2011 30th IEEE International Symposium on Reliable Distributed Systems

Process Implanting: A New Active Introspection Framework for Virtualization

Zhongshu Gu, Zhui Deng, Dongyan Xu
Department of Computer Science
Purdue University
West Lafayette, IN, USA, 47907-2107
{gzs, deng14, dxu}@purdue.edu

Xuxian Jiang
Department of Computer Science
North Carolina State University
Raleigh, NC, USA, 27695-8206
jiang@cs.ncsu.edu

***Abstract*—Previous research on virtual machine introspection proposed “out-of-box” approach by moving out security tools from the guest operating system. However, compared to the traditional “in-the-box” approach, it remains a challenge to obtain a complete semantic view due to the semantic gap between the guest VM and the hypervisor.**

In this paper, we present Process Implanting, a new active VM introspection framework, to narrow the semantic gap by implanting a process from the host into the guest VM and executing it under the cover of an existing running process. With the protection and coordination from the hypervisor, the *implanted process* can run with a degree of stealthiness and exit gracefully without leaving negative impact on the guest operating system. We have designed and implemented a proof-of-concept prototype on KVM which leverages hardware virtualization. We also propose and demonstrate application scenarios for Process Implanting in the area of VM security.

***Keywords*-Security; Virtualization; Active VM introspection**

I. Introduction

Many security tools for malware detection are vulnerable to attacks because they are exposed to malwares. The most common technique used by a malware to hide itself is to dysfunction anti-malware engines. In order to be tamper-resistant and stealthy, current approaches leveraging virtualization technology to detect and prevent malware attacks usually try to move the monitoring tools from the untrusted guest virtual machine (VM) to underlying hypervisor or to another isolated trusted VM to prevent the tools from being tampered with. By reconstructing the semantic view of guest VM from host through the technique of virtual machine introspection, we can gain a large (yet incomplete) amount of semantic information. The semantic gap [1] between the guest operating system (OS) and host is a known barrier to services operating below the abstractions of guest OS and applications. This problem becomes more challenging with the widely deployment of hardware virtualization, whose goal is to run most of the native instructions directly on the CPU and expose as few details as possible to the virtual machine monitor to gain higher performance. Previous approaches to virtual machine introspection can passively detect [2]–[5] or actively monitor attacks with effectiveness [6]–[8]. Yet we wish to act even more actively in the counter-attacks by obstructing, analyzing and subverting the malware attacks. This requires a more complete, native semantic view of the guest VM.

In this paper, we present Process Implanting, a general-purpose active VM introspection framework. The idea is to implant a process directly from the host into the guest under the cover of an existing process inside the guest OS to narrow the semantic gap and gain in-context knowledge of the running VM. Instead of leaving the *implanted process* alone inside the guest VM, we design a series of coordination and protection mechanisms supported by the higher-privileged hypervisor to exempt the *implanted process* from malware’s tampering and leave minimum negative impact on the normal execution of guest OS and applications after it exits.

The rest of this paper is organized as follows. Section II proposes application scenarios for the Process Implanting framework. Section III provides security requirements and the detailed design of our system. Section IV describes the implementation of our prototype. Section V evaluates performance and presents some security application cases. Section VI discusses the limitations and coding requirements for *implanted process*. Section VII describes related work and we conclude in Section VIII.

II. Application Scenarios

There are several security implications of our Process Implanting technique, as illustrated in the following application scenarios:

Computer Forensics and Monitoring: Computer forensics is the practice to derive an anatomical view of an attack. When a process exhibits suspicious behavior, forensic tools can be used to perform static or dynamic analysis. For example, tracer is a specialized forensic tool to record the execution of a program for the purpose of monitoring and debugging. Tools such as *ltrace* and *strace* are widely used to monitor signals and library/system calls issued by a specific process during runtime. These tools get the in-context, semantics-rich tracing information by executing inside the guest OS. If the behavior of a process running inside the guest VM has been found suspicious, the tracing tool can be implanted into the guest and attached to this process to gain more detailed evidence of its malicious operations. The result of tracing can be sent from guest VM to hypervisor directly through hypercall. The host-based auditing system analyzing the logs sent from the implanted tracer can identify malware more accurately.

System Recovery/Patching: If the system has already been compromised by the malware, Process Implanting can be utilized to recover the system to its normal state by removing affected files, quarantining the suspicious malware binary image and restarting the security services that have been disabled. If any critical security vulnerability of guest OS is disclosed, the host of the guest VM is expected to patch the applications or the guest OS through Process Implanting.

III. Process Implanting Overview

A. Security Requirements

In the traditional "in-the-box" approach to detecting and subverting malware, the existence of anti-malware software is explicitly visible to the attacker. It is not difficult for a malware to identify processes or services that belong to a specific anti-malware software. The most common obfuscating technique of a malware is to disable its adversaries from executing normal operations of scanning and detecting.

Traditional anti-malware software usually runs in the user mode. It has no advantage over the malware on the same system even if it can elevate to root privilege because some malware can also obtain the same privilege by exploiting system vulnerabilities.

The "out-of-box" approach addresses the problem by moving the anti-malware software out to the host to elevate it to a higher privilege, but at the same time it loses the in-context semantics-rich view and needs to reconstruct the view from outside.

If we want to send a monitoring process into the guest OS, we need to meet a number of security requirements to make sure that this process is protected, hard to detect and tamper-resistant to attacks. Otherwise, it will have no advantage over the traditional "in-the-box" approach.

Considering that the *implanted process* still runs upon the guest OS, it will have some interactions with other components and rely on some services offered by the guest kernel. We do not want to add too many constraints on the coding of the *implanted process*, which would make it totally isolated from the environment. We want to reuse existing programs as *implanted process* with only minor modification to make it compatible with our framework. We make the assumption that the integrity of the guest kernel is not in a compromised state for the duration of implanting. The techniques introduced by NICKLE [9] and HookSafe [10] can be leveraged to guarantee the kernel integrity before and during the implanting.

We state the security requirements in four aspects, stealthiness, isolation, robustness and completeness.

