Introspections on the Semantic Gap

Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion | Stony Brook University

An essential goal of virtual machine introspection (VMI) is security policy enforcement in the presence of an untrustworthy OS. One obstacle to this goal is the difficulty in accurately extracting semantic meaning from the hypervisor's hardware-level view of a guest OS.

Virtual machine introspection (VMI) techniques allow an external security monitor to observe software behavior inside a virtual machine (VM), including the guest OS. For example, we can use VMI to list programs running inside a VM—comparable to `ps` on Unix systems or Windows Task Manager. Obtaining a process list outside a VM is appealing from a security perspective because security administrators can identify illicit programs on a system, even if the OS kernel is compromised. There are also nonsecurity benefits to listing processes outside the VM, such as standardization of administrative utilities across multiple guest OSs.

A simple VMI-based process list would identify process descriptors' memory addresses and typecast them (in C parlance) to interpret their content. VMI developers must find the kernel data structures, such as process descriptors, by searching publicly available symbols for the addresses of the process descriptors' data structure.

Any guest OS abstraction can be introspected, including open file descriptors, network sockets, and interprocess communication abstractions. For instance, storage system prototypes have used VMI to track whether disk writes are data or metadata, writing metadata changes to disk more aggressively than data.[1] In this article, we focus on in-memory data structures and CPU register state.

VMI is appealing because it can move OS security monitoring out of the OS. Widely used OS kernels are generally very large and afford little fault or security isolation among components; are written in languages such as C or C++ that offer little protection against exploitable programmer errors; and have complex, hard-to-secure APIs. Thus, if any OS kernel component has an exploitable bug, all OS-level security measures are easily disabled.

In our process listing example, a *rootkit* module could tamper with the kernel's mechanism for listing the set of running processes, often to hide other malware running on the system. Not only could an effective rootkit hide malware from a process listing utility or antivirus system inside the OS, it could avoid detection and removal. A VMI monitor can view all guest OS memory and identify rootkits.

The fundamental challenge underlying VMI is how to reliably infer what's happening in the guest OS. In our simple example, the VMI monitor has direct access only to hardware-level state, such as CPU registers and memory contents, and must make inferences about high-level abstractions, such as process descriptors and

 Copublished by the IEEE Computer and Reliability Societies

open files. This mismatch is called the *semantic gap*.[2] In this article, we summarize the major known techniques to bridge the semantic gap and discuss attacks on and defenses against these techniques.

Assumptions

Because bridging the semantic gap is such a challenging problem, most techniques have introduced assumptions that limit the threat model. In our process listing example, the VMI monitor uses knowledge obtained out of band, such as debugging symbols and structure definitions, and must assume that the potentially compromised guest OS is using these symbols and structures as expected. In some cases, we can detect deviation from assumed OS behavior, but many assumptions are hard to check. For instance, it's difficult to verify that the binary name listed in a process descriptor is an accurate description of what's running in the process. Each fragile assumption is a potential vector for adversaries to confuse and evade the VMI monitor.

As a result, most VMI techniques assume the guest OS is benign—initially not malicious, but potentially compromised after boot. VMI designs currently assume generous limits on the degree to which a compromised guest OS can actively work to confuse a VMI monitor, and these limits aren't always explicated. Nonetheless, VMI tools designed under this threat model can still have practical value, as OSs can be benign in practice.

Basic VMI System Design

The first consideration in VMI design is where to place the monitor, which directly influences how the monitor accesses guest memory and CPU state. Figure 1 illustrates options for VMI monitor placement, including in the hypervisor (with possible hardware assistance), in the guest OS, in a sibling VM, or outside the hypervisor (in a Type 2, or hosted, hypervisor only—not shown).

To access hardware information, such as CPU register contents, an in-hypervisor monitor can directly access hypervisor-internal data structures. When the monitor is moved out of the hypervisor, the hypervisor must export other hardware information to the monitor via an additional interface.

Placing the introspection tool in a sibling VM is particularly popular for several reasons. First, a sibling VM can have a read-only or copy-on-write mapping of the guest's memory, creating a high-bandwidth channel to traverse data structures. Second, this design requires minimal changes to the hypervisor and protects it from bugs in the VMI monitor. Finally, a VMI monitor developer can use a familiar environment, such as a comparable OS kernel and helper functions.

