Toward Guest OS Writable Virtual Machine Introspection

Zhiqiang Lin
The University of Texas at Dallas
zhiqiang.lin@utdallas.edu

Abstract

Over the past decade, a great deal of research on virtual machine introspection (VMI) has been carried out. This approach pulls the guest OS state into the low-level hypervisor and performs external monitoring of the guest OS, thereby enabling many new security applications at the hypervisor layer. However, past research mostly focused on the read-only capability of VMI; because of inherent difficulties, little effort went into attempting to interfere with the guest OS. However, since hypervisor controls the entire guest OS state, VMI can go far beyond read-only operations. In this paper, we discuss writable VMI, a new capability offered at the level of the hypervisor. Specifically, we examine reasons of why to embrace writable VMI, and what the challenges are. As a case study, we describe how the challenges could be solved by using our prior EXTERIOR system as an example. After sharing our experience, we conclude the paper with discussions on the open problems and future directions.

1. Introduction

By virtualizing hardware resources and allocating them based on need, virtualization [18] [19] [28] has significantly increased the utilization of many computing capacities, such as available computing power, storage space, and network bandwidth. It has pushed our modern computing paradigm from multi-tasking computing to multi-operating-system computing. Located one layer below the operating system (OS), virtualization enables system developers to achieve unprecedented levels of automation and manageability—especially for large scale computing systems—through resource multiplexing, server consolidation [32], machine migration [4], and better security [16] [15] [14] [3] [34], reliability, and portability [2]. Virtualization has become ubiquitous in the realm of enterprise computing today, underpinning cloud computing and data centers. It is expected to become ubiquitous on the desktop and mobile devices in the near future.

In terms of security, one of the best applications enabled by virtualization is virtual machine introspection (VMI) [16]. VMI pulls the guest OS state into the outside virtual machine monitor (VMM), or *hypervisor* (the terms VMM and hypervisor are used interchangeably in this paper), and performs external monitoring of the runtime state of a guest OS. The introspection can be placed in a VMM, in another virtual machine (VM), or within any other part of the hypervisor, as long as it can inspect the runtime state of the guest OS—including CPU registers, memory, disk, and network. Because of such strong isolation, VMI has been widely adopted in many security applications such as intrusion detection (e.g., [16] [24] [25]), malware analysis (e.g., [22] [5] [6] [9]), process monitoring (e.g., [30] [31]), and memory forensics (e.g., [20] [7] [9]).

However, past research in VMI has primarily focused on read-only inspection capability for the guest OS. This is reasonable, because intuitively any writable operation to the guest OS might disrupt the kernel state and even crash the kernel. In other words, in order to perform writable operations, the VMM must know precisely which guest virtual address it can safely write to, and when it can perform the write (i.e., what the execution context is). Unfortunately, this is challenging because of the well-known semantic gap problem [2]. That is, unlike the scenario with the in-guest view—where we have rich semantics such as the type, name, and data structure of kernel objects—at the VMM layer, we can view only the low-level bits and bytes. Therefore, we must bridge the semantic gap.

Earlier approaches to bridging the semantic gap have leveraged kernel-debugging information, as shown in the pioneer work Livewire [16]. Other approaches include analyzing and customizing kernel source code (e.g., [26] [21]), or simply manually writing down the routines to traverse kernel objects based on the kernel data structure knowledge (e.g., [22] [24]). Recently, highly automated binary-code-reuse-based approaches have been proposed that either retain the executed binary code in a re-executable manner or operate through an online kernel data redirection approach utilizing dual-VM support.

Given the substantial progress in the all possible approaches to bridging the semantic gap at the VMM layer, today we are almost certain of the semantics of the guest OS virtual addresses that we may or may not write to. Then can we go beyond read-only VMI? Since the VMM controls the entire guest computing stack, VMM certainly can do far more than that, such as perform guest OS memory-write operations. Then what are the benefits of writable VMI? What is the state of the art? What are the remaining challenges that must be addressed to make writable VMI deterministic? How can we address them and realize this vision?

