A Survey on Hypervisor-Based Monitoring: Approaches, Applications, and Evolutions

ERICK BAUMAN, GBADEBO AYOADE, and ZHIQIANG LIN, University of Texas at Dallas

When designing computer monitoring systems, one goal has always been to have a complete view of the monitored target and at the same time stealthily protect the monitor itself. One way to achieve this is to use hypervisor-based, or more generally out of virtual machine (VM)-based, monitoring. There are, however, challenges that limit the use of this mechanism; the most significant of these is the semantic gap problem. Over the past decade, a considerable amount of research has been carried out to bridge the semantic gap and develop all kinds of out-of-VM monitoring techniques and applications. By tracing the evolution of out-of-VM security solutions, this article examines and classifies different approaches that have been proposed to overcome the semantic gap—the fundamental challenge in hypervisor-based monitoring—and how they have been used to develop various security applications. In particular, we review how the past approaches address different constraints, such as practicality, flexibility, coverage, and automation, while bridging the semantic gap; how they have developed different monitoring systems; and how the monitoring systems have been applied and deployed. In addition to systematizing all of the proposed techniques, we also discuss the remaining research problems and shed light on the future directions of hypervisor-based monitoring.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection

General Terms: Security

Additional Key Words and Phrases: Virtualization, hypervisor, introspection, semantic gap, isolation, integrity, virtual machine monitor, VM, monitoring, detection, malware

ACM Reference Format:
Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. 2015. A survey on hypervisor-based monitoring: Approaches, applications, and evolutions. ACM Comput. Surv. 48, 1, Article 10 (August 2015), 33 pages.
DOI: http://dx.doi.org/10.1145/2775111

1. INTRODUCTION

Computer system monitoring is a fundamental mechanism for maintaining systems security. Intrusion detection, access control (e.g., DAC, MAC, and RBAC), sandboxing, inlined reference monitors, firewalls, and antiviruses all involve security monitoring. An ideal monitoring system should have both a complete view of the monitored target and the ability to (stealthily) protect the monitoring system itself. Although there are many ways to do so, it is not a simple task. Over the past few decades, a large amount of research has been carried out to search for better and more secure ways to develop computer systems are designed in a hierarchical structure with layers [Dijkstra 1968].

Authors' addresses: E. Bauman, G. Ayoade, and Z. Lin, Computer Science Department, University of Texas at Dallas, 800 West Campbell Road, Richardson, TX 75080, USA; emails: {exb131030, gga110020, zxl111930}@utdallas.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2015 ACM 0360-0300/2015/08-ART10 $15.00
DOI: http://dx.doi.org/10.1145/2775111

WIZ, Inc. EXHIBIT - 1037
WIZ, Inc. v. Orca Security LTD.

Traditionally, from top to bottom, there is the application layer, operating system (OS) layer, and hardware layer. In general, each layer implements the abstractions and interface for the layer above it, and it uses the predefined interface of the underlying layer to perform its own functions. Each layer is well isolated from the layer above it, and the lower the layer, the more control of the system it has—but with fewer abstractions.

As a layer that runs in between the hardware and OS layers, the hypervisor was first proposed in the 1960s [Popek and Goldberg 1974]. A hypervisor—also called a *virtual machine monitor* (VMM)—enables a computing environment to run multiple independent OSes at the same time in a single physical computer, which leads to more effective use of the available computing power, storage space, and network bandwidth. The fundamental reason for introducing a hypervisor layer into our computing stack is that the capacity in a single server is so large that it is almost impossible for most workloads to efficiently use it; consequently, virtualization becomes the best way to improve resource utilization while also simplifying and automating computing unit management. Although this computing model was originally designed to logically divide the resources for time sharing of different applications in mainframes, it now underpins today's cloud computing and data centers.

In addition to pushing our computing paradigm from multitasking to multi-OS, hypervisors have also pushed system monitoring from traditional in–virtual machine (VM) monitoring to out-of-VM, hypervisor-based monitoring. This is because guest OSes run on the virtual resources [Barham et al. 2003] that a VMM provides, which gives new opportunities for flexibility and control because VMM essentially is a software layer, and software is easier to modify, migrate, and monitor. Through extracting and reconstructing the guest OS states at the VMM layer, out-of-VM monitors become possible, empowering the monitoring system to control, isolate, interpose, inspect, secure, and manage a VM from the outside [Chen and Noble 2001]. The seminal paper of Garfinkel and Rosenblum [2003] on this topic, introducing the first hypervisor-based monitoring system, "call[ed] this approach of inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it *virtual machine introspection* (VMI)." We will use the more general term *out-of-VM* to refer to hypervisor-based monitoring, including VMI, throughout this article.

However, all out-of-VM solutions have to solve the semantic gap problem due to being located one layer below the guest OS. Specifically, the semantic gap exists because at the hypervisor layer, we have access only to the raw data of the hardware level state of a VM—namely its CPU registers and physical memory, although we can also access the instruction-level execution state for certain types of hypervisors (e.g., binary translation–based VMs). However, what we want is the semantic information about the guest OS state, such as the variables being accessed, variable types, and guest OS kernel events. Therefore, we must bridge the semantic gap to obtain meaningful guest OS state information. In the past decade, there have been many proposed approaches for solving this problem, operating under different constraints and varying from being purely manual to fully automated.

Out-of-VM monitors offer many advantages over traditional in-VM monitors because they run at a higher privilege level and are isolated from attacks within the guest OSes they monitor, and also because they are one layer below the guest OS and can interpose all guest OS events. Consequently, out-of-VM monitors have been widely used in many security applications, ranging from read-only introspection (e.g., Garfinkel and Rosenblum [2003]) to writable reconfiguration and repair (e.g., Fu and Lin [2013b] and Lin [2013]), passive intrusion detection (e.g., Joshi et al. [2005]) to active prevention (e.g., Payne et al. [2008]), defense from malicious applications (e.g., Dinaburg et al. [2008] and Srinivasan et al. [2011]) to defense from malicious OSes (e.g., [Chen et al.

2008]), malware analysis (e.g., Lanzi et al. [2009] and Yin et al. [2007]) to memory forensics (e.g, Dolan-Gavitt et al. [2011b] and Hay and Nance [2008]), and so forth.

Given such a significant amount of research in out-of-VM monitoring, there is a pressing need to systematize the knowledge in this domain. As such, in this article, we would like to follow the technical footprints and trace the evolution of out-of-VM monitoring, revisit what has been done, discuss where we are, and shed light on where to go. We begin with why we need out-of-VM monitoring and discuss the technical background in Section 2. We then describe the semantic gap challenge faced in all out-of-VM monitoring in Section 3. Next, we summarize how this challenge has been solved under different constraints in Section 4 and classify all applications of out-of-VM monitoring in Section 5, as well as how they have been deployed in Section 6. We discuss the remaining research problems and future trends for out-of-VM monitoring in Section 7, followed by a related work review in Section 8. Finally, we conclude in Section 9.

2. BACKGROUND

In this section, we discuss the motivations of why we need out-of-VM monitoring. We first review in-VM monitoring and discuss its pros and cons in Section 2.1. We then introduce out-of-VM monitoring and discuss its advantages and disadvantages in Section 2.2. Finally, we discuss the scope of this work in Section 2.3.

2.1. In-VM–Based Monitoring

In general, a security monitoring system can be defined as

$$M(S, P) \rightarrow \{True, False\}, \tag{1}$$

where M denotes the security enforcing mechanism, S denotes the current system state, and P denotes the predefined policy. If the current state S satisfies the security policy P, then it is in a secure state ($True$), and if M is an online mechanism, it can allow continued execution. Otherwise, it is insecure ($False$); an attack[1] is detected, and M can halt the execution (for prevention) or report that there is an attack instance. For example, in an antivirus system, S can denote the current memory and disk state, and P the signatures of viruses; if M identifies that there is any running process or suspicious file having one of the signatures defined in P, the antivirus will raise an alarm. In a system call–based intrusion detection system, S can denote the current system call and P can denote the correct state machines for S; if M identifies that S deviates from P, then it can raise an intrusion alert. Besides these, a wide variety of other security tools (e.g., security reference monitors) all fall under the security monitoring category.

A crucial step in any monitoring system is how to collect the state information S from predefined sensors, such as those embedded in the running OS or processes, then monitor them with a well-defined security policy P. Prior to the advent of out-of-VM monitoring, most monitoring systems were in-VM based, running normally as OS applications or as loadable kernel modules in the OSes they monitor.

Advantages. Since in-VM based monitoring resides within the OS, it makes state collection easy and fast. Specifically, in-VM implementations have the following advantages:

—*Rich abstractions*: There are plenty of abstractions for in-VM monitors from which to extract the OS and process state. They can use critical kernel variables, exported

[1]Note that in this article, we use the general term *attack* to represent all kinds of cyber breaches that violate the security policy, such as malware (e.g., viruses, bots, rootkits, backdoors) that intrude and harm the system, as well as memory exploits that compromise the system.

registry or `proc` files—or even log files (on disk)—and system calls or library calls provided by OS or third-party vendors. At an even higher level of abstraction, they can also use the available in-VM security tools to extract the state. The monitoring mechanism M can trivially intercept system calls or library calls and inspect their arguments, return values, or sequences (e.g., Forrest et al. [1996]). They can also easily extract the known signatures (e.g., code hash) of running processes from memory or disk, and verify their integrity (e.g., Tripwire [Kim and Spafford 1994]). They can also monitor fine-grained control flow transfer and check its integrity (e.g., Abadi et al. [2005]).

—*Fast speed*: In-VM monitoring also executes quickly. Compared to out-of-VM solutions, in-VM state acquisition, security checks and enforcement are all executed natively (without any world switch). For instance, for an in-VM monitor program, the in-VM state can be directly accessed, and in-VM enforcement can be instantly executed without any trapping into hypervisor.

Disadvantages. Although in-VM systems have been very successful, they have a fundamental weakness: they can be attacked because they are executing at the same privilege level as the system they are protecting (unless special care is taken to protect the monitor, e.g., using special memory protection enforced by hypervisor). Malware, such as kernel rootkits, or more generally intrusions or attacks, can often tamper with all components involved in $M(S, P)$, such as the sensors that collect state information and the monitoring tools that enforce the security policy. More specifically, they can:

—*Generate the false state S*: To mislead the monitoring systems, attackers can modify logs, the registry, `proc` files, or any other state information of interest with false data (or even the code responsible for generating the data), as long as the system can continue to function (e.g., no crashes).

—*Tamper with the security policy P*. Attackers can also modify the security policy P if it is known. For instance, the attackers can modify the signature database to evade their attacks.

—*Tamper with the enforcing mechanism M*. A wide range of methods can be used here. For instance, if the security mechanism is based on system call hooking, attackers can then modify the system call tables to bypass the security check; if the security mechanism is deployed using a kernel module or individual monitoring process, attackers can simply remove or shut down the monitoring module or process.

2.2. Out-of-VM–Based Monitoring

Due to the shortcomings of in-VM systems, researchers proposed moving monitoring outside of the VM. In particular, hypervisor-based, or the so-called VMM-based, systems use the hypervisor layer to secure the monitoring system. In theory, since hypervisors operate at a lower level than the monitored system, they too are isolated and become more secure. Hypervisor-based monitoring has been the subject of extensive research in the past decade, as briefly mentioned in the previous section.

A hypervisor runs either directly on the host hardware (bare metal) or in another host OS, and provides a software-controlled layer that resembles the host hardware. Depending on *where* the hypervisor is located, hypervisors can be classified into two types [Popek and Goldberg 1974]:

(1) *Type 1* (*bare metal*) hypervisors, which run directly on the host's hardware to control the hardware and monitor the guest OS. Typical examples of such hypervisors include Xen, VMware ESX, and Microsoft Hyper-V.
(2) *Type 2* (*hosted*) hypervisors, which run within a traditional OS. In other words, a hosted hypervisor adds a distinct software layer atop the host OS, and the guest

OS becomes a third software layer above the hardware. Well-known examples of type 2 hypervisors include KVM, VMware Workstation, VirtualBox, and QEMU.

Although the preceding *type 1* and *type 2* hypervisor classification has been widely accepted (e.g., Barham et al. [2003] and Wang and Jiang [2010]), sometimes it insufficiently differentiates among hypervisors of the same type (e.g., KVM vs. QEMU). Therefore, based on *how* the virtualization gets designed (hardware vs. software) [Adams and Agesen 2006] and the guest OS and its application code is executed, we can have another type of classification of hypervisors that will be used throughout this article:

(1) *Native hypervisors* that directly push the guest code to execute natively on the hardware using hardware virtualization [Adams and Agesen 2006].
(2) *Emulation hypervisors* that translate each guest instruction for an emulated execution using software virtualization [Adams and Agesen 2006].