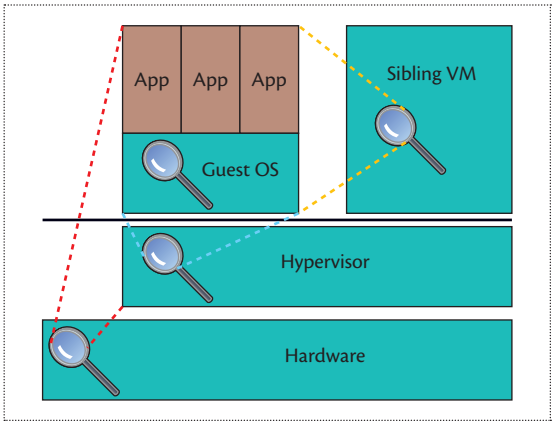


Figure 1. Monitor placement options in virtual machine introspection (VMI): in a sibling virtual machine (VM), the hypervisor, the guest OS, or the hardware. In-guest and hardware solutions require assistance from the hypervisor.

Trading Risk for Performance with Asynchrony

The second question is when to introspect. In our example, suppose we want to know each time a process is created or destroyed. A synchronous mechanism requires one or more triggering events, such as changing the process descriptor list or scheduling a process. When a triggering event occurs, the hypervisor pauses the VM, and the VMI tool introspects the process descriptor list. In contrast, an asynchronous mechanism would introspect memory concurrently with guest execution, generally at a configurable interval.

A typical introspection pass that checks data structure invariants takes from milliseconds to minutes; pausing the VM for this length of time in a synchronous design is unacceptable. Asynchrony limits CPU overhead, generally to a few percent, by adjusting the frequency of checks.

Asynchrony's primary disadvantage is that it must handle transient OS states, whereas a carefully placed synchronous triggering event can avoid transient states. While executing inside a critical section, an OS might violate its own invariants temporarily. A correct OS will, of course, restore the invariants before exiting the critical section. If an introspection monitor searches memory during a kernel critical section, the monitor might observe benign violations of these invariants. Current approaches to this problem include looking for repeated violations of an invariant (leaving the system vulnerable to race conditions with attackers) or introspecting only when the OS can't be in critical sections, for example, by preempting each CPU while out of the guest kernel.

Hardware Acceleration

Several VMI prototypes have used hardware to accelerate or offload introspection. One major approach is

Table 1. VMI techniques, their underlying trust assumptions, and monitor placement.

Technique	Assumptions	Monitor placement
Automated learning and reconstruction (source analysis or offline training)	Benign copy of OS for training; OS will behave similarly during learning phase and monitoring; security-sensitive invariants can be automatically learned; and attacks will persist long enough for periodic scans	Sibling VM, hypervisor, or hardware
Code implanting (hypervisor protects monitor inside guest OS)	Malicious guest schedules monitoring tool and reports information accurately	Guest with hypervisor protection
Process outgrafting (reuse monitoring tools from sibling virtual machine [VM] with shared kernel memory)	Live, benign copy of OS behaves identically to monitored OS	Sibling VM

snapshotting, wherein a device takes a snapshot of RAM, using, say, a tool on the PCI bus, and offloads the snapshot to another machine for asynchronous introspection.

More recent systems have used snooping on the system memory bus as a lightweight triggering mechanism. On commodity hardware, page protections are the primary technique to monitor access to many memory locations. The coarse granularity of page protections leads to many needless checks triggered by memory accesses adjacent to the monitored structure. Unlike page protections, snooping systems can monitor writes at the finer granularity of cache lines, reducing needless checks.

Initial snooping systems used customized hardware, although a recent design leveraged best-effort hardware transactional memory on commodity chips to implement snooping, at the cost of one dedicated core.[4] Snooping can be synchronous or asynchronous.

Prevention versus Detection

Some introspection tools prevent security policy violations, such as execution of unauthorized code, whereas others detect a compromise only after the fact. Clearly, prevention is a more desirable goal but requires a mechanism to identify and interpose on low-level operations that might violate a system security policy. Certain goals map naturally onto hardware mechanisms, such as page protections on kernel code. Other goals, such as upholding kernel data structure invariants, are open questions. All current prevention systems employ some form of memory protection to synchronously interpose on sensitive data writes.