This paper tries to answer these questions. Based on our prior experiences with EXTERIOR [10], we argue that writable VMI is worthwhile and can be realized. In particular, we show that there will be many exciting applications once we can enable writable VMI, such as guest OS reconfiguration and repair, and even applications for guest OS kernel updates. However, there are still many challenges to solve before we can reach that point.

The rest of the paper is organized as follows: Section 2 addresses further the need of writable VMI. Section 3 discusses the challenges we will be facing. In Section 4, we present the example of a writable-VMI prototype that we built to support guest OS reconfiguration and repair. Section 5 discusses future directions, and finally Section 6 concludes the paper.

2. Further Motivation

Past research on VMI primarily focused on retrieving the guest OS state, such as the list of running processes, active networking connections, and opening files. None of these operations requires modification of the guest OS state, which has consequently limited the capabilities of the VMI. By enabling VMI to write to the guest OS, we can support many other operations on the guest OS, such as configuring kernel parameters, manipulating the IP routing table, or even killing a malicious process.

For security, writable VMI would certainly share all of the benefits of readable VMI, such as strong isolation, higher privilege, and stealthiness. In addition, it can have another unique advantage—high automation. In the following, we discuss these benefits in greater detail. More general discussion of the benefits of hypervisor-based solutions can be found in other papers (c.f., [2] [17]).

- **Strong isolation** – The primary advantage of using the VMM is the ability to shift the guest OS state out of the VM, thereby isolating in-VM from out-of-VM programs. It is generally believed to be much harder for adversaries to tamper with programs running at the hypervisor layer, because there is a world switch from in-VM to out-of-VM (unless the VMM has vulnerabilities). Consequently, we can gain higher trustworthiness of out-of-VM programs. For instance, if we have a VMM-layer guest OS process kill utility, we can guarantee that this utility is not tampered before using it to kill the malicious processes inside the guest OS.
- **Higher privileges and stealthiness** – Traditional security software (e.g., antivirus) runs inside the guest OS, and in-VM malware can often disable the execution of this software. By moving the execution of security software to the VMM layer, we can achieve a higher privilege (same as the hypervisor's) for it and make it invisible to attackers (higher stealthiness). For instance, malicious code (e.g., a kernel rootkit) often disables the `rmmod` command needed to remove a kernel module. By enabling the execution of these commands at the VMM layer, we can achieve a higher privilege. Also, the VMM-layer `rmmod` command would certainly be invisible (stealthy) to the in-VM malware because of the strong isolation.
- **High Automation** – A unique advantage of writable VMI is the enabling of automated responses to guest OS events. For instance, when a guest OS intrusion is detected, it often requires an automated response. Current practice is to execute an automated response inside the guest OS and/or notify the administrators. Again, unfortunately, any in-VM responses can be disabled by attackers because they run at the same privilege level. However, with writable VMI, we can quickly take actions to stop and prevent the attack without the assistance from any in-VM programs and their root privileges. Considering that there are a great deal of read-only VMI-based intrusion-detection systems (e.g., [6] [8] [9] [16] [22] [23] [24]), writable VMI can be seamlessly integrated with them and provide a timely response to attacks—such as `kill`-ing a rootkit-created hidden process and running `rmmod` against a hidden malicious kernel module.

3. Challenges

However, it is non-trivial to realize writable VMI at the hypervisor layer. As in all the read-only VMI solutions, we must bridge the semantic gap and reconstruct the guest OS abstractions. In addition, we will also face a concurrency problem while performing guest OS writable operations.

