2014 IEEE Symposium on Security and Privacy

SoK: Introspections on Trust and the Semantic Gap

Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion
Stony Brook University
{*bpjain, mbaig, dozhang, porter, sion*}*@cs.stonybrook.edu*

***Abstract*—An essential goal of Virtual Machine Introspection (VMI) is assuring security policy enforcement and overall functionality in the presence of an untrustworthy OS. A fundamental obstacle to this goal is the difficulty in accurately extracting semantic meaning from the hypervisor's hardware-level view of a guest OS, called the semantic gap. Over the twelve years since the semantic gap was identified, immense progress has been made in developing powerful VMI tools.**

Unfortunately, much of this progress has been made at the cost of reintroducing trust into the guest OS, often in direct contradiction to the underlying threat model motivating the introspection. Although this choice is reasonable in some contexts and has facilitated progress, the ultimate goal of reducing the trusted computing base of software systems is best served by a fresh look at the VMI design space.

This paper organizes previous work based on the essential design considerations when building a VMI system, and then explains how these design choices dictate the trust model and security properties of the overall system. The paper then observes portions of the VMI design space which have been under-explored, as well as potential adaptations of existing techniques to bridge the semantic gap without trusting the guest OS.

Overall, this paper aims to create an essential checkpoint in the broader quest for meaningful trust in virtualized environments through VM introspection.

***Keywords*-VM Introspection, semantic gap, trust.**

I. Introduction

Virtualization has the potential to greatly improve system security by introducing a sensible layering—separating the policy enforcement mechanism from the component being secured.

Most legacy OSes are both monolithic and burdened with a very wide attack surface. A legacy OS, such as Linux, executes all security modules in the same address space and with the same privilege level as the rest of the kernel [91]. When this is coupled with a porous attack surface, malicious software can often load code into the OS kernel which disables security measures, such as virus scanners and intrusion detection. As a result, users have generally lost confidence in the ability of the OS to enforce meaningful security properties. In cloud computing, for instance, customers' computations are isolated using virtual machines rather than OS processes.

In contrast, hypervisors generally have a much narrower interface. Moreover, bare metal, or Type I [76], hypervisors generally have orders of magnitude fewer lines of code than a legacy OS. Table I summarizes the relative size of a representative legacy OS (Linux 3.13.5), and a representative bare-metal hypervisor (Xen 4.4), as well as comparing the number of reported exploits in both systems over the last 8 years. Perhaps unsurprisingly, the size of the code base and API complexity are strongly correlated with the number of reported vulnerabilities [85]. Thus, hypervisors are a much more appealing foundation for the trusted computing base of modern software systems.

This paper focuses on systems that aim to assure the functionality required by applications using a legacy software stack, secured through techniques such as **virtual machine introspection (VMI)** [46]. A number of valuable research projects observe that a sensitive application component, such as a random number generator or authentication module, requires little functionality, if any, from the OS, yet are vulnerable to failures of the OS [68, 69]. These projects are beyond the scope of this paper, which instead focuses on systems that leverage virtualization to ensure security properties for applications that require legacy OS functionality.

VMI has become a relatively mature research topic, with numerous projects. This paper distills key design points from previous work on VMI—providing readers and system designers with a framework for evaluating design choices.

Moreover, we observe an unfortunate trend in the literature: many papers do not explicate their assumptions about the system, trusted computing base, or threat models. Although an attentive reader can often discern these facts, this trend can create confusion within the field. Thus, this survey carefully explicates the connection between certain design choices and the fundamental trust assumptions underlying these designs. One particularly salient observation is that all current solutions to the **semantic gap** problem [34] implicitly assume the guest OS is benign. Although this is a reasonable assumption in many contexts, it can become a stumbling block to the larger goal of reducing the size of the trusted computing base.

Finally, after identifying key design facets in previous work, this paper identifies promising under-explored regions of the design space. The paper discusses initial work in these areas, as well as the applicability of existing techniques and more challenging threat models.

