UNOBSERVABLE INTRUSION DETECTION BASED ON CALL TRACES IN PARAVIRTUALIZED SYSTEMS*

Carlo Maiero, Marino Miculan
Department of Mathematics and Computer Science, University of Udine, Italy
carlo.maiero@uniud.it, marino.miculan@uniud.it

Keywords: Intrusion detection systems, paravirtualization, system call trace analysis.

Abstract: We present a non-invasive system for intrusion and anomaly detection, based on system call tracing in paravirtualized machines over Xen. System calls from guest user programs and operating systems are intercepted stealthy within Xen hypervisor, and passed to a detection system running in Dom0 via a suitable communication channel. Guest applications and machines are left unchanged, and an intruder on the virtual machine cannot tell whether the system is under inspection or not. As for the detection algorithm, we present and study a variant of Stide, which we verify experimentally to have a good performance on intrusion detection with an acceptable overhead—in fact, online real-time intrusion detection feasible. However, since the interception mechanism is kept separated from the detection system, the latter can be replaced according to further needs.

1 INTRODUCTION

Nowadays, the most common form of intrusion and anomaly detection is *host-based*, which means that the detection system runs on the same machine under inspection. A serious drawback of host-based detection is that it is *invasive*, since the computation environment is modified by the presence of the detection system itself (e.g. as a program in the file system, or as a process running in memory, or as a kernel module, or some open ports...). As a consequence, an intruder can notice the detection system and modify her actions accordingly, e.g. by changing the attack methodology in order to avoid detection. This can be particularly serious in "honeypots", which are systems built on the purpose of gathering information about attacker's motives and tactics. Moreover, once the detection system has been discovered, it can be attacked, e.g. by disabling it or altering its data.

In order to avoid these problems, an intruder on the machine under inspection must not be able to notice the existence of the detection system. Ideally, from the intruder's point of view, a monitored machine should be observationally equivalent to an unmonitored machine. This means that a monitored machine and an unmonitored one cannot differ neither in the file system, nor in the user-space and kernel-space memory, nor in devices, etc.. In particular, the detection system must execute *outside* the machine under inspection, and still be able to observe the behavior of processes *in* the machine (which is unmodified), and possibly to take appropriate countermeasures.

In this paper we show how to solve this conundrum by adopting *(para)virtualized machines* in conjunction with *syscall-based anomaly detection*. In paravirtualized systems (like Xen (Barham et al., 2003)), a suitable layer, called *virtual machine monitor* or *hypervisor*, allows to run an operating systems on a virtual version of the underlying hardware, while maintaining the same architecture of the real one. Privileged instructions cannot be executed by the guest operating system, since this runs without the necessary privileges. Therefore, when a process in a guest machine executes a system call to its operating system, the trap triggers the hypervisor which takes care of the usual context-switch between user and kernel space. In that instant, the hypervisor can collect data about the system call, as done in (Laureano et al., 2007), and forward these data to a detection system for analysis. In case the trace appears to be not correct, the detection system can apply suitable countermeasures, e.g. it can abort the offending system call, or "suspend" the machine (possibly) under attack.

*Work funded by MIUR project 20088HXMYN *SisteR*.

This solution presents several interesting aspects. First, system call interception is completely unobservable to the guest operating system and processes. This means that an intruder cannot tell whether the system is being monitored. Of course, the system appears to be paravirtualized (e.g., due to the presence of some specific daemons), but this is not uncommon nowadays, especially for hosted services and cloud computing. Secondly, the interception mechanism is kept separated from the detection system, which can be easily replaced and upgraded, accordingly to further needs. Moreover, our approach is "black-box", i.e., we do not assume any *a priori* knowledge about the internals of the virtual machine nor the applications therein running. Finally, the overhead introduced by the interception mechanism is quite low, so that on-line, real-time intrusion detection is feasible.