As a result, current VMI tools detect only violations of more challenging properties, generally using periodic introspections. Periodic checks are a good fit for malware that leaves persistent modifications, but can miss transient modifications. A straw man approach to prevent violations of data structure invariants might trigger synchronous introspection on all writes to all security-relevant objects—which would be prohibitively expensive. Moreover, because some invariants span multiple writes, the straw man approach would likely yield false negatives without deeper analysis of the code behavior. Prevention techniques based on memory bus snooping might be more efficient, but this is an open research question.

Bridges across the Semantic Gap

To cross the semantic gap, a VMI system must extract high-level abstractions from the running guest system. We describe the three primary techniques to bridge the semantic gap—learning and reconstruction, code implanting, and process outgrafting—and their underlying trust assumptions in Table 1.

One assumption common to all these techniques is that the executable kernel code doesn't change between introspection tool creation and guest OS monitoring. This requires a measure of kernel integrity protection, discussed in more detail in "SoK: Introspections on Trust and the Semantic Gap."[3]

Learning and Reconstruction

The first technique reconstructs data structures from memory contents. Data structure reconstruction can be divided into learning and searching phases. The learning phase creates data structure signatures, using techniques including expert knowledge, source analysis, and dynamic analysis. A signature identifies and defines data structure instances.

The search phase uses the signatures to identify and interpret data structures. A search can be either a linear scan of kernel memory or a traversal of data structure pointers, starting with public symbols. It is arguable which approach is more efficient, because many kernel data structures can have cyclic or invalid pointers but might require traversing less total memory. However, the linear scan of kernel memory is robust in the presence of "disconnected" structures or other attempts to obfuscate pointers. Both techniques can observe transient states when searching concurrently with OS operation.

There are the three major techniques for learning data structure signatures.

Handcrafted signatures. Introspection and forensic analysis tools initially used handcrafted signatures, based on expert knowledge of the internal workings of an OS. Handcrafted signatures have an inherent limitation: each change to an OS kernel requires an expert to update the tools. For instance, a new version of the Linux kernel is released every two to three months; bug-fix updates can be as frequent as every few weeks. Each of these releases can change a data structure layout or invariant. Similarly, different compilers or versions of the same compiler can change the layout of a data structure in memory, frustrating handwritten tools. Automated techniques have become popular to keep pace with these release schedules and the variety of OS kernels and compilers.

Source code analysis. Automated reconstruction tools might rely on source code analysis or debugging information to extract data structure definitions and leverage source invariants to reduce false positives during the search phase.

A basic application of source analysis identifies all kernel object types, and then traverses the graph of pointers, starting from global symbols. A key challenge in creating this graph of data structures is that not all pointers in a data structure point to valid data. For example, the Linux `dcache` uses deferred memory reclamation of a directory entry structure, called a `dentry`, to avoid synchronization with readers. When a `dentry` is on a to-be-freed list, it might point to memory that has already been freed and reallocated for another purpose; an implicit invariant is that these pointers will no longer be followed once the `dentry` is on this list. Unfortunately, these implicit invariants can thwart simple pointer traversal.

An alternative is to use the structure of this pointer graph as a signature.[5] For instance, the pointers among `task_struct` structures in Linux form a different graph from pointers connecting `inode` structures.

Dynamic learning. Rather than identifying code invariants from kernel source code, we can observe a running OS instance to learn data structure invariants.[5,6] Analogous to supervised machine learning, the VMI tool trains on a trusted OS instance, and then classifies the data structures of potentially untrusted OS instances. During the training phase, these systems often control the stimuli by running programs that will manipulate a data structure of interest or incorporating debugging symbols to discern more quickly which memory regions might include a structure of interest.

Some dynamic systems have also developed robust signatures, which are immune to malicious changes to live data structure instances.[6] The primary utility of robust signatures is detecting when a rootkit attempts to hide persistent data by modifying data structures in ways that the kernel doesn't expect. However, these attempts are fruitful only if they don't crash the OS kernel. Thus, robust signatures leverage invariants an attacker can't safely violate.