The contributions and insights of this work are as follows:

- A thorough survey of research on VMI, and a distillation of the principal VMI design choices.
- An analysis of the relationship between design choices

DOI 10.1109/SP.2014.45

Codebase	Lines of code
Xen hypervisor 4.4	0.50 Million
Linux kernel 3.13.5	12.01 Million
Codebase	**No. of CVE**
Xen hypervisor	24
Linux kernel	903

Table I
SIZE AND DOCUMENTED VULNERABILITIES OF A REPRESENTATIVE BARE-METAL HYPERVISOR (XEN) AND LEGACY OS (LINUX). CODE SIZES WERE CALCULATED BASED ON XEN 4.4 AND LINUX 3.13.5. CVES WERE COLLECTED FOR ALL VERSIONS OF THESE CODE BASES OVER THE PERIOD FROM 01/01/2006 TO 03/03/2014.

and implicit assumptions and trust. We observe that existing solutions to the semantic gap problem inherently trust the guest OS, often in direct contradiction to the underlying motivation for using VM introspection.

- The observation that the semantic gap problem has evolved into two separate issues: an engineering challenge and a security challenge. Existing solutions address the engineering challenge.
- Identifying a connection between techniques that protect memory and prevent attacks.
- Exploring the applicability of current techniques to new problems, such as removing the guest OS from the trusted computing base without removing OS functionality.
- Identifying additional points in the design space that are under-explored, such as hardware-support for mutual distrust among system layers and dynamic learning from an untrusted OS.

II. BACKGROUND

The specific goals of VM introspection systems vary, but commonly include identifying if a malicious loadable kernel module, or **rootkit**, has compromised the integrity of the guest OS [75]; identifying malicious applications running on the system [65]; or ensuring the integrity or secrecy of sensitive files [51]. In these systems, a monitor tracks the behavior of each guest OS and either detects or prevents policy violations. Such a monitor may be placed in the hypervisor, a sibling VM, in the guest itself, or in the hardware, as illustrated in Figure 1. This process of looking into a VM is Virtual Machine Introspection (VMI).

A fundamental challenge to using VMI for security policy enforcement is that many desirable security policies are expressed in high-level, OS abstractions, such as files and processes, yet the hypervisor only has direct visibility into hardware-level abstractions, such as physical memory contents and hardware device operations. This disparity in abstractions is known as the **semantic gap**.

As an example of how the semantic gap creates challenges for introspection, consider how a hypervisor might go about listing the processes running in a guest OS. The hypervisor can access only hardware-level abstractions, such as the CPU registers and contents of guest memory pages. The hypervisor must identify specific regions of guest OS

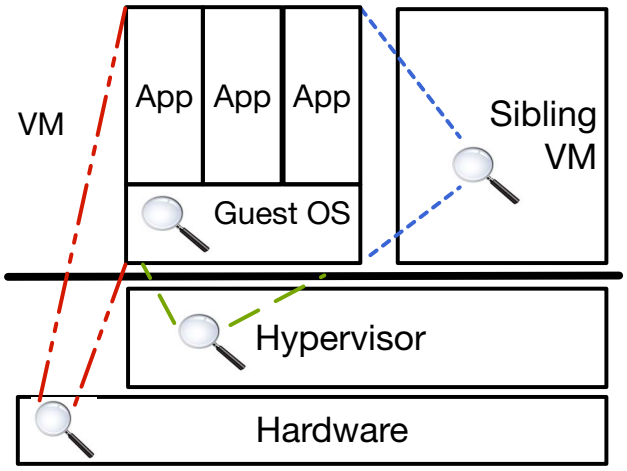

Figure 1. Monitor placement options in VMI: in a sibling VM, the hypervisor, in the guest OS itself, or in hardware. In-guest and hardware solutions require some assistance from the hypervisor.

memory that include process descriptors, and interpret the raw bytes to reconstruct semantic information, such as the command line, user id, and scheduling priorities.

As a result of the semantic gap, much of the VMI development effort goes into reconstructing high-level semantic information from low-level sources. VMI tools attempt to reconstruct a range of information, including the set of running processes, sensitive file contents, and network sockets. For brevity, we limit this paper to memory introspection, where the hypervisor draws inferences about guest behavior from the contents of memory and CPU registers. A range of work has also introspected disk contents [54, 87, 93] and network traffic [48, 56]; at this boundary, we limit discussion to in-memory data structures, such as those representing file metadata (`inode`) or a socket (`sk_buff`).

As we discuss in the next section, many of these semantic reconstruction techniques rely on fragile assumptions or are best-effort. Unfortunately, errors in reconstructing semantic information can be exploited by malware to trick an introspection-based security monitor.