The paper is organized as follows. In Section 2, we describe the architecture of the unobservable intrusion detection system: the mechanism added to the Xen hypervisor in order to gather information about system calls, the channels for communicating with the detection system, etc. In Section 3 we consider some possible algorithms for the detection system. Basically, the idea is to compare the system calls traces with respect to a set of "well-behaved" traces, as done e.g. in (Hofmeyr et al., 1998). As we will show in Section 4, the resulting system has good performances, concerning both the attacks it can recognize (with very low false positive rates), and the computational overhead. Final remarks, related work and directions for further work are discussed in Section 5.

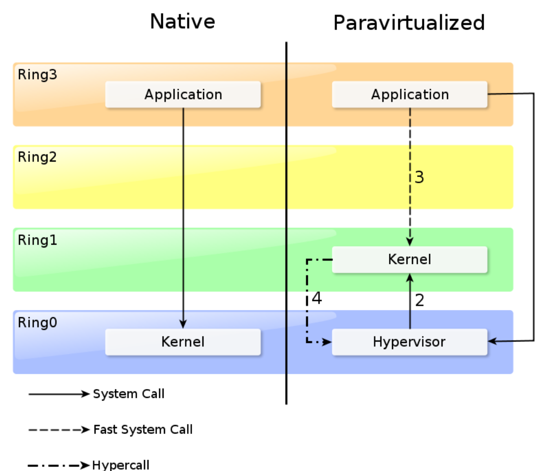

Figure 1: System call handling in a native system (left) and paravirtualized Xen system (right)

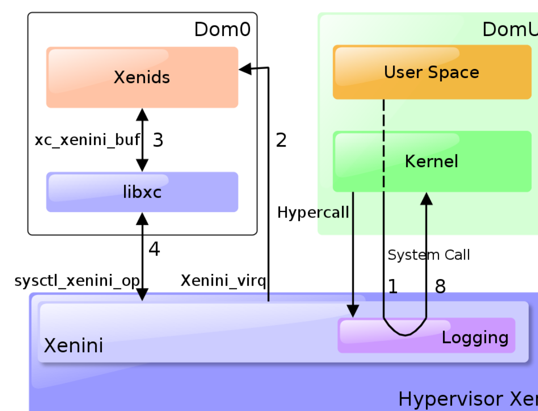

Figure 2: Architecture of the Xenini-XenIds system.

2 XENINI: UNOBSERVABLE SYSCALL INTERCEPTION

In this section we present *Xenini*, a patch to Xen hypervisor implementing the stealthy gathering of information of the behavior of virtual machines. This patch is designed to have the following features:

Independence and flexibility Xenini must be able to operate in complete autonomy, without any cooperation with the machines being monitored.

Security the patch must affect only the hypervisor and must not use any guest kernel data structure.

Independence from memory Differently from other solutions using techniques of guest machine memory and virtual disk introspection, Xenini observes only the system calls (as in (Laureano et al., 2007)).

Isolation the attacker is not aware of being monitored, because the patch is at hypervisor level without *any* change to the virtual machine.

Simplicity the system requires only an initial patch to the hypervisor, and does not put any constraint on virtual machines.

Architecture In Xen, a guest application executing a system call raises a trap which yields the hypervisor to be executed (Figure 1). At this point, before passing the control to the guest kernel, we can intercept and log the system call. More precisely, we can observe the syscall number and the caller process ID. This information is passed for evaluation to the detection system, called *XenIds*, which is a process running in Dom0. XenIds will scrutiny the syscall trace, and possibly take suitable countermeasures.

The overall architecture is shown in Figure 2. Before describing the execution flow, let us describe the main components of the system.

System calls interception First, in order to handle all system calls by the hypervisor, we disable "fast" system calls in Xen (without modifying the guest operating system). Then, for each paravirtualized domain DomU, Xen installs a "local" Interrupt Descriptor Table (IDT) which it is looked up by hypervisor when a trap is invoked by domU. The interrupt 0x80 is not part of the traps handled in local IDTs, and it is managed by hypervisor. The traps that must be managed at level 0 are handled by the function `do_guest_trap()` (file `xen/arch/x86/traps.c`). In this function, the case of interrupt 0x80 is modified into a call to Xenini. The system call number is in the `%eax` register, which is stored in a buffer, together with the process ID. Here, we can also terminate the system call by changing the value of the virtual processor register `%eax`.