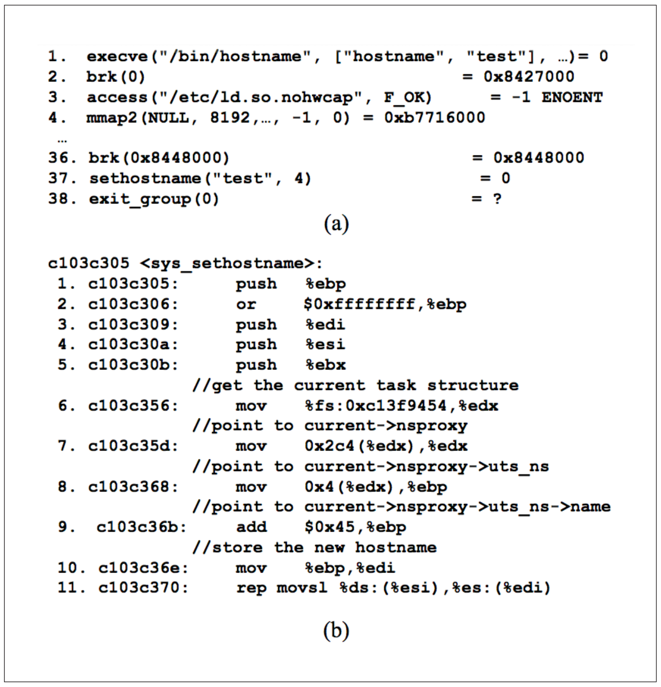
3.1 Reconstructing the Guest OS Abstractions

Essentially, a hypervisor can be considered to be programmable hardware. Therefore, the view at the hypervisor layer is at a very low level. Specifically, we can observe all the CPU registers and all of the physical memory cells of the guest OS. Also, we can observe all the instruction executions if the hypervisor is an instruction-translation-based VMM; otherwise we can only observe some special VMM-level instructions (e.g., Intel VT-x instructions) and special kernel events such as page faults if the hypervisor is a hardware-virtualization-based VMM.

However, what we want is the semantic information of the guest OS abstractions. For instance, for a memory cell, we want to know the meaning of that cell—for example, what is the virtual address of this memory cell? Is it a kernel global variable? If so, what does this global variable stand for? For a running instruction inside the guest OS, we also would like to know if it is a user-level instruction or a kernel-level instruction? Which process does the instruction belong to? If the instruction belongs to kernel space, is it a system-call-related instruction, a kernel-module instruction, kernel-interrupt handler, or something else? For a running system call, we also want to know the semantics of this system call, such as the system call number and the arguments.

Therefore, we must bridge the semantic gap for this low-level data and these events. In general, we must be armed with detailed knowledge of the algorithms and data structures of each OS component in order to rebuild high-level information. However, due to the high complexity of modern OSs, acquiring such knowledge is a tedious and time-consuming operation, even for open source OSs. When the source code is not available, sustained effort is needed to reverse engineer the undocumented kernel algorithms and data structures.

Because of the importance of this problem, significant research in the past has focused on how to bridge the semantic gap more efficiently and with less constraint. Currently, the state of the art includes the kernel-data-structure-based approach (e.g., [16] [26] [21] [22] [24] [1]), and the binary-code-reuse-based approach (e.g., [6] [9] [10] [11] [29]). Each has its own pros and cons. The data-structure-assisted approach is flexible, fast, and precise, but it requires access to kernel-debugging information or kernel source code; a binary-code-reuse-based approach is highly automated, but it is slow and can only support limited functionality (e.g., with fewer than 20 native utilities supported so far in VMST [9] and EXTERIOR [10]).



```
1.  execve("/bin/hostname", ["hostname", "test"], …)= 0
2.  brk(0)                                    = 0x8427000
3.  access("/etc/ld.so.nohwcap", F_OK)        = -1 ENOENT
4.  mmap2(NULL, 8192,…, -1, 0) = 0xb7716000
 …
36. brk(0x8448000)                            = 0x8448000
37. sethostname("test", 4)                     = 0
38. exit_group(0)                             = ?
```

(a)

```
c103c305 <sys_sethostname>:
 1. c103c305:     push   %ebp
 2. c103c306:     or     $0xffffffff,%ebp
 3. c103c309:     push   %edi
 4. c103c30a:     push   %esi
 5. c103c30b:     push   %ebx
               //get the current task structure
 6. c103c356:     mov    %fs:0xc13f9454,%edx
               //point to current->nsproxy
 7. c103c35d:     mov    0x2c4(%edx),%edx
               //point to current->nsproxy->uts_ns
 8. c103c368:     mov    0x4(%edx),%ebp
               //point to current->nsproxy->uts_ns->name
 9.  c103c36b:    add    $0x45,%ebp
               //store the new hostname
 10. c103c36e:    mov    %ebp,%edi
 11. c103c370:    rep movsl %ds:(%esi),%es:(%edi)
```

