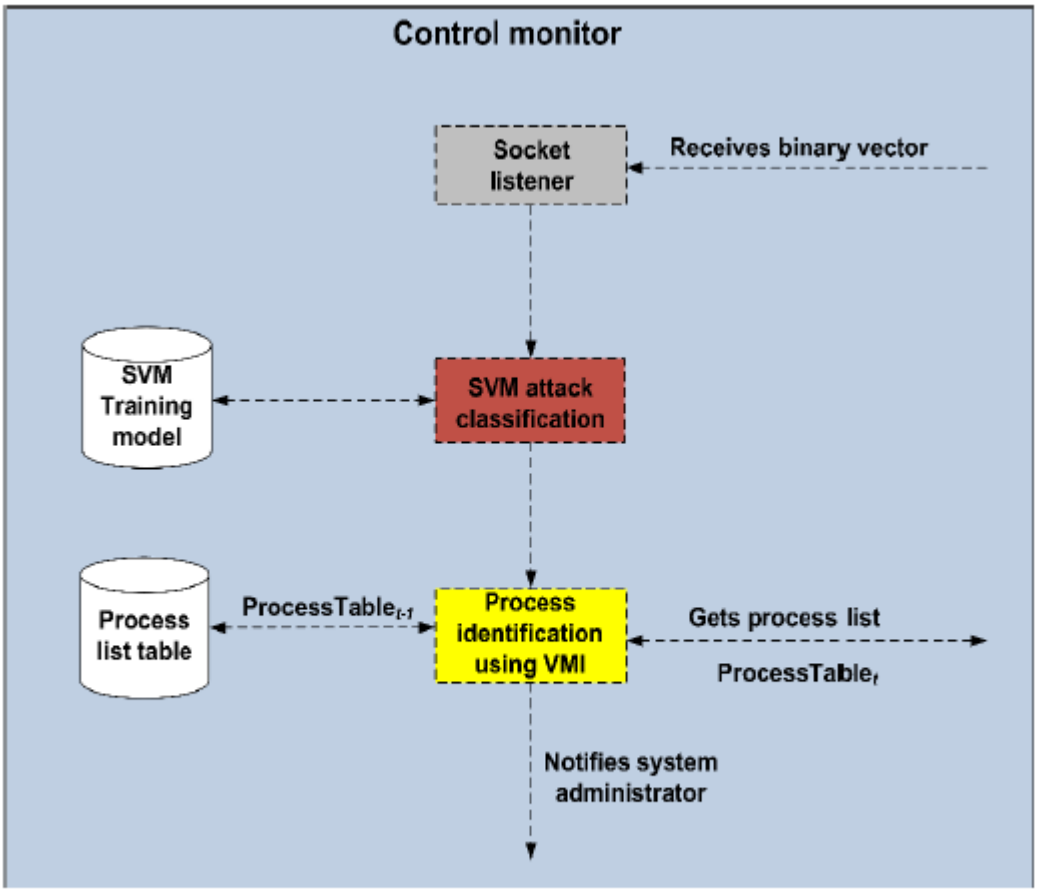

Fig. 4. Control monitor

Process identification using VMI: The process identification module obtains the latest memory process table ProcessTable$_t$ from the guest using the LibVMI API. It then performs a diff operation between the ProcessTable$_{t-t}$ and ProcessTable$_t$ files to identify the offending process before notifying the system administrator.

IV. TEST SCENARIOS

We evaluate our implemented malware and rootkit detection system by creating two guest VMs running CentOS 6.5 as well as another two guests running Ubuntu 12.04 (64-bit) on one of the aforementioned HPC server nodes. We evaluate the performance of our system by measuring its ability to detect malware and rootkit attacks, as well as measuring the performance of both the protected in-VM monitor and the VMI-based control monitor.

A. Detecting userspace malware and kernel-level rootkits

We determine the ability of our solution to detect both userspace malware as well as kernel-level rootkits by running them on the guests. We use a number of publicly available malware as well kernel-level rootkits from PacketStorm [3] in addition to one we devise on our own. We run both malware and kernel-level rootkits on the aforementioned guests and determine the ability of our system to detect them.