Code Implanting

A simpler approach to bridging the semantic gap is to inject code into the guest OS that reports semantic information back to the hypervisor. For instance, Syringe implants functions into the kernel, which can be called from the VM.[7] A challenge to implanting code is ensuring that the implanted code isn't tampered with and actually executes, and that the guest OS components it uses report correct information. Most of these implanting techniques ultimately rely on the guest kernel to faithfully represent information, such as the process list, to the injected code.

Process Outgrafting

To overcome the challenges with running a trusted process inside an untrusted VM, process outgrafting relocates a monitoring process from the monitored VM to a second, trusted VM.[8] The trusted VM has some visibility into the monitored VM's kernel memory, allowing VMI tools to access any kernel data structures without direct interference from an adversary in the monitored VM.

The Virtual Machine Space Traveler generalizes this approach by running a trusted, clean copy of the OS with a roughly copy-on-write view of the monitored guest.[9] Monitoring applications, such as `ps`, simply execute in a complete OS environment on the monitoring VM; each executed system call actually reads state from the monitored VM. This approach bridges the semantic gap by repurposing existing OS code. However, it has open problems, such as reconciling divergences in the guest kernel's copy-on-write views.

Attacks, Defense, and Trust

Here, we explain the three major classes of attacks against VMI—kernel object hooking (KOH), dynamic kernel object manipulation (DKOM), and direct kernel structure manipulation (DKSM)—known defenses against those attacks, and how these attacks relate to trust placed in the guest OS. These issues are summarized in Table 2 and illustrated in Figure 2.

Kernel Object Hooking

KOH attack modifies function pointers (hooks) located in the kernel text or data sections, such as those used to implement an extensible virtual file system model. As Figures 2a and 2b illustrate, attackers might replace the `iterate` function call pointer to filter malware from

Table 2. VMI attacks, defenses, and underlying trust assumptions.

Attack	Defense	Trust assumption
Kernel object hooking (KOH; code and hooks)	Memory-protect hooks from text section modification, or whitelist loadable modules	Pristine initial OS copy and administrator's ability to discern trustworthy kernel modules
Dynamic kernel object manipulation (heap)	Identify data structure invariants, or detect violations by scanning memory snapshots	Guest kernel exhibits only desirable behavior during training, or source is trustworthy; all security-relevant data structure invariants can be identified a priori; all malware will leave persistent modifications that violate an invariant; all invariants can be checked in a single search; and attackers can't win races with the monitor
Direct kernel structure manipulation	Prevent bootstrapping through KOH or return-oriented programming	OS is benign and behaves identically during training and classification

monitoring software. Defenses against KOH attacks generally depend on whether the hook is located in the kernel's text or data segment.

Text section hooks. The primary text section hooks are the system call table and interrupt descriptor table. For instance, attackers could interpose on all file `open` system calls simply by replacing the function pointer with the `sys_open()` function in the system call table. To prevent malware from overwriting these hooks, most kernels now place them in the read-only text segment. In a VMI system, the hypervisor can prevent malware from changing read-only page permissions.

Data section hooks. Kernel data section hooks are more difficult to protect than text section hooks, because they place function pointers in objects to facilitate extensibility. For instance, the Adore-ng rootkit replaces the directory listing function of the `/proc` directory (see Figure 2b), hiding itself from the output.[10] The fundamental challenge is that, although these hooks generally do not change during the object's lifetime, they are often located on the same page or even in the same cache line with fields that must change, thwarting defenses based on simple page protections.

To defend against such attacks, function pointers must be protected from modification once initialized. Because of the high cost of moderating all writes to these data structures, most defenses either move the hooks to different locations that can be write protected[11] or augment hooks in the kernel with checks against a whitelist of trusted functions.[12]

Trust. Preventing text section modification is a prerequisite for current VMI techniques. Defenses against KOH on data hooks effectively assume that kernel modules are benign, in order to provide meaningful protections without solving the significantly harder problem of kernel control flow integrity in the presence of untrusted modules.

Dynamic Kernel Object Manipulation

DKOM attacks modify the kernel heap through a loaded module or an application accessing `/dev/mem` or `/proc/kcore` on Linux.[13] DKOM attacks modify only data values and thus are distinct from attacks that modify the control flow through function hooks (KOH).