Examples of native hypervisors include Xen, KVM, VMware ESX, and Microsoft Hyper-V, and emulation hypervisors include QEMU, Bochs, and the very early versions of VMware-Workstation and VirtualBox (note that recent VMware-Workstation and VirtualBox are able to execute the guest OS code natively). Since there is no binary code translation involved, native hypervisor runs much faster than emulation hypervisor.

With a hypervisor, system developers have an additional control layer that allows them to multiplex resources (e.g., scheduling the VMs in a way similar to scheduling the processes) and migrate and control the VMs using software. In native hypervisors, system developers can get control at certain hardware events (e.g., interrupt, page fault exception). For emulation hypervisors, system developers can get the control any time they want because all instructions can be conveniently trapped.

Advantages. Because of the great opportunities provided by hypervisors, we can push the security monitoring into the hypervisor (namely from in-VM to out-of-VM). In-VM monitors run inside the OS, whereas out-of-VM monitors run outside of the OS and are located at the hypervisor layer. Although the two types of security monitors can perform most of the same functions (e.g., identifying malware, detecting intrusions), moving monitoring functionality out-of-VM has tremendous benefits. In particular, we can have:

—*Strong isolation (tamper resilience)*: Since the guest OS runs on a separate level above the hypervisor, there is a world switch whenever control passes between the two. The hypervisor thus provides strong isolation between the security monitors and the attacks present in the guest OS. Unless the hypervisor has vulnerabilities, it makes M and P tamper resilient, as they are located below the guest OS. Although attackers may still generate false data, if the out-of-VM solutions directly extract S from the raw data, they could also largely defend against false data generation attacks.
—*Transparent deployment*: To deploy a security monitor at the hypervisor layer, we have no need for an account in the guest OS, neither do we have any need to install the software inside the OS. Instead, everything can happen transparently at the hypervisor layer without even disrupting services (e.g., many read-only introspection techniques can be transparently deployed during runtime).
—*Complete view*: Another advantage of out-of-VM over in-VM monitors is that the hypervisor has full access to all of the memory, register, and disk state of the VM on which the OS runs. We can observe each application's state, as well as the kernel state, including those invisible ones hidden by attackers, which is often challenging to achieve through in-VM approaches.
—*High cost savings*: Out-of-VM also provides system developers unrestricted accesses to virtualized resources. For instance, they can create a sandboxed environment, let

real malware execute and observe its behavior, and then simply discard the malware-damaged VM. They can also save a snapshot of the state of the guest OS, which can be analyzed later without affecting the performance of the running VM. These are features that in-VM systems often lack.

—*Less vulnerability*: In-VM systems usually have to trust the entire guest OS kernel, which tends to have a huge code base. However, out-of-VM often only needs to trust the underlying hypervisor, which has a smaller code base. For example, the Xen hypervisor has less than one twelfth the number of lines of code than the Linux kernel; this smaller attack surface leads to fewer vulnerabilities [Jain et al. 2014].

Disadvantages. Although out-of-VM monitoring offers many advantages over in-VM monitoring, it also has limitations. Specifically,

—*No abstractions*: To out-of-VM monitors, there are no guest OS abstractions. Therefore, all out-of-VM solutions face a challenge that must be addressed to perform effective monitoring; they must bridge the semantic gap caused by moving monitors outside of the guest OS. The details about the semantic gap problem are discussed in Section 3.

—*Slow speed*: In addition, out-of-VM monitoring has to perform additional address translation (what it observes is physical memory addresses, and it has to translate between those and the guest's virtual addresses) and world switching that traps to the hypervisor for security checks and monitoring. It therefore usually is slower compared to in-VM monitoring, although recently there were efforts (e.g., Li et al. [2015]) to improve performance of the world switching.

2.3. Scope

In this article, we focus on the out-of-VM solutions that are below the OS layer but still monitor the internal contents of the machine. Thus, we do not consider the out-of-VM solutions that run on network devices (e.g., firewalls, gateways, and switches).

Strictly speaking, we technically should also exclude the hardware solutions that often use extra hardware (e.g., PCI devices) for monitoring, since hardware usually does not belong to the hypervisor layer, which is software by definition. However, we still include them because out-of-VM monitoring can also be implemented using purely hardware approaches. Not only do these hardware solutions also face the semantic gap challenge, it can be even more difficult for them if a hypervisor is present in the system; they must bypass the hypervisor layer to introspect the guest OS. Therefore, for a complete view of all kinds of out-of-VM monitoring techniques, we include these extra hardware approaches.

3. THE SEMANTIC GAP PROBLEM

The primary advantage of in-VM systems is their direct access to all kinds of OS-level abstractions. However, when using a hypervisor, access to all of the rich semantic abstractions inside the OS is lost. Although hypervisors have a grand view of the entire state of the VMs they monitor, this grand view unfortunately consists of just ones and zeros with no context. Therefore, there is a semantic gap between what we can observe (Section 3.1) and what we want (Section 3.2), as illustrated in Figure 1, and we must bridge it to provide effective monitoring services. In this section, we present the semantic gap problem in greater detail.

3.1. What We Observe

Since hypervisors virtualize all of the computing resources and provide abstractions to the guest OS, they can theoretically view any aspect of the guest OS's state. However,

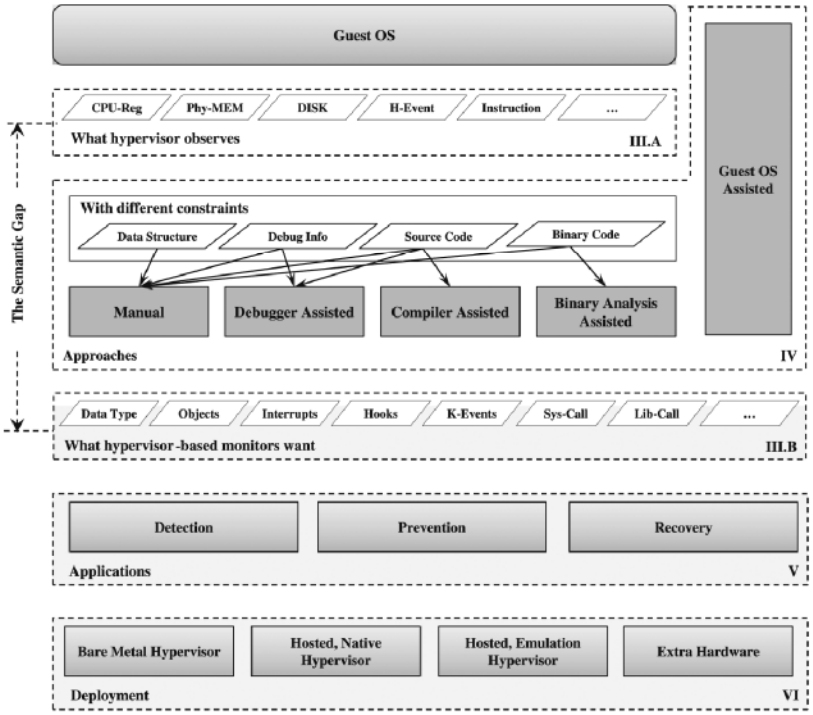

Fig. 1. An overview of various approaches, applications, and deployment of hypervisor-based monitoring.

as shown in Table I, different types of hypervisors virtualize different logical resources; consequently, they have different views.

Native hypervisors directly push guest code to run on the hardware. As such, they cannot continuously monitor the execution of the guest OS (unless they use single-step execution). Instead, they intercept control from interrupts or other hardware events. They therefore have a *snapshot view* of the VM state, which is often acquired when certain hardware events occur and allow the hypervisor to regain control. Hypervisors of this type can observe the following:

—*CPU registers*: All of the CPU registers can be read by the hypervisor when it gains control because it runs at the highest privilege level.
—*Guest OS memory*: The entire guest OS memory state can also be observed. However, hypervisors only have access to physical addresses, which have to be translated to virtual addresses while accessing each specific memory cell.
—*Hard disk contents*: Similar to the memory image, the content of the guest OS's disk image, if not encrypted, is also visible to the hypervisor.
—*Hardware events*: All hardware-level events, including timers, interrupts, and exceptions, can also be observed.
—*I/O traffic*: The hypervisor also oversees all I/O traffic, including network traffic, disk I/O, and keystrokes.

Note that native hypervisors technically can obtain a contiguous view of the guest OS (e.g., through single-step execution), but doing so is much too slow; it would necessitate trapping most events to the hypervisor, requiring constant world switching. Therefore, most of the time, native hypervisors can only observe some special VMM-level instructions (e.g., Intel VT instructions) and special kernel events such as interrupts and exceptions (e.g., page faults if there is no extended or nested page table enabled in the hardware).

Table I. Low-Level Data and Desired Abstractions in Out-of-VM Monitoring

What We Observe	Snapshot View					Contiguous View				
	CPU Registers	Physical Memory	Disk Data	Hardware Events	I/O Data	Program Counter	Opcode & Operand	Control Flow Transfer	Call-Stack	Context-Switch
Native Hypervisor	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗
Emulation Hypervisor	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Emulation Hypervisor	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Native Hypervisor	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗
What We Want	Variables, Objects	Variables and Types	File System and Files	Interrupt/Exceptions	Packets, Buffers	Instruction Semantics	Variables, Pointers	Calls, Hooks, Branches	Execution Context	Processes, Threads
	Data and Control State Abstractions									

Emulation hypervisors can have a *contiguous view* of the guest-OS execution in addition to the *snapshot view* observed by native hypervisors. In particular, they can observe the following:

—*Program counter*: They can know which instructions get executed and their disassembly code.
—*Instruction opcode and operand*: For each executed instruction, they can observe its opcode and operand.
—*Control flow transfer*: All control flow transfers (e.g., `call/jmp/ret`, conditional branches) can be observed, along with their source and destination addresses (if applicable).
—*Call stack*: The stack can be traversed if a stack frame pointer exists, or instructions can be transparently instrumented to build the call stack information.
—*Context-switch*: Each specific process or thread execution context can also be observed.

3.2. What We Want

Given that hypervisors only have access to low-level binary data, we have to translate the data into higher-level abstractions to provide useful monitoring services. In general, these abstractions can be broken down into two types: data state abstractions and control state abstractions.

Data state abstractions. Data state consists of the state of the current kernel objects or the objects inside a monitored process (basically the memory state) as well as the current *CPU execution* state. More specifically, we are interested in the following:

—*Variables, objects, and virtual address spaces*: Given the physical memory of a guest OS, we would like to know the location of kernel or monitored process variables (or objects) of interest, the virtual address of a variable or object, and how to locate virtual addresses in the physical memory.
—*Data structure types and their connections*: We also would like to know object types, data structures, and their point-to relations (such that we can traverse them and verify their integrity).
—*File systems and files*: Given the disk image, we are interested in the type of file system being used and where files are located.
—*Interrupts, exceptions, and other kernel events*: For an observed hardware event, we would like to obtain additional details about it; for an interrupt or exception, we want to know which specific interrupt or exception it is. In addition, if possible, we would like to recognize other kernel events, such as when a certain kernel object is created or destroyed.
—*Packets and buffers*: From observed I/O data, we would like to differentiate between network packets and DMA buffers. Even further, we would like to determine the content and context of packets if possible.

Control state abstractions. Whereas *data state* provides a snapshot view of the running VMs, its granularity depends on when and how often the hypervisors take control. In native hypervisors, hypervisors can regain control only when certain hardware events occur, whereas with emulation hypervisors, we can take control at arbitrary times (depending on our interests). As mentioned in Section 3.1, obtaining a contiguous view with native hypervisors, although possible, is far too inefficient for most applications. Therefore, we mainly focus on emulation hypervisors for the *control state* abstractions. More specifically, from fine grained to coarse grained, we are interested in the following:

—*Instructions, control path, and call stack*: Knowledge of which instruction the VM is executing, the control path to which it belongs, and the calling context can help the out-of-VM monitor precisely understand the current state of the guest OS. These fine-grained control states can often be observed by emulation hypervisors or single-stepped native hypervisors.
—*Function calls, system calls, library calls, and hooks*: As instruction-level monitoring usually significantly slows down the VM execution, we could instead monitor at a coarse-grained granularity (e.g., at level of function call execution, or at certain system calls, library calls, or hooks of monitor interest).
—*Processes, threads, and execution context*: When there is a context switch, we would like to know which process (thread) is switched (from) to, as control flow is often thread specific. For a given executing instruction, we also would like to know which process or thread is executing this instruction, whether the execution is in user space or kernel space.

4. APPROACHES

In this section, we summarize the general approaches that have been proposed for bridging the semantic gap, as well as the specific applications the approaches are targeting. In total, we have analyzed 64 papers (projects) that were published between 2003 and 2014 from several highly selective security and system venues.[2] Note that we do not aim to exhaustively examine all of the papers, as there are too many other

[2]The security venues include IEEE Security and Privacy (SP), ACM CCS, USENIX Security (in short USENIX-SEC), NDSS, DSN, ESORICS, RAID, and ACSAC; system venues include SOSP, OSDI, ASPLOS, USENIX-ATC, EuroSys, and VEE.