Detecting userspace malware: We determine the ability of our system to detect user-level malware by running two pieces of publicly available malware, one of which is a reverse shellcode that gives a remote server shell access to a guest and the other a botnet code that creates a C & C (Command & Control) botnet, on the guests. We also devise a malware which performs privilege escalation and modifies the /etc/passwd file using ROP (Return-oriented Programming). We implement this as a local application and run it locally on the guest.

We run the userspace malware 5 times and the system calls which help in detecting them are shown in Table II.

TABLE II. MALWARE TESTED

Malware	Detecting system call
Reverse shellcode	`sys_connect, sys_execve`
C & C botnet	`sys_connect`
ROP-based shellcode	`sys_execve, sys_open`

We are able to detect both the reverse shellcode as well as the C & C botnet attacks by monitoring their respective port numbers through monitoring the sys_connect system call. In the case of the former, however, we are able to detect it through the sys_execve system call, as it executes the Linux shell (/bin/sh) once the remote port connection is established. As for our own ROP-based attack, we are able to detect it based on the file path we obtain from monitoring sys_open as well as sys_execve system calls.

Detecting kernel-level rootkits: We determine the ability of our system to detect kernel-level rootkits by running three publicly available rookits, namely *Azazel*, *Enyelkm*, and *XingYiQuan* on the guests. These rootkits take control of the guest by modifying the underlying system call table (sys_call_table) entries and establishing external network connections using the *Netfilter* kernel module.

By monitoring changes to the monitored system call table entries, we are able to detect them since they modify the system call table enteries to hide their activities within the guest. We are able to detect both by monitoring the parameters of the system calls as shown in Table III.

TABLE III. ROOTKITS TESTED

Rootkits executed	Detecting system call
Azazel	`sys_connect, sys_open`
Enyelkm	`sys_connect, sys_open`
XingYiQuan	`sys_connect, sys_execve`

B. *Impact of the protected in-VM monitor on guest VM performance*

CPU time for system call monitoring: In order to determine the impact of the protected in-VM monitor on the guest VM performance, we evaluate the amount of time taken by the CPU to open one of the monitored files (/etc/passwd). To that end, we implement a userspace C program which opens the file under three cases, namely without the monitor installed; with monitor installed and hashing enabled; and with monitor installed and hashing disabled. We run 3 sets of 100 times for each case using the program and the CPU times averaged over the 300 repeated runs are tabulated in Table IV.

TABLE IV. EXECUTION TIMES FOR `SYS_OPEN` AVERAGED OVER 300 REPEATED RUNS

Case tested	Execution time (ms)
Without monitor	0.21
With monitor but without hashing	1.78
With monitor and with hashing	1.82

At the first glance, there seems to be a significant increase in CPU time when the monitor is installed in the guest VM compared to that without it. This is due to the fact that the monitor has to intercept every sys_open system call request and determine if the file opened belongs to the monitored files list.