DKOM attacks invalidate latent assumptions in unmodified kernel code. A classic DKOM example is hiding a malicious process. The Linux kernel tracks processes in two separate data structures: a linked list for process listing and a tree for scheduling (see Figure 2c). A rootkit can hide malicious processes by taking the process out of the linked list but leaving the malicious process in the scheduler tree. Interestingly, loading a module is sufficient to alter the behavior of unrelated, unmodified kernel code.

DKOM attacks are hard to prevent because they are a needle in a haystack of expected kernel heap writes. As a result, most practical defenses attempt to identify data structure invariants by hand, static, or dynamic analysis, and then detect data structure invariant violations asynchronously. Because attackers can create objects from any memory, not just the kernel heap allocator, data structure detection is a salient issue for detecting DKOM attacks.

DKOM defenses introduce additional trust in the guest beyond a KOH defense and make several assumptions that attackers can violate. Most DKOM defenses work by identifying security-related data structure invariants. Because it is difficult for defenders to have confidence that all security-relevant invariants have been identified, this approach will likely be best effort and reactive in nature.

Another problematic assumption is that all kernel data structures' security-sensitive fields have invariants

that can be checked easily in a single memory snapshot or scan. For instance, a VMI-based approach to detect network sockets could be thwarted by a rootkit that copies packets directly from the heap of an application to the outgoing network driver. In this example, the inconsistency between outgoing packets and open sockets spans a sequence of operations, which can't be captured with one snapshot.

DKOM defenses cement trust that the guest kernel is benign. These defenses train data structure classifiers on a clean kernel instance or derive the classifiers from source code, which is assumed to demonstrate only desirable behavior during the training phase.

The interesting contrast between KOH and DKOM is that DKOM defenses can detect invalid data modification in the presence of an untrustworthy module, whereas common KOH defenses rely on module whitelisting. Thus, if a DKOM defense intends to tolerate untrusted modules, it must build on a KOH defense that's robust to untrusted modules as well, which might require substantially stronger control flow integrity protection.

Finally, these detection systems explicitly assume malware will leave persistent, detectable modifications and implicitly assume malware can't win races with the detector. DKOM detectors rely on invariant violations being present in the view of memory they analyze—either a snapshot or a concurrent search. Because DKOM detectors run in increments of seconds, short-lived malware could evade detection. If a rootkit can reliably predict when a DKOM detector will view kernel memory, it can temporarily repair data structure invariants—racing with the detector. To our knowledge, no work has successfully exploited this race condition, but this issue deserves further investigation.

Direct Kernel Structure Manipulation

Direct kernel structure manipulation (DKSM) attacks change the interpretation of a data structure between training a VMI tool and classifying memory regions into data structures.[14] Figure 2d illustrates a simple DKSM attack by a malicious kernel, which selectively swaps two data structure fields to hide the presence of malware from a VMI tool based on standard headers.

Because most VMI tools assume a benign kernel, successful DKSM attacks hinge on changing kernel control flow without changing kernel text. Two previously proposed bootstrapping mechanisms are KOH attacks and return-oriented programming—both of which have known countermeasures.

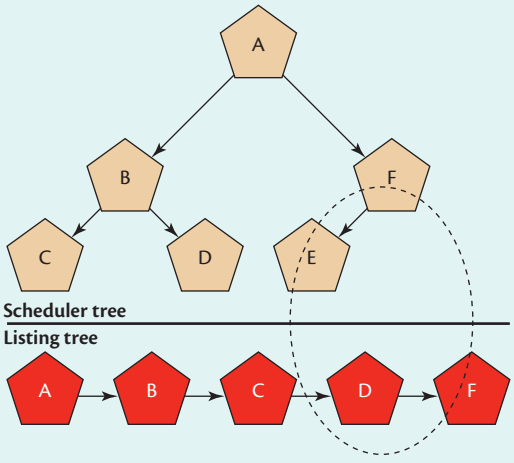
DKSM is an oddity in the literature because it's effectively precluded by a generous threat model. However, a realistic threat model might allow an adversarial OS to demonstrate different behavior during the data structure training and classification phases—analogous

```
int
proc_pid_read_dir(file *f, ...) {
  foreach task in task_list
    return task.tgid;
}
```

(a)

```
int iterate_dir(file *f, ...) {
  f->f_op->iterate(f, ...);
}
```

(b)