Communication between hypervisor and Dom0 The communication mechanism between Xenini and intrusion detection systems residing in Dom0, can be divided into two parts: the *event channel* and the library `libxc` for message handling.

Event channels are a simple asynchronous notification mechanism provided by Xen. Each domain has a set of ports that can be associated with an event source, which can be an interrupt, a physical IRQ, a virtual interrupt IRQ or port to another domain. The events are then handled asynchronously, so the delivery of the message does not necessarily wait for a response from Xen. In this case the two domains connected are the hypervisor and Dom0. This event channel notifies the IDS that some new interception data is present in the buffer of Xenini. The communication takes place via an *ad hoc* virtual interrupt `VIRQ_XENINI` added in the file `xen/include/xen/xen.h`.

Libxc provides an API for Xen control, e.g., for communicating with Xen through Dom0 and for controlling (e.g., creating, destroying, suspending) other virtual machines. We have extended the functionalities of this library by adding an internal protocol for communication between the IDS and Xenini.

Execution flow is as follows (Figure 2):

1. Xenini intercepts the system call or the hypercall
2. Xenini alerts XenIds via a VIRQ
3. XenIds makes a request get info to libxc
4. Libxc requires data to Xenini
5. Xenini transmits the data to libxc
6. Libxc returns data to IDS
7. the IDS processes the data and gives an answer.
8. the control flow returns to the guest VM

Xenini can act synchronously (i.e., blocking) or asynchronously, depending on the system call. Synchronous operation is intended for system calls (or hypercalls) which are considered "critical"; in this case, Xenini waits for the answer from XenIds before executing the call. Asynchronous operation is for system calls which have to be logged only; in this case, Xenini communicates the data to XenIds and proceeds immediately. We will see in Section 4 that asynchronous operation yields much less overhead, still allowing for a high level of recognized attacks.

3 CALL-TRACE BASED INTRUSION DETECTION

In this Section we discuss the algorithm implemented in the intrusion detection system XenIds. As for any IDS, our aim is to recognize when the behavior of a system is "non-standard", which may indicate that the system is being attacked.

Let us recall that Xenini passes to the IDS only the system call numbers (i.e., the names), together with the ID of the processes. The IDS does not have access to guest machines memory, nor it can analyze the programs which are running on these machines. In fact, we cannot even know which program a given process is running. Therefore, we have to follow a *black-box* approach: we can only analyze the sequences of system calls that are invoked by each process, without the possibility of inspecting the process' memory.

State of art Considerable work has been carried out about the analysis of system call traces. In general, the principal black-box approaches for the analysis of system calls traces use statistical approaches or machine learning techniques for extracting the basic "correct" structures of call traces; then, the trace of a single program at once is analyzed, usually offline. Due to lack of space, we can give only a very short description of the principal approaches.

Frequency-based methods observe the occurrences of certain patterns in a sequence of system calls. (Helman and Bangoo, 1997) propose to classify each sequence on the frequency that it appears in traces of intrusions: sequences that occur more frequently in intrusions and less frequently in normal behaviors are considered suspect. Unfortunately, the frequency of each sequence in the intrusion traces is not known a priori.

Data mining can be used to recognize certain types of patterns present in a large amount of data. In the case of intrusion detection, this leads to a definition of "normal" (self) lower than that obtained by

simply listing all sequences that occur in normal behavior. We mention Cohen's RIPPER (Cohen, 1995), which has been adapted for syscall interception in (Lee et al., 1997; Lee and Stolfo, 1998).

Finite state machines can be used for recognizing a language of "normal traces". An intrusion yields a sequence which is not recognized. There are many techniques for building DFA and NDFA to solve this problem; we mention the Hidden Markov Model (HMM). (Zhang et al., 2007) suggests an approach that deals with the grammar of system calls.

Enumerate sequences Stide. (Forrest et al., 1996) introduced an intrusion detection method based on observation of system calls used by privileged processes. The approach is inspired by the natural immune system, which has several aspects in common with security services (Forrest et al., 1997).