Continuing our example of listing processes in a guest OS, a typical introspection strategy would be to identify the definition of a process descriptor (e.g., a `task_struct` on Linux) from the source code, and then walk the list of runnable processes by following the global root of the process list `init_task`, overlaying this structure definition over the relevant memory addresses. This strategy faces a number of challenges. First, one must either assume all process descriptors are in this list—even in a compromised or malicious OS—or one must detect hidden processes, using techniques such as scanning all of guest memory looking for potential process descriptors or detecting inconsistencies between the currently loaded page tables and the purported process descriptor [55]. Hidden process detection faces additional challenges, such as false positives from scanning memory during a critical section which temporarily

violates some internal invariant the introspection tool is checking. In order to prevent the guest OS from using a hidden process descriptor, the introspection must identify all context switching code in the kernel, possibly including dynamically loaded code which manually context switches a hidden process. Finally, a rootkit might hide itself in a subtle and unexpected manner, such as loading itself as a thread in the address space of a benign system process, or placing its code in the memory image of a common library and scheduling itself by changing the address of a signal handling function.

These subtleties make robustly bridging the semantic gap quite a challenge. The next section organizes current strategies to solve this problem.

III. Bridges across the semantic gap

Modern OSes are complex systems consisting of thousands of data structure types, and many instances of each type. A typical running instance of the Linux kernel was found to have a core set of 29,488 data structure instances belonging to 231 different types that enable scheduling, memory management, and I/O [77]. Each of these structures consists of many fields. For instance, a *task_struct* in Linux 3.10 contains more than 50 fields [14], many of which are pointers to other structures. A key ingredient to any solution to the semantic gap problem is **reconstruction** of kernel data structures from memory contents.

This section begins with explaining techniques to reconstruct kernel data structures (III-A), followed by additional introspection techniques that do not directly reconstruct data structures (§III-B–III-C), and then techniques that assure the integrity of the kernel binary (§III-D). As the section explains each technique, it highlights the underlying trust assumption(s)—most commonly that the guest OS is benign. We will revisit these trust assumptions as we explain VMI attacks and defenses (§V). as well as discussing how one might adapt VMI to a stronger threat model where these assumptions do not hold (§VI).

A. Learning and Reconstruction

Data structure reconstruction generally relies on a learn and search methodology. A **learning phase** is used to extract information relevant to data structures, generally a data structure **signature**. A signature can be used to identify and reconstruct data structure instances within kernel memory contents. Signatures are created using techniques such as expert knowledge, source analysis, or dynamic analysis—each described in this subsection (§III-A1–III-A3).

A second **search phase** identifies instances of the data structure. The two most common search strategies are to either linearly scan kernel memory or to traverse data structure pointers, starting with public symbols. Depending on the OS, public symbols may include debugging symbols or the dynamic linking tables exposed to loadable kernel modules. It is arguable which approach is more efficient, since many kernel data structures can have cyclic or invalid pointers, but may require traversing less total memory. However, the linear scan of kernel memory has the advantage that it is robust to "disconnected" structures or other attempts to obfuscate pointers. Both techniques can observe *transient* states when searching concurrently with OS operation, discussed further in §IV-A.

Several linear scanning techniques limit the search space by introspecting on the kernel memory allocators—either by interpreting allocator data structures [51] or by placing debugging breakpoints on the allocator [77]. OS kernels commonly use a different slab or memory pool for each object type; this information can be used to further infer data structure types. An advantage of leveraging heap-internal information for search is more easily identifying transient data structures which have been freed but may be pointed to—a challenge for other search approaches. An inherent risk of this approach is missing data structures allocated in an unorthodox manner.

Searching overheads: In practice, searching for data structures in a kernel memory snapshot can take from tens of milliseconds [51] up to to several minutes [31]. Thus, most systems reduce overheads by searching periodically and asynchronously (§IV-A). Periodic searches fundamentally limit these approaches to detecting compromises after the fact, rather than preventing policy violations. Moreover, these approaches can only reliably detect compromises that make persistent changes to a data structure. Transient malware can slip through the cracks between two searches of kernel memory.

The rest of this subsection describes the three major approaches to learning data structure signatures.

1) Hand-crafted data structure signatures: Introspection and forensic analysis tools initially used hand-crafted signatures, based on expert knowledge of the internal workings of an OS. For instance, such a tool might generate the list of running processes, similar to `ps`, by walking a global task list in Linux. Examples of this approach include Memparser [11], KNTLIST [9], GREPEXEC [4] and many others [1, 2, 3, 5, 10, 12, 13, 16, 17, 81, 82].