(b)

Figure 1. (a) System call trace of the hostname command (b) Disassembled instructions for sys_sethostname system call

3.2 Addressing the Concurrency Issue

Unlike with read-only VMI, if we aim to perform writable operations on the guest OS, we must ensure that the memory write is safe. By *safe*, we mean that the newly written value should reflect the original OS semantics. In particular, for a memory write, even though we can bridge its semantic gap, we still need to know when it is the safe moment to launch the write operation. For instance, as shown in the Figure 1(b), when writable VMI executes the `rep movsl` instruction at the hypervisor layer to set the host name of the guest OS, we need to ensure there is no concurrent execution of this instruction inside the guest OS.

In addition, the OS is designed to manage hardware resources such as CPU and memory, which are often shared by multiple processes or threads (for multiplexing). Therefore, the OS kernel is full of synchronization or lock primitives against the concurrent access of the shared resources. These synchronization mechanisms (e.g., `spinlock` and `semaphores`) would set yet another obstacle when implementing writable VMI.

Note that the concurrency issue happens at very fine granularity—that is, the memory-cell level for a particular variable. Based on the semantics, if we are sure that there is no such concurrency, we can safely perform the memory write. In other words, the outside writable operation should be like a transaction (e.g., [27]), and self-contained. For instance, the execution of the `ps` command would not affect the kernel state, and this "transaction" is self-contained and can happen multiple times even inside the guest OS.

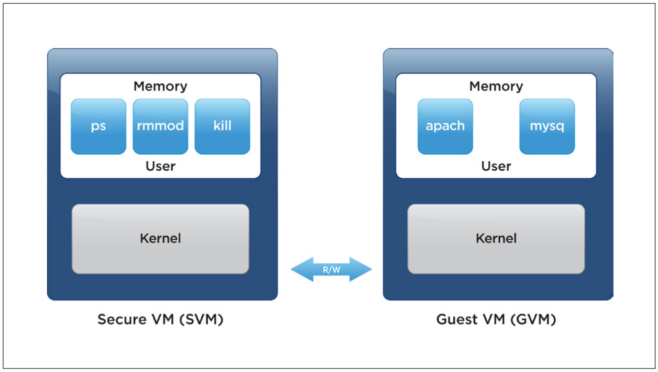


Figure 2. Overview of EXTERIOR.

There are also some other related issues, such as the performance trade-off. One intuitive approach for avoiding concurrency would be to stop guest OS execution and then perform the writable operation solely from VMI. This is doable if the performance impact on the guest OS is not so critical and if inside the guest OS there is no similar behavior to the outside writable operation.

4. Examples

In this section, we share our experiences in realizing a writable VMI system named EXTERIOR [10]. We first give an overview of our system in Section 4.1, and in Section 4.2 explain how we addressed the semantic-gap and concurrency challenges. Finally, we discuss the limitations of EXTERIOR in Section 4.3.

4.1 EXTERIOR Overview

Recently, we presented EXTERIOR, a dual-VM, binary-code-reuse-based framework for guest OS introspection, configuration, and repair. As illustrated in Figure 2, EXTERIOR enables native OS utilities such as `ps`, `rmmod`, and `kill` to execute in a secure VM (SVM) but transparently inspect and update the OS state in the guest VM (GVM). There are two requirements for EXTERIOR to work: (1) The OSs running in the two VMs must be the exact same version and (2) the SVM's hypervisor is an instruction-translation-based VMM.

The SVM is used to create the necessary running environment for the utility processes. The binary-translation-based VM is used to monitor all the instruction execution in the SVM, resolve the instruction execution context, and dynamically and transparently redirect and update the memory state at the hypervisor layer from SVM to GVM when the execution context of interest is executed, thus achieving the same effect—in terms of kernel state updates—as running the same utility inside the GVM.

