EXTERIOR: Using A Dual-VM Based External Shell for Guest-OS Introspection, Configuration, and Recovery

Yangchun Fu
Department of Computer Science
The University of Texas at Dallas
800 West Campbell RD
Richardson, TX 75080
yangchun.fu@utdallas.edu

Zhiqiang Lin
Department of Computer Science
The University of Texas at Dallas
800 West Campbell RD
Richardson, TX 75080
zhiqiang.lin@utdallas.edu

Abstract

This paper presents EXTERIOR, a dual-VM architecture based external shell that can be used for trusted, timely out-of-VM management of guest-OS such as introspection, configuration, and recovery. Inspired by recent advances in virtual machine introspection (VMI), EXTERIOR leverages an isolated, secure virtual machine (SVM) to introspect the kernel state of a guest virtual machine (GVM). However, it goes far beyond the *read-only* capability of the traditional VMI, and can perform automatic, fine-grained guest-OS writable operations. The key idea of EXTERIOR is to use a dual-VM architecture in which a SVM runs a kernel identical to that of the GVM to create the necessary environment for a running process (e.g., `rmmod`, `kill`), and dynamically and transparently redirect and update the memory state at the VMM layer from SVM to GVM, thereby achieving the same effect in terms of kernel state updates of running the same trusted in-VM program inside the shell of GVM. A proof-of-concept EXTERIOR has been implemented. The experimental results show that EXTERIOR can be used for a timely administration of guest-OS, including introspection and (re)configuration of the guest-OS state and timely response of kernel malware intrusions, without any user account in the guest-OS.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection; D.3.4 [*Software*]: Processors—Code generation; Translator writing systems and compiler generators

General Terms Security, Management

Keywords Virtual machine introspection, Semantic-gap, Binary code reuse

1. Introduction

Virtual machines (VMs) [18, 19] have significantly reshaped the landscape of our modern computer systems. They have provided a variety of new opportunities for operating system (OS) developers to deploy innovative, previously infeasible out-of-VM solutions, such as server consolidation [48], machine migration [10], strong isolation, better security [8, 15–17, 51], reliability and portability [7]. Undoubtedly, system virtualization has pushed our computing paradigm from multi-tasking computing to multi-OS computing, and has already become ubiquitous in the realm of enterprise computing infrastructure (underpinning our cloud computing and data centers today).

In the physical world (without virtualization) all the application software runs on top of a particular OS that is installed on a particular physical machine. Both the application and OS kernel state is updated and stored in the system memory during their instruction execution. With the introduction of virtualization, all the hardware resources including physical memory and even CPU instructions are virtualized in a VM monitor (VMM) [18, 19], and today we still follow such a traditional program execution model. Namely, each of the program state (including kernel) in-VM is also stored in the system memory, and is updated through the execution of in-VM program code. For instance, to configure the host name, we have to execute `hostname` in-VM to update the kernel state; to remove a malicious kernel module (e.g., a detected installed rootkit), we have to execute `rmmod` in-VM to unload it from the OS kernel; and to stop a malicious process, we have to execute `kill` in-VM to remove it.

However, from a security perspective, the traditional program execution model has at least the following issues: (1) In-VM programs (e.g., `hostname`,`rmmod`, `ps`) and kernel states are directly faced by user level, as well as kernel level malware, and they can often be attacked. For instance, malicious processes and device drivers (or kernel modules) can hide from in-VM system enumeration tools (e.g., `ps`,`lsmod`) and can be immune to attempts of removal or disabling [24, 41, 42]. (2) End-users or administrators often have to be authenticated before running in-VM programs to update the kernel state, which may not be ideal for a timely response to intrusions (e.g., `kill` a rootkit hidden process), especially for cloud providers who in many cases do not have a user account in the guest-OS.

Therefore, in this paper we introduce EXTERIOR, a new program execution model in which we can run programs in an outer-shell for a guest-OS administration, with the same effect in terms of kernel state updates akin to running the programs inside the GVM. As such, unlike traditional in-VM program execution model, we update the in-VM kernel state entirely from the outside. We can thus ensure the trustworthiness of our out-of-VM programs because they are located out-of-VM and there is a world switch (far from reaching) with the in-VM programs such as the in-VM malware. Also, we do not have to be authenticated, and we execute the trusted out-of-VM programs in our outer-shell which is outside control of the in-VM software.

VEE'13, March 16–17, 2013, Houston, Texas, USA.

To realize EXTERIOR, we have to develop the out-of-VM programs that can precisely identify the memory locations that reflect the in-VM kernel state and update them correspondingly at the VMM layer. Unfortunately, this is challenging because of a semantic gap [7]. In particular, to the VMM, the view of the guest-OS is just the raw bits and bytes of the physical memory and register state. In contrast to the in-VM environment, where we have rich semantics (e.g., files, APIs, and system calls), at the VMM layer there is no OS-level abstraction. Consequently, we have to bridge the semantic gap and identify the in-VM kernel state variables at the VMM layer, which is typically a tedious, time-consuming, and error-prone process [13, 15, 24, 25, 39].

Fortunately, by tracing how the traditional in-VM program executes and updates the kernel state, we observe that OS kernel state is often inspected and updated at certain kernel system call (syscall for short in the rest of the paper) execution context. For instance, syscall `getpid` will return the `pid` of the running process, and syscall `sysctl` will update kernel parameters. If we are able to precisely identify these in-VM syscall execution contexts when the corresponding trusted utilities are executed at the VMM layer, and if we are able to maintain a secure duplicate of the running VM, through redirecting both their memory read and write operation from the secure VM to the running VM, then we can transparently update the in-VM kernel state of the running VM from the outside via the secure VM. In other words, the semantic-gap (e.g., the memory location of in-VM kernel state) is automatically bridged by the intrinsic instruction constraints encoded in the binary code in the duplicated VM.

As a result, it leads to a dual-VM architecture for EXTERIOR with a secure VM (SVM) and guest VM (GVM). Specifically, in EXTERIOR, we need a trusted, corresponding guest-OS kernel with the same version installed in a separate SVM, in which not only we have the full control and can ensure its integrity but also in which we can run the native administration utilities and perform the memory redirection. Through running the trusted binary code in the monitored SVM, EXTERIOR transparently redirects the memory read and write operations of kernel memory from SVM to GVM, thus modifying the state of the GVM. Therefore, the outer-shell for the GVM is actually located in our SVM, and now we can *execute* trusted, native, widely tested administration utilities in SVM to *timely* supervise the state of GVM, including *introspection* and *(re)configuration* of guest-OS kernel state as well as *recovery* and response to intrusions (EXTERIOR is named based on these underscored characters).

In a nutshell, EXTERIOR has reshaped the traditional program execution model. Normally, a given program runs on top of a given OS within a shell. EXTERIOR changes this model and supports running programs completely outside of the OS with the same effect as running the program inside in terms of kernel state update, thanks to the powerful, programmable VMM. A direct outcome of EXTERIOR is that we can now run the trusted administration utilities to (re)configure the guest-OS and respond quickly to intrusions such as recovering the system from attacks (e.g., `kill` a rootkit created process, and `rmmod` a malicious kernel module) entirely from out-of-VM, without any user account inside the guest-OS. Therefore, EXTERIOR will significantly ease the administration of the guest-OS. When combined with the intrusion detection techniques from VM Introspection [13–15, 24, 25, 39], EXTERIOR can also provide a timely response to intrusions detected in the guest-OS.

In summary, this paper makes the following contributions:

- We present a dual-VM architecture based external shell, and demonstrate for the first time that using such a shell we can run a program completely out-of-VM to achieve the same effect of running it in-VM in terms of kernel state update, but with greater privilege depth.
- We systematically explore the principles behind such architecture, and devise a number of enabling techniques. In particular, unlike all of the existing VMI techniques, which only extract and distinguish coarse-grained process information, we are the first to present a new and binary code analysis technique to extract and isolate even the fine-grained thread information at VMM layer.
- We have built a prototype of EXTERIOR and performed an empirical evaluation with Linux kernels. Our experimental results show that EXTERIOR can be used in a timely, trusted manner for guest-OS administration, including but not limited to system *introspection*, *(re)configuration* and *recovery* of kernel intrusions.

The rest of the paper is organized as follows: §2 begins with a description of the further motivation and challenges we will be facing, and an overview of our approach. §3 elaborates by presenting a detailed design of EXTERIOR, including its kernel system call identification, kernel data identification and redirection, and GVM memory address mapping and updating. §4 shares the details on how we implement each component of EXTERIOR. §5 reports our evaluation results. §6 discusses known limitations of our system, and §7 provides direct comparisons to related work. Finally, §8 concludes.

2. Background and Overview

The central goal of EXTERIOR is to support the trusted, timely introspection (note that the term introspection means inspecting kernel state from outside, whereas inspection is still inside), (re)configuration, and recovery of guest-OS kernel state from out-of-VM. In this section, we first describe further why we need EXTERIOR in §2.1. Then we examine the challenges faced when realizing it as well as how we will address them in §2.2. Finally, we give an overview of our system in §2.3.

2.1 Further Motivation

There are a number of reasons why we need out-of-VM program execution to manage the guest-OS. Besides all the benefits such as isolation, portability, and reliability [7] while implementing the service out-of-VM, we could have the following additional benefits:

Trustworthiness Recent cyber attacks such as kernel rootkits have pushed our defense software into the hypervisor or even hardware layers (i.e., out-of-VM). It is generally believed to be much harder for attackers to tamper with the software running out-of-VM, because there is a world switch for the attacks from in-VM to out-of-VM (unless the VMM has vulnerabilities). Therefore, we can gain a higher trustworthiness of the out-of-VM software. For instance, we can guarantee that the administration utilities (e.g, `ps`) are not tampered before using them to manage a guest-OS in our SVM as our SVM is not directly faced by attackers.