The Stide method of anomaly/intrusion detection defined in (Hofmeyr et al., 1998) creates a profile defined as "normal" and the substantial deviations from this profile are defined as "abnormal". Stide works in two steps. First, a database of "normal behaviors" is created by analysing system call traces generated by processes running in a safe environment. Then, the real system, possibly subject to attacks, is monitored looking for abnormal behaviors. This method analyzes only short sequences of system calls, in chronological order. Thus, the only information necessary to the algorithm are the names of system calls performed by each process; system call parameters are ignored.

The algorithm used to create the database of normal behavior is very simple: we scan the traces of system calls generated by a particular process and put in the database all unique sequences of length k, for k a fixed parameter. For instance, let us suppose we observe the following trace:

open, write, mmap, open, write, mmap

If $k = 3$ we get the following unique sequences:

(open, write, mmap)
(write, mmap, open)
(mmap, open, write)

Once the database of normal behaviors is created, the same method is used to check the traces under inspection. Given a trace, we look-up in the database all unique sequences of length k in it; the sequences which do not appear in the database are considered mismatches. The "strength" of anomaly "signal" can be measured as the number of events, i.e. mismatches, in a trace. The anomalies are counted on the last n system calls. The algorithm raises an anomaly when this number is over a given threshold, which is usually specified as a percentage of n.

Comparison According to (Warrender et al., 1999), there is no "best" algorithm in terms of false positives and false negatives; however, it is also possible to say that RIPPER and Stide perform better on a threshold value of 6.

For our application the best algorithm seems to Stide, for two reasons. First, we can train its database using directly the data provided by Xenini, during normal activity of the system. Secondly, we can implement the detection algorithm quite efficiently, allowing for a real-time, online intrusion detection. In fact, sequences are stored in the database as trees, whose nodes are labelled with system calls. Hence, there are as many trees as system calls; moreover, the upper limit for the database size is $O(Nk)$, where k is the sequence length, and N is the number of unique sequences. In practice the requirements are lower because the sequences are stored in trees.

However, the modular structure of XenIds/Xenini allows to replace easily the detection algorithm.

Implementation Differently from other work in literature, our aim is to identify intrusions on online systems, in (almost) real time. When an anomaly is detected, XenIds can perform several actions, from very simple ones (such as reporting the intrusion) to quite intrusive, such as aborting the compromising system call or suspending the corresponding virtual machine to preserve its data.

Operating principles Stide is intended to analyze each process separately, which means that one should build a database for each program, and compare the traces from a process with respect to the database of the corresponding program. However, this is problematic in our setting, because we do not know which program is running a given process: Xenini can intercept the process ID, but not the executable file name.

Therefore, we decide to keep a single database, i.e. a single tree, for all sequences, coming from any program. This is suggested by the experimental observation that different programs have only few sequences in common. In other words, a program can be identified by looking the set of its "good" sequences. Clearly, keeping all sequences from all programs in the same database could increase the number of false negatives (because a sequence which is abnormal for a program could be normal for another), but as we will see in the next Section this is not the case. In fact, this problem can be solved by suitably choosing the length k. Let us denote by p_n the probability that at some point in a sequence, an abnormal syscall is found in the database; then, the probability of recognizing as normal a k-length abnormal sequence, is p_n^k. The possibility of a false negative can be reduced exponentially by increasing the length of sequences.

XenIds works as follows. When it receives a triple $(VM_ID, SysCall_number, process_ID)$ from Xenini, it appends the *SysCall_number* to a buffer of length k associated to $(VM_ID, process_ID)$ (the buffer is created if not already existing). Then the buffer is searched in the database; if it is not present, the number of anomalies for that $(VM_ID, process_ID)$ is incremented. If the anomalies ratio exceeds the threshold, XenIds will perform the corresponding action.

However, some system calls have a special meaning, which must be taken into account in the IDS. In particular a process can create other processes, or execute a different program without changing its ID. In these cases, a specific action is required on buffers. For instance, on Unix-like machines processes are created using a *fork* or *vfork* system call. While *fork* creates another instance of the same program which will be tracked separately, the *vfork* replace the current program with a new one without changing id. So in the case of *vfork* the system calls recorded in the buffer have no relevance, and hence the buffer is cleared. The same applies to the *exec* system call, which replaces the calling program with a new one program code; also in this case we clear the buffer.