(c)

```
set_task_comm(task *tsk, char *name,...) {
  strlcpy(tsk->comm, name, sizeof(tsk->comm));
  if (malware) {
    tsk->notifier = kmalloc(strlen(name));
    strncpy(tsk->notifier, tsk->comm,
            strlen(name)+1);
    strncpy(tsk->comm, "browser", 7);
  }
}
```

(d)

Figure 2. Overview of kernel process listing. (a) Pseudocode to list running process IDs by reading the `/proc` directory. (b) Virtual file system–level pseudocode for reading a directory, which calls low-level file system calls, such as `proc_pid_read_dir`, in Figure 2a. A kernel object hooking (KOH) attack replaces the iterate function pointer in the file handle for `/proc`. (c) A dynamic kernel object manipulation (DKOM) attack selectively violates data structure invariants, such as the assumption that all processes are on a list (for listing) and a tree (for scheduling). (d) Pseudocode example of a direct kernel structure manipulation (DKSM) attack, where process initialization changes the interpretation of process descriptor fields for a program name to confuse a tool searching for known malware.

to "split personality" malware that behaves differently when it detects that it is under analysis. Under a stronger threat model, a malicious OS could actively mislead VMI tools to violate a security policy.

The Semantic Gap Is Really Two Problems

In the VMI literature, the semantic gap problem evolved to refer to two distinct issues: the largely solved engineering challenges of generating introspection tools, possibly without source code, and a malicious or compromised OS's ability to exploit fragile assumptions underlying many introspection designs to evade a security measure.

We suggest a clearer nomenclature for the two subproblems: the *weak* and *strong semantic gap* problems, respectively. The weak semantic gap is a solved engineering problem. The strong semantic gap problem is, to our knowledge, unsolved, and a solution would also prevent or detect DKSM attacks launched by malicious guest OSs. Our paper, "SoK: Introspections on Trust and the Semantic Gap," provides a more complete treatment of these issues.[3]

Toward an Untrusted OS

Some techniques from related research might help bridge the strong semantic gap.

Paraverification

Many VMI systems have an implicit design goal of working with an unmodified OS, which induces trust in the guest OS to simplify the problem. A useful stepping-stone might be to modify the OS to aid in its own introspection.

InkTag introduced the idea of *paraverification*, in which a guest OS provides a hypervisor with evidence that it is servicing an application's request correctly.[15] The hypervisor can easily check the evidence offered by the guest OS without trusting the guest OS. For instance, if a trusted application requests a memory mapping of a file, the application would also report the request to the hypervisor. The OS then submits evidence to the hypervisor that changes to hardware-level page tables are an appropriate response to the memory-mapping request, which the hypervisor then verifies. Although InkTag's goals differ from VMI's, the idea of forcing an untrusted OS to aid in its own introspection could be fruitful if the techniques were simple enough to adopt.

Mutual Distrust in Hardware

Intel has recently taken an interesting direction, developing a mutual distrust model for hardware memory protection called Software Guard Extensions (SGX). SGX lets an OS or hypervisor manage an application's virtual-to-physical OS mappings, but the lower-level software can't access memory contents. In the context of introspection or the strong semantic gap, hardware like SGX could be useful for creating a finer-grained protection domain for code implanted in the guest OS.

Reconstruction from Untrusted Sources

Current tools that automatically learn data structure signatures assume the OS will behave similarly during training and classification. Among the assumptions in current VMI tools, this one has the best chance of being incrementally removed. For example, one approach might train VMI classifiers on the live OS and continue incremental training as the guest OS runs. Similarly, continuous monitoring might detect inconsistencies between the VMI's training and classification stages.

Virtual machine introspection is a relatively mature research topic that has made substantial advances in the 12 years since the semantic gap problem was posed. However, efforts in this space should focus on removing trust from the guest OS to strengthen overall system security. ■

Acknowledgments

We thank Virgil Gligor, Bill Jannen, and the anonymous reviewers for their insightful comments on earlier drafts. This research was supported in part by NSF grants CNS-1149229, CNS-1161541, CNS-1228839, CNS- 1318572, CNS-1223239, and CCF-0937833; the US ARMY award W911NF-13-1-0142; the Office of the Vice President for Research at Stony Brook University; and gifts from Northrop Grumman Corporation, Parc/Xerox, Microsoft Research, and CA.