Higher Privilege and Stealthiness Traditional security software (e.g., anti-virus, or host intrusion detection) runs inside the guest-OS, and in-VM malware can often disable the execution of these software. By moving the execution of security software out-of-VM, we can achieve higher privilege (same as hypervisor) and stealthiness, and make them invisible to attackers. For instance, malicious code (e.g., kernel rootkit) often disables the `ps` command from showing the running malicious process, and disables the `rmmod` command needed to remove a kernel module. Through enabling the execution of these commands out-of-VM, we can achieve higher privilege and stealthiness, and prevent the rootkits from tampering with the security software.

Automation When an intrusion is detected, it often requires an automated response. Current practice is often to notify the administrators or execute some automated responses inside the guest-OS. Unfortunately, again any in-VM responses can be disabled by attackers because they run at the same privilege level. However, with EXTERIOR we support running software out-of-VM, and we can quickly take actions to stop and prevent the attack without the help from any in-VM root privileges. Considering there are a great deal of VMI based intrusion detection systems (e.g., [13–15, 24, 25, 39]), EXTERIOR can be seamlessly integrated with them and provide a timely response to attacks, such as `kill` a rootkit created hidden process and `rmmod` a hidden malicious kernel module.

2.2 Challenges and Our Approach

Observation It is truly feasible to realize our out-of-VM guest-OS state introspection and update. The best example is to consider the operation we did for disk data inspection. Normally, a given OS usually has a mounted persistent disk storage; guest OS can use in-guest utilities to manage the disk data (e.g., `ls`, `rm`, `touch` a file). Meanwhile, we can also mount the disk into other OSes, and use the same utilities to manage these disks. From the state perspective, the disk data itself does not sense whether the management (e.g., inspection or editing) is performed in the guest OS or in other OSes, as long as we ensure the disk data remains in a consistent state.

For precision and clarity, we define a few formal notations for our following discussion. In general, a program $\mathcal{P}$ (including an OS kernel) is often composed of *code* $\mathcal{C}$ and *data* $\mathcal{D}$ (i.e., $\mathcal{P} = \mathcal{C}(\mathcal{D})$), where $\mathcal{C}$ determines how the software executes and $\mathcal{D}$ reflects the state of the execution. Normally, $\mathcal{C}$ and $\mathcal{D}$ reside in the same physical machine, and $\mathcal{D}$ gets updated when $\mathcal{C}$ gets executed.

We can further split $\mathcal{D}$ into user-level data $\mathcal{D}_{user}$ and kernel-level data $\mathcal{D}_{kernel}$, as illustrated in Fig. 1. At run-time, these data can be further classified into stack, heap, and global at both the user and kernel level. Thus, we denote $\mathcal{D}_{user} = \{\mathcal{D}^{stack}_{user}, \mathcal{D}^{heap}_{user}, \mathcal{D}^{global}_{user}\}$, and $\mathcal{D}_{kernel} = \{\mathcal{D}^{stack}_{kernel}, \mathcal{D}^{heap}_{kernel}, \mathcal{D}^{global}_{kernel}\}$, respectively. Also, note that $\mathcal{D}_{kernel}$ represents the run-time kernel state, is internally maintained by the kernel itself, and is indirectly accessed through OS system calls invoked by $\mathcal{P}$'s $\mathcal{C}$. Meanwhile, $\mathcal{D}_{kernel}$ can flow to $\mathcal{D}_{user}$ and get updated by $\mathcal{P}$'s $\mathcal{C}$. Therefore, we can have

$$\begin{aligned}\mathcal{P} &= \mathcal{C}(\mathcal{D}) \\ &= \mathcal{C}(\mathcal{D}_{user}, \mathcal{D}_{kernel}) \\ &= \mathcal{C}(\{\mathcal{D}^{stack}_{user}, \mathcal{D}^{heap}_{user}, \mathcal{D}^{global}_{user}\}, \{\mathcal{D}^{stack}_{kernel}, \mathcal{D}^{heap}_{kernel}, \mathcal{D}^{global}_{kernel}\})\end{aligned} \quad (1)$$

Meanwhile, a process always consumes data within its own physical machine, and has its own $\mathcal{D}^{stack}_{user}$, $\mathcal{D}^{heap}_{user}$ and $\mathcal{D}^{global}_{user}$. When a syscall is trapped from user space to kernel space, there will be a $\mathcal{D}^{stack}_{kernel}$ to maintain the kernel execution state for this particular syscall. During the execution of a syscall, many other kernel events can occur, such as interrupts, exceptions, and context switches, causing kernel control to jump to the execution of these events.

During the execution of one syscall, the other syscalls for this running process will not be executed until it returns (multi-thread process is discussed in §3.1). Also, there is an active kernel stack at kernel level for a running process (with limited size) that stores the return addresses of the called functions and the local variables when each syscall gets executed. When a syscall returns, the kernel stack for that specific process is empty, and the kernel stack frame is dynamically allocated from the kernel heap.

There exist a number of approaches that may be feasible to achieve our goal of timely out-of-VM kernel state introspection and update. In the following, we study these possible solutions and outline our approach.

Approach-I: Memory Diffing Given the dual-VM architecture, one intuitive approach is to take the snapshot of the GVM memory at state $\mathcal{M}_0$, and then resume the GVM execution with the state of $\mathcal{M}_1 = \mathcal{M}_0 + \mathcal{M}_\delta$. To acquire $\mathcal{M}_\delta$, we can run our SVM and take its memory snapshot before ($\mathcal{M}_{before}$) and after ($\mathcal{M}_{after}$) executing a configuration utility, and then we can get $\mathcal{M}_\delta = \mathcal{M}_{after} - \mathcal{M}_{before}$.

However, such an approach only works if and only if $\mathcal{M}_\delta$ is located in$\mathcal{D}^{global}_{kernel}$. If $\mathcal{M}_\delta$ includes any $\mathcal{D}^{heap}_{kernel}$ data, it will fail unless it also identifies heap data differences and resolves the mapping.Also, this intuitive approach might only work for kernel parameter reconfiguration (e.g., `hostname`,`chrt`,`sysctl`). If we want to kill a malicious process, we cannot use this memory diffing approach as our SVM does not contain the to-be-killed malicious process. We also cannot use this approach for state introspection (e.g., `ps`) as we do not know what will be the involved $\mathcal{M}_\delta$ in GVM.

```
 1 execve("/sbin/sysctl",["sysctl", "-w","kernel..=1"],...) = 0
 2 brk(0)                                  = 0x604000
 3 access("/etc/ld.so.nohwcap",F_OK)       = -1 ENOENT
 4 mmap(NULL, 8192, PROT_READ|.., -1,0) = 0x7f07b1749000
 5 access("/etc/ld.so.preload",R_OK)       = -1 ENOENT
 6 open("/etc/ld.so.cache", O_RDONLY)      = 3
 ...
47 open("/proc/sys/kernel/randomize_va_space",O_WRONLY|...) = 3
48 fstat(3, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
49 mmap(NULL, 4096, PROT_READ|.., -1, 0) = 0x7f07b1748000
50 write(3, "1\n", 2)                        = 2
51 close(3)                                 = 0
...
57 exit_group(0)                           = ?
```

Figure 2. Syscall trace of running `sysctl -w` to turn on the address space randomization in Linux kernel 2.6.32.

Approach-II: Process Implanting Because of the semantic-gap from the outside world, the second approach is to implant a configuration and recovery process into the guest OS, as demonstrated in Process Implanting [21]. However, the biggest problem for this approach is it has to pick up a victim process from the guest-OS and replace its program execution context with the implanted one. Thus, to the victim process, its semantics has been completely disrupted. Meanwhile, it will also execute the in-guest user-library and kernel code which may have been altered by malware. As such, this approach cannot guarantee the trustworthiness from the out-of-VM injected process except with other code integrity protection techniques from GVM.

Approach-III: Our Approach Inspired by our recent VM space traveler (VMST [14]) work, our approach is to use a *dual-VM* execution architecture with a kernel syscall *context aware* scheme that monitors the instruction execution of the trusted utilities at SVM, and transparently redirects each individual piece of memory update $\mathcal{M}_\delta$, at *binary code* instruction level from SVM to GVM when the syscall of interest gets executed, to achieve our state introspection, (re)configuration and recovery for GVM.

For instance, considering running `sysctl(8)` to configure the kernel parameters, as shown in Fig. 2, there are in total 57 syscalls, and only 4 of them (highlighted in the figure) are of our interest because these syscalls are responsible to tune the kernel parameters. If we can redirect the kernel data access of these four syscalls, we can achieve the same effect of configuring the kernel from outside VM.