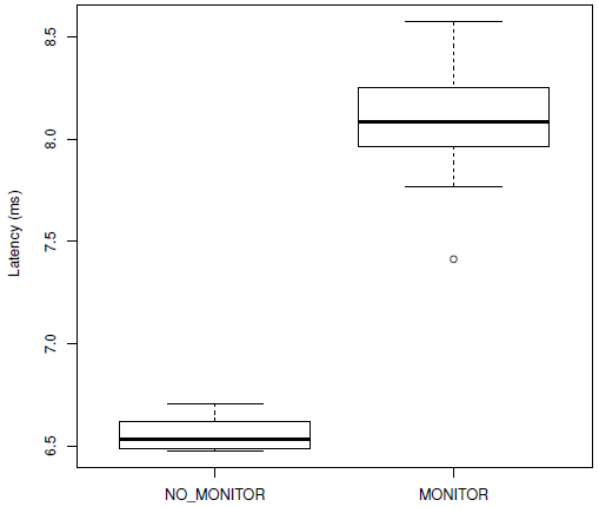

Fig. 5. File I/O latency with 10 *DBench* processes

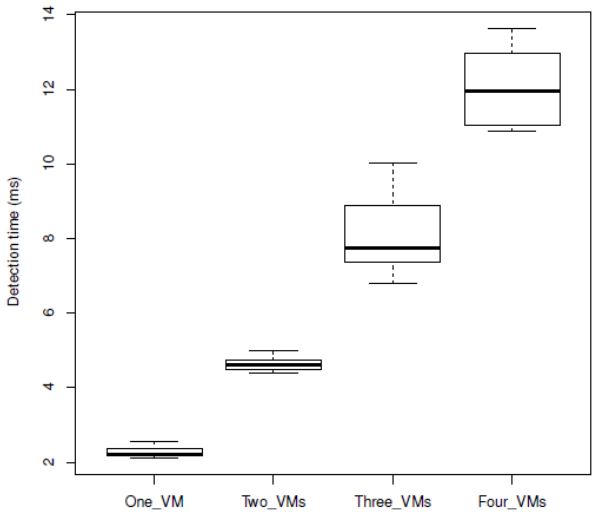

Fig. 6. Attack detection time of control monitor

In addition, there is an overall increase in CPU time when hashing is enabled compared to it disabled. This is also expected, since it tends to take time for the CPU to calculate the hash value both on initial installation as well as for every sys_open invocation. Since system files such as /etc/passwd are not frequently accessed in a typical guest VM, it will not have a negative overall impact on the guest performance.

Guest VM file access latency: We also measure the amount of file access overhead (latency) both in the presence and absence of the protected in-VM monitor using *DBench*, which uses different file I/O system calls including the monitored sys_open to measure the filesystem performance. We use *DBench* to set up 10 processes to emulate simultaneous access to the guest file system, and then run the access by the emulated processes 10 times.

The file I/O latency is plotted using standard box plots in Figure 5. The lines at the bottom and top of a box represent the minimum and maximum file I/O latencies, respectively while the line in the box represents the median latency. The range from the minimum line to the lower edge of the box is the 1st quartile, while the range from the lower edge of the box to the median represents the 2nd quartile. The range from the median line to the upper edge of the box is the 3rd quartile, with the range from the upper edge of the box to the maximum line representing the 4th quartile.

The median file I/O latency in the presence as well as the absence of the protected in-VM monitor are 8.1 ms and 6.6 ms respectively, indicating an increase of 22.7%. This is because the in-VM monitor intercepts every sys_open system call request and checks its parameters before execution. In addition there is an outlier average latency of approximately 7.4 ms with the monitor running, due to the randomized manner in which the *DBench* processes trigger different file I/O system calls to simulate random file access. Since the maximum file I/O latency in the presence of the protected in-VM monitor is approximately only 8.6 ms on average, the presence of our protected in-VM monitor will not have negative effect on the overall guest system performance.

C. Performance of the VMI-based control monitor

In order to measure the performance of the control monitor, we measure the amount of time taken to detect an attack within the guest VM against the number of guests running on the host. This was done by running the *XingYiQuan* rootkit on the guest and measuring the detection time with increasing guests.

We run the protected in-VM monitor with hashing enabled, and we notify the control monitor for any abnormal guest behaviour. We run the test 30 times on the aforementioned guests and plot the detection times in Figure 6.

There is a linear increase in detection time with increasing number of guests. When we run our experiment on two guests running the same guest operating system (32-bit CentOS), the detection time is approximately 4.2 ms with the second and third quartiles relatively close to the median. This is because the LibVMI API can use the same data structures to access their process tables without having to switch between different architectural spaces.

With the introduction of a guest running Ubuntu 64-bit, however, we find that the maximum execution time increases from approximately 5.1 ms to 10 ms with approximately 75% of the execution times falling within the third quartile. This is because the API needs to switch between 32- and 64-bit kernel spaces to read the respective guest kernel symbol tables to obtain the process tables. When we introduce a fourth guest running the same 64-bit operating system, the distribution of the average execution times in the second and third quartile relative to the median becomes relatively even due to it having to navigate an equal number of different guest operating system architectures.