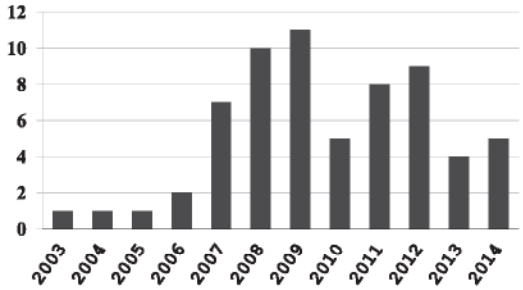

Fig. 2. Number of published out-of-VM monitoring papers in the past decade.

papers published in other venues. The distribution of the number of papers published per year along the out-of-VM monitoring topic is presented in Figure 2.

In total, there are five major approaches to bridging the semantic gap: the manual approach (Section 4.1), debugger-assisted approach (Section 4.2), compiler-assisted approach (Section 4.3), binary analysis–assisted approach (Section 4.4), and guest-assisted approach (Section 4.5), as illustrated in Figure 1 and also reported in Table II. The first four approaches are classified based on the constraints that an implementer faces when building an out-of-VM monitor. At a high level, these constraints are based on whether there is access to guest OS *data structures*, *debug information*, *source code*, or *binary code*. In addition, there is an option to avoid the semantic gap altogether, at the cost of potentially sacrificing the security advantages from VMI. This approach, the *guest-assisted* approach, modifies a guest OS kernel or places a program inside to pass information to the out-of-VM monitors.

All of these approaches vary in difficulty (when developers implement them) and practicality. For example, although some manual approaches utilize detailed information about a specific OS, they may be easier to implement. Different approaches also vary in terms of the degree of automation. The manual approach by definition involves little automation, whereas the other approaches may use varying amounts of automation in the way in which they bridge the gap, and the way in which they port their implementation from one to another. In the following sections, we will examine each approach in greater detail, and we will compare them in Section 4.6.

4.1. Manual Approaches

The most straightforward approach is to manually reconstruct the guest OS abstractions at the hypervisor layer, as long as we have the guest OS data structure information. The layout, offset, and field type for each individual data structure, as well as the connections (e.g., the point-to relations) between data structures of a guest OS, can be determined through manual analysis of either documentation, debug information, source code, or binary code. Then, by accessing the memory of the guest OS, hypervisor programmers are able to use this detailed understanding of kernel structures to extract information about the state of the guest OS, which can be used for various introspection purposes. For example, the Linux kernel task structure (i.e., `task_struct`) is organized in a double-linked list that contains all running processes. By traversing through this list, we can retrieve all running processes in a system if there is no direct kernel object manipulation (DKOM) [Petroni and Hicks 2007] attack inside the guest OS. This approach can also be used to retrieve information about the disk image containing files stored on a system. As long as access to the disk image is available, knowledge of the file system structure (e.g., `ext2`) can be used to traverse and access files.

Although manually determining data structures can be tedious, it is not a difficult technical problem in that a human is able to perform the semantic interpretations.

Table II. Summary of Different Out-of-VM Approaches in Bridging the Semantic Gap, Their Applications, and Deployment

System		Approaches					Application			Deployment			
Name	Venue	Manual	Debugger Assisted	Compiler Assisted	Binary Analysis	Guest Assisted	Detection	Prevention	Recovery	Bare Metal	Hosted Native	Hosted Emulation	Extra Hardware
LIVEWIRE [Garfinkel and Rosenblum 2003]	NDSS '03		✓				✓					✓	
COPILOT [Petroni et al. 2004]	USENIX-SEC '04	✓					✓						✓
INTROVIRT [Joshi et al. 2005]	SOSP '05		✓				✓				✓		
ANTFARM [Jones et al. 2006]	USENIX-ATC '06	✓					✓			✓			
PFWA [Petroni et al. 2006]	USENIX-SEC '06	✓					✓						✓
EKKYS [Egele et al. 2007]	USENIX-ATC '07	✓					✓					✓	
VMSCOPE [Jiang and Wang 2007]	RAID '07	✓					✓					✓	
VMWATCHER [Jiang et al. 2007]	CCS '07	✓					✓					✓	
PANORAMA [Yin et al. 2007]	CCS '07	✓					✓					✓	
SBCFI [Petroni and Hicks 2007]	CCS '07			✓			✓			✓	✓	✓	
SECVISOR [Seshadri et al. 2007]	SOSP '07	✓						✓		✓			
XENACCESS [Payne et al. 2007]	ACSAC '07	✓					✓			✓			
HOOKFINDER [Yin et al. 2008]	NDSS '08	✓					✓					✓	
LYCOSID [Jones et al. 2008]	VEE '08	✓					✓			✓			
OVERSHADOW [Chen et al. 2008]	ASPLOS '08	✓						✓			✓		
LARES [Payne et al. 2008]	SP '08					✓		✓		✓			
PATAGONIX [Litty et al. 2008]	USENIX-SEC '08	✓					✓	✓		✓			
HOOKMAP [Wang et al. 2008]	RAID '08		✓				✓					✓	
NICKLE [Riley et al. 2008]	RAID '08	✓						✓				✓	
ETHER [Dinaburg et al. 2008]	CCS '08	✓					✓			✓			
VICI [Fraser et al. 2008]	ACSAC '08	✓							✓	✓			
GIBRALTAR [Baliga et al. 2008]	ACSAC '08			✓			✓			✓	✓	✓	
ANUBIS [Bayer et al. 2009]	NDSS '09	✓					✓					✓	
KTRACER [Lanzi et al. 2009]	NDSS '09	✓					✓					✓	
POKER [Riley et al. 2009]	EuroSys '09		✓				✓					✓	
RKPROFILER [Xuan et al. 2009]	RAID '09	✓					✓					✓	
KOP [Carbone et al. 2009]	CCS '09			✓			✓			✓	✓	✓	
HOOKSAFE [Wang et al. 2009]	CCS '09					✓		✓		✓			
SLCL [Sharif et al. 2009]	CCS '09					✓	✓	✓			✓		
DGSTG [Dolan-Gavitt et al. 2009]	CCS '09	✓					✓			✓	✓		
OSVM [Oliveira and Wu 2009]	ACSAC '09					✓		✓				✓	
MAVMM [Nguyen et al. 2009]	ACSAC '09	✓					✓			✓			
HIMA [Azab et al. 2009]	ACSAC '09	✓					✓	✓		✓			
PMPDFGJ [Paleari et al. 2010]	USENIX-SEC '10	✓							✓			✓	
HYPERCHECK [Wang et al. 2010]	RAID '10	✓					✓						✓
LIVEDM [Rhee et al. 2010]	RAID '10			✓			✓					✓	
TRAILOFBYTES [Krishnan et al. 2010]	CCS '10	✓					✓			✓			
PEDA [Zhang et al. 2010]	ACSAC '10	✓					✓			✓		✓	
HUKO [Xiong et al. 2011]	NDSS '11					✓		✓		✓			
SIGGRAPH [Lin et al. 2011]	NDSS '11			✓			✓			✓	✓	✓	
GATEWAY [Srivastava andGiffin 2011]	NDSS '11			✓			✓	✓		✓			
OSCK [Hofmann et al. 2011]	ASPLOS '11			✓			✓				✓		
VIRTUOSO [Dolan-Gavitt et al. 2011a]	SP '11				✓		✓			✓	✓	✓	
SHELLOS [Snow et al. 2011]	USENIX-SEC '11	✓					✓				✓		
SWJX [Srinivasan et al.2011]	CCS '11					✓	✓				✓		
GDXJ [Gu et al. 2011]	SRDS '11					✓	✓		✓		✓		
KRUISER [Tian et al. 2012]	NDSS '12					✓	✓			✓			
V2E [Yan et al. 2012]	VEE '12	✓					✓				✓	✓	
VMST [Fu and Lin 2012]	SP '12				✓		✓					✓	
MAS [Cui et al. 2012]	USENIX-SEC '12			✓			✓			✓	✓	✓	
DROIDSCOPE [Yan and Yin 2012]	USENIX-SEC '12	✓					✓					✓	

(Continued)

Table II. Continued

System		Approaches					Application			Deployment			
Name	**Venue**	**Manual**	**Debugger Assisted**	**Compiler Assisted**	**Binary Analysis**	**Guest Assisted**	**Detection**	**Prevention**	**Recovery**	**Bare Metal**	**Hosted Native**	**Hosted Emulation**	**Extra Hardware**
SYRINGE [Carbone et al. 2012]	RAID '12					✓	✓			✓			
VIGILARE [Moon et al. 2012]	CCS '12	✓					✓						✓
BLACKSHEEP [Bianchi et al. 2012]	CCS '12				✓		✓					✓	
SENTRY [Srivastava and Giffin 2012]	ACSAC '12					✓	✓	✓		✓			
EXTERIOR [Fu and Lin 2013b]	VEE '13				✓		✓		✓			✓	
KI-MON [Lee et al. 2013]	USENIX-SEC '13	✓					✓						✓
MOSS [Prakash et al. 2013]	DSN '13		✓				✓			✓	✓	✓	
TZB [Dolan-Gavitt et al. 2013]	CCS '13				✓		✓					✓	
HYBRIDBRIDGE [Saberi et al. 2014]	NDSS '14				✓		✓				✓	✓	
RTKDSM [Hizver and Chiueh 2014]	VEE '14	✓					✓			✓			
HYPERSHELL [Fu et al. 2014]	USENIX-ATC '14					✓	✓		✓		✓		
WCLM [Wu et al. 2014]	DSN '14					✓	✓		✓		✓		
TZ-RKP [Azab et al. 2014]	CCS '14	✓					✓						✓
Total	64	33	5	8	6	12	55	12	6	27	19	30	6

Essentially, a human does the work to solve the semantic gap problem instead of relying on other tools. However, the biggest limitation for such an approach is its extremely low scalability. To support a large volume of different OSes, it requires tremendous amounts of effort to manually build the data structure knowledge for each kernel and rewrite the corresponding monitor programs.

Examples. Surprisingly, as reported in Table II, many out-of-VM monitors (33 out of 64) actually adopted such a manual approach. For instance, the first manual approach, COPILOT [Petroni et al. 2004], retrieves the Linux kernel text and system call tables, then verifies their integrity using an external PCI device (invisible to the guest OS). The knowledge about the OS kernel data structures, such as where the code and system call tables are located, is manually reconstructed. COPILOT specifically targets the 2.4 and 2.6 series of Linux kernels. It utilizes the fact that Linux kernel memory is not paged and that kernel virtual addresses are linear mapped. Certain Linux kernel text and data structures, including page tables, are located at a specific invariant location in virtual memory and are mapped linearly, allowing for the retrieval of page tables and thus the locations of data structures that would otherwise be difficult to determine. Once the locations are known, it then reconstructs the guest OS semantics based on kernel data structure knowledge that was also manually extracted. Some examples of extracted abstractions are the organization of the kernel `task_struct` and what offsets allow access to other data structures, such as `mm_struct`.

Followed by the COPILOT approach, dynamic spyware analysis (referred to as EKKYS[3]) [Egele et al. 2007] and PANORAMA [Yin et al. 2007] also manually reconstruct the guest OS abstractions (e.g., processes, files, browser, or kernel objects) to facilitate malware analysis. VMWATCHER [Jiang et al. 2007] uses a guest view casting technique to infer the state of the guest OS, and the casting is guided by manually retrieved kernel data structure knowledge. XENACCESS [Payne et al. 2007] is a library for the monitoring of guest OSes running on the Xen hypervisor. It provides a framework for accessing the state of the guest OS and reducing the amount of work for guest introspection.

[3]Note that throughout the article, we refer to each discussed paper by its system name, if there is any, and otherwise by the author's initials.

However, the library itself is developed based on the manually retrieved data structure knowledge of the monitored kernel. Other manual approaches to bridge the semantic gap include OverShadow [Chen et al. 2008], Lycosid [Jones et al. 2008], VMScope [Jiang and Wang 2007], HookFinder [Yin et al. 2008], among others. The list of all of these manual approaches is presented in the third column of Table II.

4.2. Debugger-Assisted Approaches

As data structure knowledge is usually available in the debugging symbols if a program or OS kernel is compiled with the debug option, we can bridge the semantic gap with the assistance of debuggers. This approach is also very straightforward. Specifically, if the kernel is compiled with debug option, we can obtain debugging information from off-the-shelf debuggers designed to analyze kernel dumps or live memory. From this information, we can further derive the guest OS abstractions. Note that sometimes we can directly retrieve the debugging symbols from software vendors (e.g., Microsoft actually does release the Windows kernel symbols to the public), but for this case we still have to develop the introspection routine based on the retrieved data structure knowledge, which essentially is a manual approach. Similarly to the manual approach, using a debugger makes this approach OS specific. In addition, the guest OS kernel has to be recompiled with debugging symbols, which greatly limits the practicality of this approach.