More specifically, suppose we want to implement a new out-of-VM program $\mathcal{P}_{out}$, which could be a state inspection program (e.g., `ps`, `lsmod`, `netstat`) , a configuration or attack recovery program (e.g., `kill`, `rmmod`). We can reuse the execution context of the original in-VM program $\mathcal{P}_{in} = \mathcal{C}_{in}(\mathcal{D}_{user}, \mathcal{D}_{kernel})$ with the same $\mathcal{D}_{user}$, but with different $\mathcal{D}'_{kernel}$. However, we cannot reuse the $\mathcal{D}'^{stack}_{kernel}$ because the data in the stack is transient and mostly related to kernel control flow. Therefore, in order to implement $\mathcal{P}_{out}$, we can have

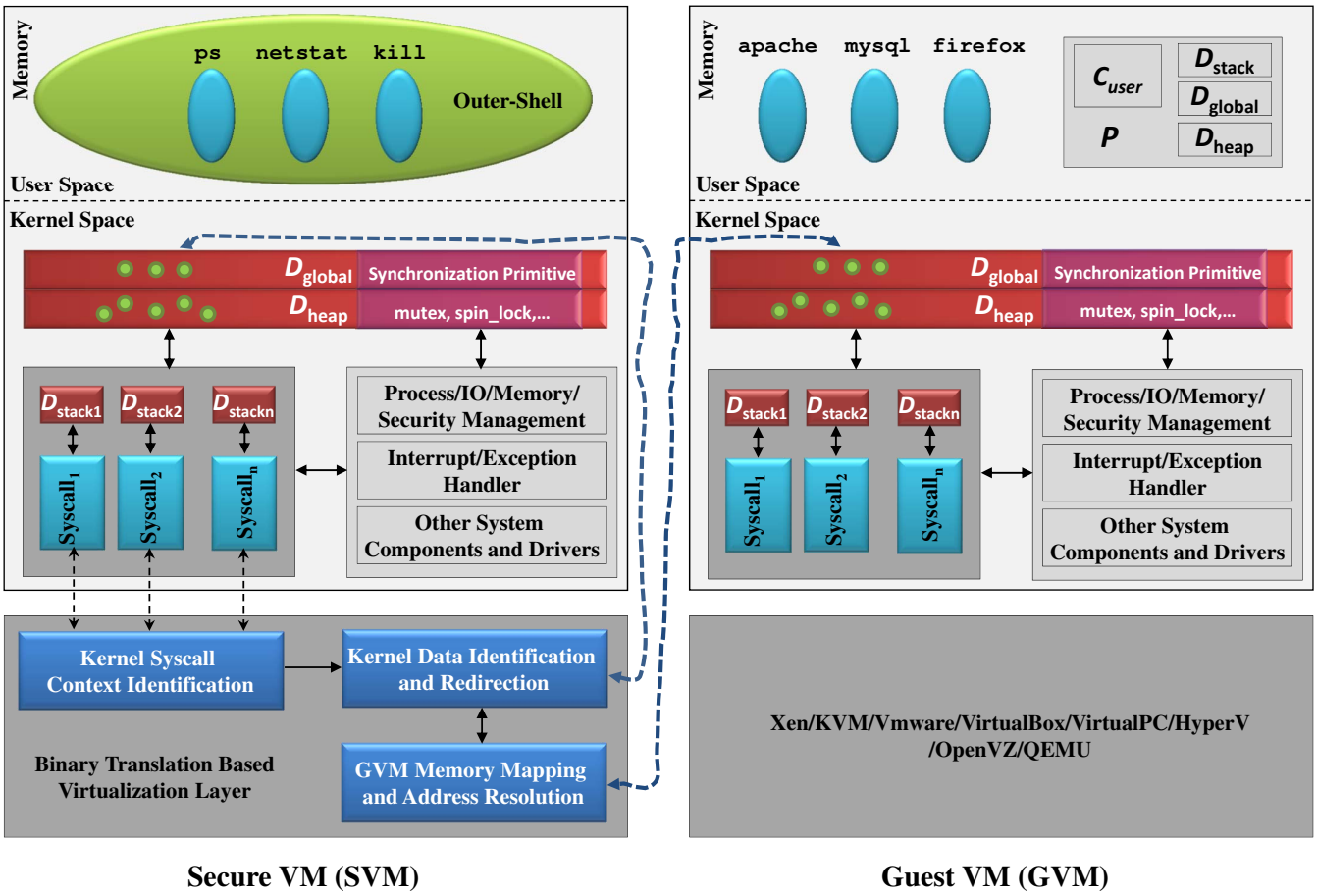

Figure 1. Overview of Our Dual-VM based EXTERIOR, which supports the execution of native administration utilities in the outer-shell of SVM to manage GVM. At a high level, suppose syscall_2 in SVM is of interest and executed in process `kill` context, then all the accessed $\mathcal{D}^{global}_{kernel}$ and $\mathcal{D}^{heap}_{kernel}$ (denoted as circle) under this process context will be redirected to GVM except those synchronization primitives such as `spin_lock` which often have unique instruction sequences.

$$\begin{aligned}\mathcal{P}_{out} &= \mathcal{C}_{out}(\mathcal{D}_{user}, \mathcal{D}_{kernel}) \\ &= \mathcal{C}_{in}(\mathcal{D}_{user}, \mathcal{D}'_{kernel}) \\ &= \mathcal{C}_{in}(\mathcal{D}_{user}, \{\mathcal{D}^{stack}_{kernel}, \mathcal{D}'^{heap}_{kernel}, \mathcal{D}'^{global}_{kernel}\})\end{aligned} \tag{2}$$

where $\mathcal{P}_{out}$ is the new out-of-VM program; $\mathcal{C}_{out} = \mathcal{C}_{in}$, $\mathcal{D}'^{heap}_{kernel}$ and $\mathcal{D}'^{global}_{kernel}$ are from the GVM; and $\mathcal{C}_{in}$, $\mathcal{D}_{user}$ and $\mathcal{D}^{stack}_{kernel}$ are from the SVM.

Interestingly, from formula (2) we can see that the semantic gap is automatically bridged for the out-of-VM program $\mathcal{P}_{out}$ that is running in our SVM. This is because the new $\mathcal{P}_{out}$ satisfies $\mathcal{C}_{out}(\mathcal{D}_{user}, \mathcal{D}_{kernel}) = \mathcal{C}_{in}(\mathcal{D}_{user}, \mathcal{D}'_{kernel})$ by reusing the legacy binary code $\mathcal{C}_{in}$ of $\mathcal{P}_{out}$. In other words, $\mathcal{P}_{out}$ can use all the syscall, APIs invoked by itself in SVM, and it transparently updates the state of $\mathcal{D}'_{kernel}$ of GVM and achieves the same effect of running the corresponding $\mathcal{P}_{in}$ in GVM, but with higher trustworthiness.

Challenges While EXTERIOR is inspired by our own VMST, to realize it there are still many new challenges including: (**C1**) How to precisely isolate the target process execution context in SVM, given that OS can run multiple processes and threads. While our VMST is able to identify the process level context, it does not have techniques to identify thread-level context, which is crucial for memory update in GVM. (**C2**) What those syscalls of our interest are. In VMST, we only identified introspection related syscall, and we have not extended it further to identify the configuration and recovery related syscalls. (**C3**) How to identify the corresponding $\mathcal{M}_\delta$ in $\mathcal{D}^{global}_{kernel}$ and $\mathcal{D}^{heap}_{kernel}$ while executing the program $\mathcal{P}$ in SVM. Note that in VMST we have developed techniques to solve this challenge, and EXTERIOR will directly leverage them. (**C4**) What would happen if there is any synchronization primitives (such as `mutex` and `spin_lock` in $\mathcal{D}'^{global}_{kernel}$ and $\mathcal{D}'^{heap}_{kernel}$. While VMST also faces this issue, it is less severe as many of these primitives are accessed in interrupt handlers. However, in EXTERIOR, we have to investigate this problem further as some memory write operations do involve synchronization primitives (e.g., when removing a kernel module, `delete_module` syscall needs to lock the `module list`. (**C5**) How to reflect the $\mathcal{M}_\delta$ from SVM to GVM, such that GVM feels that $\mathcal{P}$ is executed in its own VM. In VMST, we do not need to update the GVM memory as it is only used for *read-only* introspection.

2.3 Architecture Overview

An overview of EXTERIOR is presented in Fig. 1. Since we need to monitor the kernel instruction level memory access of $\mathcal{M}_\delta$, our SVM is based on the instruction translation based VMM. In our system design, we use the open source QEMU [1]. The GVM could be any VMM, such as Xen/KVM/vSphere/HyperV. Note that the

design of EXTERIOR is only bounded with SVM, which will only be used for security and administration.

There are three key components designed in our SVM at its binary translation based VMM layer. Specifically, to precisely isolate the target process execution context in kernel space (addressing **C1**), we devise a *Kernel Syscall Context Identification* (§3.1) component, which identifies the target process and thread execution context in the kernel space at the syscall granularity in SVM. During the execution of $\mathcal{P}$, not all the syscall related data is of interest to our $\mathcal{M}_\delta$ (as discussed in the above `sysctl` example), our *Kernel Process-Context Identification* will also pinpoint which syscall context needs the $\mathcal{D}_{kernel}^{global}$ and $\mathcal{D}_{kernel}^{heap}$ redirection (addressing **C2**). In addition, it will identify those interrupt execution context to filter the redirection of synchronization primitives (addressing **C4**). After that, our second component *Kernel Data Identification and Redirection* (§3.2) intercepts the data access of in guest $\mathcal{D}_{kernel}^{global}$ and $\mathcal{D}_{kernel}^{heap}$ (addressing **C3**), when the particular syscall of interest gets executed. In the meantime, it sends the GVM data read-and-write request to our third component *GVM Memory Mapping and Address Resolution* (§3.3). Our third component is responsible for mapping the physical memory of GVM, resolving the corresponding kernel virtual address, and performing the read and write operations of the involved $\mathcal{M}_\delta$ to GVM (addressing **C5**).