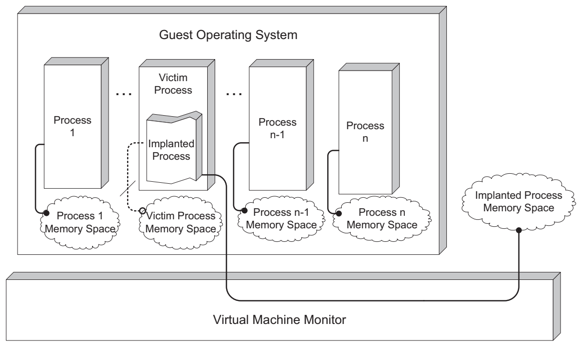


Figure 1. Overall design of Process Implanting framework

S1. Stealthiness: The *implanted process* should run without being detected by other processes in the guest OS.

S2. Isolation: It should rely on as few services of the guest OS as possible. Also it should have the minimum interactions with other processes. This can reduce the level of trustworthiness we require on the guest OS and other applications.

S3. Robustness: The *implanted process* should not be terminated by other processes in the guest VM when it is running.

S4. Completeness: When the *implanted process* runs to completion or the hypervisor needs to call it back, it should exit gracefully without any impact on the stability of the guest OS and other running applications.

B. Overall Design

The key idea of Process Implanting is to load a prepared program image from the host into the guest VM and run it with the camouflage of an existing process inside the guest OS. We denote the process to be loaded into the guest as the *implanted process* and the process used as camouflage as the *victim process*. The *implanted process* and the *victim process* can be chosen by the host administrator at runtime. Memory regions are allocated on the host separately for segments of *implanted process*. When the *victim process* is scheduled and the context switch is captured by the hypervisor, we save its CPU context and replace it with the initial context of the *implanted process*, e.g., replace the instruction pointer with the entry address of the *implanted process* binary executable and the stack pointer with the starting address of the stack, etc. Then we modify the related page table entries in both guest page table and shadow page table to enforce them pointing to the memory where the *implanted process* is loaded. When the guest OS resumes to continue the context switch operation, it will return to the user space and begin to execute the *implanted process*. When the *implanted process* has run to completion or the hypervisor wants to perform mandatory restoration of the *victim process*, we recover the *victim process* by restoring the saved context. Then the *victim process* will restart its execution. From the view of the *victim process*, it is "frozen" for a specific time and some of its time quanta are "stolen" by the *implanted process*. Figure 1 illustrates the overall design.

The complexity of this approach comes from the need to satisfy the security requirements proposed in the previous section. We will present the detailed design for fulfilling the security requirements respectively.

Random selection of victim process: The reason why traditional anti-malware software is easy to be subverted is because of its visible presence. The attacker can easily determine its existence by reading the software configuration from the OS. In order to be more stealthy, the host administrator can select the *victim process* and restart the *implanted process* from scratch by switching to another *victim process* at runtime. This property of randomness eliminates the possibility that malware can locate its opponents by querying the system.

Single virtual CPU for guest OS: If the guest VM has multiple vcpus, the *implanted process* running on one vcpu could be detected by other processes running on another vcpu simultaneously. SMP support for guest OS is disabled in our design to eliminate the possibility of other running processes detecting the existence of the *implanted process*.

Camouflage of implanted process: The *implanted process* will reuse all the data structures, e.g., process descriptor, page table, heap, etc., of the pre-existing *victim process* in the guest OS. From the view of guest VM, it can not tell the difference from the normal execution of *victim process*. The *implanted process* just "steal" several time quanta from the *victim process* to run its own program and restore the *victim process'* execution at a later time.

Moreover, the *implanted process* is not backed by a file-based binary executable on the guest OS. If the malware wants to conduct binary analysis to determine the property of this process, it can only find the executable file of the *victim process* rather than the *implanted process*.

Our design tries to satisfy the stealthiness requirement with the camouflaging of the *victim process*. Forking a new process is not permitted during the execution of the *implanted process* as it will leave obvious fingerprint, i.e., one more running process, in the guest OS. This will violate the stealthiness requirement.

Self-contained executable of implanted process: When compiling the *implanted process* executable, we choose to link the library routines statically. Though it increases the size of its binary image, it brings benefits that the *implanted process* does not need to rely on the services of library functions offered by the guest OS. Otherwise, if the library in the guest is compromised, the result generated by the *implanted process* can not be fully trusted. Then the assumption of trust level has to be elevated. With self-contained binary image, we only need to make the assumption that the system services used by the *implanted process* are trusted. This satisfies the security requirement of isolation.

Invisible memory space: We allocate three memory regions for the *implanted process*, i.e., code, data and stack segment. These memory regions are located at the host level and beyond the physical memory range of the guest VM. The guest OS only checks the physical memory at its booting time and indexes it into its kernel data structure of memory pages. Adding more memory to the guest OS during its runtime is like plugging more memory into the memory slots. The guest OS has no knowledge of the newly registered memory regions and it will not touch the memory located beyond its memory scope. This satisfies the security requirement of stealthiness because of the transparency of these memory regions.

Timing of implanting: The timing of implanting is critical to our system in order to satisfy the security requirement of completeness. From the view of the guest VM, the operation of *victim process* "freezes" when it is being implanted. Remember we choose to implant the process when there is context switch happening and the next scheduled process is the *victim process*. However, as a side effect, part of the execution in the kernel space for *victim process* will be lost because when it enters back to user space, the *implanted process* will substitute the *victim process* to execute. We make three design decisions to address this problem. We first check if this context switch is triggered by a system call from *victim process*. When we restore the *victim process* after the *implanted process* exits, we set the instruction pointer on the kernel stack backwards to restart the system call. Secondly, If the *victim process* is waiting for resources and has already set its state as *uninterruptible*, we will enter guest VM silently without implanting. Thirdly, if the context switch is caused by kernel preemption, we will not implant at this time.

Frequent scene restoration: We only permit the *implanted process* itself to read/write/execute its own memory regions and other processes should not have access to these regions. Leakage of information should be prevented in the situation that malware is capable of scanning other processes' page table to detect the modification. We design a mechanism called Frequent Scene Restoration (FSR) to recover all the states we have modified during implanting when the *implanted process* is scheduled out. After the context switch, the *victim process* is not modified in the guest's view. Though it will bring some performance degradation, it helps to meet the security requirement of stealthiness.

Checkpoint and restart: Checkpoint and restart is an optional design for some specific *implanted process*es. As the user stack of *implanted process* is allocated independently, we can checkpoint the execution status by recording the register status. The *implanted process* can restart from this checkpoint and continue its execution at a later time when we want to implant it again. This is also designed to fulfill the security requirement of stealthiness.