We demonstrated that EXTERIOR can be used for automated management of a guest OS, including introspection (e.g., `ps`, `lsmod`, `netstat`) and reconfiguration (e.g., `sysctl`, `hostname`, `renice`) of the guest OS state without any user account in the guest OS. It also supports end users developing customized programs to repair the kernel damage inflicted by kernel malware, such as contaminated system-call tables.

4.2 Solutions to the Challenges

To bridge the semantic gap, EXTERIOR uses a binary-code-reuse-based approach. The key insight is that for compiled software, including an OS kernel, variables are usually updated by the compiled instructions within a certain execution context. More specifically, by tracing how the traditional native program executes and updates the kernel state, we observe that OS kernel state is often updated within a certain kernel system call execution context. For instance, as shown in Figure 1, the `sethostname` utility, when executed with the `test` parameter, invokes the `sys_sethostname` system call to update the kernel host name with the new name. The instructions from line 6 to line 11 are responsible for this.

Therefore, if at the VMM layer we can precisely identify the instruction execution from line 6 to line 11 when system call `sys_sethostname` is executed, and if we can maintain a secure duplicate of the running GVM as a SVM, through redirecting both their memory read and write operations from the SVM to the running GVM, we can transparently update the in-VM kernel state of the GVM from the outside SVM. In other words, the semantic gap (e.g., the memory location of in-VM kernel state) is automatically bridged by the intrinsic instruction constraints encoded in the binary code in the duplicated VM. That is why it eventually leads to a dual-VM based architecture.

Regarding the concurrency issues, it is rare to execute these native utilities simultaneously in both SVM and GVM. For instance, the probability would be extremely low of executing a utility such as `sethostname` in the SVM at the same time that it is executed in the GVM. Meanwhile, the utilities that EXTERIOR supports are self-contained, and they can be executed multiple times in one VM. For instance, we can execute `kill` multiple times to kill a process, and we can also execute `ps` multiple times to show the running-processes list. Also, we can execute `kill` at an arbitrary time to kill a running process, because this operation is self-contained. That is why these operations can be considered transactions. A rule of thumb is that if we can execute a command multiple times in a VM and can get the same result in terms of kernel-state inspection or update, then that command can be executed in a SVM.

4.3 Limitations of EXTERIOR

The way in which EXTERIOR bridges the semantic gap and how it addresses the concurrency issue naturally lead to a number of limitations. First, it requires the two kernels to have identical kernel versions because of the nature of binary code reuse. Any new patch to the GVM kernel must be applied to the SVM. Second, it also requires the address space of kernel global variables not be randomized; otherwise it must derandomize it. Third, the execution of the monitored system call (e.g., `sys_sethostname`) will not be blocked, and the monitored system call should only operate on memory data.

Because of the above constraints, EXTERIOR cannot support the running of arbitrary administration utilities with arbitrary kernels. Also, EXTERIOR must precisely identify the instruction execution context. Currently, it can precisely identify the system-call execution context. However, a given system call can contain certain nonredirectable data, such as the variables accessed by `spin_lock` and `spin_unlock`, or semaphores accessed by `__up` or `__down`. If we cannot precisely identify the execution context of these functions, EXTERIOR is highly likely to make these relevant kernel lock primitives inconsistent when redirecting kernel data access. Currently, EXTERIOR uses a manual approach to derive the signatures for all those observed kernel locks. Such a manual approach is tedious and error-prone, and it must be repeated for different kernels.

5. Future Research

There are many directions to go in order to realize writable VMI. The two most urgent steps are to (1) push the technology further based on different constraints and (2) demonstrate the technology with more compelling applications. This section discusses both steps in more detail.

5.1 Improving the Techniques

Whether the hypervisor can access the guest OS kernel source code determines which of the two following strategies are possible: we can either retrofit the kernel source code to make it more suitable for writable VMI, or we can improve the binary code analysis of the OS kernel to automatically recognize a more fine-grained execution context such as `spin_locks`.