Examples. Not many systems use the debugger-assisted approach, and in total there are only five such projects, as reported in Table II. Specifically, the first debugger-assisted approach, Livewire [Garfinkel and Rosenblum 2003], which pioneered the concept of VMI, leverages a modified kernel `crash` dump analysis tool (essentially a debugger) to interpret the raw binary data of a memory dump generated by a kernel compiled with debugging information. The advantage of this approach is that the debugger tool can directly return the guest OS abstractions without developing any additional code. For instance, with `crash` tool [Anderson 2003], Livewire uses the built-in commands to retrieve the list of running process (by invoking the `ps` command), opened files (with the `files` command), live kernel objects (with the `kmem` command), and the kernel call stack for each process (with the `bt` command). Using the semantic information obtained from these commands, Livewire then determines whether the guest OS has been compromised.

Additionally, IntroVirt [Joshi et al. 2005] leverages the debugging symbols to set "breakpoints" to inspect and execute vulnerability specific predicates at the hypervisor layer. HookMap [Wang et al. 2008] resolves kernel symbols by querying from the `system.map` file for kernel text and `nm` utilities for kernel modules. As a rootkit profiler, PoKeR [Riley et al. 2009] also compiles the Linux kernel with debugging flags and uses a customized debugger (i.e., `gdb`) to traverse kernel objects and provide kernel object maps. These object maps are used to facilitate the understanding of kernel rootkit behavior. Moss [Prakash et al. 2013] leverages the Windows debugging symbols to mutate the fields of the kernel data structure out-of-VM with duplicate-value–directed semantic field fuzzing to estimate the severity of kernel data structure manipulation attacks.

4.3. Compiler-Assisted Approaches

In the compiler-assisted approach, the source code of the OS is used (or modified) to automatically construct a data structure graph (or inform the hypervisor with the data of interest). This approach leverages the power of the compiler to infer where control will potentially flow to as the OS runs, taking advantage of type definitions and context information in the OS source to determine the types of generic pointers.

The compiler also helps to automate the process of finding abstractions, or the kernel is modified (or automatically instrumented) to generate the abstractions if the original

source code does not contain them, but this approach relies on having access to the source. This approach also requires one to repeat the analysis on any new kernels that need the VMI solutions.

Examples. In total, there are eight projects that use the compiler-assisted approach. The first approach of this type, SBCFI [Petroni and Hicks 2007], obtains the locations of kernel abstractions by first extracting global variables and creating a graph of kernel variable types from the kernel source code, then using that data to generate code to traverse pointers accessible from those variables. The goal of SBCFI is to check the kernel control flow integrity (CFI). To this end, it generates the control flow graph from kernel source code. By performing periodic checks on the snapshots of the kernel against an initially generated flow graph, it can determine whether the kernel has been compromised. CFI [Abadi et al. 2005] of the kernel can be determined by verifying static and dynamic pointers used by the kernel. Static pointers include function pointers stored in system call tables or jump tables used in the program. Dynamic function pointers include generic pointers like `void*` in C and indirect function calls like in select statements or unions, which is determined during the execution. To resolve dynamic pointers, SBCFI requires source annotations to recognize certain pointer types, including generic pointers.

KOP [Carbone et al. 2009] recognizes the limitation in SBCFI and proposes to compute all possible types of generic pointers using interprocedural point-to analysis, resolve type ambiguities with pattern matching, and determine the boundary of dynamic arrays with kernel memory pool knowledge. MAS [Cui et al. 2012] further improves KOP to provide a more reliable type graph with a new memory traversal algorithm that supports error correction (i.e., cutting off invalid pointers) and stops error propagation.

Other compiler-assisted approaches include GIBRALTAR [Baliga et al. 2008], which leverages CIL [Necula et al. 2002] to acquire data structure definitions, and SIGGRAPH [Lin et al. 2011], which uses `gcc` to derive kernel data structure invariants. LIVEDM [Rhee et al. 2010] leverages kernel source code to distinguish different heap data structures and then constructs the kernel heap graph to facilitate in understanding kernel malware behavior. OSCK [Hofmann et al. 2011] extracts the memory management data structures from kernel source code and then verifies the type safety properties for the kernel heap using a clever linear scanning approach. GATEWAY [Srivastava and Giffin 2011] recompiles kernel source code with special padding such that a binary rewriter can be applied and the control flow transfer can be observed and enforced at the hypervisor layer.

4.4. Binary Analysis–Assisted Approaches

The binary analysis–assisted approach is used in cases where only the compiled binary of an OS is available. It does not require access to any special versions of the OS, any debug information, or any defined kernel data structures. Instead, this approach analyzes the compiled (symbol stripped) OS kernel binary code and reconstructs the guest OS abstractions.

This approach is quite sophisticated because it does not obtain abstractions or definitions from any other sources. Everything has to be determined either by performing dynamic binary analysis on register values, accessed memory, and executed instructions, or by performing static binary analysis on raw code and data. However, this makes the final result more practical for VMI, as this can be performed by anyone and does not rely on a special version of an OS.

Analysis of binary code can be done both offline and online. For offline analysis, abstractions are developed by observing how a trusted version of an OS executes while inspecting the kernel states. Then, from the information obtained by these observations, a tool can be trained to assist out-of-VM monitors. Online analysis, on the other

hand, can occur dynamically with a live system. Specifically, it observes how the data gets accessed, then reconstructs the abstractions via online dataflow analysis and data redirection.

Examples. Interestingly, six projects used the binary analysis–assisted approach. In particular, the first binary code analysis approach, VIRTUOSO [Dolan-Gavitt et al. 2011a], demonstrates that it is possible to extract the instruction traces from the execution of native inspection commands and the OS kernel, then translate the traces into an introspection program. Three different phases are involved while using VIRTUOSO: training, analysis, and runtime environment generation. Training involves taking multiple traces of an introspection program (e.g., `ps`). The analysis phase involves looking through instructions in the trace to identify unwanted instructions like `malloc`. The instructions needed are then used to generate an executable. The generated program can then run as a stand-alone outside the guest OS in the hypervisor layer to perform the introspection.

As VIRTUOSO involves an offline training phase to generate the trace that suffers the coverage issues, VMST [Fu and Lin 2012, 2013a] shows an online binary code reuse approach without training to enable native inspection programs to automatically become introspection programs. With VMST, no training is needed, as the data of introspection interest is automatically identified and accessed by the native inspection process. EXTERIOR [Fu and Lin 2013b] essentially uses the same binary analysis technique from VMST but extends it for guest OS writable introspection. BLACKSHEEP [Bianchi et al. 2012] avoids the complicated binary code analysis and instead uses a novel memory comparison approach to identify the kernel rootkit attack points from binary data. TZB [Dolan-Gavitt et al. 2013] shows that we can also use a mining approach to identify hook points in both the OS kernel and applications from binary executions. Combining the strengths of both the training-based approach from VIRTUOSO and the online kernel data redirection approach from VMST, HYBRIDBRIDGE [Saberi et al. 2014] improves the performance of VMST by one order of magnitude through training memoization and decoupled execution.

4.5. Guest-Assisted Approaches

The guest-assisted approach is different from the other approaches in that it avoids the semantic gap altogether, at the cost of potentially sacrificing the security advantages that come from working outside the VM (unless certain special care is taken). In this approach, a program is placed inside the guest OS to give the monitor information about the system. It may also involve adding hooks into the guest or simply running the monitoring system inside it. Since the program or the hook is running inside the OS, it has full access to all abstractions that normally are lost when moving the monitor outside the VM. However, this also potentially leaves it vulnerable to all of the same dangers as an in-host monitor, and thus all of the information it gives to the hypervisor-based monitor becomes unreliable if the system is compromised and no special care is taken to protect the inside component.

This approach is easier compared to other approaches; it avoids the semantic gap problem, it can easily work with any OS that can run the in-guest program, and it has access to all guest system information needed. However, the security concerns raised by using this approach are the same concerns that led to out-of-VM monitoring in the first place. Therefore, implementations often seek to use the hypervisor for additional protection for their in-VM monitor code.

Examples. There are 12 projects that use the guest-assisted approach. The pioneer work, LARES [Payne et al. 2008], inserts hooks in a guest VM and protects its guest component by using the hypervisor for memory isolation with the goal of supporting active monitoring. Unlike passive monitoring, active monitoring requires the interposition of

Table III. Definitions of the Metrics Used to Compare Out-of-VM Approaches

Metric	Definition
Flexibility	How many constraints the approach relies on (e.g., if the approach relies on access to OS source code, it is less flexible)
Coverage	How many abstractions can be derived; the scope of the approach
Easiness	How difficult the approach is to implement
Practicality	How useful and adoptable the approach is for real-world applications
Automation	How much can be done automatically instead of by hand

kernel events. As a result, it requires the monitoring code to be executed inside the guest OS, which is why it essentially leads to the solution of inserting certain hooks inside the guest VM. The hooks are used to trigger events that can notify the hypervisor or redirect execution to an external VM. More specifically, LARES design involves three components: a guest component, a secure VM, and a hypervisor. The hypervisor helps to protect the guest VM component by memory isolation and acts as the communication component between the guest VM and the secure VM. The secure VM is used to analyze the events and take actions necessary to prevent attacks.

In another work, secure in-VM monitoring (SLCL) [Sharif et al. 2009] places the monitoring code within the guest OS and uses the power of the hypervisor to isolate the region occupied by the monitor from attackers in the guest OS. HOOKSAFE [Wang et al. 2009] inserts an in-guest short-lived kernel module to set up the hook indirection layer, then enforces hook protection at the hypervisor layer. HUKO [Xiong et al. 2011] inserts a trusted driver into the guest OS to notify the hypervisor about the allocation and reclamation of kernel memory, and also labels the owner subject of each kernel page for the hypervisor to enforce a mandatory access control. Process implanting (GDXJ) [Gu et al. 2011] leverages the execution context of an in-guest process and replaces its code with inspection or management utilities to perform introspection and recovery. Process outgrafting (SWJX) [Srinivasan et al. 2011] inserts a kernel module into a guest VM and redirects the system call execution of a monitored process to a guest VM for behavior analysis. SYRINGE [Carbone et al. 2012] uses a novel guest assisted function-call injection and localized shepherding technique to verify the execution of guest code. Rather than redirecting the data access (as in VMST), HYPERSHELL [Fu et al. 2014] and WCLM [Wu et al. 2014] redirect the execution of system call from the secure VM to the guest VM, with an assisted helper process at the guest VM, to bridge the semantic gap.

More broadly, the approach that requires kernel source code modification (or instrumentation) to proactively report state changes to the hypervisor is also guest assisted. For instance, OSVM [Oliveira and Wu 2009] modifies guest kernel source code to add the Biba integrity level for kernel subjects and objects, then collaboratively enforces the Biba integrity model at the hypervisor layer. KRUISER [Tian et al. 2012] modifies kernel source code, especially the heap management data structure to add canaries, then verifies the canaries at the hypervisor layer to detect heap overflow. SENTRY [Srivastava and Giffin 2012] modifies the kernel source code to partition the kernel data structure layout such that it can detect and prevent malicious modifications to critical kernel data structures that are protected by the hypervisor.

4.6. Summary and Comparison

As discussed earlier, different approaches have different constraints, pros, and cons. In the following, we compare them in greater detail. We provide definitions of each metric in Table III and compare them in Table IV. Specifically, we compare each approach from the degree of *flexibility* (flexibility of the approach; less constraints, more flexible), *coverage* (how many abstractions can be reconstructed, and how much of the guest OS

Table IV. Comparison Between Different Out-of-VM Approaches

Approaches	Year Started	Flexibility	Coverage	Easiness	Practicality	Capability	Automation
Debugger Assisted	2003	◐	●	●	○	◐	●
Manual	2004	●	○	●	◐	○	○
Compiler Assisted	2007	○	●	◐	○	●	●
Guest Assisted	2008	◐	●	●	◐	●	◐
Binary Analysis Assisted	2011	●	●	○	●	●	●

Note: The ○ symbol denotes a low degree for that comparison item, ◐ denotes a medium degree, and ● denotes a high degree.

state can be observed), *easiness* (how easy it is to implement this approach), *practicality* (how usable and adoptable this is for real-world applications), and *automation* (how much can be done automatically instead of by hand). We order these approaches based on the year they first appeared.

Debugger assisted. The debugger-assisted approach uses the debugging facilities to help provide semantic information about the guest OS. Since it relies on access to debug info, it has only *moderate flexibility*, but the rich amount of data from debugging information also provides *high coverage*. Out-of-VM tools are relatively easy to write for this approach (*high easiness*), as it can use existing OS debugging tools (*high automation*). For instance, LIVEWIRE directly reuses the available built-in commands to retrieve the kernel semantic information. With the debugging information, the monitoring tools can also be used to analyze a snapshot of the VM after obtaining system information. Obtaining debug information, however, may involve recompiling the source in debug mode or running a special debug version of the OS if the source is not available. It is not always practical to expect a user to run a debug version of an OS, and such an OS often has to be recompiled from kernel source code. Due to these weaknesses, this approach has *low practicality*.