Coordination between implanted process and hypervisor: The coordination mechanism between the *implanted process* and the hypervisor is designed for solving the concrete problem encountered when implanting some specific processes. For example, trace programs like *ltrace* and *strace* will attach to a running process inside the guest OS and monitor its behavior. If we are going to execute mandatory restoration of *victim process*, it will cause the process being traced to behave incorrectly because the tracer has not detached it. A similar problem will arise for implanting a multi-threaded process. If all the child threads spawned by the *implanted process* are still running when the mandatory restoration happens, they will run on the address space of *victim process* after the restoration and this will cause serious problems. We design a mechanism of coordination between the *implanted process* and the hypervisor to eliminate such problems. A covert channel is created by setting a control bit on the argument part of user stack. It can be read by the *implanted process* and written by the hypervisor. The *implanted process* should check it periodically. Instead of restoring the *victim process* immediately, the hypervisor will set this control bit to tell the *implanted process* that it can exit. When the *implanted process* read this bit, it should clean up, e.g., let all the child threads exit or detach the process being traced, and then exit. The exit operation will be intercepted by the hypervisor and will be described shortly.

The other coordination mechanism is that we modify the source code of existing tools to let them send string pointers through hypercalls to the hypervisor instead of printing in the guest VM. The hypervisor can read these strings by translating guest virtual addresses to host virtual addresses.

These coordination mechanisms help meet the requirements for completeness and stealthiness.

Graceful exiting: When the *implanted process* is exiting, the guest OS will try to free the memory space of the *implanted process* we have allocated at the host level, and this will lead to a crash as the physical address of that memory space exceeds the maximum physical memory the guest could see. This will break our security requirements of stealthiness and completeness and is thus undesirable. Instead of letting the process complete its exiting, we should stop the exiting attempt and restore the *victim process* to maintain stealthiness and completeness. To perform the interception and restoration, the hypervisor needs to know exactly when the *implanted process* is going to exit. Although it is possible to modify the source code of the implanted program to let it inform the hypervisor actively, this would be inconvenient and not applicable to closed-source programs. Instead, we choose to set a trap through debug register for the exiting event of the *implanted process*, and the hypervisor will be notified as soon as the trap is triggered.

Protection from the hypervisor: The *implanted process* is not acting alone in its mission. It is backed by the hypervisor. We add protection to the *implanted process* from the hypervisor level to satisfy the security requirement of robustness. Two mechanisms are designed in the Process Implanting framework. First we elevate the privilege level of the *implanted process* to root by modifying the credential entry in the process descriptor. This has the same effect of switching a user to root in the guest OS. With root privilege, user level malware is not capable of killing it by sending a termination signal. This is also useful for some application scenarios because monitoring/patching operations can only be performed with the highest privilege in the guest VM.

However, if the malware also has the root privilege, it still can kill the *implanted process*. To strengthen robustness, we design a second approach for stronger protection. This approach is to set the *unkillable* flag for this process and check its status for every context switch. The *unkillable* flag is used only by the *init* process in Linux to prevent it from being killed in any situation. We also utilize it to make our *implanted process unkillable*. If this bit is cleared by other processes, we will check it during every context switch to make sure that it is set/reset before the running of the *implanted process*.

Multi-threaded program implanting: Multi-threading is widely used in programs nowadays, and Process Implanting should support such programs for broader range of application scenarios. However, special care is needed for both selecting a multi-threaded *victim process* and implanting a multi-threaded program.

To illustrate this problem, let us take a close look at the scene of selecting a multi-threaded *victim process*. When implanting happens, we choose a thread of the *victim process* to execute the *implanted process*. We denote this specific thread as the *victim thread*, and other threads of the *victim process* as *innocent threads*. We modify the address space and the execution context of the *victim thread* to provide an execution environment for the *implanted process*. Note that such modification to the address space is shared between all threads of the *victim process*, but the modification to the execution context is only done to the *victim thread*. When those *innocent threads* begin to execute, inconsistency between the address space and their execution contexts will lead to a crash. There are two ways to solve this problem, either by "freezing" all *innocent threads*, or by restoring the address space when there is a context switch to an *innocent thread*. We choose the latter one because it is more stealthy and easier to implement.

Implanting a multi-threaded program is more complicated. The threads created by the *implanted process* require the address space for the *implanted process* while *innocent threads* of the *victim process* require the original address space of the *victim process*, so we have to switch address space when there is a context switch between these two kinds

Algorithm 1: Multi-threaded program implanting handling on context switch

```
if $next$ is victim and $next.pid = next.tgid$ and
$imp =$ false then
    $implant()$
    $imp \leftarrow$ true
    $maxpid \leftarrow get\_maxpid\_in\_group(next)$
    $vicpid \leftarrow next.pid$
else if $imp =$ true then
    $ptype \leftarrow$ OTHER
    $ntype \leftarrow$OTHER
    if $prev.tgid = vicpid$ then
        if $prev.pid = prev.tgid$ then
            $ptype \leftarrow$VICTIM
        else if $prev.pid \leq maxpid$ then
            $ptype \leftarrow$INNOCENT
        else
            $ptype \leftarrow$IMPNEW
        end if
    end if
    if $next.tgid = vicpid$ then
        if $next.pid = next.tgid$ then
            $ntype \leftarrow$VICTIM
        else if $next.pid \leq maxpid$ then
            $ntype \leftarrow$INNOCENT
        else
            $ntype \leftarrow$IMPNEW
        end if
    end if
    if ($ptype =$VICTIM or $ptype =$IMPNEW) and
    ($ntype =$INNOCENT or $ntype =$OTHER) then
        $restore\_scene()$
    else if ($ptype =$INNOCENT or $ptype =$OTHER)
    and ($ntype =$VICTIM or $ntype =$IMPNEW) then
        $load\_scene()$
    end if
end if
```

of threads. However, there is no simple way to differentiate between these two kinds of threads because they belong to the same thread group. Our solution leverages the fact that if no *innocent thread* is created after implanting, then any thread created by the *implanted process* should have greater process id (pid) than any of the *innocent threads* (assuming the pid is in the same order of process creation time). Note that most programs only create threads in their main threads, so if we choose the main thread as the *victim thread*, the above condition is naturally met. In this way we could find the maximum pid of *innocent threads* before implanting and use it as a boundary between the two kinds of threads. Algorithm 1 shows the comprehensive logic we design based on the above methods to handle multi-threaded programs in Process Implanting. We denote the previous task as $prev$, next task as $next$, type of previous task as $ptype$, type of next task type as $ntype$, *victim thread* pid as $vicpid$, process implanted flag imp and maximum pid in *victim thread* group as $maxpid$.