Scope and Assumptions As a proof-of-concept, we focus on the x86 architecture and Linux kernel. Currently, we do not support *arbitrary* state updates from the outside, and we *only* focus on the state update related to our memory introspection, configuration and recovery utilities we tested in §5. Disk data introspection and update is out-of-scope of EXTERIOR.

Also, we assume we know the specific kernel version running in the GVM, and we have a corresponding trusted kernel copy in SVM. The specific kernel version could be either reported by the VM administrator, or through guest-OS fingerprinting if there is any potential cheating (or the administrator cannot report). For example, recently proposed techniques such as CPU context-based approaches (e.g., [44]) or memory-only approaches (e.g., [20]) could be leveraged.

In addition, we assume our SVM is trusted, and we are able to maintain a clean state. Note that our SVM only consumes data from GVM, and will not execute any code from GVM (we can achieve this because we control each instruction execution in SVM). Finally, we assume there is no ASLR [4] inside the guest-OS kernel. That is, the virtual address of the kernel global variables should be always identical across different machines for the same version of the OS. In fact, this is true for many OSes including all of the available Linux kernels, and pre-Windows 7 kernels from Microsoft.

3. Detailed Design

In this section, we present the detailed design of the three key components of our SVM. We first describe how we identify the specific process execution context at VMM layer in §3.1; then describe how we intercept the kernel instruction execution, and identify the global as well as heap data in §3.2; finally we present how we map and resolve the data access (including both read and write) of the physical memory from SVM to GVM in §3.3.

3.1 Kernel Syscall Context Identification

The goal of our *Kernel Syscall Context Identification* is to identify the target-process kernel-level execution context, and pinpoint the exact syscall context at the VMM layer. We present how we achieve this in greater detail below.

Identifying Process Kernel Execution Context All the modern OSes running in the x86 architecture grant each process a private page directory that is often pointed by a control register CR3, and the value of the CR3 can hence naturally be used to differentiate the process execution context. This observation has been widely used in many of the VM introspection systems (e.g., [14, 25, 26, 39, 40]).

In addition to using CR3 to differentiate the process execution context, we have to further retrieve process name to pin-point our targeted process (such as `ps`,`kill`,`rmmod`). While we could traverse kernel data structures (e.g., `task_struct`) to retrieve such information, our approach is to inspect the system call arguments (e.g., the argument of `execve(2)`) of process creation to make our system more OS-agnostic.

However, there is still a caveat. A process could run with multiple threads. Using CR3 and process name can only pin-point the process execution context, and it cannot precisely isolate the specific syscall context yet. This is because all of the threads for the same process can execute syscalls. As such, we have to differentiate the thread context for the same process at VMM layer. But the Linux kernel does not have any thread specific support (to Linux kernel, a thread is uniformly treated as a process) and multi-threading is implemented at user level (e.g., `pthread` library which takes care of creating unique stack address for each thread). In fact, when using `pthread_create` to create a new thread, this function will use syscall `clone(2)` that has a user specified virtual address for child stack, instead of the default process `fork(2)`.

Fortunately, we have a new observation: while multi-threads for the same process share the same CR3 (threads share the same virtual address), each process at kernel level has a unique kernel stack (this is dynamically allocated) which can be used to isolate the thread execution context at kernel level. Therefore, we propose to use CR3, process name, and kernel `esp` register (with a lower 12bits cleared by mask) together, to uniquely differentiate and isolate the fine-grained thread execution context. To the best of our knowledge, none of the existing VMI techniques (e.g., [11, 13–15, 24–26, 35, 39, 40]) except vProbe [2] have reached this level of granularity. While vProbe is able to retrieve thread level information, it requires the access to kernel data structure information.

Identifying Specific Syscall Execution Context After having been able to identify the *fine-grained* process context, we need to further identify the specific syscall context under the target process execution. Note that syscalls are the exported OS services. As illustrated in Fig. 2, user level processes must invoke syscalls to request the OS services, such as file access.

Since our SVM monitors all the instructions executed inside the computer system, it is trivial to intercept the entry point and exit point of the syscall execution. Specifically, in the x86 architecture, syscall execution has unique instruction pairs. In the Linux kernel, they are `int 0x80/iret` and `sysenter/sysexit` (this pair is used since kernel-2.5). The specific syscall is indexed by register `eax` when invoking a syscall. Therefore, by monitoring these instructions, we can detect the entering (`int 0x80/sysenter`) and exiting (`iret/sysexit`) of a syscal.

Unfortunately, the kernel level execution between a syscall entry point and a syscall exit point is not entirely for the execution of this syscall as discussed in our VMST [14]. Besides the normal control flow such as `call/ret/jump`, as illustrated in Fig. 1, kernel control flow is also driven by the asynchronous event: interrupt (e.g., context switch timer) and exception (e.g., page fault). These events will be responsible for managing the system resources and executing device drivers. Certainly, we have to precisely identify these execution contexts and exclude their data access of $\mathcal{D}_{kernel}^{global}$ and $\mathcal{D}_{kernel}^{heap}$ (because our analysis with kernel source code reveals that many of the `spin_locks` and `mutexes` are accessed in these context). Otherwise, when reading these data from GVM, most likely our SVM kernel will lead to an inconsistent state (such as

dead lock) and even crash during the execution of these execution contexts. For instance, if the page fault handler of SVM is about to allocate new pages for a process, but if it reads a different state from GVM, it will likely render the page fault handler unusable.

Whereas the kernel has such a very complicated, unpredictable control flow, we can precisely identify the syscall execution context. All of these asynchronous events are driven by interrupts and exceptions, and our SVM emulates all these hardware level resources. As such, we are able to identify the beginning execution of these events because the SVM controls the hardware. For the end of these events, they will have an `iret` instruction, which we can also capture precisely. Meanwhile, for the bottom up handlers of an interrupt and exception, they will be executed during context switch. Our SVM controls the interrupt and timer, and we hence control the context switch. Therefore, our SVM is able to precisely identify the syscall execution of the target process, and keep it running successfully in SVM. Note that this syscall context identification approach has been proposed in our VMST [14], and we directly leverage this technique when building our EXTERIOR.

Eventually, the output of our first component will precisely tell the exact execution context of the syscalls, excluding any other kernel execution such as context switch and interrupt (and exception) handler. Next, our second component will perform the identification of $\mathcal{D}^{global}_{kernel}$ and $\mathcal{D}^{heap}_{kernel}$ accessed during the syscall execution of our interest (recall only 4 out of 57 system calls are of our interest when executing `sysctl(8)`).

3.2 Kernel Data Identification and Redirection

The goal of this component is to intercept all the data access, pinpoint the precise $\mathcal{D}^{global}_{kernel}$ and $\mathcal{D}^{heap}_{kernel}$, and read data from or write data to the memory in GVM with the facilitation of our third component, while executing the monitored syscall of our interest.

Identifying $\mathcal{D}^{global}_{kernel}$ and $\mathcal{D}^{heap}_{kernel}$ during a Syscall Execution Similar to user level stack data, $\mathcal{D}^{stack}_{kernel}$ is also transient. While $\mathcal{D}^{stack}_{kernel}$ does contain some localized state variables, it actually does not contribute to the state of kernel introspection, configuration, and recovery. $\mathcal{D}^{global}_{kernel}$ and $\mathcal{D}^{heap}_{kernel}$ are the memory regions that store the persistent kernel state. Therefore, our primary focus is to precisely identify these $\mathcal{D}^{global}_{kernel}$ and $\mathcal{D}^{heap}_{kernel}$ when the syscall of our interest gets executed.

After a kernel is compiled, the addresses of $\mathcal{D}^{global}_{kernel}$ become literal values in kernel instructions. As such, it is trivial to identify $\mathcal{D}^{global}_{kernel}$ by simply looking at the address ranges of the literal values. Then the real challenge comes from how to identify $\mathcal{D}^{heap}_{kernel}$. Fortunately, based on the observation that it will have the same effect as precisely identifying all the $\mathcal{D}^{stack}_{kernel}$ and excluding them, since a kernel data x either belongs to $\mathcal{D}^{stack}_{kernel}$, $\mathcal{D}^{global}_{kernel}$, or $\mathcal{D}^{heap}_{kernel}$.