Manual. The manual approach can be taken regardless of what resources are available or constraints exist, and therefore it has *high flexibility*. It can analyze source code or debugging information, or it can even reverse engineer the binary code to determine the forms and locations of kernel data structures of interest. However, in this case, this approach has *low coverage* because each data structure must be manually identified. Alternatively, usable definitions of data structures may already exist. Out-of-VM tools have to be manually developed from scratch, resulting in *low automation*. However, although it is tedious, it is not difficult to create these tools (*high easiness*). This approach has *moderate practicality*, as it can be conducted by anyone under most circumstances, but its low scalability could hinder its practical usage.

Compiler assisted. The compiler-assisted approach requires access to the source code of the OS, meaning that it has *low flexibility* and *low practicality*. Because it attempts to automate the process by using the compiler on the source to follow potential program flow and reason about the abstractions, it has *high automation*. Some data structures can be automatically generated by following static information in the source, whereas some will also depend on certain values at runtime. Besides the complete view of static data, some implementations have managed to track dynamic data, and therefore this approach has *high coverage*. However, a compiler pass is not easy to implement and dynamic data can also be hard to determine, so it has *moderate easiness*.

Guest assisted. The guest-assisted approach often installs a helper program inside a guest OS or modifies the existing system to export the useful information to out-of-VM monitors. Since this approach avoids the semantic gap problem, it has *high*

easiness. It has *moderate flexibility*, as it either requires the authorization of the guest OS to install the helper software or recompiles the kernel. These requirements lead to *moderate practicality*. If it needs to, it can cover almost everything (*high coverage*), because it can access all of the guest OS abstractions. The automation is not as high as purely out-of-VM solutions (*moderate automation*), as it requires a step to install software in the guest OS or recompile the guest OS kernel.

Binary analysis assisted. The binary analysis–assisted approach is used when only binary code is available, and therefore it has *high flexibility*. It attempts to determine abstractions through either static analysis or dynamic analysis by observing the behavior of compiled code as it runs. As such, it can be used to effectively introspect closed-source OSes. Binary code analysis can be performed both in real time in a live system or can be used to train tools for later use. The approach is much harder to implement (*low easiness*), but it has *high practicality* in that it can perform many out-of-VM monitoring functions. Being able to reason the abstractions faithfully from the binary code and even reuse certain legacy binaries, it also has *high coverage* and *high automation*.

5. APPLICATIONS

Over the past decade, out-of-VM monitoring has been adopted for many security applications due to the huge benefits of moving in-VM monitors out of VM (as discussed in Section 2). In general, we can classify the security applications into attack detection (Section 5.1), attack prevention (Section 5.2), and attack recovery (Section 5.3); these are the three dimensions of responding to attacks [Bishop 2002]. We will discuss how out-of-VM solutions have been applied in these dimensions in greater detail.

5.1. Detection

The first step for any monitoring system is to collect the state S to allow M to make decisions (recall the definition in (1)). The observed state can be of the kernel level or user level and may be benign or malicious. Any observed changes can be classified as either code modification or data modification.

After observing benign or malicious behaviors in a system, we can use the information obtained from these observations to detect attacks. Based on where attacks occur, we classify them into either kernel- or user-level attacks. In addition, based on whether attacks tamper with code or data, we classify them as code attacks or data attacks. In addition, out-of-VM monitors can also be used purely for behavior analysis (e.g., malware analysis) and forensic investigations (after an attack has happened). In the following paragraphs, we review these specific applications.

Kernel level. Kernel-level attacks have administrative privileges and can tamper with both the OS kernel and application processes. Kernel rootkits are examples of such attacks. Detecting kernel anomalies often requires building a model that captures benign kernel behavior. Any deviation from benign behavior can be flagged as attacks. Several systems are concerned with observing benign kernel data structures. In particular, SBCFI [Petroni and Hicks 2007], KOP [Carbone et al. 2009], and MAS [Cui et al. 2012] statically analyze kernel source code to build the benign kernel data structure graph. OSCK [Hofmann et al. 2011] also first uses static analysis to extract the heap management data structure of benign kernels. LIVEDM [Rhee et al. 2010] uses dynamic analysis to build a kernel heap graph. DGSTG [Dolan-Gavitt et al. 2009] applies fuzzing techniques to derive value-invariant signatures of kernel data structures, and SIGGRAPH [Lin et al. 2011] uses benign snapshots and kernel data structures extracted from source code to derive the graph signatures.

Another option is to observe the malicious behavior caused by kernel malware (for misuse detection). HOOKFINDER [Yin et al. 2008] observes how hook behavior gets

affected by kernel malware. KTRACER [Lanzi et al. 2009] extracts malicious behaviors from kernel rootkits; POKER [Riley et al. 2009] and RKPROFILER [Xuan et al. 2009] also dynamically observe how rootkits execute in OS kernels. This information is often helpful for deriving the effective signatures to capture the attacks.

User level. There are also numerous user-level attacks, such as viruses, worms, or spyware, that harm the system at the user level. To determine whether a process is malicious, a monitor has to analyze its behavior. Although in-VM solutions might use something like `strace` to trace in-VM system call behaviors, out-of-VM solutions can also achieve such functionality. For example, VMSCOPE [Jiang and Wang 2007] interprets system call events at the hypervisor layer, and process outgrafting (SWJX) [Srinivasan et al. 2011] uses dual VMs to trace the system call behavior of benign processes. EKKYS [Egele et al. 2007] analyzes browser plugins to detect spyware. PANORAMA [Yin et al. 2007] analyzes the whole systemwide information flow to analyze the malicious behavior inflicted by malware.

Code modification. Code modification detection involves the identification of kernel code changes. By performing hashing of kernel text, periodic recalculation is done to determine whether kernel text has changed—LIVEWIRE [Garfinkel and Rosenblum 2003] and COPILOT [Petroni et al. 2004] use hashing to detect any unauthorized code modification. SBCFI [Petroni and Hicks 2007], PATAGONIX [Litty et al. 2008], HIMA [Azab et al. 2009], RKPROFILER [Xuan et al. 2009], POKER [Riley et al. 2009], OSCK [Hofmann et al. 2011], VIGILARE [Moon et al. 2012], and KI-MON [Lee et al. 2013] also feature such *strong* code modification detection. In contrast, some systems offer very *weak* detection in code modification, such as ANTFARM [Jones et al. 2006], which is just a framework, and therefore users must extend it to obtain sufficient detection.

Data modification. Data modification includes changes to critical kernel objects such as function pointers and system call tables. Data modifications can be detected by a signature-based approach [Dolan-Gavitt et al. 2009; Petroni et al. 2004]. Kernel memory can also be traversed to discover kernel data structures that contain information about the kernel state—for example, traversing `task_struct` to obtain a list of processes or `module list` to obtain a list of loaded kernel modules. PFWA [Petroni et al. 2006], VMWATCHER [Jiang et al. 2007], KOP [Carbone et al. 2009], DGSTG [Dolan-Gavitt et al. 2009], POKER [Riley et al. 2009], RKPROFILER [Xuan et al. 2009], LIVEDM [Rhee et al. 2010, 2011], SIGGRAPH [Lin et al. 2011], VIRTUOSO [Dolan-Gavitt et al. 2011a], VMST [Fu and Lin 2012], MAS [Cui et al. 2012], BLACKSHEEP [Bianchi et al. 2012], SENTRY [Srivastava and Giffin 2012], and KI-MON [Lee et al. 2013] are examples of systems that detect kernel data modification. Among them, PFWA, KOP, POKER, RKPROFILER, LIVEDM, MAS, and KI-MON have *strong* data detection because they can even detect the modification of dynamic kernel objects.

Besides directly modifying the function pointer or system call table, there is another type of attack that can be triggered by exploiting memory errors such as buffer overflows. To detect such a type of attack, KRUISER [Tian et al. 2012] instruments and recompiles the kernel source code, and at the hypervisor layer, it detects whether there are any heap overflows.

Forensics and analysis. Forensics involves investigative analysis of how a system was attacked after the fact. The first step for any forensics investigation is to collect the state of an infected system. Signature-based scanning (e.g., DGSTG and SIGGRAPH) or other systems such as TRAILOFBYTES [Krishnan et al. 2010], VIRTUOSO [Dolan-Gavitt et al. 2011a], SHELLOS [Snow et al. 2011], and VMST [Fu and Lin 2012] can all be used to perform forensic analysis on the victim VM.

Out-of-VM monitors have also been used for malware analysis. For instance, ANUBIS [Bayer et al. 2009] uses the VMM to observe system calls and APIs executed by malware, and it supports other advanced program analysis such as slicing and taint

analysis. Ether [Dinaburg et al. 2008] provides a high-fidelity environment to analyze malware. Ekkys [Egele et al. 2007], Panorama [Yin et al. 2007], and V2E [Yan et al. 2012] can also be used in such a scenario, and so can malware profilers such as Ktracer [Lanzi et al. 2009], PoKeR [Riley et al. 2009], Rkprofiler [Xuan et al. 2009], and Moss [Prakash et al. 2013, 2014].

5.2. Prevention

The ultimate goal for any security mechanism is to prevent attacks. This often requires active monitoring and the ability to intercept an attack. Similarly to our classification of detection applications in Section 5.1, we also classify existing prevention applications into kernel- and user-level protection, as well as code and data protection.

Kernel level. In addition to kernel code and data protection described later, several projects aim to separate kernel modules from the rest of the kernel, because kernel modules are often the vulnerable points of an OS kernel. In particular, kernel module isolation often allocates special memory address spaces for kernel modules. Verification is done to ensure that the kernel modules do not perform any unauthorized access to the kernel area. This helps to protect the kernel from malicious modules. HIMA [Azab et al. 2009] and Huko [Xiong et al. 2011] use hardware-assisted paging to isolate kernel extensions from the kernel. Gateway [Srivastava and Giffin 2011] isolates devices drivers in a similar way and at the same time patches indirect calls in driver code for the interception.

User level. Similarly to kernel module isolation, process isolation is achieved by running a process in an isolated address space, which is protected by a hypervisor from unauthorized modification. Implementations include Slcl [Sharif et al. 2009] and HIMA [Azab et al. 2009]. Although in SIM this technique is used to protect the monitoring process, the idea can be extended for use in monitoring and isolating ordinary processes as well, as shown in HIMA.

Code protection. Code protection involves protecting the kernel text by making the kernel text memory region unchangeable. Another technique under code protection is to encrypt the guest OS memory to prevent unauthorized access. For instance, OverShadow [Chen et al. 2008] encrypts the guest VM pages such that out-of-VM tampering becomes impossible. In another approach, SecVisor [Seshadri et al. 2007] uses hardware-assisted memory protection and verifies code integrity as well. Patagonix [Litty et al. 2008] verifies the authenticity of the code before it is allowed to execute, as does HIMA. These code protections are very *strong*.

Data protection. Data protection aims to protect data structures, system call tables, and kernel objects in memory, including both static and dynamic data. Protecting dynamic data involves traversing the kernel heap to discover dynamic data objects. HookSafe [Wang et al. 2009] moves all kernel hooks to a separate address space and prevents memory access writes to this region. Although HookSafe protects the kernel hooks completely, it does not protect other kernel data. It thus has *moderate* completeness, as shown in Table II. NICKLE [Riley et al. 2008], Lares [Payne et al. 2008], OSVM [Oliveira and Wu 2009], and Sentry [Srivastava and Giffin 2012] all use memory access restrictions to protect kernel data. OverShadow [Chen et al. 2008] encrypts the pages to defeat any malicious process attempting to view critical data. It hence offers *strong* protection, as it is very challenging for attackers to modify the data.

5.3. Recovery

The ability of a system to automatically recover from an attack helps reduce the need of human intervention whenever there is a problem. Once we have detected an attack, if it is possible to directly fix it, the system can remain online without any interruption in service. This will be particularly useful for services that require high availability.

Table V. Definitions of the Metrics Used to Compare Out-of-VM Monitor Deployment Types

Metric	Definition
Flexibility	How many constraints are imposed on the monitor
Security	How well the deployment type provides for security coverage
Invisibility	How difficult the presence of the monitor is to detect from within the VM
Speed	How much system slowdown occurs compared to no monitor running
Space	How much storage capability the deployment type possesses

Table VI. Comparison Between Different Out-of-VM Monitor Deployment Types

Approaches	Flexibility	Security	Invisibility	Speed	Space
Bare Metal	◐	◐	◐	●	●
Hosted, Native Hypervisor	●	◐	◐	●	●
Hosted, Emulation Hypervsior	●	◐	◐	○	●
Extra Hardware	○	●	●	●	○

Note: The ○ symbol denotes a low degree for that comparison item, ◐ denotes a medium degree, and ● denotes a high degree.