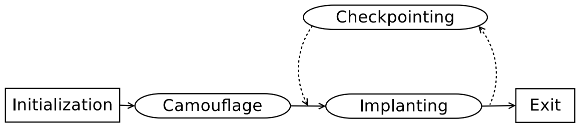


Figure 2. Basic procedure of Process Implanting

IV. Implementation

We have implemented a proof-of-concept Process Implanting prototype as an extension of KVM [11](kernel-based virtual machine) leveraging the Intel virtualization technology [12]. The host OS is Ubuntu 10.04 32bit (Linux kernel 2.6.32-23) distribution and the guest OS is Ubuntu 9.10 32bit (Linux kernel 2.6.31-14) distribution.

The basic procedures can be divided into five phases: initialization, camouflage, implanting, checkpointing and exit, as illustrated in Figure 2. The dotted line for the phase of checkpointing means that it is optional. We will explain each phase in detail below:

A. Initialization

We choose ELF (Executable and linkable format) as the file format for the *implanted process* image. The binary image is compiled beforehand by statically linking all the library routines to make it self-contained. A program loader is implemented to load the code/data/stack segments into the memory allocated on the host.

Then we register the memory slots in KVM for these three memory segments and assign them to guest physical addresses beyond the boundary of existing memory size of the guest OS. These memory segments are transparent to the guest OS which has no knowledge that they are "plugged in" during runtime. The guest OS calculates the memory pages at booting time. Only the *implanted process* can touch this part of memory in the later phases.

B. Camouflage

In the camouflage phase, the *victim process* name can be determined at runtime of guest OS by writing into the victim configuration file which is read by the hypervisor periodically. The guest virtual addresses of exported kernel functions can be read from the system map for guest kernel. *__switch_to* is the function responsible for context switch in Linux kernel. After finding its entry address by searching the system map, we set debug register at this address in the guest kernel. Every context switch will cause debug exception and can be captured by the hypervisor. In our previous design, we regarded the setting of new cr3 register as the symbol of context switch. This can not meet the security requirements because if the thread scheduled after the *implanted process* is a kernel thread or a user thread in the same thread group, it will reuse the previous cr3 and no VM exit will happen. If it is a malicious thread, it can scan the page table of the previous thread to dump the code and data segments of the *implanted process*. With the VM exit for every context switch and single virtual cpu support, if the previous thread is the *implanted process*, we can restore both the page table and the modified entries in process descriptor before the execution of the next thread. Before implanting, we need to fill the upper part of the user stack by copying the content from the *victim process'* user stack. These are arguments, environments and auxiliary array which will be read by the *implanted process* at its loading time.

C. Implanting

When a context switch happens and the *victim process* is the next thread to be scheduled, the guest VM exits to the hypervisor. All the user registers' value for the *victim process* are stored on the kernel stack. The steps of implanting are:

1) Save user registers' value of *victim process*
2) Save the memory region descriptor's list
3) Save the original affected address mapping
4) Adjust physical page table of the *victim process* to point to implant process' memory
5) Update related entries in shadow page table
6) Adjust the memory region descriptor's list to adapt to the new address space
7) Set user registers' value on the kernel stack with the user registers' value of *implanted process*

After completing 7 steps above, when the guest VM resumes guest mode, the *victim process* is completely replaced with the *implanted process*. The procedure of switching to address space of *implanted process* is illustrated in Figure 3.

After the first-time implanting, when the *implanted process* is scheduled out, we will restore the *victim process'* physical page table and memory region descriptor list via FSR. If the *implanted process* is scheduled again, we will load the *implanted process*' physical page table and memory region descriptor list.

D. Checkpointing

Checkpointing phase is optional and its main purpose is to raise the bar of stealthiness. *Implanted process* can be checkpointed at a specific time by saving its execution state, i.e., user registers, memory region descriptor list and restore the execution of *victim process*. The *implanted process* can restart at the checkpoint and continue execution after the *victim process* runs for several time quanta.

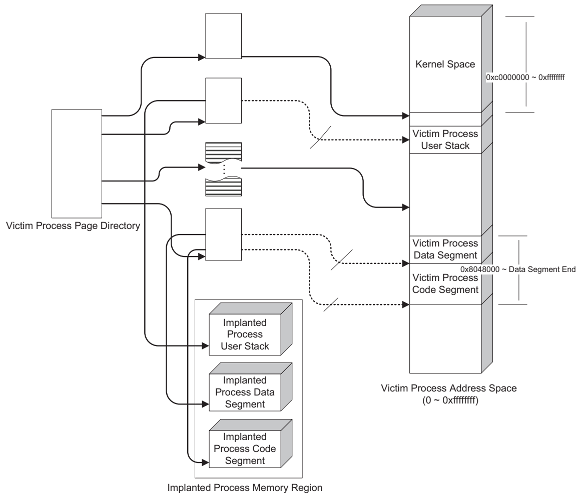

Kernel Space
0xc0000000 ~ 0xffffffff
Victim Process User Stack
Victim Process Page Directory
Victim Process Data Segment
0x8048000 ~ Data Segment End
Victim Process Code Segment
Implanted Process User Stack
Implanted Process Data Segment
Implanted Process Code Segment
Victim Process Address Space (0 ~ 0xffffffff)
Implanted Process Memory Region

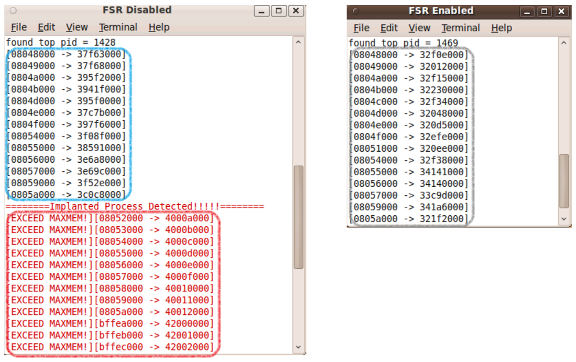
Figure 3. Address space of implanted process