It may seem to be trivial to identify the $\mathcal{D}^{stack}_{kernel}$ as we monitor all the instruction, we can check whether $x \in \mathcal{D}^{stack}_{kernel}$ by looking at the address range. However, it turns out that such an approach will not work considering the fact that $\mathcal{D}^{stack}_{kernel}$ is also dynamically allocated from $\mathcal{D}^{heap}_{kernel}$. On the other hand, $\mathcal{D}^{stack}_{kernel}$ often has data dependencies with the kernel stack pointer (`esp`). Therefore, we leveraged a stack data dependence tracking algorithm from our VMST [14] to track the data directly and indirectly derived from kernel stack pointer `esp`. This algorithm is a variant of standard taint analysis, which has been widely investigated and used in many applications (e.g., [9, 32, 37]). In our scenario, any data derived from stack pointer `esp` as well as their propagations will be tainted by instrumenting data movement and data arithmetic instructions. Then for a given kernel address x, if its taint bit is set, then it belongs to $\mathcal{D}^{stack}_{kernel}$; otherwise, it is $\mathcal{D}^{global}_{kernel}$, or $\mathcal{D}^{heap}_{kernel}$.

```
<spin_lock> in 2.6.34
0xc0129950: 55                push ebp
0xc0129951: ba 00 01 00 00    mov edx, 0x100
0xc0129956: 89 e5             mov ebp, esp
0xc0129958: 3e 66 0f c1 10    xadd word ptr ds[eax], dx
0xc012995d: 38 f2             cmp dl, dh
0xc012995f: 74 06             jz 0xc0129967
0xc0129961: f3 90             pause
0xc0129963: 8a 10             mov dl, byte ptr ds[eax]
0xc0129965: eb f6             jmp 0xc012995d
0xc0129967: 5d                pop ebp
0xc0129968: c3                ret

<spin_lock> in 3.0.4
0xc1026a70: 55                push  ebp
0xc1026a71: ba 00 01 00 00    mov edx, 0x100
0xc1026a76: 89 e5             mov ebp, esp
0xc1026a78: 3e 66 0f c1 10    xadd word ptr ds[eax], dx
0xc1026a7d: 38 f2             cmp  dl, dh
0xc1026a7f: 74 06             jz 0xc1026a87
0xc1026a81: f3 90             pause
0xc1026a83: 8a 10             mov dl, byte ptr ds[eax]
0xc1026a85: eb f6             jmp 0xc1026a7d
0xc1026a87: 5d                pop  ebp
0xc1026a88: c3                ret
```

Figure 3. Disassembled Instruction Sequence of `spin_lock` Primitive in different Linux Kernels.

Enumerating Syscall of Interest Recall as illustrated in Fig. 2, not all the syscall contributes to the kernel state inspection and update, and we have to systematically enumerate the syscalls of our interest. As discussed in VMST [14], this enumeration is often application-specific and has to be done by kernel experts, not the end users of our EXTERIOR. In particular, after manually examining all the syscalls, we classify those of our interest into the following three categories:

(1) **Inspection** In order to reconfigure the OS or recovery from an attack, we have to introspect the OS to get its current status and perform the response. Many user level utilities such as `ps(1)`, `lsmod(8)`, `lsof(8)`, `netstat(8)` are designed for this inspection purpose. Interestingly, all these utilities read `proc` files to inspect the kernel state. Therefore, file access related syscalls: `open(2)`, `read(2)`, `fstat(2)`, `stat(2)`, `lseek(2)`, `readv(2)`, `readdir(2)`, `close(2)` are of our particular interest. Note that Linux kernel leverages `proc` files to enable user-level program accessing kernel state.

(2) **Configuration** Similar to the inspection, many configuration utilities such as `sysctl(8)` use `write(2)` to change the kernel state through `proc` file system. Therefore, `write(2)` is of interest. In addition, there is also a `sysctl(2)` syscall for kernel to directly change its parameters. Meanwhile, we are interested in other syscalls such as `socket(2)`, `ioctl(2)` (for `route(8)`) and `nice(2)` because they can also dynamically change the kernel state.

(3) **Recovery** Once we have detected a kernel attack such as a hidden malicious process or a hidden device driver in GVM (using the inspection utility in SVM to introspect GVM for instance), we need to quickly remove them from the guest kernel. Therefore, syscalls `kill(2)` and `delete_module(2)` are also of our interest.

Identifying Synchronization Primitives in Syscalls While many synchronization primitives are executed in the interrupt context, we also find some syscalls do contain them. For instance, `delete_module(2)` will call `spin_lock`, `spin_unlock`, two functions widely used in kernel synchronization, to lock and unlock the `modlist_lock` that is a kernel global variable. As such, we have to filter the data redirection of `modlist_lock`. It might be viable by white-listing the program counters (PCs) of the

involved instructions. However, this is tedious, challenging, and also kernel-specific (we have to perform such analysis for each kernel to filter these PCs).

After analyzing the instruction sequences of these synchronization primitives, we realize that we can still have a systematic solution to identify their execution contexts by looking for the particular instruction sequences of the synchronization primitives. Specifically, as illustrated in Fig. 3, when executing a function prologue in SVM (say `push ebp` or even at `xadd` instruction in Fig. 3), we forward scan these instruction sequences (the scanning window is determined by each specific primitive), and if they fall into the sequences of kernel synchronization primitives such as `spin_lock` (that has 25 bytes with byte sequence `55 ba 00 .. f6 5d c3`) and `spin_unlock`, or `__up` and `__down` (for a semaphore), we will filter the data redirection for these primitive functions. Our experiment with a number of Linux kernels confirms these instruction sequence patterns are also stable across different kernels.

3.3 Mapping the GVM Memory Address

Having identified a given kernel address x in the syscall of our interest belongs to $\mathcal{D}^{global}_{kernel}$, or $\mathcal{D}^{heap}_{kernel}$, we will dynamically instrument the executing instruction to make it fetch the data from and write the data to the physical memory (PM) of GVM. This is achieved by our third component *GVM Memory Mapping and Address Resolution*. Also, this component makes EXTERIOR substantially different compared to VMST [14] which has just read-only capability of guest-OS.

3.3.1 GVM Memory Mapping

We devise two approaches to map the PM of GVM to our SVM. One is the *online mapping* and we directly map the pages that belong to GVM to our SVM with the support from the VMM (i.e., hypervisor) . The other is the *offline mapping* and we directly take the memory snapshot of GVM and attach it to SVM; once we finish the update in SVM, we restore the updated memory to GVM. Since the second approach is very simple, we focus on the first approach in our following design.

As our SVM uses binary code translation based virtualization (or emulation), it has to run in a host OS. Depending on whether the underneath hypervisor of GVM is hardware-based or software-based, we have two different strategies.

Mapping Software Virtualization Based GVM When a GVM uses software virtualization (such as QEMU), we also have two situations. One is if the GVM also resides in the host OS with our SVM, then to our SVM, the GVM is just another process and we can use inter-process communication between the two VMMs to share the physical memory of GVM. The other is that the GVM resides in a different host OS, and we have to transfer the memory snapshot of GVM to the SVM, or just the references and updates to save the network bandwidth. For both situations, we develop a host or network stub in the VMMs of our SVM and GVM for the communication.

Mapping Hardware Virtualization Based GVM A GVM could also run on top of hardware virtualization such as Xen. In this case, the hypervisor underneath is able to identify the page frames which belong to the GVM. Also, if the host VM of the SVM running in the same hypervisor with GVM, then the hypervisor is able to map the memory of GVM to the SVM. Otherwise, we transfer the memory images of GVM to SVM through network communications.

3.3.2 GVM Memory Address Resolution

After having performed the mapping of the GVM physical memory (G-PM) to our SVM, the G-PM is just another piece of added

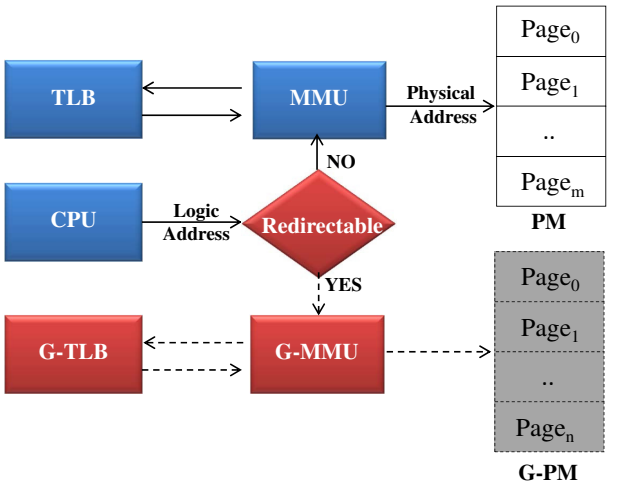

Figure 4. The VMM Extension of our SVM to Map and Resolve the Physical Memory Address of GVM.

physical memory (PM) no matter whether the GVM is software or hardware virtualization based. Note that PM and G-PM could have different sizes since they are in two different machines. Next, we have to instrument the hypervisor of our SVM to transparently access it.

More specifically, as illustrated in Fig. 4, CPU in x86 operates with virtual address (i.e., logic address), and MMU (a hardware component) together with a TLB is responsible for translating the virtual address to physical address (V2P). The TLB is used as a cache to avoid the expensive page table lookup while performing the V2P.

For a given redirectable kernel address x, we could just traverse the page tables to perform its V2P. However, this will be very expensive as each time there will be three memory references. Therefore, adopted from VMST [14], we extend the software-translation based VMM with a G-MMU (GVM's MMU) and G-TLB (GVM's TLB) component, which performs V2P translation in G-PM instead of the original PM, as shown in Fig. 4.

Also, while performing the V2P for a redirectable kernel address x, we need the address of the page directory (PGD) of GVM. In x86 architecture, the PGD is stored in the control register CR3. Therefore, we will retrieve the value of CR3 from GVM when we perform the mapping.

The GVM Status during the SVM Updating When SVM is updating the memory of GVM, there could be some concurrent issues if we keep GVM running as well. Therefore, during the update, our current design is to pause the GVM execution and resume it once the update finishes. Although the proper execution free of concurrency introduced by EXTERIOR is not guaranteed, we have tested with our utilities and confirmed that we can also run them without pausing the GVM.

4. Implementation

We have implemented a prototype of EXTERIOR to support the out-of-VM execution of trusted utilities. In the following, we share the implementation details of interest. As there are two VMs (SVM and GVM) in our dual-VM architecture, we present how we instrument and modify them below. All of our implementation is at the VMM layer, and we will not introduce any components to, or modify, the guest OS.