The most sophisticated frameworks for hand-crafted reconstruction use a wide range of subtle invariants and allow users to develop customized extensions. FACE/Ramparser [37] is a scanner that leverages invariants on values within a data structure, such as enumerated values and pointers that cannot be null. Ramparser can identify running processes (*task_struct*), open network sockets (*struct_sock*), in-kernel socket buffers (*sk_buff*), loaded kernel modules (struct *module*), and memory-mapped and open files for any given process (*vm_area_struct*). Similarly, Volatility [15] is a framework for developing forensics tools that analyze memory snapshots, with a focus on helping end-users to write extensions. Currently, Volatility includes tools that

extract a list of running processes, open network sockets and network connections, DLLs loaded for each process, OS kernel modules, system call tables, and the contents of a given process's memory.

Hand-crafting signatures and data structure reconstruction tools creates 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 2–3 months; bugfix updates to a range of older kernels are released as frequently as every few weeks. Each of these releases may 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 hand-written tools. Hand-written tools cannot keep pace with this release schedule and variety of OS kernels and compilers; thus, most introspection research has instead moved toward automated techniques.

2) Source code analysis: Automated reconstruction tools may rely on source code analysis or debugging information to extract data structures definitions, as well as leverage sophisticated static analysis and source invariants to reduce false positives during the search phase. Examples of source code analysis tools include SigGraph [64], KOP [31], and MAS [38].

One basic approach to source analysis is to identify all kernel object types, and leverage points-to analysis to identify the graph of kernel object types. Kernel Object Pinpointer (KOP) [31] extended a fast aliasing analysis developed for non-security purposes [49], with several additional features, including: field-sensitivity, allowing KOP to differentiate accesses made to different fields within the same struct; context-sensitivity, differentiating different uses of `union` types and `void` pointers based on type information in code at the call sites; as well as inter-procedural and flow-insensitive analysis, rendering the analysis robust to conditional control flow, e.g., `if` statements. In applying the static analysis to a memory snapshot, KOP begins with global symbols and traverses all pointers in the identified data structures to generate a graph of kernel data structures.

A key challenge in creating this graph of data structures is that not all of the pointers in a data structure point to valid data. As a simple example, the Linux `dcache` uses deferred memory reclamation of a directory entry structure, called a `dentry`, in order to avoid synchronization with readers. When a `dentry` is on a to-be-freed list, it may 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. MAS [38] addresses the issue of invalid pointers by extending the static analysis to incorporate value and memory alias checks.

Systems like MAS [38], KOP [31] and LiveDM [77] also improve the accuracy of type discovery by leveraging the fact that most OSes create object pools or slabs for each object type. Thus, if one knows which pages are assigned to each memory pool, one can reliably infer the type of any dynamically allocated object. We hasten to note that this assumption can be easily violated by a rootkit or malicious OS, either by the rootkit creating a custom allocator, or allocating objects of greater or equal size from a different pool and repurposing the memory. Thus, additional effort is required to detect unexpected memory allocation strategies.

SigGraph [64] contributed the idea that the graph structure of the pointers in a set of data structures can be used as a signature. As a simple example, the relationships of pointers among `task_struct` structures in Linux is fundamentally different than among `inode` structures. SigGraph represents graph signatures in a grammar where each symbol represents a pointer to a sub-structure. This signature grammar can be extended to encode arbitrary pointer graphs or encode sub-structures of interest. SigGraph is designed to work with a linear scan of memory, rather than relying on reachability from kernel symbols.

3) Dynamic Learning: Rather than identifying code invariants from kernel source code, VMI based on dynamic analysis learns data structure invariants based on observing an OS instance [24, 41, 64].

By analogy 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 more quickly discern which memory regions might include a structure of interest. Tools such as Daikon [43] are used to generate constraints based on observed values in fields of a given data structure.

Several dynamic systems have created **robust signatures**, which are immune to malicious changes to live data structure instances [41]. More formally, a robust signature identifies any memory location that could be used as a given structure type without false negatives. A robust signature can have false positives. Robust signatures are constructed through fuzz testing during the training phase to identify invariants which, if violated, will crash the kernel [41, 64]. For instance, RSFKDS begins its training phase with a guest in a clean state, and then attempts to change different data structure fields. If the guest OS crashes, this value is used to generate a new constraint on the potential values of that field. 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. The key insight is that these attempts are only fruitful inasmuch as they do not crash the OS kernel. Thus, robust signatures leverage invariants an attacker cannot safely violate.