References

1. V. Tarasov et al., "Improving I/O Performance Using Virtual Disk Introspection," *Proc. 5th Usenix Workshop Hot Topics in Storage and File Systems* (HotStorage 13), 2013, p. 11.
2. P.M. Chen and B.D. Noble, "When Virtual Is Better Than Real," *Proc. 8th Workshop Hot Topics in Operating Systems*, (HotOS 01), 2001, pp. 133–138.
3. B. Jain et al., "SoK: Introspections on Trust and the Semantic Gap," *IEEE Symp. Security and Privacy*, 2014, pp. 605–620.
4. Y. Liu et al., "Concurrent and Consistent Virtual Machine Introspection with Hardware Transactional Memory," *Proc. IEEE 20th Int'l Symp. High Performance Computer Architecture*, 2014, pp. 416–427.
5. Z. Lin et al., "SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-Based Signatures," *Proc. 18h Ann. Network and Distributed System Security Symp.*, 2011, www.internetsociety.org/sites/default/files/lin.pdf. .

6. B. Dolan-Gavitt et al., "Robust Signatures for Kernel Data Structures," *Proc. 16th ACM Conf. Computer and Communications Security*, 2009, pp. 566–577.
7. M. Carbone et al., "Secure and Robust Monitoring of Virtual Machines through Guest-Assisted Introspection," *Research in Attacks, Intrusions, and Defenses*, LNCS 7462, Springer, 2012, pp. 22–41.
8. D. Srinivasan et al., "Process Out-Grafting: An Efficient 'Out-of-VM' Approach for Fine-Grained Process Execution Monitoring," *Proc. 18th ACM Conf. Computer and Communications Security*, 2011, pp. 363–374.
9. Y. Fu and Z. Lin, "Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection," *Proc. IEEE Symp. Security and Privacy*, 2012, pp. 586–600.
10. J. Corbet, "A New Adore Root Kit," *Linux Weekly News*, Mar. 2004; http://lwn.net/Articles/75990.
11. Z. Wang et al., "Countering Kernel Rootkits with Lightweight Hook Protection," *Proc. 16th ACM Conf. Computer and Communications Security*, 2009, pp. 545–554.
12. N.L. Petroni Jr. and M. Hicks, "Automated Detection of Persistent Kernel Control-Flow Attacks," *Proc. 14th ACM Conf. Computer and Communications Security*, 2007, pp. 103–115.
13. J. Butler and G. Hoglund, "Vice—Catch the Hookers," *Black Hat USA*, vol. 61, 2004, pp. 17–35.
14. S. Bahram et al., "DKSM: Subverting Virtual Machine Introspection for Fun and Profit," *Proc. 29th IEEE Symp. Reliable Distributed Systems*, 2010, pp. 82–91.
15. O.S. Hofmann et al., "InkTag: Secure Applications on an Untrusted Operating System," *Proc. 18th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 2013, pp. 265–278.

Bhushan Jain is a PhD candidate in computer science at Stony Brook University. His research interests include virtualization security, memory isolation, and system security. Jain received a B.Tech in computer engineering from College of Engineering Pune. Contact him at bpjain@cs.stonybrook.edu.

Mirza Basim Baig is an MS candidate in computer science at Stony Brook University. His research interests include data mining, machine learning, and graph theory. Basim Baig received a BS in computer science from Lahore University of Management Sciences, School of Science and Engineering (LUMS-SSE). Contact him at mbaig@cs.stonybrook.edu.

Dongli Zhang is a PhD candidate in computer science at Stony Brook University. His research interests include system security, virtualization, and cloud computing. Zhang received an MS in computer science from Stony Brook University. Contact him at dozhang@cs.stonybrook.edu.

Donald E. Porter is an assistant professor of computer science at Stony Brook University. His research interests include system security, operating systems, and virtualization. Porter received a PhD in computer science from The University of Texas at Austin. Contact him at porter@cs.stonybrook.edu.

Radu Sion is an associate professor of computer science at Stony Brook University. His main interests lie in systems, cybersecurity, and efficient and large-scale computing. Sion received a PhD in computer science from Purdue University. Contact him at sion@cs.stonybrook.edu.