Category	Utility	Description	Effective? Syntactics	Effective? Semantics
Introspection	ps (1)	report a snapshot of the current processes	✗	✓
	pstree (1)	display a tree of processes	✗	✓
	lsmod (8)	show the status of modules in the Linux Kernel	✓	✓
	dmesg (1)	print or control the kernel ring buffer	✓	✓
	vmstat (8)	Report virtual memory statistics	✗	✓
	netstat (8)	show network connections, routing tables, interface statistics, etc.	✓	✓
	lsof (8)	list open files	✗	✓
	uptime (1)	tell how long the system has been running	✗	✓
	df (1)	report file system disk space usage	✗	✓
Configuration	sysctl (8)	configure kernel parameters at runtime	✓	✓
	route (8)	show / manipulate the IP routing table	✓	✓
	hostname (1)	show or set the system's host name	✓	✓
	chrt (1)	manipulate real-time attributes of a process	✓	✓
	renice (1)	alter priority of running processes	✓	✓
Recovery	kill (1)	send a signal to a process	✓	✓
	rmmod (8)	simple program to remove a module from the Linux Kernel	✓	✓

Table 1. Effectiveness evaluation of our EXTERIOR. The syntactics and semantics is compared with the result of running the corresponding utility inside and outside of the GVM.

SVM Our SVM is built on top of a most recent QEMU-1.0 [1], which is an emulation-based VMM. There are three key components designed in §3 to be added to QEMU. The code size of our own implementation is about 9,300 LOC.

In particular, for the first component, we instrumented the interrupt/exception handler (`do_interrupt_all` function) and `sysenter/sysexit` instructions in QEMU to identify the exact syscall execution context. Our second component *Kernel Data Identification and Redirection* needs to perform run-time taint analysis. To this end, we instrument the instruction translation of QEMU (`disas_insn`) such that we can intercept each instruction, and based on the instruction semantics (e.g., data movement or data arithmetic), we perform data dependence tracking. For our third component, we extended the `i386-softmmu` component in QEMU, and leveraged the original TLB and MMU translation code to implement our G-TLB and G-MMU. Also, for simplicity, we implemented a snapshot-based stub to map and update the GVM memory. That is, we will map the physical memory which is a snapshot of the GVM.

GVM There are a number of hardware virtualization or software virtualization based GVM. For proof-of-concept and simplicity, we took two typical VMs: a KVM based, and a QEMU based. Both VMs already support pausing the execution of the machine, taking the snapshot of the physical memory, and resuming the machine with a new snapshot. As such, we just extended KVM and QEMU with a CR3 value collection component (with less than 100 LOC).

5. Evaluation

This section presents the experimental result. We first evaluate what kind of introspection, (re)configuration and recovery EXTERIOR can do, and how much user input it requires in §5.1, then we evaluate the performance costs of EXTERIOR in §5.2, and finally we evaluate the generality of our approach regarding different guest-OS kernels in §5.3. Our host environment runs Ubuntu 11.04 with Linux kernel 3.1.0, on Intel Core i7 CPU with 8G memory.

5.1 Effectiveness

Our goal is to support the out-of-VM execution of a given utility, to achieve the same effect of running it in-VM. To this end, as presented in Table 1, we took 16 commonly used administration utilities as the benchmarks and classified them into three categories: introspection, (re)configuration, and recovery. Our experiment is to verify whether we are able to run these utilities in our external shell and achieve the same effect while running these software inside the VM. The testing guest OS is debian 6.0 with Linux kernel 2.6.32 and 512M memory, and we tested our GVM with both KVM and QEMU.

5.1.1 Introspection

Native inspection utility allows users to examine the state of an OS, which includes the running processes and their relationship (`ps,pstree`), the opened files by the running processes (`lsof`), the network connections (`netstat`), the list of the device drivers (i.e., kernel modules) running in the system (`lsmod`), and how long the system has been running (`uptime`), etc.

Interestingly, these *inspection* utilities automatically become *introspections* for GVM kernel when running them in our SVM shell. As shown in the 4^{th}-column of Table 1, some of them do not have the equivalent syntactic output with the result running inside GVM. More specifically, there will be one running process less while testing `ps,pstree` in SVM to introspect the state of GVM. This missing process is `ps,pstree` itself when we executing them inside GVM. Because of the different timings of the snapshots (we took the snapshot after we run `ps,pstree`), it is not in the snapshot. Similarly while running `lsof` to show all the opened files for all the processes, there are some syntactic differences as well because of this missing running `lsof` process. For the syntactics difference of `uptime,vmstat`, it is due to the time differences of executing them in SVM and GVM. For all other commands (i.e., `lsmod,dmesg,netstat`), we get equivalent syntactics results. For the semantics (i.e., the meanings), these commands achieve the same effect respect to the inspections as shown in the 5^{th}-column of Table 1.

5.1.2 Configuration

We also tested seven configuration utilities to reconfigure kernel parameters (`sysctl`), routing table (`route`), host name (`hostname`), and process priority (`chrt,renice`). All of these five utilities successfully reconfigured the kernel state, and achieve the same effect while executing them inside.

5.1.3 Recovery

Removing the malicious processes/modules The OS kernel is often contaminated by kernel rootkit that hides malicious processes or kernel modules. When a hidden process is detected, an immediate step would be to remove the malicious process (`kill`). Similarly, when a malicious module is detected, we also need to immediately remove it from the kernel (`rmmod`). In the extreme case, we may have to reboot the system to stop the attack. We also evaluated these capabilities, and our experiments show that EXTERIOR directly supports run these utilities out-of-VM, and achieves the same effect of running in-VM. Note that for the `reboot`, since at the VMM layer we control everything, we can directly reset GVM without executing the `reboot` inside.

Recovering the contaminated function pointers However, it is just a first step to `kill` the malicious process or `rmmod` the malicious modules for the attack recovery. We also need to recover the kernel function pointers that have been modified by kernel rootkits. To this end, we developed a static memory mapping-based, kernel function pointer recovery tool (MAKEUP), which runs in the VMM of our SVM to recover the rootkit attacks (by updating the G-PM).

Most kernel rootkits (over 96% [43]) hijack the runtime OS kernel control flow by modifying such as the syscall table and other kernel function pointers. These syscall tables and other global kernel function pointers are usually static, having the same virtual addresses and values across different machines for the same kernel. Therefore, we can directly compare the values of these static function pointers guided by the `system.map` in the physical memory of SVM which is trusted, with the memory from GVM, to identify which kernel function pointer has been contaminated, and then recover it from the trusted value which is present in our SVM.

In particular, if the value of a function pointer does not match in PM and G-PM, and if this function pointer is not pointed to kernel modules (usually in kernel heap) in our trusted kernel, then our MAKEUP tool will recover this contaminated function pointer. The reason why we cannot compare the pointers which are pointing to heap is because kernel modules usually have different dynamic addresses across different machines including the SVM and GVM.

We took 12 open source rootkits from from *packetstormsecurity.org* and tested them with our MAKEUP tool. By verifying with source code, we can hence make sure we indeed identified all of the contaminated function pointers. Surprisingly, as presented in Table 2, MAKEUP performs incredibly well and it can recover all of the contaminated kernel pointers for 11 rootkits, leaving only one of the rootkits, adore-2.6, unrecovered. This is because adore-2.6 modifies the function pointers in both kernel global and kernel heap. However, MAKEUP cannot identify the kernel heap pointers statically. All other rootkits which modify either system call table, interrupt descriptor table (IDT), or some global kernel function pointer (e.g., `tcp4_seq_show`), MAKEUP successfully recovered the kernel from these attacks by comparing the static pointers in SVM and GVM.

5.1.4 Automation

As we have demonstrated in §5.1.2 and §5.1.3 that we can use EXTERIOR to perform the guest-OS introspection, reconfiguration and recovery, but we have not evaluated how much user input it requires. In fact, much like all the utilities can be automated using script in-VM, all of our tested utilities can also be automated (without any user input) out-of-VM.

To demonstrate this feature, we developed a component with scripts to enable our SVM to periodically introspect the GVM (using commands `ps` and `lsmod`), and then perform a cross-view comparison with the result from a signature scanning approach [12, 34] (in this way we do not have to install anything inside GVM): if there is a hidden process or device driver, we `kill` or `rmmod` it and at the same time using our MAKEUP tool to recover from the attack. We tested this component with both a process hidden rootkit and a malicious driver, our EXTERIOR quickly identified the hidden object and removed it from the GVM kernel. Depending on the time window we set for the periodically checking, the quickest one takes less than a second.

Rootkit	Targeted Function Pointer	Succeed?
adore-2.6	kernel global, heap object	✗
hookswrite	IDT table	✓
int3backdoor	IDT table	✓
kbdv3	syscall table	✓
kbeast-v1	syscall table, tcp4_seq_show	✓
mood-nt-2.3	syscall table	✓
override	syscall table	✓
phalanx-b6	syscall table, tcp4_seq_show	✓
rkit-1.01	syscall table	✓
rial	syscall table	✓
suckit-2	IDT table	✓
synapsys-0.4	syscall table	✓

Table 2. Rootkit Recovery with Our MAKEUP Tool.