E. Exit

When the *implanted process* attempts to exit, we need to restore the *victim process*. The hypervisor intercepts the exiting attempt by setting traps on the system calls of *sys_exit* and *sys_exit_group*. All user mode processes in Linux call either of these two system calls when they are going to exit. We set two debug registers to the entry points of these two system calls. So any call or jump to them will trigger a VM exit and be captured by the hypervisor. When the hypervisor intercepts an exiting event and finds the exiting process to be the *implanted process*, it will restore the *victim process*. Note that restoration here is slightly different from what we do in the checkpointing phase. In the checkpointing phase, the user mode registers saved in kernel stack will be restored directly when the kernel is returning to user mode. However, in the exiting phase, the user mode *eax* register will be set to the return value of the *sys_exit* or *sys_exit_group* system call by the kernel. This is unexpected since the call was invoked by the *implanted process* but not the *victim process*, and the user mode *eax* register of the *victim process* should not be tampered. To solve this problem, we set the kernel mode *eax* register, which is used to store the return value of the system call, to the same value as the user mode *eax* register of the *victim process*. In this way the user mode *eax* register of the *victim process* will remain unmodified even if it is set to the return value of the system call. Also, because the *sys_exit* or *sys_exit_group* function should not be actually executed, we set the instruction pointer and the stack pointer to the frame of the function's caller, so the result looks like we return from the function without executing its code.

V. Evaluation

In this section, we evaluate our Process Implanting framework in three aspects. First, we evaluate how well our



```
found top pid = 1428
[08048000 -> 37f63000]
[08049000 -> 37f68000]
[0804a000 -> 395f2000]
[0804b000 -> 3941f000]
[0804d000 -> 395f0000]
[0804e000 -> 37c7b000]
[0804f000 -> 397f6000]
[08054000 -> 3f08f000]
[08055000 -> 38591000]
[08056000 -> 3e6a8000]
[08057000 -> 3e69c000]
[08059000 -> 3f52e000]
[0805a000 -> 3c0c8000]
========Implanted Process Detected!!!!!========
[EXCEED MAXMEM!][08052000 -> 4000a000]
[EXCEED MAXMEM!][08053000 -> 4000b000]
[EXCEED MAXMEM!][08054000 -> 4000c000]
[EXCEED MAXMEM!][08055000 -> 4000d000]
[EXCEED MAXMEM!][08056000 -> 4000e000]
[EXCEED MAXMEM!][08057000 -> 4000f000]
[EXCEED MAXMEM!][08058000 -> 40010000]
[EXCEED MAXMEM!][08059000 -> 40011000]
[EXCEED MAXMEM!][0805a000 -> 40012000]
[EXCEED MAXMEM!][bffea000 -> 42000000]
[EXCEED MAXMEM!][bffeb000 -> 42001000]
[EXCEED MAXMEM!][bffec000 -> 42002000]
```

```
found top pid = 1469
[08048000 -> 32f0e000]
[08049000 -> 32012000]
[0804a000 -> 32f15000]
[0804b000 -> 32230000]
[0804c000 -> 32f34000]
[0804d000 -> 32048000]
[0804e000 -> 320d5000]
[0804f000 -> 32efe000]
[08051000 -> 320ee000]
[08054000 -> 32f38000]
[08055000 -> 34141000]
[08056000 -> 34140000]
[08057000 -> 33c9d000]
[08059000 -> 341a6000]
[0805a000 -> 321f2000]
```

Figure 4. Scan the page table of *victim process*

system meets the security requirements. Then we present some active VM introspection cases. Finally we present the results of performance measurements.

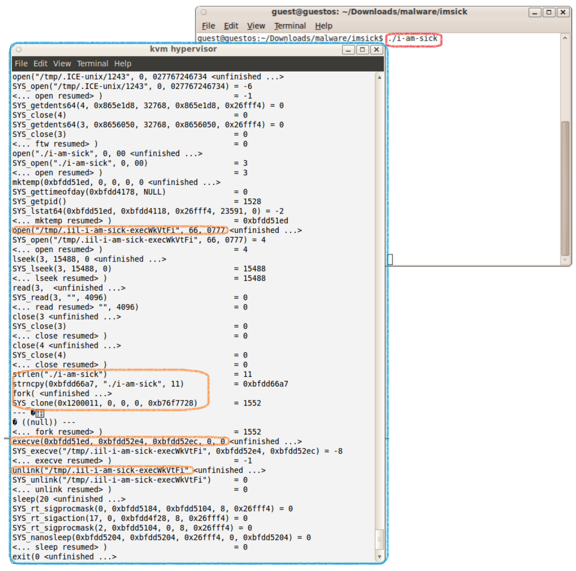
A. Security

Experiment I: Scan the page table of victim process

To demonstrate that even if the guest kernel is trusted, our mechanism of FSR is still crucial and necessary to maintain the stealthiness of the *implanted process*, we have implemented a small userspace program to simulate the potential attack by scanning the page table of the *victim process* repeatedly. The feature that allows userspace programs to read page tables of other processes by reading */proc/pid/pagemap* is granted in the Linux kernel of version 2.6.25 and newer. We assume that the attacker knows all about our design, details of our implementation and even which process we may choose as a *victim process*.

The detection follows this observation: if a process is implanted, some of its virtual pages will be mapped to physical pages exceeding the maximum physical memory allocated for the guest VM. For normal processes, no such mapping exists. This is because the memory used to store code, data segments and stack of the *implanted process* is allocated on host, which falls out of the border of guest memory. The scanner will claim that a process is implanted if it finds such suspicious page mappings in the page table of that process.