Modes of operation As mentioned before, the intrusion detection system can operate in two modes: synchronous or asynchronous. In the synchronous modality, each system call intercepted by Xenini is delayed until the IDS has verified that the behavior is normal. In the asynchronous modality, Xenini sends to the IDS the data about the intercepted system call, and proceeds immediately with the execution of the system call. Thus, the decision (and possibly the action) of the IDS is decoupled from the interception, as it may be delayed with respect to the compromising system call. As we will see in the next section, these two modes present quite different performances.

4 DISCUSSION

Detection Testing In order to evaluate the detection ability of XenIDS, we can refer to two datasets of syscall traces available from the Artificial Intelligence Laboratory at MIT, and the Department of Computer Science, University of New Mexico (UNM) (Hofmeyr et al., 1998; Hofmeyr et al., 1999).

Since the database of XenIds collects sequences from different programs, we have created a system call trace by combining a trace of 300,000 system calls obtained by logging an "idle" Linux virtual machine, with about 1900 unique sequences from MIT and UNM datasets for the program "sendmail" (which contain also sequences arising from various attacks). The resulting trace contains system calls from 18 Linux services, including sendmail and apache2.

This trace has been submitted both to the original Stide algorithm (which uses the database for sendmail only) and to XenIds (which uses instead the database containing sequences from all processes), with the following parameters:

- sequence length (stide) $k = 6$;
- locality frame count equal to 100;
- threshold = 6%.

The following table summarize the results of this test, for several attacks to sendmail.

Attack	mismatch	Stide	XenIds
decode	8%	yes	yes
syslog	30%	yes	yes
fw-loop	16%	yes	yes
sscp	26%	yes	yes
sm5x	35%	yes	yes

Therefore, we can conclude that on these sample trace sets, our IDS has the same detection skill of the original Stide algorithm: no false negative has been found. This does not guarantee that using a single sequence database for all programs is *always* as precise as using a separate database for each program: a sequence which could be suspicious for a process, could be valid for another. It is worthwhile to notice that using a single sequence database does not affect the false positive rate. Indeed, if a valid call sequence is not recognized by our IDS, it means that it does not appear in the trace set of any program; but then, also the original Stide algorithm would not recognized it.

As another test, we have installed a simple server running on a Linux virtual machine, which receives strings over the network and saves them on a file. The IDS has been trained with the trace coming from the following cases:

- strings of 10 characters, and of 200 characters;
- termination of the server using SIGINT.

Then, we have changed the behavior of the process in two ways. First, we have not changed the program, but only the data it has to deal with, and how it is terminated:

- strings of 25 characters, and of 100 characters;
- termination using SIGKILL.

Secondly, we have modified the program, in order to simulate an intruder that introduces malicious changes. The modifications we have introduced are:

- write a copy of the data in a different location;
- opens a backdoor (a system shell);
- sends a copy of the data to a remote address.

The following table summarize the answer of XenIds; for this experiment the length k is 6, the size of LFC is 100 and the threshold value is 15%.

Change	Mismatch	Anomaly?
strings of 25 chars	$< 15\%$	No
strings of 100 chars	$< 15\%$	No
closing using kill	$< 15\%$	No
local copy string	20%	Yes
open a system shell	50%	Yes
remote copy string	30%	Yes

In all cases, the IDS detects correctly the modified sequences. We must observe that the accuracy of the system depends mainly on two factors: the threshold, and the accuracy of data collected. We noticed that tests carried out without adequate preparation led to a false positive rate of 50% for some processes. The same applies to a very low threshold, i.e. of less than 6%, the false positive rate is always high, and stands at 15-20% for new jobs. On the other hand, with the threshold at 15% and a "well-trained" database, we found no false positives for the traces under test. However, XenIds stays "on the safe side", since has identified all the attacks carried out in programs, i.e., there have been no false negatives.