However, as shown in Table II, even among repairable attacks there has been significantly less attention in attack recovery.

VICI [Fraser et al. 2008] made an early attempt for VMM-based guest OS recovery. It performs code repair that reverts kernel text to its original state once modified kernel code has been detected. It also can use a rollback mechanism, which involves periodically taking a snapshot of the VM as it runs. These snapshots can be used to restore the VM to a known good state in the event of an attack. PMPDFGJ [Paleari et al. 2010] automatically generates malware remediation procedures using an emulated VM for malware attacks, then executes them to restore the infected machine to a good state. GDXJ [Gu et al. 2011] is able to inject a cleanup process into the guest OS and hence features some recovery capability. EXTERIOR [Fu and Lin 2013b] repairs a limited number of kernel objects, such as the system call tables, through cross-machine checking. We must note that none of the existing work can achieve a *high* degree of recovery except with rollback, as there are certainly some attacks that cannot be fixed and require either a system reinstall or restoring from an earlier snapshot.

6. DEPLOYMENT

Another important consideration when implementing an out-of-VM monitor is where it will be deployed. Factors considered when determining which layer to deploy to include flexibility, security, visibility, speed, and space constraints. The detailed definition about these metrics is presented in Table V. There are advantages and disadvantages to each deployment type, and we have categorized the different deployment methods in Table VI. Note that all hypervisor deployments have *high space* because they are not working with limited memory and *moderate invisibility* due to hypervisor detection techniques such as timing analysis. In the following, we discuss them in greater detail.

Bare metal. One deployment method uses bare metal hypervisors such as Xen. Such deployment offers *moderate flexibility* and improved performance (*high speed*) because the guest code is run directly on the hardware. It also allows for monitoring of privileged instructions and hardware events. This deployment lacks the degree of security coverage compared to some of the other deployment methods (*moderate security*), however, as it sees a snapshot view of the guest OS only when certain kernel events occur (e.g., certain interrupts). Typical implementations include SECVISOR [Seshadri et al.

2007], Ether [Dinaburg et al. 2008], XenAccess [Payne et al. 2007], MAVMM [Nguyen et al. 2009], and Huko [Xiong et al. 2011], among others.

Hosted, native hypervisor. Another deployment method is to use a hosted, native hypervisor such as KVM. This type of hypervisor offers the *high flexibility* of using host OS abstractions to implement the hypervisor service. It also allows for the full-speed execution of guest code (*high speed*), although it still shares the same snapshot view as bare metal hypervisors (*moderate security*). Implementations that use this type of hypervisor include Slcl [Sharif et al. 2009], HookSafe [Wang et al. 2009], Gateway [Srivastava and Giffin 2011], OSck [Hofmann et al. 2011], ShellOS [Snow et al. 2011], and SWJX [Srinivasan et al. 2011].

Hosted, emulation hypervisor. Unlike the two previous methods, deploying alongside a hosted emulation hypervisor offers the ability of full software control of the monitoring service. A typical hypervisor for this type of deployment is QEMU or the earlier versions of VMware Workstation and VirtualBox. Emulation hypervisors allow all of the instructions to be intercepted, which offers *high flexibility* and good security coverage. However, this technique suffers from performance degradation (*low speed*) because of the overhead of emulating instructions at the software layer. This slowdown also results in *moderate security* due to it increasing the feasibility of timing attacks. Implementations using this deployment method include Livewire [Garfinkel and Rosenblum 2003], VMwatcher [Jiang et al. 2007], HookMap [Wang et al. 2008], and Vmst [Fu and Lin 2012].

In addition, from Table II, we can observe that there are some systems that are agnostic to any particular type of hypervisor and can be deployed into any of them. This is especially true for memory-based monitoring systems, such as Sbcfi [Petroni and Hicks 2007], Gibraltar [Baliga et al. 2008], Kop [Carbone et al. 2009], Dgstg [Dolan-Gavitt et al. 2009], SigGraph [Lin et al. 2011], Virtuoso [Dolan-Gavitt et al. 2011a], and Mas [Cui et al. 2012].

Extra hardware. A more secure method is to deploy the out-of-VM monitoring on external hardware. By using a separate device for deploying the monitoring service, the risk of attack is greatly reduced. The primary shortcoming is that this deployment approach is less flexible (*low flexibility*). Many external devices may also have limited memory, and this limits the capabilities of a monitor (*low space*). External hardware involved in this type of deployment includes PCI cards, which have been used in Copilot [Petroni et al. 2004], Pfwa [Petroni et al. 2006], and HyperCheck [Wang et al. 2010]. Extra hardware may not always have sufficient space due to the high cost compared to other deployments, but it can offer comparatively *high security*, *high invisibility*, and *high speed*. In addition, recently there have been special extra hardware components designed to integrate with the system bus to verify the kernel integrity using a bus snooping technique (e.g., Lee et al. [2013] and Moon et al. [2012]).

7. DISCUSSIONS AND FUTURE DIRECTIONS

Although there has been a large amount of research in out-of-VM monitoring, there are still lots of opportunities for future research. In this section, we discuss the remaining research problems and shed light on future directions.

7.1. Improving Hypervisor Security

When performing out-of-VM monitoring, it is important to determine what to trust when designing tools to derive guest OS abstractions, as introspection depends on the integrity of the reconstructed data to understand the state of the guest OS. Jain et al. [2014] recently emphasized that many introspection techniques make assumptions about the guest OS that may allow a compromised or malicious guest OS to circumvent detection. These assumptions include the OS being benign during training, that

it is possible for abstractions to be learned by an automated process (for nonmanual approaches), that attacks are long-lived enough to be detected (if only checking system state periodically), and that it is feasible to whitelist modules in deployment. For example, as mentioned earlier, the snapshot view of native hypervisors might miss highly transient attacks. Depending on the approaches taken, the validity of assumptions varies. It will be important for future out-of-VM solutions to clearly state their security assumptions and expected use cases to better define their capabilities.

On the other hand, the security of out-of-VM monitors relies on the security of hypervisors. As discussed in Section 2.2, out-of-VM has significantly fewer attack vectors compared to in-VM solutions, but recently there have been attacks that aim to compromise hypervisors and carry out malicious functionality more stealthily. Notable attacks include Blue Pill [Laurie and Singer 2008], SMM-based rootkits [Embleton et al. 2008], PCI-based rootkits [Heasman 2006], SubVirt [King et al. 2006], SubXen [Wojtczuk 2008], and reactive VMI [Fu et al. 2013].

To combat the threat against hypervisors, several solutions from different angles have been proposed. They either push the out-of-VM monitors one layer down into logically isolated hardware, improve the security of the hypervisor, reduce the code base and attack surface of hypervisors, or force the OS to verify itself. More specifically:

—*Pushing one layer down*: Since the hypervisor layer can secure the execution of the guest OS one layer above it, naturally we can protect hypervisor execution by pushing the monitor one layer down into the hardware. Recent efforts include HyperSentry [Azab et al. 2010], CloudVisor [Zhang et al. 2011], HyperCoffer [Xia et al. 2013], and mGuard [Liu et al. 2013]. HyperSentry leverages the system management mode (SMM) of the x86 system to monitor hardware state. SMM can access the host memory and CPU registers to check the integrity of a running system. CloudVisor introduces a tiny security monitor underneath the commodity hypervisor using nested virtualization to protect both the VMMs and the VMs. To guard the privacy and integrity of the guest VMs, HyperCoffer introduces a VM-Shim mechanism in between a guest VM and the hypervisor using memory encryption and integrity checking. With a drop-in memory controller, mGuard monitors the memory traffic to prevent any illegal modifications of the hypervisor code.

—*Improving hypervisor code*: Another solution is to improve the hypervisor code. Formal methods and type verification have been applied to prove the absence of certain vulnerabilities as shown in SeL4 [Klein et al. 2009] and Verve [Yang and Hawblitzel 2010]. In addition, as it is challenging to eliminate the software vulnerabilities in hypervisor code, HyperSafe [Wang and Jiang 2010] instruments the hypervisor code and then runtime verifies its integrity. Recently, Xmhf [Vasudevan et al. 2013] proposes an extensible and modular hypervisor framework that can verify the hypervisor memory integrity and ensure that the hypervisor memory is not modified by software running at a lower privilege level.

—*Deprivileging the hypervisors*: The third solution is to minimize the trusted computing base (TCB) of the hypervisor. Nova [Steinberg and Kauer 2010] implements a thin bare metal hypervisor that moves the virtualization support to user level, resulting in at least one order of magnitude smaller TCB size than that of existing systems. NoHype [Keller et al. 2010; Szefer et al. 2011] eliminates the hypervisor layer by strictly partitioning the hardware resources and rewriting the guest OS kernel. HyperLock [Wang et al. 2012] creates a separate address space to confine the hypervisor execution. DeHype [Wu et al. 2013] directly deprivileges most of the hypervisor code without any cooperation from the guest OS kernel.

—*Paraverification*: Another solution, related to the concept of paravirtualization, considers verification even in the face of a potentially malicious or compromised OS.

This solution requires slight modifications to the OS, as it must be designed to report its activities to the hypervisor. Even though the OS cannot be trusted, if it is forced to report on its own behavior, the hypervisor may check for inconsistencies. Work still needs to be done in this area to determine the potential of this approach [Jain et al. 2014].

7.2. Providing High-Fidelity Hypervisors

Similar to the "cat and mouse" games of in-VM attacks and in-VM defenses, by detecting the presence of a hypervisor's execution, attackers can change their behaviors so as to evade the out-of-VM monitor. For instance, by performing timing analysis of instruction execution, an emulation hypervisor can be easily detected (because of its slow speed) [Garfinkel and Rosenblum 2003; Jiang et al. 2007]. Additionally, by observing unusual changes to memory and CPU register values, malware can detect the presence of certain hypervisors [Dinaburg et al. 2008].

To combat such monitoring-aware attacks, one viable approach is to develop efficient detection techniques (e.g., behavior differencing [Balzarotti et al. 2010]) or stealthily fight against malware (e.g., Vasudevan and Yerraballi [2006]), whereas the other direction is to provide high-fidelity hypervisors. Ether [Dinaburg et al. 2008] has made a first step in removing side effects that are unconditionally detectable by malware, resulting in a highly transparent monitoring system. PokeEMU [Martignoni et al. 2012] leverages path lifting and symbolic execution to cross validate the implementation of emulation hypervisors, thereby improving their fidelity. TxIntro [Liu et al. 2014] uses the hardware transactional memory (HTM) to actively monitor updates to critical kernel data structures. With the HTM, it can achieve concurrent, timely, and consistent introspection of guest VMs. Despite the progress being made, tremendous efforts are still required to reach the stage of having a high-fidelity and highly transparent hypervisor.

7.3. Complete Memory Monitoring

Although a hypervisor can access all of the physical memory of a guest OS, the memory it observes unfortunately is incomplete. Specifically, if a certain page is not used and the OS is starving for memory, it usually swaps out the pages from physical memory to disk, resulting in pages that are invisible to the hypervisor. Whereas the Linux kernel does not swap out kernel memory, the Windows kernel does. To have a complete view of the entire guest OS including both the kernel and its running applications, we have to address swapped-out memory issues.

Although guest OS–assisted approaches (e.g., GDXJ [Gu et al. 2011], Slcl [Sharif et al. 2009], Lares [Payne et al. 2008]) can solve this problem by injecting an agent to access the virtual addresses (they will be automatically swapped back by the guest OS), we are not aware of any pure out-of-VM solutions that have solved this problem. Intuitively, it requires the nontrivial effort of traversing the guest OS page swapping data structures as well as additional progress from disk introspection (to be discussed in Section 7.4). It would not be a surprise if there were more research efforts in this direction in the future.

7.4. Complete Disk Monitoring

Interestingly, to our knowledge, all of the current out-of-VM monitors primarily focus on memory, and except for XenAccess [Payne et al. 2007], which provides a library to access VM disk images, and VMwatcher [Jiang et al. 2007], which also examines disk files, little light has been shed on accessing VM disk data, including swapped memory. However, disk data does contain much valuable information about the state of an OS,

considering that nearly all in-VM antivirus software today actually scans both disk and memory to find viruses.

One may argue that to introspect the disk data of a guest OS is trivial, as we can directly mount it to a second computer and use a native disk scanning tool to analyze its state. Unfortunately, this only works if the disk has a known file system, as shown in XenAccess [Payne et al. 2007] and VMwatcher [Jiang et al. 2007], and if the files have not been encrypted. If the guest OS uses a private unknown file system, or the files have been encrypted by the underlying file systems such as with full disk encryption, which tends to be a common practice for publicly outsourced VMs in today's cloud, this introduces significant challenges, and consequently more research efforts are needed to address this problem.