We performed the experiment with FSR enabled and disabled, respectively. The comparison of these two results is shown in Figure 4. When FSR was disabled, before the implanting, the scanner found the original page mappings of the *victim process* as shown in upper box in the left window. Then, right after the implanting, changes made to the page table and suspicious page mappings shown in the lower box in the left window were discovered. On the contrary, when FSR was enabled, the attacker could only find the original page mappings of the *victim process* shown in the right window during the entire experiment. This is because when the scanning process was running, the *implanted process* must had been scheduled out at an earlier time and we had already recovered all the information we modified during implanting using FSR.


guest@guestos: ~/Downloads/malware/imsick
File Edit View Terminal Help

```
guest@guestos:~/Downloads/malware/imsick$ ./i-am-sick
```

kvm hypervisor
File Edit View Terminal Help

```
open("/tmp/.ICE-unix/1243", 0, 027767246734 <unfinished ...>
SYS_open("/tmp/.ICE-unix/1243", 0, 027767246734) = -6
<... open resumed> )                                = -1
SYS_getdents64(4, 0x865e1d8, 32768, 0x865e1d8, 0x26fff4) = 0
SYS_close(4)                                        = 0
SYS_getdents64(3, 0x8656050, 32768, 0x8656050, 0x26fff4) = 0
SYS_close(3)                                        = 0
<... ftw resumed> )                                 = 0
open("./i-am-sick", 0, 00 <unfinished ...>
SYS_open("./i-am-sick", 0, 00)                      = 3
<... open resumed> )                                = 3
mktemp(0xbfdd51ed, 0, 0, 0, 0 <unfinished ...>
SYS_gettimeofday(0xbfdd4178, NULL)                  = 0
SYS_getpid()                                        = 1528
SYS_lstat64(0xbfdd51ed, 0xbfdd4118, 0x26fff4, 23591, 0) = -2
<... mktemp resumed> )                              = 0xbfdd51ed
open("/tmp/.iil-i-am-sick-execWkVtFi", 66, 0777 <unfinished ...>
SYS_open("/tmp/.iil-i-am-sick-execWkVtFi", 66, 0777) = 4
<... open resumed> )                                = 4
lseek(3, 15488, 0 <unfinished ...>
SYS_lseek(3, 15488, 0)                              = 15488
<... lseek resumed> )                               = 15488
read(3,  <unfinished ...>
SYS_read(3, "", 4096)                               = 0
<... read resumed> "", 4096)                        = 0
close(3 <unfinished ...>
SYS_close(3)                                        = 0
<... close resumed> )                               = 0
close(4 <unfinished ...>
SYS_close(4)                                        = 0
<... close resumed> )                               = 0
strlen("./i-am-sick")                               = 11
strncpy(0xbfdd66a7, "./i-am-sick", 11)              = 0xbfdd66a7
fork( <unfinished ...>
SYS_clone(0x1200011, 0, 0, 0, 0xb76f7728)           = 1552
--- �
� ((null)) ---
<... fork resumed> )                                = 1552
execve(0xbfdd51ed, 0xbfdd52e4, 0xbfdd52ec, 0, 0 <unfinished ...>
SYS_execve("/tmp/.iil-i-am-sick-execWkVtFi", 0xbfdd52e4, 0xbfdd52ec) = -8
<... execve resumed> )                              = -1
unlink("/tmp/.iil-i-am-sick-execWkVtFi" <unfinished ...>
SYS_unlink("/tmp/.iil-i-am-sick-execWkVtFi")        = 0
<... unlink resumed> )                              = 0
sleep(20 <unfinished ...>
SYS_rt_sigprocmask(0, 0xbfdd5184, 0xbfdd5104, 8, 0x26fff4) = 0
SYS_rt_sigaction(17, 0, 0xbfdd4f28, 8, 0x26fff4) = 0
SYS_rt_sigprocmask(2, 0xbfdd5104, 0, 8, 0x26fff4) = 0
SYS_nanosleep(0xbfdd5204, 0xbfdd5204, 0x26fff4, 0, 0xbfdd5204) = 0
<... sleep resumed> )                               = 0
exit(0 <unfinished ...>
```

Figure 5. Implanting ltrace to trace malware

B. Active VM introspection case study

Ltrace is a tool to intercept and record the library/system calls and the signals of a specific process. It can attach to a process and monitor its behavior during runtime. We performed two experiments to demonstrate its usage in our framework.

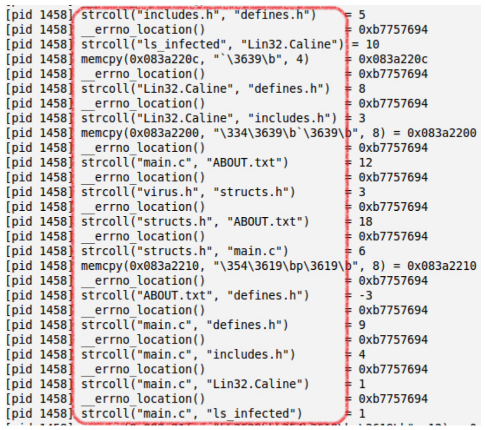
Experiment II: Implanting ltrace to trace malware

In this experiment, we implanted *ltrace* into the guest OS and leveraged it to trace both the library and system calls of a piece of real-world malware. The results were transferred to the hypervisor on the host through hypercalls.

The right window in Figure 5 presents the malware whose name is *i-am-sick*. Its main function is to infect the files under the */tmp* directory by copying code into them and execute the infected files afterwards. With the implanted *ltrace*, we could attach to the malware at runtime and monitor the library calls and system calls. The left window in Figure 5 is the console of KVM hypervisor. It received logs directly from the implanted *ltrace*. After inspecting the logs in this console, the execution path and malicious behavior of this malware can be easily identified.