Performance In order to run a sample performance analysis, we installed a Linux virtual machine with the Apache web server v2.2, that responds to requests for a simple PHP page. To simulate the requests we have used the "Apache Benchmark" `ab` tool. The tests have been performed for four different scenarios:

1. Xen (version 4.0.1) native, without modifications (i.e., without Xenini).
2. Xen with Xenini, but interception is not activated: the hypervisor is patched with Xenini, and XenIDS is running on Dom0, but interception is disabled. This test is carried out to check the impact of additional controls.
3. Xen with interception activated, and IDS in asynchronous mode.
4. Xen with Xenini active and IDS in synchronous mode (i.e., the hypervisor waits for the answer from the IDS for each system call).

For each case, we have run three tests:

- 50,000 requests, concurrency level=2 (i.e., 2 parallel requests at a time);
- 50,000 requests, concurrency level=50;
- 100,000 requests, concurrency level=5.

The results are shown in Figure 3. These tests point

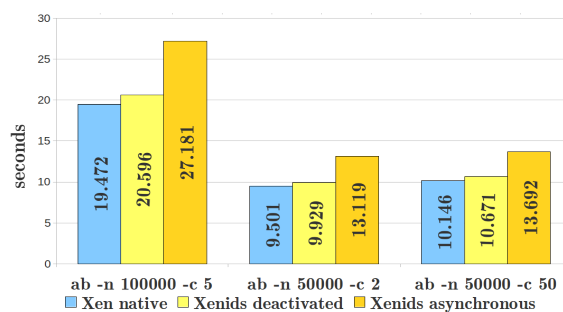

Figure 3: Performance analysis.

out that the patch, when inactive, introduces an overhead of about 5-6%, which is due to controls on interrupts 0x80 and 0x82 and inspections on some flags for intercepting calls. Instead, when the interception is activated, and XenIds operates asynchronously, the overhead is about 40%; this is due to monitoring, reporting and analysis of all system call. However, we consider this overhead still acceptable for a real-time, online monitoring. Finally, the tests with XenIds in synchronous modality have exhibited a drastic overhead; in fact, the system becomes so slow that seems to be in deadlock. Thus, we can say that synchronous modality cannot be used for on-line production systems, but only for testing purposes.

5 CONCLUSIONS

In this work we have described Xenini and XenIds, a system for detecting intrusion in paravirtualized Xen, unobservable by guest applications. This has been possible by intercepting system calls in the hypervisor, and passing this information to a IDS running in Dom0, through a Xen communication channel. The algorithm for the analysis of system call traces that we have implemented is a variant of Stide, which we have experimentally verified to have a good performance on intrusion detection, at the price of an acceptable overhead.

Related work One of the first implementations of using virtual machines for intrusion detection and prevention is Livewire (Garfinkel and Rosenblum, 2003). Their approach differs from ours in many aspects; e.g. the IDS can inspect directly the virtual machine's memory, but only the low-level internal state of the virtual machine is observed, not the activity of each guest process. Also the actions that the IDS can take are quite limited: it can either suspend or restart a virtual machine. Memory inspection of Xen virtualized machines is also used in PsycoTrace (Baiardi et al., 2009), which adopts a "white box" approach, i.e., it

analyzes the program memory in order to detect possible anomalies. Moreover PsycoTrace is not transparent, since it requires a module to be installed on the monitored machines. The same issue affects the solution proposed by (Payne et al., 2008), where an active monitoring for Xen is implemented by a specific "security driver" to be loaded in the kernel of monitored machines. However, (Bahram et al., 2010) have shown that approaches based on virtual memory introspection can be fooled by the "DKSM attack"

On the other hand, VMScope (Jiang and Wang, 2007) only logs system calls and their arguments (on QEMU), without any memory inspection. However, VMScope is used only for *a posteriori* trace analysis and attack reconstruction, since it lacks a real IDS.