Tested-Item	Original-Qemu	EXTERIOR	Slowdown
Context switch (ms)	18.3	18.8	2.7%
Mmap (ms)	23300	23600	1.3%
Sh proc (ms)	10000	10000	0.0%
10k file create (ms)	372.7	380.4	2.0%
Mem read (MB/s)	1629	1621	0.4%
Bcopy (libc) (MB/s)	901	883	2.0%

Table 3. Overhead of micro-benchmarks

Finally we emphasize again that to run the introspection, reconfiguration and recovery commands, there is no requirement of the `root` privilege (as well as the guest-OS root password) from the GVM. This is one of the unique benefits of our system, which can enable a timely response for attacks. One may wonder whether our SVM has broken the administrator trustworthiness of the GVM. However, we have to note that to all the in-VM programs including OS kernel, they often trust their underlying VMM. Our SVM is located in its VMM, and it should therefore be trusted.

5.2 Performance Costs

Next, we measure the performance overhead of EXTERIOR. From the system design in §3, we can notice that the major system overhead of EXTERIOR is only at SVM side, and the overhead comes from the instrumentation of our syscall context identification, data dependence tracking, and GVM mapping and update. In the following, we use two sets of benchmarks to evaluate the performance overhead introduced by our instrumentation.

Micro-benchmarks To evaluate the OS-primitive level performance slowdown of our SVM, we use the standard micro-benchmark LMBench suites to estimate our instrumentation impacts on various aspects of OS operations in SVM.

Interestingly, as shown in Table 3, we could see there is little impact on the regular kernel primitives, such as the latency of context switch and `mmap` syscall, the time spent to execute the C library function system (the `sh proc` row), the latency to create a 10KB file, the memory read bandwidth, and the time to copy 1MB data using the C library function. The explanation is that all of our instrumentation such as process context identification, data

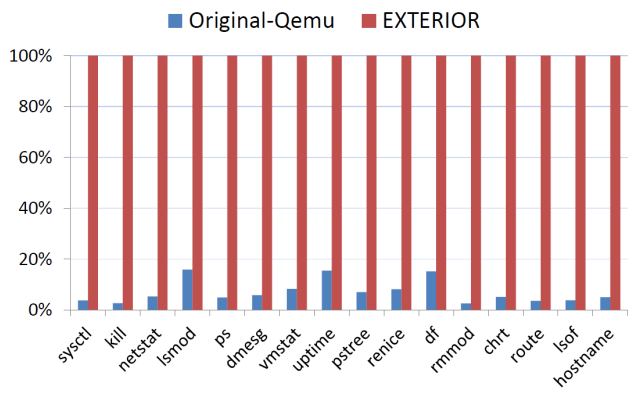

Figure 5. Overhead of macro-benchmarks.

dependence tracking and memory redirection is only active for some particular process in its particular syscall of interest. The LM-Bench does not contain any syscall context of our interest (introspection, configuration, and recovery), and these little performance slowdown actually comes from the check of the need of process context identification, the data dependence tracking, and memory redirection.

Macro Benchmarks We use the same set of effectiveness benchmarks in §5.1 to quantify the performance slowdown of our instrumentation. More specifically, we measure the performance of these 16 utilities in Table 1 by running each of them 100 times, and calculate the running time with Linux `time` command and report the average time.

The experimental result is reported in Fig. 5. We could see that our instrumentation has introduced 23X overhead on average compared with the programs running inside the OS. The major overhead comes from our dynamic dependence tracking in which we have to instrument all the data movement, data arithmetic instructions when running at kernel mode for a particular syscall of interest. Therefore, if a utility is syscall intensive, it tends to have larger overhead. For instance, there is a 37.5X overhead when running `kill`. Note that `kill` is interesting as it has to traverse kernel running list to find the target process (data redirection intensive) and then modify the `pending` field in the `task_struct`. While we have such large overhead, we have to emphasize that the absolute running time is small, which is only 1.83 seconds on average. Also, our SVM will not be used for any production runs and solely for EXTERIOR, such as serving as a web server.

5.3 Generality

While we have evaluated EXTERIOR with 16 native, trusted utilities to manage the guest-OS out-of-VM, we have not answered the question of how general our techniques are with respect to different kernels. To this end, we took 22 Linux distributions from four major vendors, and installed them on top of our SVM and GVM.

From our technical design, we have already noticed that EXTERIOR does not require both kernels to have the same configuration. To validate this, we explicitly configure each GVM with different physical memory size, and different kernel modules. Then we execute these 16 utilities to manage the GVM from SVM. As described in Table 4, our SVM directly supports running these kernels without any modification (transparent). Our explanation is static computer configurations are often stored in $\mathcal{D}^{global}_{kernel}$ and dynamic kernel modules are often stored in $\mathcal{D}^{heap}_{kernel}$. Our SVM just creates the skeleton execution context with the necessary binary as well as kernel code to execute the administration utilities of our interest, and these redirected $\mathcal{D}^{global}_{kernel}$ and $\mathcal{D}^{heap}_{kernel}$ from GVM will be reflected in our SVM, and thus they can fulfill our introspection, configuration, and recovery tasks.

Linux Distribution	Kernel Version	Release Date	Transparent?
Debian 4.0	2.6.26	2007-04-06	✓
Debian 5.0	2.6.28	2009-02-12	✓
Debian 6.0	2.6.32	2010-01-22	✓
Fedora-8	2.6.23	2007-11-08	✓
Fedora-10	2.6.27	2008-11-25	✓
Fedora-12	2.6.31	2009-11-17	✓
Fedora-14	2.6.35	2010-11-02	✓
Fedora-16	3.1.0	2011-11-08	✓
OpenSUSE-10.3	2.6.22	2007-10-04	✓
OpenSUSE-11.0	2.6.25	2008-06-19	✓
OpenSUSE-11.1	2.6.27	2008-12-18	✓
OpenSUSE-11.2	2.6.31	2009-11-12	✓
OpenSUSE-11.3	2.6.34	2010-07-15	✓
OpenSUSE-12.1	3.1.0	2011-11-16	✓
Ubuntu-8.04	2.6.24	2008-04-24	✓
Ubuntu-8.10	2.6.27	2008-10-30	✓
Ubuntu-9.04	2.6.28	2009-04-23	✓
Ubuntu-9.10	2.6.31	2009-10-29	✓
Ubuntu-10.04	2.6.32	2010-04-29	✓
Ubuntu-10.10	2.6.35	2010-10-10	✓
Ubuntu-11.04	2.6.38	2011-04-28	✓
Ubuntu-11.10	3.0.4	2011-10-13	✓

Table 4. OS-Agnostic Testing of EXTERIOR.

6. Limitations and Future Work

As we have demonstrated in §5, our EXTERIOR can be used for out-of-VM guest-OS introspection, (re)configuration, and recovery. In this section, we describe what EXTERIOR cannot do as well as our future work.

6.1 Limitations

From the technical point of view, for our EXTERIOR to succeed, we must require (1) the two kernels to have identical $\mathcal{D}^{global}_{kernel}$, (2) the identical instructions of the monitored syscall, (3) the monitored syscall will not be blocked, and (4) the monitored syscall only operates on memory data. Therefore, we cannot run arbitrary administration utilities and support arbitrary kernels.

More specifically, if two kernels have different $\mathcal{D}^{global}_{kernel}$, when we redirect the data access from SVM to GVM we will read different data. Consequently, it will not only impact $\mathcal{D}^{global}_{kernel}$, but also impact $\mathcal{D}^{heap}_{kernel}$ as $\mathcal{D}^{heap}_{kernel}$ is often dereferenced from $\mathcal{D}^{global}_{kernel}$. Therefore, if there is any address space randomization (e.g., [4, 49]) or data structure layout randomization [33] for $\mathcal{D}^{global}_{kernel}$, our current EXTERIOR cannot support this.

Meanwhile, if the monitored syscall has different instructions such as calling some additional functions in GVM kernel modules to fulfill certain tasks, but our SVM does not contain these modules, certainly EXTERIOR cannot support such syscalls. In addition, if the monitored syscall can be blocked, EXTERIOR will fail as well, because our SVM disables the context switch during the execution of the syscall of interest, and the monitored process cannot be woken up. Fortunately, all the 16 utilities we tested do not involve the execution of the syscall context to these execution states.

Finally, if the monitored syscall accesses disk data, EXTERIOR will fail because we do not transfer disk image from GVM to SVM.

Therefore, currently we do not support running any disk related utilities (e.g., `ls`,`fsck`). Also, the GVM kernel state should not be swapped to disk. For the Linux kernels, we confirmed from kernel source code that Linux kernel state will not be swapped.

6.2 Future Work

Dealing with Kernel ASLR EXTERIOR assumes there is no ASLR for kernel global variables. In fact, this is true for many OSes including all the available Linux kernels, and pre-Windows 7 kernels from Microsoft. Fortunately, we have an observation that for Windows 7 the randomization of kernel global variables is through a sliding window. That is, the relative offset between two global variables are not randomized. Thus, as long as we can align one of the kernel global variables, we shall be able to align all others. One of our future efforts will address this issue and enable EXTERIOR to support Microsoft Windows systems including kernel ASLR-enabled Windows 7.

Introducing Primitives for Synchronization EXTERIOR only supports the kernel state introspection, (re)configuration and recovery utilities, and our simple design is to *pause* the GVM execution when we update its kernel state. Therefore, we would like to transparently introduce lock primitives at the VMM layer to the kernel such that we can keep executing the GVM without pausing its execution.

In fact, we believe this is an interesting research problem, as essentially we are working on *a new system architecture that has multiple-CPUs where each has its own OS with the same version with others and they share the same kernel global and heap data*. For security reasons, the OS code and private kernel stack data for each CPU are isolated. We are realizing this architecture using a dual-VM approach. One of our future efforts will investigate other alternative approaches.