Experiment III: Implanting ltrace to trace infected application

In this experiment, we implanted *ltrace* to trace the *ls* which had been infected by *caline*. *Caline* is an ELF infector using S.P.I (segment padding infection) technique. It inserts virus code after the code segment of an ELF binary to change its behavior. Through tracing the infected *ls*, we could easily identify the deviated execution path by checking



```
[pid 1458] strcoll("includes.h", "defines.h")    = 5
[pid 1458] __errno_location()                    = 0xb7757694
[pid 1458] strcoll("ls_infected", "Lin32.Caline") = 10
[pid 1458] memcpy(0x083a220c, "`\3639\b", 4)      = 0x083a220c
[pid 1458] __errno_location()                    = 0xb7757694
[pid 1458] strcoll("Lin32.Caline", "defines.h")  = 8
[pid 1458] __errno_location()                    = 0xb7757694
[pid 1458] strcoll("Lin32.Caline", "includes.h") = 3
[pid 1458] memcpy(0x083a2200, "\334\3639\b`\3639\b", 8) = 0x083a2200
[pid 1458] __errno_location()                    = 0xb7757694
[pid 1458] strcoll("main.c", "ABOUT.txt")        = 12
[pid 1458] __errno_location()                    = 0xb7757694
[pid 1458] strcoll("virus.h", "structs.h")       = 3
[pid 1458] __errno_location()                    = 0xb7757694
[pid 1458] strcoll("structs.h", "ABOUT.txt")     = 18
[pid 1458] __errno_location()                    = 0xb7757694
[pid 1458] strcoll("structs.h", "main.c")        = 6
[pid 1458] memcpy(0x083a2210, "\354\3619\bp\3619\b", 8) = 0x083a2210
[pid 1458] __errno_location()                    = 0xb7757694
[pid 1458] strcoll("ABOUT.txt", "defines.h")     = -3
[pid 1458] __errno_location()                    = 0xb7757694
[pid 1458] strcoll("main.c", "defines.h")        = 9
[pid 1458] __errno_location()                    = 0xb7757694
[pid 1458] strcoll("main.c", "includes.h")       = 4
[pid 1458] __errno_location()                    = 0xb7757694
[pid 1458] strcoll("main.c", "Lin32.Caline")     = 1
[pid 1458] __errno_location()                    = 0xb7757694
[pid 1458] strcoll("main.c", "ls_infected")      = 1
```

Figure 6. Trace the library call of infected application

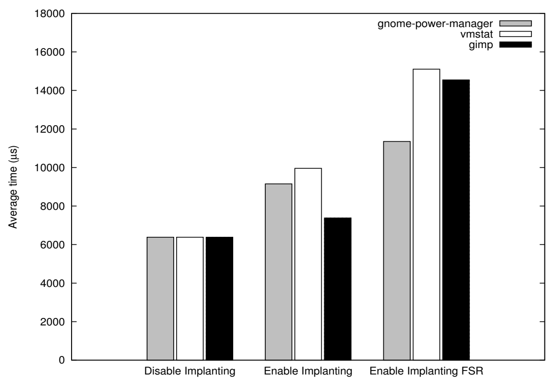


Figure 7. Performance comparison of implanted process

the arguments of library calls. The box in Figure 6 presents the suspicious execution result.

System events tracing is one of the most important techniques for computer forensics to collect evidence of malware and still requires better support by virtual machine introspection, especially in the era when hardware virtualization technology is widely deployed. Traditional methods used in QEMU [13] based system to intercept system calls can not be readily used because a system call is no longer via a privileged instruction and will not cause VM exit under hardware virtualization. The common technique now is to set a trap point at the system call table entry address or set up page faults manually to cause the VM exit. This type of methods will introduce performance degradation because VM entry and exit are heavy weight and expensive operations [14]. Furthermore, there has been no introspection technique that can track user-level events such as library function calls. Process Implanting provides a useful option for such needs.

C. Performance measurements

Our testing platform for host is Dell Optiplex 755 with Intel Core(TM)2 Quad Q6600 2.40GHz CPU and 3GB memory. We allocated 1GB memory to the guest OS. The performance of *implanted process* was measured in three scenarios:

Implanting disabled: in this scenario, all the capabilities of process implanting were disabled. It was used as the base case for performance measurement.

Implanting enabled: the function of process implanting was enabled in this scenario. But we still disabled the feature of FSR which is used to enhance the security by restoring the execution scene when the *implanted process* is scheduled out.

Enable both implanting and FSR: The FSR was enabled along with implanting in this scenario to measure the performance overhead that was introduced for this feature.

We have implemented a program as micro-benchmark to test the performance. Its main function is to read and write the entries in the */proc* file system, allocate/free memory. This program has been run 1000 times to get the average running time. Three different kinds of applications in guest OS were used as *victim processes*, *gnome-power-manager*, *vmstat* and *gimp*.

Gnome-power-manager is a session daemon to manage the power for the laptop or desktop. It is a good candidate for the *victim process* because it is scheduled periodically to check the status of battery and AC power. *Vmstat* is a console tool to report the virtual memory statistics. It has no interaction with the user. *Gimp* is a widely used image manipulation program under Linux. It is an interactive GUI program. Figure 7 shows the performance result of this micro-benchmark. The y-axis is the average time to run the micro-benchmark once. If we only enable the implanting without FSR, it introduced 43.4%, 55.6%, and 15.4% performance overhead respectively for these three *victim processes* comparing with running the process directly on the guest OS. With FSR enabled, the performance overhead increased by 24.1%, 51.6%, and 97% respectively, comparing with the system with only implanting enabled. The performance overhead comes from two sources. The first one is introduced by the virtual machine introspection. Debug register is set at the entry address of context switch function of the guest kernel. Guest VM exits when there is a process scheduled to run. The other source of performance overhead is from the FSR. FSR will restore the execution context of the *victim process* to eliminate the possibility for other processes to detect the occurrence of implanting. The effectiveness of FSR has been demonstrated in Experiment I. In FSR, the memory region list is restored every time when a context switch occurs. It will introduce more overhead if the *victim process* itself is more complex. When the *victim process* was *gimp*, we can see from Figure 7 that the running time jumped by 97% after FSR was enabled.

We point out that the performance overhead is incurred only at the time of implanting. Process Implanting framework is designed to be modular and decoupled with other functionalities of KVM hypervisor. The implanting capability can be turn off easily without impacting other components in the hypervisor. When the *implanted process* exits, the function of implanting can be disabled by removing the breakpoints set by the debug registers and the system can return to its original performance level.

VI. Discussion

A. Limitations

Although our approach tries to maximize the stealthiness of the *implanted process*, during our experiments we find out that it is necessary to limit the behavior of *implanted process*, otherwise it will still incur the risk of being detected. In our experiment of implanting *ltrace* to trace malware, *ltrace* uses the ptrace system call which will reveal the pid of the tracer to the program being traced. The malware could check if it is being traced and which process is tracing it. With the FSR feature enabled, the malware can only detect that the *victim process* has some abnormal behavior, but can not detect the behavior of an *implanted process*. We leave the further improvement of *implanted process* stealthiness to our future work.