5.1.1 Retrofitting Kernel Source Code

As with writable VMI, we want to perform transaction-like operations. Also, at the binary-code level it is challenging to recognize the kernel-synchronization primitives. Then why not to retrofit the kernel source code to add hooks or wrappers such that, at the hypervisor layer, we can easily detect these events? This is certainly doable. For instance, much as in paravirtualization [32], we can modify kernel source code (with a compiler pass) to automatically recognize certain functions based on certain rules, and add hooks (e.g., [13]), or even rewrite some part of kernel code if the transaction-like behavior is missing (c.f., TxOS [27]).

On the other hand, to perform writable VMI at the hypervisor layer, essentially we are executing a program at the hypervisor layer to update kernel variables. Another route would be to change the binary code output (by compilers) of the given kernel in order to assist our VMI. For instance, if we can relocate the kernel variables to certain pages (instead of mixing them with all other unrelated kernel variables, which is the current practice), it would be much easier for the hypervisor to recognize and update the kernel introspection related information. For instance, through program analysis such as program slicing, if we can precisely identify the variables involved in the memory write and relocate them into special pages, we could map the pages between the SVM and the GVM such that the operation happening in the SVM is directly reflected in the GVM's state. It might be trivial to relocate the global variables, but for heap we might have to dynamically track them through pointer references. Part of our current research is working in this direction.

5.1.2 Recognizing Fine-Grained Execution Context

When we cannot retrofit the kernel source code, the only way we can move forward is to improve the binary code analysis to recognize the more fine-grained execution context. Currently, we can recognize the beginning and ending point of a system call, interrupt, and exception, through instrumenting kernel binary code and hardware events generation [9] [10] [11] [29]. We cannot recognize many other kernel functions such as context switch, the bottom half of the interrupt handler, and many of the kernel synchronization primitives. Although we have mitigated the identification of these functions by instrumenting the timer to further disable the context switch, this is certainly not general enough and has limited functionality.

Therefore, determining how to identify fine-grained kernel function execution contexts and their semantics is a challenging problem. Manually inspecting each kernel function will not scale, and automatic techniques are needed. Having source code access is much easier, because we can instrument the code and inform the hypervisor of the execution context. When given only binary, we must automatically infer them from the information we gather.

5.2 Exploring More Applications

There will be many new exciting applications once we are able to perform fine-grained (i.e., memory-address-level) writable VMI. We have demonstrated that we can do guest OS reconfiguration such as resetting certain kernel parameters. We can also perform guest OS repair to clean the attack footprints.

Other possible applications include kernel updates. If we can quantify that a new kernel patch changes the kernel state in a transactional way, then we can certainly perform writable VMI to update the kernel defined in the patch. Other applications could be forensic applications that bypass authentication [12]. Again, the biggest advantage for writable VMI is that no explicit root privilege is required to perform a task (because it has the highest, hypervisor-level privilege).

In a broader scope, we can view writable VMI as a new program execution model that has certain components executed in-VM and certain components executed out-of-VM. These two types of components work together but have different trustworthiness and privileges. Some typical problems, such as the consumer and producer model, might be good examples of using writable VMI. For instance, we can use writable VMI to produce the kernel data for consumers inside the guest OS to consume. It might also be useful for high-performance computing (HPC), because this model splits program execution into two parts. With the support of special APIs, we might be able to improve the performance of specific HPC applications.

6. Conclusion

VMI has been an appealing application for security, but it only focuses on the guest OS read-only capability provided by the hypervisor. This paper discusses the possibility of exploring approaches for writable-capability that is necessary for changing certain kernel state. In particular, we discussed the demand for highly automated writable-VMI approaches that can be used as transactions. We also walked through the challenges that we will be facing, including the well-known semantic-gap problem, and the unique concurrency issues that occur when both in-VM and out-of-VM programs write the same kernel variables at the same time. We discussed how we can solve these challenges by using our prior EXTERIOR system as an example. Finally, we believe writable VMI is useful. It will open many new opportunities for system administration and security. There are still many open problems to work on in order to realize full-fledged writable-VMI.

References

1 Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., and Jiang, X. Mapping kernel objects to enable systematic integrity checking. In *The 16th ACM Conference on Computer and Communications Security (CCS'09)* (Chicago, IL, USA, 2009), pp. 555–565.