B. Code Implanting

A simpler approach to bridging the semantic gap is to simply inject code into the guest OS that reports semantic information back to the hypervisor. For example, Process implanting [47] implants and executes a monitoring process within a randomly-selected process already present in the VM. Any malicious agent inside the VM is unable to predict which guest process has been replaced and thus the injected code can run without detection. Rather than implant a complete process, SYRINGE [30] implants functions into the kernel, which can be called from the VM.

A challenge to implanting code is ensuring that the implanted code is not tampered with, actually executes, and that the guest OS components it uses report correct information. SIM [84] uses page table protections to isolate an implanted process's address space from the guest OS kernel. Section III-D discusses techniques to ensure the integrity of the OS kernel. Most of these implanting techniques ultimately rely on the guest kernel to faithfully represent information such as its own process tree to the injected code.

C. Process Outgrafting

In order to overcome the challenges with running a trusted process inside of an untrusted VM, process outgrafting [86] relocates a monitoring process from the monitored VM to a second, trusted VM. The trusted VM has some visibility into the kernel memory of the monitored VM, allowing a VMI tools to access any kernel data structure without any direct interference from an adversary in the monitored VM.

Virtuoso [40] automatically generates introspection programs based on dynamic learning from a trusted VM, and then runs these tools in an outgrafted VM. Similarly, OSck [51] generates introspection tools from Linux source which execute in a monitoring VM with a read-only view of a monitored guest.

VMST [44] generalizes this approach by eliminating the need for dynamic analysis or customized tools; rather, a trusted, clean copy of the OS runs with a roughly copy-on-write view of the monitored guest. Monitoring applications, such as `ps`, simply execute in a complete OS environment on the monitoring VM; each system call executed actually reads state from the monitored VM. VMST has been extended with an out-of-VM shell with both execute and write capabilities [45], as well as accelerated by using memoization, trading some accuracy for performance [80]. This approach bridges the semantic gap by repurposing existing OS code.

The out-grafting approach has several open problems. First, if the monitoring VM treats kernel data as copy-on-write, the monitoring VM must be able to reconcile divergences in the kernel views. For example, each time the kernel accesses a file, the kernel may update the inode's `atime`. These atime updates will copy the kernel data, which must be discarded for future introspection or view of the file system will diverge. VMST does not address this problem, although it might be addressed by an expert identifying portions of the kernel which may safely diverge, or resetting the VM after an unsafe divergence. Similar to the limitations of hand-crafted introspection tools, each new OS variant may require hand-updates to divergent state; thus, automating divergence analysis is a useful topic for future work. Finally, this approach cannot handle policies that require visibility into data on disk—either files or swapped memory pages.

D. Kernel executable integrity

The introspection approaches described above assume that the executable kernel code does not change between creation of the introspection tools and monitoring the guest OS. Table II lists additional assumptions made by these techniques.

In order to uphold the assumption that the kernel has not changed, most hypervisor-based security systems must also prevent or limit the ability of the guest OS to modify its own executable code, e.g., by overwriting executable pages or loading modules. This subsection summarizes the major approaches to ensuring kernel binary integrity.

1) The (Write $\oplus$ Execute) principle: The W $\oplus$ X principle prevents attacks that write the text segment by enforcing a property where all the pages are either writable or executable, but not both at the same time. For instance, SecVisor [83] and NICKLE [79] are hypervisors that enforce the W $\oplus$ X principle by setting page table permissions on kernel memory. SecVisor will only set executable permission on kernel code and loadable modules that are approved by an administrator, and prevent modification of this code.

Although non-executable (NX) page table bits are ubiquitous on modern x86 systems, lack of NX support complicated the designs of early systems that enforced the W $\oplus$ X principle. Similarly, compilers can mix code and data within the same page, although security-conscious developers can also restrict this with linker directives.

2) Whitelisting code: As discussed above, SecVisor and NICKLE policies require a notion of approved code, which is represented by a whitelist of code hashes created by the administrator.