7.5. Comparable Performance Metrics

Unfortunately, in our study, we found that many of the current solutions for out-of-VM introspection have performance evaluations that are not comparable to other solutions. Usually, a solution determines its performance overhead using various calculations, benchmarks, or common tasks, but no standard set of tests are used. In addition, unless identical hardware and underlying software (e.g., the hypervisor or OS) are used, comparing results for the same test is less useful. In addition to time overhead, it is also important to be able to compare code size to determine how much additional attack surface a solution adds, and not all works give this information. Therefore, it may be useful for future work to create some form of standard testing metrics that can allow researchers to directly compare solutions.

7.6. Beyond Read-Only Introspection

Nearly all past research in out-of-VM monitoring has been read only, mostly avoiding making changes to the guest OS. This is reasonable because unless a VMM knows exactly which guest addresses can be safely modified and when it is safe to write to them, it runs the risk of disrupting kernel state or even crashing the kernel.

Due to the substantial advances in bridging the semantic gap, recent implementations have nearly complete semantic information about the guest OS, and thus we have the opportunity to go beyond traditional read-only introspection. Considering the power of the hypervisor layer, it is possible to make more advanced guest OS modifications than in-VM security systems can. As demonstrated by Exterior [Fu and Lin 2013b], it is possible to perform operations such as configuring kernel parameters, changing the IP routing table, or even killing malicious processes.

The most appealing aspect of writable VMI is its ability to immediately respond to and prevent attacks without any cooperation from the guest OS or guest OS applications. It does not require root privileges from the guest OS to perform root-level operations, such as `kill`-ing a rootkit-created hidden process or running `rmmod` against a hidden malicious kernel module. Considering that there are many read-only VMI implementations (e.g., Dolan-Gavitt et al. [2011a], Fu and Lin [2012], Garfinkel and Rosenblum [2003], Jiang et al. [2007], and Jones et al. [2008]), it should be possible to merge writable VMI seamlessly with them. Given the incentives of writable VMI and advances in bridging the semantic gap, we expect that there will be more research in this direction.

7.7. Beyond the Guest OS Kernel and Traditional Platform

Currently, the majority of out-of-VM monitors are interested in the behavior of the guest OS kernel, which is just one layer above the monitor. As a progression of this concept, a natural question is why not monitor two layers up to observe application-level behavior running inside the guest OS? In fact, this certainly is possible. A recent

work, TZB [Dolan-Gavitt et al. 2013], has demonstrated that by identifying the hook points of a Web browser through highly automated program analysis, we are able to observe Web browser behavior at the hypervisor layer through active monitoring (e.g., Payne et al. [2008]). We believe that there will be more future efforts pushing forward on the monitoring of guest OS applications, such as Web browsers.

In addition, whereas current out-of-VM monitoring primarily focuses on the traditional platform, there is a pressing need on mobile platforms. One earlier attempt was DroidScope [Yan and Yin 2012], which monitors both the Android kernel and the apps running in the Dalvik VM. Given more mobile devices with virtualization support, and more stealthy malware fighting into mobile OS kernels [Zhou and Jiang 2012], it will be no surprise that more out-of-VM monitors will be developed to introspect mobile devices. For instance, a recent effort, Tz-Rkp [Azab et al. 2014], has demonstrated the use of hypervisors to host kernel protection tools for the Android platform.

8. RELATED WORK

Recently, there have been several efforts focusing on systemizing the knowledge of VMI, especially from the forensics and secure cloud computing perspective. In particular, Kuhn and Taylor [2011] reviewed introspection for the specific purpose of forensics. They also discussed the various ways in which VMI is designed and implemented, and gave an overview of the applications of VMI rather than different implementation techniques.

Denz and Taylor [2013] summarized introspection from the perspective of cloud security. They focused first on the threat model for VMs in the cloud, including current detection and prevention techniques. They then discussed secure hypervisors, various attempts at hardening hypervisors, and cloud resilience. They ended with an attempt to make a comparison of attack surface size and performance overhead, but they could only compare code size for open source systems, and performance overhead data was missing for some systems.

A similar work to ours is Jain et al. [2014], which also focused on the semantic gap problem in VMI. However, their paper focused on trust, security concerns, and attacks, whereas our work performs a more systematic study, classification, and summary of existing approaches, as well as an overview of potential applications and deployment strategies, which we believe will be useful for researchers attempting to contribute to the field.

9. CONCLUSION

Out-of-VM monitoring has been an appealing alternative to in-VM monitoring since the first day it emerged. Over the past decade, a significant amount of research has been carried out to extend the security applications of out-of-VM monitors and make them more practical and efficient. In this article, we systematized the knowledge in the domain of out-of-VM monitoring by examining and classifying the different approaches that have been proposed to overcome the semantic gap and develop various security applications. Specifically, we reviewed how the existing approaches bridge the semantic gap while addressing different constraints such as flexibility, coverage, practicality, and automation; how they have developed different monitoring systems; and how the monitoring systems have been applied and deployed. For the future of out-of-VM monitoring, we believe that there are several interesting avenues to explore, such as protecting the hypervisor itself, having complete memory and disk monitoring, and developing more introspection capabilities including writable VMI for both kernel- and user-level programs, as well as expanding the out-of-VM monitors for mobile platforms.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This research was partially supported by AFOSR grant FA9550-14-1-0119 and NSF grant 1453011. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily those of the AFOSR and NSF.

REFERENCES

Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. ACM, New York, NY, 340–353. DOI: http://dx.doi.org/10.1145/1102120.1102165

Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. 2–13. DOI: http://dx.doi.org/10.1145/1168919.1168860

David Anderson. 2003. White Paper: Red Hat Crash Utility. Retrieved July 1, 2015, from http://people.redhat.com/anderson/crash_whitepaper/.

Ahmed M. Azab, Peng Ning, Emre C. Sezer, and Xiaolan Zhang. 2009. HIMA: A hypervisor-based integrity measurement agent. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'09)*. 461–470. DOI: http://dx.doi.org/10.1109/ACSAC.2009.50

Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*. ACM, New York, NY, 90–102. DOI: http://dx.doi.org/10.1145/2660267.2660350

Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. 2010. HyperSentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. 38–49. DOI: http://dx.doi.org/10.1145/1866307.1866313

Arati Baliga, Vinod Ganapathy, and Liviu Iftode. 2008. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC'08)*. IEEE, Los Alamitos, CA, 77–86. DOI: http://dx.doi.org/10.1109/ACSAC.2008.29

Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2010. Efficient detection of split personalities in malware. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*. http://www.isoc.org/isoc/conferences/ndss/10/pdf/24.pdf.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. 164–177. DOI: http://dx.doi.org/10.1145/945445.945462

Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Krügel, and Engin Kirda. 2009. Scalable, behavior-based malware clustering. In *Proceedings of the 2009 Annual Network and Distributed System Security Symposium (NDSS'09)*. http://www.isoc.org/isoc/conferences/ndss/09/pdf/11.pdf.

Antonio Bianchi, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2012. Blacksheep: Detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. ACM, New York, NY, 341–352. DOI: http://dx.doi.org/10.1145/2382196.2382234

Matt Bishop. 2002. *Computer Security: Art and Science*. Addison-Wesley Professional.

Martim Carbone, Matthew Conover, Bruce Montague, and Wenke Lee. 2012. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses (RAID'12)*. 22–41. DOI: http://dx.doi.org/10.1007/978-3-642-33338-5_2

Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. 2009. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*. 555–565. DOI: http://dx.doi.org/10.1145/1653662.1653729

Peter M. Chen and Brian D. Noble. 2001. When virtual is better than real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HOTOS'01)*. 133. http://dl.acm.org/citation.cfm?id=874075.876409

Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. 2008. 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)*. ACM, New York, NY, 2–13. DOI: http://dx.doi.org/10.1145/1353536.1346284

Weidong Cui, Marcus Peinado, Zhilei Xu, and Ellick Chan. 2012. Tracking rootkit footprints with a practical memory analysis system. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security'12)*. 42–42. http://dl.acm.org/citation.cfm?id=2362793.2362835

Robert Denz and Stephen Taylor. 2013. A survey on securing the virtual cloud. *Journal of Cloud Computing* 2, 1, 1–9.

Edsger W. Dijkstra. 1968. The structure of the THE-multiprogramming system. *Communications of the ACM* 11, 5, 341–346.

Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*. 51–62. DOI: http://dx.doi.org/10.1145/1455770.1455779

Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. 2013. Tappan Zee (north) bridge: Mining memory accesses for introspection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'13)*. ACM, New York, NY. 839–850. DOI: http://dx.doi.org/10.1145/2508859.2516697

Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011a. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA, 297–312. DOI: http://dx.doi.org/10.1109/SP.2011.11

Brendan Dolan-Gavitt, Bryan Payne, and Wenke Lee. 2011b. *Leveraging Forensic Tools for Virtual Machine Introspection*. Technical Report GT-CS-11-05. Georgia Institute of Technology.

Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. 2009. Robust signatures for kernel data structures. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'09)*. ACM, New York, NY. 566–577. DOI: http://dx.doi.org/10.1145/1653662.1653730

Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. 2007. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference (ATC'07)*. Article No. 18. http://dl.acm.org/citation.cfm?id=1364385.1364403

Shawn Embleton, Sherri Sparks, and Cliff Zou. 2008. SMM rootkits: A new breed of OS independent malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks (SecureComm'08)*. Article No. 11. DOI: http://dx.doi.org/10.1145/1460877.1460892

Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. 1996. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy (SP'96)*. 120–128. DOI: http://dx.doi.org/10.1109/SECPRI.1996.502675

Timothy Fraser, Matthew R. Evenson, and William A. Arbaugh. 2008. VICI virtual machine introspection for cognitive immunity. In *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC'08)*. 87–96. DOI: http://dx.doi.org/10.1109/ACSAC.2008.33

Yangchun Fu and Zhiqiang Lin. 2012. Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA, 586–600. DOI: http://dx.doi.org/10.1109/SP.2012.40

Yangchun Fu and Zhiqiang Lin. 2013a. Bridging the semantic gap in virtual machine introspection via online kernel data redirection. *ACM Transactions on Information and System Security* 16, 2, Article No. 7. DOI: http://dx.doi.org/10.1145/2505124

Yangchun Fu and Zhiqiang Lin. 2013b. EXTERIOR: Using a dual-VM based external shell for guest-OS introspection, configuration, and recovery. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'13)*. 97–110. DOI: http://dx.doi.org/10.1145/2451512.2451534

Yangchun Fu, Zhiqiang Lin, and Kevin Hamlen. 2013. Subverting systems authentication with context-aware, reactive virtual machine introspection. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13)*. 229–238. DOI: http://dx.doi.org/10.1145/2523649.2523664

Yangchun Fu, Junyuan Zeng, and Zhiqiang Lin. 2014. HYPERSHELL: A practical hypervisor layer guest OS shell for automated in-VM management. In *Proceedings of the 2014 USENIX Conference (USENIX ATC'14)*. 85–96. http://dl.acm.org/citation.cfm?id=2643634.2643644

Tal Garfinkel and Mendel Rosenblum. 2003. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS'03)*. http://www.isoc.org/isoc/conferences/ndss/03/proceedings/papers/13.pdf.

Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. 2011. Process implanting: A new active introspection framework for virtualization. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems (SRDS'11)*. 147–156. DOI: http://dx.doi.org/10.1109/SRDS.2011.26

Brian Hay and Kara Nance. 2008. Forensics examination of volatile system data using virtual introspection. *SIGOPS Operating System Review* 42, 3, 74–82. DOI: http://dx.doi.org/10.1145/1368506.1368517

John Heasman. 2006. Implementing and Detecting a PCI Rootkit. Retrieved July 1, 2015, from http://www.blackhat.com/presentations/bh-dc-07/Heasman/Paper/bh-dc-07-Heasman-WP.pdf.