Handling Other Attacks Obviously, current EXTERIOR cannot recover the direct kernel object manipulation (DKOM) attack, and our MAKEUP tool for kernel pointer recovery does not check any pointers in kernel heap. Thus, part of our future work will investigate how to fix the kernel heap pointers.

Finally, our design assumes there is no virtualization based attacks to target either our SVM or GVM (such as Bluepill [45], SubVert [29], or DKSM [3]). Defending these virtualization based attacks is another venue of our future work.

7. Related Work

Virtual machine introspection (VMI) Our EXTERIOR is closely related to the VMI [15] in the sense that we all monitor the guest OS state at the outside VMM layer. Some of the VMI techniques only perform purely monitoring (i.e., *read-only*), such as Liveware [15], Antfarm [25], XenAccess [39], VMwatcher [24], Ether [11], Virtuoso [13], and VMST [14]. Some of them will take active actions once an attack is detected. For instance, IntroVirt [27] will execute the vulnerability-related predicates at VMM level to detect and respond to an intrusion, Lycosid [26] will patch the executable code to disable the malicious code execution once a hidden process is detected, and Manitou [35] will make a corrupted instruction page non-executable when the instruction page mismatches are detected.

Compared to all of these VMI techniques including the recent VMST [14], the key difference is: EXTERIOR for the first time enables an out-of-VM shell with a guest-OS writable and executable capability. Moreover, the writable is at the fine-grained memory cell level, and guided by the intrinsic constraints of the kernel code itself.

Intrusion Recovery To recover from attacks, a common approach is through monitoring and logging the behavior of intrusions to replay and restore the infected data. Such an approach has been recently instantiated to recovery from host intrusion (e.g., [23, 28]) or web applications (e.g, [6]), and repair malware damages (e.g., [38]). Unlike these techniques that must perform the logging and replay, EXTERIOR introduces a new technique to remediate the intrusions out-of-VM.

In general, EXTERIOR is also related to the system failure recovering [5], and system self-healing [36, 46], in that we all allow the system to keep running and we all are able to recover from attacks. The difference is EXTERIOR complements all of these existing techniques and exploits new dimensions from an out-of-VM perspective to recover from attacks.

Kernel Rootkit Defense Kernel-level rootkits are one of the biggest threats to the integrity of the modern OS. To detect their presence, a large number of techniques have been proposed, such as using a specification-based approach [22, 41, 42]), or VMI-based (e.g., [15, 24, 40]), or binary analysis-based [30], or using the signatures (e.g., field value invariants [12], or structural-invariants [34]). Meanwhile, as kernel hooks are directly related to the intrusions of kernel rootkits, there are also many techniques focusing on kernel hook identification [50] and protection [47], and profiling [31].

The substantial difference compared with these techniques is EXTERIOR not only focuses on the kernel rootkit detection by exploring new techniques such as our out-of-VM MAKEUP, but also introduces a new technique towards the recovery from a kernel rootkit attack.

8. Conclusion

We have presented EXTERIOR, a novel dual-VM based external *shell* for trusted, native, out-of-VM program execution for guest-OS administration including introspection, configuration, and recovery. The key idea is to leverage an identical trusted kernel with the guest-OS to create the necessary environment for a running process in a SVM, and dynamically and transparently redirect and update the memory at the VMM level to a GVM, thereby achieving the same effect in terms of kernel state updates of running a program inside a guest-OS. We have implemented EXTERIOR and demonstrated that EXTERIOR can be used for (automatic) introspection, (re)configuration of the guest-OS state (in the cloud), and can perform a timely response such as recovery from a kernel malware intrusion. In addition to the trustworthiness, higher privilege and stealthiness gained from out-of-VM, another distinctive feature of EXTERIOR is that it does not require any user account in the guest-OS to perform these tasks. Finally, we believe EXTERIOR has demonstrated a new program execution model on top of virtualization, and it will open new opportunities for system administration and security.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and suggestions. We are also grateful to Haibo Chen, Yuan Ding, Kenneth Miller, Richard Wartell, and Junyan Zeng for their invaluable feedback on an early draft of this paper. This research was supported in part by a AFOSR grant FA9550-12-1-0077 and a research grant from VMware Inc. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the AFOSR and VMware.

References

[1] QEMU: an open source processor emulator. *http://www.qemu.org/*.

[2] Vprobe toolkit. https://github.com/vmware/vprobe-toolkit.

[3] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu. Dksm: Subverting virtual machine introspection for fun and profit. In *The 29th IEEE Symposium on Reliable Distributed Systems*, 2010.

[4] E. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, 2003.

[5] A. B. Brown and D. A. Patterson. Undo for operators: building an undoable e-mail store. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, San Antonio, Texas, 2003.

[6] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 101–114, Cascais, Portugal, 2011. ACM. ISBN 978-1-4503-0977-6.

[7] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HOTOS'01)*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.

[8] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 2–13, Seattle, WA, USA, 2008. ACM.

[9] J. Chow, B. Pfaff, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole-system simulation. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286. USENIX Association, 2005.

[11] 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 (CCS'08)*, pages 51–62, Alexandria, Virginia, USA, 2008. ISBN 978-1-59593-810-7.

[12] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, pages 566–577, Chicago, Illinois, USA, 2009. ACM. ISBN 978-1-60558-894-0.

[13] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the* 32^{nd} *IEEE Symposium on Security and Privacy*, pages 297–312, Oakland, CA, USA, 2011.

[14] Y. Fu and Z. Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of* 33^{rd} *IEEE Symposium on Security and Privacy*, May 2012.

[15] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings Network and Distributed Systems Security Symposium (NDSS'03)*, February 2003.

[16] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 193–206, Bolton Landing, NY, USA, 2003. ACM. ISBN 1-58113-757-5.

[17] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*, May 2007.

[18] R. P. Goldberg. Architectural principles of virtual machines. PhD thesis, Harvard University. 1972.

[19] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer Magazine*, pages 34–45, June 1974.

[20] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin. Os-sommelier: Memory-only operating system fingerprinting in the cloud. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SOCC'12)*, San Jose, CA, October 2012.

[21] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process implanting: A new active introspection framework for virtualization. In *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems (SRDS 2011)*, pages 147–156, Madrid, Spain, October 4-7, 2011.

[22] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with osck. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 279–290, Newport Beach, California, USA, 2011. ISBN 978-1-4503-0266-1.

[23] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su. Back to the future: A framework for automatic malware removal and system repair. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 257–268, 2006. ISBN 0-7695-2716-7.

[24] 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 (CCS'07)*, pages 128–138, Alexandria, Virginia, USA, 2007. ACM. ISBN 978-1-59593-703-2.

[25] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, Boston, MA, 2006. USENIX Association.

[26] 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*, VEE '08, pages 91–100, Seattle, WA, USA, 2008. ACM. ISBN 978-1-59593-796-4.

[27] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP'05)*, pages 91–104, Brighton, United Kingdom, 2005. ISBN 1-59593-079-5.

[28] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, Vancouver, BC, Canada, 2010. USENIX Association.

[29] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 314–327, 2006. ISBN 0-7695-2574-1.

[30] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference(ACSAC'04)*, pages 91–100, 2004. ISBN 0-7695-2252-1.

[31] A. Lanzi, M. I. Sharif, and W. Lee. K-tracer: A system for extracting kernel malware behavior. In *Proceedings of the 2009 Network and Distributed System Security Symposium, San Diego, California, USA,*, 2009.

[32] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008.

[33] Z. Lin, R. D. Riley, and D. Xu. Polymorphing software by randomizing data structure layout. In *Proceedings of the 6th SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'09)*, Milan, Italy, July 2009.

[34] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA, February 2011.

[35] 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, ASID '06, pages 6–11, San Jose, California, 2006. ISBN 1-59593-576-2.

[36] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Software self-healing using collaborative application communities. In *In Proceedings of Network and Distributed Systems Security Symposium*, pages 95–106, 2006.

[37] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'05)*, San Diego, CA, February 2005.

[38] R. Paleari, L. Martignoni, E. Passerini, D. Davidson, M. Fredrikson, J. Giffin, and S. Jha. Automatic generation of remediation procedures for malware infections. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, Washington, DC, 2010. ISBN 888-7-6666-5555-4.

[39] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, December 2007.

[40] B. D. Payne, M. Carbone, M. I. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of 2008 IEEE Symposium on Security and Privacy*, pages 233–247, Oakland, CA, May 2008.

[41] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - A coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, San Diego, CA, August 2004.

[42] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, B.C., Canada, August 2006. USENIX Association.

[43] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, pages 103–115, Alexandria, Virginia, USA, October 2007. ACM. ISBN 978-1-59593-703-2.

[44] N. A. Quynh. Operating system fingerprinting for virtual machines, 2010. In DEFCON 18.

[45] J. Rutkowska. Introducing blue pill, June 2006. http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html.

[46] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: automatic software self-healing using rescue points. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 37–48, Washington, DC, USA, 2009. ISBN 978-1-60558-406-5.

[47] 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*, CCS '09, pages 545–554, Chicago, Illinois, USA, 2009. ISBN 978-1-60558-894-0.

[48] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 195–209, Boston, Massachusetts, 2002. ACM. ISBN 978-1-4503-0111-4.

[49] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 260–269. IEEE Computer Society, 2003.

[50] H. Yin, Z. Liang, and D. Song. Hookfinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[51] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 203–216, Cascais, Portugal, 2011. ACM. ISBN 978-1-4503-0977-6.