V. RELATED WORK

A. In-box monitoring

Secure In-VM Monitoring (SIM) is implemented which places the monitoring tool within the guest VM by allocating a separate virtual memory segment within it [4]. The monitoring tool intercepts all system call requests made via the invocation hooks and verifies them before passing them onto the hypervisor for execution. While this protects the monitoring agent against malware compromise, it requires modifications to the underlying hypervisor making it difficult to be deployed in a multi-host environment. Our approach overcomes this by leveraging the existing LKM (Linux Kernel Module) framework to protect the monitoring agent while using external monitoring for attack verification, eliminating the need for hypervisor modifications prior to its deployment.

The use of an in-box monitoring module for threat detection is used by Schmidt *et.al* in their implementation of a rootkit detection system for guest VMs [5]. Running as a kernel module within the guest VM, it intercepts every system call and passes its parameters to a separate analyser module using User Datagram Protocol (UDP). The analyser then uses the known attack signature database from ClamAV antivirus for threat detection. While the use of a signature database in malware detection helps to reduce the amount of false positives in determining malware presence, the need for it to be updated with the most recent malware signatures makes this vulnerable against zero-day attacks. Our implemented solution eliminates the need to use a signature database by monitoring the guest behavior through system call monitoring and using an offline SVM classifier for malware classification.

B. Virtual machine introspection (VMI)

First used in the implementation of Livewire [6], VMI monitors the presence of attacks within guest through external monitoring. While this protects the monitoring tool against compromise in the event of a malware attack, it requires constant polling of guest VM state at regular intervals. This incurs a performance overhead which increases in direct proportion to the number of guests running on the host. Through protected in-VM monitoring, our approach is able to close the semantic gap while ensuring the monitor is not compromised by malware. In addition, the VMI-based process identification is triggered only when the in-VM monitor identifies a suspicious guest behavior significantly reducing the overhead associated with polling guest VM internal state.

An alternative rootkit detection approach (VMWatcher) was implemented using VMI [7]. A technique called guest view casting was used, which involves reconstructing the storage and memory contents of a guest VM based on its external observations. While this approach is able to bridge the semantic gap, the potential reconstruction overhead in a multi-guest environment will hinder the overall performance of the virtualization environment. Our approach overcomes this limitation by eliminating the need to externally reconstruct the guest VM state and traverse the guest VM memory state, by placing hooks within it.

VI. Conclusion & Future work

In this paper, we present a rootkit and malware detection system to protect the underlying virtualization infrastructure against cyberattacks in cloud computing environments. Designed to overcome the limitations associated with existing virtualization security approaches, it uses system call body hashing as well as the use of SVM together with VMI. The use of a protected in-VM monitor ensures that the internal guest VM state can be accurately obtained without it being compromised, while the use of an offline SVM classifier in the remote control monitor allows for quick attack classification.

One future work can look at additional system call monitoring in the protected in-VM monitor to increase the accuracy of attack detection within the guest. Another may investigate the use of data mining tools within the control monitor to improve the detection performance.

REFERENCES

[1] T. Joachims, “Making large-scale svm learning practical,” Universit¨at Dortmund, LS VIII-Report, LS8-Report 24, 1998.
[2] B. Payne, S. Maresca, T. Lengyel K, and A. Saba, “Libvmi,” http://www.libvmi.com, accessed: 09-07-2014.
[3] PacketStorm, “Packetstorm,” http://tinyurl.com/qhygrsu, accessed: 29-10-2014.
[4] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, “Secure invm monitoring using hardware virtualization,” in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 477–487. [Online]. Available: http://dx.doi.org/10.1145/1653662.1653720
[5] M. Schmidt, L. Baumgartner, P. Graubner, D. Bock, and B. Freisleben, “Malware detection and kernel rootkit prevention in cloud computing environments,” in *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*. IEEE, 2011, pp. 603–610. [Online]. Available: http://dx.doi.org/10.1109/PDP.2011.45
[6] T. Garfinkel, M. Rosenblum *et al.*, “A virtual machine introspection based architecture for intrusion detection.” in *NDSS*, vol. 3, 2003, pp. 191–206.
[7] X. Jiang, X. Wang, and D. Xu, “Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction,” in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.