Finally, the work closest to ours is (Laureano et al., 2007), which adopts a mechanism for system call logging similar to ours (but for User-Mode Linux). Their intrusion detection methodology integrates Stide with access control lists: "sensible" (i.e., possibly dangerous) system calls can be executed only by some pre-determined programs; for each process, the path of the corresponding program is obtained by direct memory introspection. Therefore, this approach is not as strictly "black-box" as ours. Moreover, the DKSM attack could be used for obfuscating the program names, thus misleading the IDS.

Future developments Although the experimental evaluation carried out in this paper is encouraging, keeping the sequences from all programs in a single database may not scale up as the number of programs and guest OSs will increase. A possible approach is to create a separate database for each program (i.e., profile), but this raises the issue of how to identify a program just by its trace of system calls. To this end, one could use an Hamming distance between a given process syscall trace, and the profiles (i.e., programs) present in the database; the profile associated with the process will be the one with the lowest Hamming distance. Alternatively, one could create statistical profiles based on the distributions of system calls of each program, and classify processes accordingly.

Finally, it is interesting to investigate whether the interception mechanism could be detected by means of some "side channel". In particular, the interception process necessarily introduces some overhead, increasing the time of system call execution. An attacker could detect this extra delay and use it to distinguish an observed system from a non-observed one. However, this seems unlikely, because virtualization itself introduces a similar delay, which can be also quite varying according to several parameters (e.g. system load, number of virtualized systems, etc.).

REFERENCES

Bahram, S., Jiang, X., Wang, Z., Grace, M., Li, J., Srinivasan, D., Rhee, J., and Xu, D. (2010). DKSM: Subverting virtual machine introspection for fun and profit. *Symp. Reliable Distributed Systems*, 82–91.

Baiardi, F., Maggiari, D., Sgandurra, D., and Tamberi, F. (2009). Transparent process monitoring in a virtual environment. *ENTCS*, 85–100.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T. L., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *Proc. SOSP*, 164–177.

Cohen, W. W. (1995). Fast effective rule induction. *In Machine Learning: the 12th International Conference*.

Forrest, S., Hofmeyr, S., and Somayaji, A. (1997). Computer immunology. *Comm. ACM, 40(10)*, 88–96.

Forrest, S., Hofmeyr, S., Somayaji, A., and Longstaff, T. (1996). A sense of self for UNIX process. *Proc. IEEE Symp. on Security and Privacy*, 120–128.

Garfinkel, T. and Rosenblum, M. (2003). A virtual machine introspection based architecture for intrusion detection. In *Proc. NDSS*. The Internet Society.

Helman, P. and Bangoo, J. (1997). A statistically based system for prioritizing information exploration under uncertainty. *IEEE Transaction on System, Man and Cybernetics, 27(4)*, 449–466.

Hofmeyr, S., Somayaji, A., and Forrest, S. (1998). Intrusion detection using sequences of system calls. *Journal of computer security 6(3)*, 151–180.

Hofmeyr, S., Somayaji, A., and Forrest, S. (1999). Computer immune systems.

Jiang, X. and Wang, X. (2007). Out-of-the-box monitoring of VM-based high-interaction honeypots. *Proc. RAID'07*, 198–218.

Laureano, M., Maziero, C., and Jamhour, E. (2007). Protecting host-based intrusion detectors through virtual machines. *Computer Networks 51*, 1275–1283.

Lee, W. and Stolfo, J. (1998). Data mining approaches for intrusion detection. *Proc. 7th USENIX Security Symp*.

Lee, W., Stolfo, J., and Chan, P. (1997). Learning patterns from UNIX process execution traces for intrusion detection. *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, 50–56.

Payne, B. D., Carbone, M., Sharif, M., and Lee, W. (2008). Lares: An architecture for secure active monitoring using virtualization. *Proc. IEEE Symp. on Security and Privacy*, 233–247.

Warrender, C., Forrest, S., and Pearlmutter, B. (1999). Detecting intrusions using system calls: Alternative data models. *Proc. IEEE Symposium on Security and Privacy*, 133–145.

Zhang, X., J. Li, Z. J., and Feng, H. (2007). Black-box extraction of funtional structures from system call traces for intrusion detection. *Advanced Intelligent Computing Theories and Application*, 135–144.