Patagonix [65] extends this property to application binaries, without the need to understand the structure of processes and memory maps. Patagonix leverages no-execute page table support to receive a trap the first time data from a page of memory is loaded into the CPU instruction cache. These pages are then compared against a database of whitelisted application binary pages.

Although whitelisting code can prevent loading unknown modules which are the most likely to be malicious, the approach is limited by the administrator's ability to judge whether a driver or OS kernel is malicious *a priori*.

3) Object code hooks: A practical limitation of the W $\oplus$ X principle is that many kernels place function pointers in

Technique	Assumptions	Monitor Placement	Systems
Hand-crafted data structure signatures (Expert knowledge)	• Expert knowledge of OS internals for known kernel version • Guest OS is not actively malicious	Sibling VM, hypervisor, or hardware	[1, 2, 3, 4, 5, 9, 10, 11, 12, 13, 16, 17, 81, 82]
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 • Attacks will persist long enough for periodic scans	Sibling VM, hypervisor, or hardware	[24, 31, 38, 41, 64, 77]
Code implanting (VMM protects monitoring agent inside guest OS)	• Malicious guest schedules monitoring tool and reports information accurately	Guest with hypervisor protection	[30, 47, 84]
Process outgrafting (Reuse monitoring tools from sibling VM with shared kernel memory)	• Live, benign copy of OS behaves identically to monitored OS	Sibling VM	[40, 44, 45, 86]
Kernel executable integrity (Protect executable pages and other code hooks)	• Initial benign version of monitored OS • Administrator can white-list safe modules	Hypervisor	[65, 73, 79, 83, 89]

Table II
VMI TECHNIQUES, MONITOR PLACEMENT (AS ILLUSTRATED IN FIGURE 1, AND THEIR UNDERLYING TRUST ASSUMPTIONS.

data objects that must be writable. These function pointers are used to implement a crude form of object orientation. For instance, the Linux VFS allows a low-level file system to extend generic routines for operations such as reading a file or following a symbolic link.

Lares [73] implemented a simple page-protection mechanism on kernel object hooks, but incurred substantial performance penalties because these executable pointers are in the same page as fields which the guest kernel must be able to write, such as the file size and modification time. HookSafe [89] addresses this problem by modifying OS kernel code to relocate all hooks to a read-only, shadow memory space. All code that calls a hook must also check that the requested hook is in the shadow memory space, and some policy must also be applied to approve which code can be added to the hook section. The hook redirection and checking code is in the kernel's binary text, and is read-only. HookSafe identifies locations where hooks are called through dynamic learning (§III-A); this could likely be extended with static analysis for more complete coverage.

Ultimately, these techniques are approximating the larger property of ensuring control flow integrity (CFI) of the kernel [18]. Ensuring CFI is a broad problem with a range of techniques. For instance, Program Shepherding [59] protects the integrity of implanted functions [30] (§III-B), using a machine code interpreter to monitor all control transfers and guarantee that each transfer satisfies a given security policy. Discovering efficient CFI mechanisms is a relevant, but complimentary problem to VMI.

IV. PREVENTION VS. DETECTION

Some introspection tools prevent certain security policy violations, such as execution of unauthorized code, whereas others only detect a compromise after the fact. Clearly, prevention is a more desirable goal, but many designs accept detection to lower performance overheads. This section discusses how certain design choices fundamentally dictate whether a system can provide detection or prevention.

Prevention requires a mechanism to identify and interpose on a low-level operation within a VM which violates a system security policy. Certain goals map naturally onto hardware mechanisms, such as page protections on kernel code or hooks [73, 79, 83, 89]. Other goals, such as upholding data structure invariants the kernel code relies upon, are open questions.

As a result, violations of more challenging properties are currently only **detected** after the fact by VMI tools [24, 39, 40, 44, 51, 64, 65, 74, 75, 77, 80, 84]. In general, there is a strong connection between approaches that periodically search memory and detection. Periodic searching is a good fit for malware that persistently modifies data structures, but can miss transient modifications. To convert these approaches to prevention techniques would require interposing on every store, which is prohibitively expensive. Moreover, because some invariants span multiple writes, even this strawman approach would likely yield false negatives without even deeper analysis of the code behavior.

Current detection systems usually just power off a compromised VM and alert an administrator. Several research projects identify how systems can recover from an intrusion or other security violation [32, 42, 57, 58] In general,

general-purpose solutions either incur relatively high overheads to track update dependencies (35% for the most recent general-purpose, single-machine recovery system [58]), or leverage application-specific properties. Improving performance and generality of recovery systems is an important direction for future work.