Jennia Hizver and Tzi-Cker Chiueh. 2014. Real-time deep virtual machine introspection and its applications. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'14)*. ACM, New York, NY, 3–14. DOI:http://dx.doi.org/10.1145/2576195.2576196

Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. 2011. Ensuring operating system kernel integrity with OSck. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. 279–290. DOI:http://dx.doi.org/10.1145/1950365.1950398

Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion. 2014. SoK: Introspections on trust and the semantic gap. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP'14)*. 605–620. DOI:http://dx.doi.org/10.1109/SP.2014.45

Xuxian Jiang and Xinyuan Wang. 2007. "Out-of-the-box" monitoring of VM-based high-interaction honeypots. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection (RAID'07)*. 198–218. DOI:http://dx.doi.org/10.1007/978-3-540-74320-0_11

Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. 2007. 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)*. 128–138. DOI:http://dx.doi.org/10.1145/1315245.1315262

Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2006. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the USENIX 2006 Annual Conference (ATEC'06)*. 1. http://dl.acm.org/citation.cfm?id=1267359.1267360

Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2008. VMM-based hidden process detection and identification using Lycosid. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'08)*. 91–100. DOI:http://dx.doi.org/10.1145/1346256.1346269

Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. 2005. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. ACM, New York, NY, 91–104. DOI:http://dx.doi.org/10.1145/1095809.1095820

Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. 2010. NoHype: Virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. 350–361. DOI:http://dx.doi.org/10.1145/1816038.1816010

Gene H. Kim and Eugene H. Spafford. 1994. The design and implementation of tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS'94)*. ACM, New York, NY, 18–29. DOI:http://dx.doi.org/10.1145/191177.191183

Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. 2006. SubVirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (SP'06)*. 314–327. DOI:http://dx.doi.org/10.1109/SP.2006.38

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, New York, NY, 207–220. DOI:http://dx.doi.org/10.1145/1629575.1629596

Srinivas Krishnan, Kevin Z. Snow, and Fabian Monrose. 2010. Trail of bytes: Efficient support for forensic analysis. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM, New York, NY, 50–60. DOI:http://dx.doi.org/10.1145/1866307.1866314

Stephen Kuhn and Stephen Taylor. 2011. *A Survey of Forensic Analysis in Virtualized Environments*. Technical Report. Dartmouth College, Hanover, NH.

Andrea Lanzi, Monirul I. Sharif, and Wenke Lee. 2009. K-Tracer: A system for extracting kernel malware behavior. In *Proceedings of the 2009 Annual Network and Distributed System Security Symposium (NDSS'09)*. http://www.isoc.org/isoc/conferences/ndss/09/pdf/12.pdf.

Ben Laurie and Abe Singer. 2008. Choose the red pill and the blue pill: A position paper. In *Proceedings of the 2008 Workshop on New Security Paradigms*. ACM, New York, NY, 127–133. DOI:http://dx.doi.org/10.1145/1595676.1595695

Hojoon Lee, Hyungon Moon, Daehee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. 2013. KI-Mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Proceedings of the 22nd USENIX Conference on Security*. 511–526. http://dl.acm.org/citation.cfm?id=2534766.2534810

Wenhao Li, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2015. Reducing world switches in virtualized environment with flexible cross-world calls. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, 375–387. DOI:http://dx.doi.org/10.1145/2749469.2750406

Zhiqiang Lin. 2013. Toward guest OS writable virtual machine introspection. *VMware Technical Journal* 2, 2.

Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. 2011. 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)*. http://www.isoc.org/isoc/conferences/ndss/11/pdf/3_3.pdf.

Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. 2008. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th Conference on Security Symposium (SS'08)*. 243–258. http://dl.acm.org/citation.cfm?id=1496711.1496728

Yutao Liu, Yubin Xia, Haibing Guan, Binyu Zang, and Haibo Chen. 2014. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE, Los Alamitos, CA, 416–427. DOI:http://dx.doi.org/10.1109/HPCA.2014.6835951

Ziyi Liu, JongHyuk Lee, Junyuan Zeng, Yuanfeng Wen, Zhiqiang Lin, and Weidong Shi. 2013. CPU transparent protection of OS kernel and hypervisor integrity with programmable DRAM. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. 392–403. DOI:http://dx.doi.org/10.1145/2485922.2485956

Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. 337–348. DOI:http://dx.doi.org/10.1145/2150976.2151012

Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. 2012. Vigilare: Toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. 28–37. DOI:http://dx.doi.org/10.1145/2382196.2382202

George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*. 213–228. http://dl.acm.org/citation.cfm?id=647478.727796

Anh M. Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T. King, and Hai D. Nguyen. 2009. MAVMM: Lightweight and purpose built VMM for malware analysis. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC'09)*. 441–450. DOI:http://dx.doi.org/10.1109/ACSAC.2009.48

Daniela Oliveira and Shyhtsun Felix Wu. 2009. Protecting kernel code and data with a virtualization-aware collaborative operating system. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC'09)*. 451–460. DOI:http://dx.doi.org/10.1109/ACSAC.2009.49

Roberto Paleari, Lorenzo Martignoni, Emanuele Passerini, Drew Davidson, Matt Fredrikson, Jon Giffin, and Somesh Jha. 2010. Automatic generation of remediation procedures for malware infections. In *Proceedings of the 19th USENIX Conference on Security (USENIX Security'10)*. 27. http://dl.acm.org/citation.cfm?id=1929820.1929856

Bryan D. Payne, Martim Carbone, and Wenke Lee. 2007. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*. 385–397. DOI:http://dx.doi.org/10.1109/ACSAC.2007.10

Bryan D. Payne, Martim Carbone, Monirul I. Sharif, and Wenke Lee. 2008. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA, 233–247. DOI:http://dx.doi.org/10.1109/SP.2008.24

Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. 2004. Copilot—a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*. 179–194. http://dl.acm.org/citation.cfm?id=1251375.1251388

Nick L. Petroni Jr., Timothy Fraser, AAron Walters, and William A. Arbaugh. 2006. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th Conference on USENIX Security Symposium*. Article No. 20. http://dl.acm.org/citation.cfm?id=1267336.1267356

Nick L. Petroni Jr. and Michael Hicks. 2007. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*. ACM, New York, NY, 103–115. DOI:http://dx.doi.org/10.1145/1315245.1315260

Gerald J. Popek and Robert P. Goldberg. 1974. Formal requirements for virtualizable third generation architectures. *Communications of the ACM* 17, 7, 412–421. DOI:http://dx.doi.org/10.1145/361011.361073

Aravind Prakash, Eknath Venkataramani, Heng Yin, and Zhiqiang Lin. 2013. Manipulating semantic values in kernel data structures: Attack assessments and implications. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-PDS'13)*. 1–12. DOI:http://dx.doi.org/10.1109/DSN.2013.6575344

Aravind Prakash, Eknath Venkataramani, Heng Yin, and Zhiqiang Lin. 2014. On the trustworthiness of memory analysis-an empirical study from the perspective of binary execution. *IEEE Transactions on Dependable and Secure Computing* 1, 1. DOI:http://dx.doi.org/10.1109/TDSC.2014.2366464

Junghwan Rhee, Zhiqiang Lin, and Dongyan Xu. 2011. Characterizing kernel malware behavior with kernel data access patterns. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS'11)*. 207–216. DOI:http://dx.doi.org/10.1145/1966913.1966940

Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. 2010. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection (RAID'10)*. 178–197. http://dl.acm.org/citation.cfm?id=1894166.1894179

Ryan Riley, Xuxian Jiang, and Dongyan Xu. 2008. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08)*. 1–20. DOI:http://dx.doi.org/10.1007/978-3-540-87403-4_1

Ryan Riley, Xuxian Jiang, and Dongyan Xu. 2009. Multi-aspect profiling of kernel rootkit behavior. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)*. 47–60. DOI:http://dx.doi.org/10.1145/1519065.1519072

Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. 2014. Hybrid-Bridge: Efficiently bridging the semantic-gap in virtual machine introspection via decoupled execution and training memoization. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. http://www.internetsociety.org/doc/hybrid-bridge-efficiently-bridging-semantic-gap-virtual-machine-introspection-decoupled.

Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, NY, 335–350. DOI:http://dx.doi.org/10.1145/1294261.1294294

Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. 2009. Secure in-VM monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*. 477–487. DOI:http://dx.doi.org/10.1145/1653662.1653720

Kevin Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. 2011. ShellOS: Enabling fast detection and forensic analysis of code injection attacks. In *Proceedings of the 20th USENIX Security Symposium*. http://static.usenix.org/events/sec11/tech/full_papers/Snow.pdf.

Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. 2011. Process out-grafting: An efficient "out-of-VM" approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*. ACM, New York, NY, 363–374. DOI:http://dx.doi.org/10.1145/2046707.2046751

Abhinav Srivastava and Jonathon Giffin. 2011. Efficient monitoring of untrusted kernel-mode execution. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*. http://www.isoc.org/isoc/conferences/ndss/11/pdf/3_2.pdf.

Abhinav Srivastava and Jonathon Giffin. 2012. Efficient protection of kernel data structures via object partitioning. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*. ACM, New York, NY, 429–438. DOI:http://dx.doi.org/10.1145/2420950.2421012

Udo Steinberg and Bernhard Kauer. 2010. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. 209–222. DOI:http://dx.doi.org/10.1145/1755913.1755935

Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. 2011. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*. ACM, New York, NY, 401–412. DOI:http://dx.doi.org/10.1145/2046707.2046754

Donghai Tian, Qiang Zeng, Dinghao Wu, Peng Liu, and Changzhen Hu. 2012. Kruiser: Semi-synchronized non-blocking concurrent kernel heap buffer overflow monitoring. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*. http://www.internetsociety.org/kruiser-semi-synchronized-non-blocking-con current-kernel-heap-buffer-overflow-monitoring.

Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. 2013. Design, implementation and verification of an eXtensible and modular hypervisor framework. In

Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP'13). IEEE, Los Alamitos, CA, 430–444. DOI:http://dx.doi.org/10.1109/SP.2013.36

Amit Vasudevan and Ramesh Yerraballi. 2006. Cobra: Fine-grained malware analysis using stealth localized-executions. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (SP'06)*. 264–279. DOI:http://dx.doi.org/10.1109/SP.2006.9

Jiang Wang, Angelos Stavrou, and Anup Ghosh. 2010. HyperCheck: A hardware-assisted integrity monitor. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection (RAID'10)*. 158–177. http://dl.acm.org/citation.cfm?id=1894166.1894178

Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP'10)*. 380–395. DOI:http://dx.doi.org/10.1109/SP.2010.30

Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. 2009. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*. 545–554. DOI:http://dx.doi.org/10.1145/1653662.1653728

Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. 2008. Countering persistent kernel rootkits through systematic hook discovery. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08)*. 21–38. DOI:http://dx.doi.org/10.1007/978-3-540-87403-4_2

Zhi Wang, Chiachih Wu, Michael Grace, and Xuxian Jiang. 2012. Isolating commodity hosted hypervisors with HyperLock. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, New York, NY, 127–140. DOI:http://dx.doi.org/10.1145/2168836.2168850

Rafal Wojtczuk. 2008. Subverting the Xen hypervisor. In *Proceedings of the Black Hat Technical Security Conference*.

Chiachih Wu, Zhi Wang, and Xuxian Jiang. 2013. Taming hosted hypervisors with (mostly) deprivileged execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'13)*. http://internetsociety.org/doc/taming-hosted-hypervisors-mostly-deprivileged-execution.

Rui Wu, Ping Chen, Peng Liu, and Bing Mao. 2014. System call redirection: A practical approach to meeting real-world virtual machine introspection needs. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*. 574–585. DOI:http://dx.doi.org/10.1109/DSN.2014.59

Yubin Xia, Yutao Liu, and Haibo Chen. 2013. Architecture support for guest-transparent VM protection from untrusted hypervisor and physical attacks. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. IEEE, Los Alamitos, CA, 246–257. DOI:http://dx.doi.org/10.1109/HPCA.2013.6522323

Xi Xiong, Donghai Tian, and Peng Liu. 2011. Practical protection of kernel integrity for commodity OS from untrusted extensions. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'11)*. http://www.isoc.org/isoc/conferences/ndss/11/pdf/3_1.pdf.

Chaoting Xuan, John A. Copeland, and Raheem A. Beyah. 2009. Toward revealing kernel malware behavior in virtual execution environments. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*. 304–325. DOI:http://dx.doi.org/10.1007/978-3-642-04342-0_16

Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. 2012. V2E: Combining hardware virtualization and software emulation for transparent and extensible malware analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)*. ACM, New York, NY, 227–238. DOI:http://dx.doi.org/10.1145/2151024.2151053

Lok-Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security'12)*. 29–29. http://dl.acm.org/citation.cfm?id=2362793.2362822

Jean Yang and Chris Hawblitzel. 2010. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, New York, NY, 99–110. DOI:http://dx.doi.org/10.1145/1806596.1806610

Heng Yin, Zhenkai Liang, and Dawn Song. 2008. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 2008 Annual Network and Distributed System Security Symposium (NDSS'08)*. http://www.isoc.org/isoc/conferences/ndss/08/papers/15_hookfinder_identifying.pdf.

Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*. 116–127. http://doi.acm.org/10.1145/1315245.1315261

Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. 203–216. DOI: http://dx.doi.org/10.1145/2043556.2043576

Shengzhi Zhang, Xiaoqi Jia, Peng Liu, and Jiwu Jing. 2010. Cross-layer comprehensive intrusion harm analysis for production workload server systems. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC'10)*. 297–306. DOI: http://dx.doi.org/10.1145/1920261.1920306

Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP'12)*. IEEE, Los Alamitos, CA, 95–109. DOI: http://dx.doi.org/10.1109/SP.2012.16

Received February 2014; revised December 2014; accepted August 2015