The other limitation is that there are some programs that are not so suitable to be used as *victim processes*. (1) Processes that have no chance to be scheduled. Since our approach rely on the capture of context switch, if a process can not be scheduled, we have no opportunity to select it. (2) Processes that have IPC with other processes or have communications with devices. Because our *implanted process* actually "steal" time quanta from the *victim process*, message for the *victim process* from IPC or devices will be lost or incorrectly handled during the execution of the *implanted process*.

B. Coding requirement for implanted process

We have written a series of tools and modified some existing ones to help generate programs for process implanting. The following practice is required for the development of implanted programs.

1) Add *-static* flag to the makefile to statically link all library routines.
2) Check the control bit in the argument periodically. If it is set, it means that the hypervisor wants to restore the *victim process*. The *implanted process* should clean up and then exit.

VII. Related Work

Virtual machine introspection has been researched and deployed widely to assist enhancing system security. These approaches can be classified into two categories: passive introspection and active introspection. In the former category, virtualization is utilized to gain higher privilege over the attacks and monitoring tools are moved out from the guest VM to the outside. The semantic gap [1] makes it difficult to leverage the services offered by the guest kernel or gain assistance from other applications running inside the guest VM. Semantic information needs to be re-created at the hypervisor level which is below the guest OS. Livewire [2] is the first to propose the virtual machine introspection methodology to detect malware infections by inspecting the internal states of a guest VM. XenAccess [4], VMwatcher [3], VMscope [15], Antfarm [5], and Ether [16] are representative "out-of-box" efforts to monitor the guest VM at the hypervisor level.

In contrast, active introspection interferes with an attack when it has been detected. IntroVirt [17] applies virtual machine introspection to execute vulnerability-specific predicates in a VM to detect and respond to intrusions. Lycosid [6] detects process that is maliciously hidden by using cross-view validation techniques and then patches the executable code to affect the execution of this specific process. Manitou [18] compares instruction-page hashes with memory-page hashes at runtime. If there is no matching, it considers that the instruction page has been corrupted and marks it as non-executable. Highly efficient active monitoring from outside an untrusted VM is proposed by Lares [7] and SIM [8]. Hooks are placed inside the guest OS to intercept the executing events to invoke the security tool. Lares [7] places the security tool in another trusted guest VM and the hooked system events will trigger the VM switch. SIM [8] gains the in-context view by creating a separate guest address space that is protected by the hypervisor to gain the native speed.

Process Implanting takes a complementing approach to active VM introspection. We directly implant the whole process from the host into the guest and run it under the cover of an existing running process to gain in-VM semantic information and benefit from leveraging the services offered by the guest kernel. Additional protection and coordination are provided by the hypervisor to keep the *implanted process* safe inside the guest VM. Since an *implanted process* relies partly on the services offered by the guest OS, the integrity of the kernel should be enforced for the duration of implanting. A lot of research has been done to verify the integrity of the kernel. Copilot [19] uses a trusted PCI card to fetch the memory image of OS at runtime and detect the presence of rootkit through checking the integrity of kernel code. SecVisor [20] is a light-weight hypervisor to protect the kernel against code injection attacks. It ensures that only approved code can execute with kernel privilege over the entire lifetime of its guest VM. NICKLE [9] and Patagonix [21] enforce that only verified kernel code can be fetched for execution in the kernel space. HookSafe [10] relocates hooks in the kernel to a page-aligned memory space and regulates accesses to them with hardware-based page-level protection. Process Implanting can leverage the guest kernel integrity guarantee provided by these systems.

VIII. Conclusion

We have designed and implemented Process Implanting, a general-purpose active VM introspection framework. It provides a way to implant a process from the host into the guest and the *implanted process* will run under the cover of an existing *victim process*. Through the coordination and protection from the hypervisor, the *implanted process* can achieve a degree of tamper-resistance and stealthiness in the guest VM. We also propose a number of application scenarios for Process Implanting and demonstrate its practicality and effectiveness in those scenarios.

Acknowledgment

We would like to thank all the anonymous reviewers for their insightful comments and suggestions. This research was supported, in part, by the US Air Force Office of Scientific Research (AFOSR) under Contract FA9550-10-1-0099, the US Air Force Research Laboratory (AFRL) under Contract FA8750-09-1-0224, and the US National Science Foundation (NSF) under grants 0716444, 0546173, 0855036, 0855141. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of the AFOSR, AFRL, or NSF.

References

[1] P. Chen and B. Noble, "When virtual is better than real," in *hotos*. Published by the IEEE Computer Society, 2001, p. 0133.

[2] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Network and Distributed Systems Security Symposium*, vol. 1. Citeseer, 2003, pp. 253–285.

[3] 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.

[4] B. Payne, M. Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *acsac*. IEEE Computer Society, 2007, pp. 385–397.

[5] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Antfarm: Tracking processes in a virtual machine environment," in *Proceedings of the USENIX Annual Technical Conference*, 2006, pp. 1–14.

[6] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "VMM-based hidden process detection and identification using Lycosid," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2008, pp. 91–100.

[7] B. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 233–247.

[8] M. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 477–487.

[9] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Recent Advances in Intrusion Detection*. Springer, 2008, pp. 1–20.

[10] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 545–554.

[11] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[12] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel virtualization technology: Hardware support for efficient processor virtualization," *Intel Technology Journal*, vol. 10, no. 3, pp. 167–177, 2006.

[13] F. Bellard, "QEMU, a fast and portable dynamic translator." USENIX, 2005.

[14] L. van Doorn, "Hardware virtualization trends," in *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, 2006, pp. 45–45.

[15] X. Jiang and X. Wang, "Out-of-the-box monitoring of VM-based high-interaction honeypots," in *Proceedings of the 10th international conference on Recent advances in intrusion detection*. Springer-Verlag, 2007, pp. 198–218.

[16] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 51–62.

[17] A. Joshi, S. King, G. Dunlap, and P. Chen, "Detecting past and present intrusions through vulnerability-specific predicates," in *Proceedings of the twentieth ACM symposium on Operating systems principles*. ACM, 2005, pp. 91–104.

[18] L. Litty and D. Lie, "Manitou: a layer-below approach to fighting malware," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 2006, pp. 6–11.

[19] N. Petroni Jr, T. Fraser, J. Molina, and W. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*. USENIX Association, 2004, p. 13.

[20] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. ACM, 2007, pp. 335–350.

[21] L. Litty, H. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *Proceedings of the 17th conference on Security symposium*. USENIX Association, 2008, pp. 243–258.