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

3 Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Confer-ence on Architectural support for programming languages and oper-ating systems* (Seattle, WA, USA, 2008), ASPLOS XIII, ACM, pp. 2–13.

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

5 Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)* (Alexandria, Virginia, USA, 2008), pp. 51–62.

6 Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy* (Oakland, CA, USA, 2011), pp. 297–312.

7 Dolan-Gavitt, B., Payne, B., and Lee, W. Leveraging forensic tools for virtual machine introspection. *Technical Report*; GT-CS-11-05 (2011).

8 Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Sys-tems Design and Implementation (OSDI)* (2002).

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

10 Fu, Y., and Lin, Z. Exterior: Using a dual-vm based external shell for guest OS introspection, configuration, and recovery. In *Proceedings of the Ninth Annual International Conference on Virtual Execution Environments* (Houston, TX, March 2013).

11 Fu, Y., and Lin, Z. Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. *ACM Transaction on Information System Security* 16(2) (September 2013), 7:1–7:29.

12 Fu, Y., Lin, Z., and Hamlen, K. Subverting systems authentication with context-aware, reactive virtual machine introspection. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13)* (New Orleans, Louisiana, December 2013).

13 Ganapathy, V., Jaeger, T., and Jha, S. Automatic placement of authorization hooks in the Linux security modules framework. In *Proceedings of the 12th ACM Conference on Computer and communications security* (Alexandria, VA, USA, 2005), CCS '05, ACM, pp. 330–339.

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

15 Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and BONEH, D. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM symposium on Operating Systems Principles* (Bolton Landing, NY, USA, 2003), SOSP'03, ACM, pp. 193–206.

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

17 Garfinkel, T., and Rosenblum, M. When virtual is harder than real: Security challenges in virtual machine based computing environ-ments. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)* (May 2005).

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

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

20 Hay, B., and Nance, K. Forensics examination of volatile system data using virtual introspection. *SIGOPS Operating System Review 42* (April 2008), 74–82.

21 Hofmann, O. S., Dunn, A. M., Kim, S., Roy, I., and Witchel, E. Ensuring operating system kernel integrity with osck. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA, 2011), ASPLOS '11, pp. 279–290.

22 Jiang, X., Wang, X., and Xu, D. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communica-tions Security (CCS'07)* (Alexandria, Virginia, USA, 2007), ACM, pp. 128–138.

23 Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Ant-farm: tracking processes in a virtual machine environment. In *Proceedings USENIX '06 Annual Technical Conference* (Boston, MA, 2006), *USENIX Association*.

24 Payne, B. D., Carbone, M., and Lee, W. Secure and flexible monitor-ing of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)* (December 2007).

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

26 Petroni, JR., N. L., and Hicks, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Confer-ence on Computer and Communications Security* (Alexandria, Virginia, USA, 2007), CCS '07, ACM, pp. 103–115.

27 Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA, 2009), SOSP '09, ACM, pp. 161–176.

28 Rosenblum, M., and Garfinkel, T. Virtual machine monitors: Current technology and future trends. *IEEE Computer* (May 2005).

29 Saberi, A., Fu, Y., and Lin, Z. Hybrid-Bridge: Efficiently Bridging the Semantic-Gap in Virtual Machine Introspection via Decoupled Execution and Training Memoization. In *Proceedings Network and Distributed Systems Security Symposium (NDSS'14)* (February 2014).

30 Sharif, M. I., Lee, W., Cui, W., and Lanzi, A. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA, 2009), CCS '09, ACM, pp. 477–487.

31 Srinivasan, D., Wang, Z., Jiang, X., and Xu, D. 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) (Chicago, Illinois, USA, 2011), pp. 363–374.

32 Whitaker, A., Shaw, M., and Gribble, S. D. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference* (2002).

33 Whitaker, A., Shaw, M., and Gribble, S. D. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design And Implementation* (Boston, Massachu-setts, 2002), OSDI '02, ACM, pp. 195–209.

34 Zhang, F., Chen, J., Chen, H., and Zang, B. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011), *SOSP '11, ACM*, pp. 203–216.