A. Asynchronous Vs Synchronous Mechanisms

Synchronous mechanisms mediate guest operations inline to prevent security policy violations, or receive very low latency notification of changes. All prevention systems we surveyed [73, 79, 83, 84, 89] use synchronous mechanisms, such as page protection or code implanting. Several low-latency detection systems use customized hardware, discussed further in §IV-B. A few systems also use synchronous mechanisms on commodity hardware for detection [60, 65, 77], but could likely lower their overheads with an asynchronous mechanism.

Asynchronous mechanisms execute concurrently with a running guest and inspect its memory. These systems generally introspect into a snapshot of memory [24, 39, 64, 74] or a read-only or copy-on-write view of guest memory [40, 44, 51, 53, 75, 80]. All surveyed asynchronous systems detect rootkits after infection through passive monitoring.

On one hand, the synchronous systems gain a vantage point over their counterparts against transient attacks but increase the overhead for the guest OS being protected. On the other hand, asynchronous systems introduce lower monitoring overhead but miss cleverly built transient attacks; they are also limited due to the inherent race condition between the attacker and the detection cycle.

Synchronous and asynchronous mechanisms make fundamental trade-offs across the performance, frequency of policy-relevant events, risk, and assumptions about the behavior of the system. Synchronous mechanisms tend to be more expensive, and are generally only effective when the monitored events are infrequent, such as a change in the access pattern to a given virtual page. The cost of an asynchronous search of memory can also be quite high (ranging from milliseconds [51] to minutes [31]), but the frequency can be adjusted to an acceptable rate—trading risk for performance. Both synchronous and asynchronous systems make potentially fragile assumptions about the system to improve performance, such as knowing all hook locations or assuming all objects of a given type are allocated from the same slab. These risks could be reduced in future work by identifying low-frequency events that indicate a policy violation, are monitorable without making fragile assumptions about the system, and introduce little-to-no overheads in the common case.

A final issue with executing introspection concurrently with the execution of an OS is false positives arising because of **transient states**. In general, an OS may violate its own invariants temporarily while executing inside of a critical section. A correct OS will, of course, restore the invariants before exiting the critical section. If an introspection agent searches memory during a kernel critical section, it may observe benign violations of these invariants, which will resolve quickly. Current approaches to this problem include simply looking for repeated violations of an invariant (leaving the system vulnerable to race conditions with an attacker), or only taking memory snapshots when the OS cannot be in any critical sections (e.g., by preempting each CPU while out of the guest kernel).

> Current VMI systems face fundamental trade-offs between performance and risk, often making fragile assumptions about the guest OS.

B. Hardware-Assisted Introspection

Several research prototypes have employed customized hardware for introspection [60, 67, 71], or applied existing hardware in novel ways [22, 74, 88]. The primary division within the current design space of hardware-assisted introspection is whether the introspection tool uses memory snapshots or snoops on a bus. Snooping can monitor memory regions at finer granularity than page protections, reducing overheads.

1) Snapshotting: One strategy for hardware-assisted introspection is using a PCI device to take RAM snapshots, which are sent to a second machine for introspection (monitored and monitor, respectively). For instance, Copilot [74] adds an Intel StrongARM EBSA-285 Evaluation Board on the monitored machine's PCI bus. The PCI device on the monitored machine uses DMA requests to retrieve a snapshot of host RAM, which is sent to the monitor machine upon request over an independent communication link. The monitor periodically requests snapshots and primarily checks that the hash of the kernel binary text and certain code pointers, such as the system call table, have not changed from known-good values.

Unfortunately, a memory snapshot alone isn't sufficient to robustly reconstruct and interpret a snapshot. Of particular importance is the value of the `cr3` register, which gives the physical address of the root of the page tables. Without this CPU register value, one cannot reliably reconstruct the virtual memory mapping. Similarly, a system can block access to regions of physical memory using an IOMMU [20, 27].

HyperCheck [88] augments physical memory snapshots with the contents of the `cr3` register, using the CPU System Management Mode (SMM) [6]. SMM is an x86 CPU mode designed primarily for firmware, power management, and other system functions. SMM has the advantage of protecting the introspection code from the running system as well as giving access to the CPU registers, but must also preempt the system while running (i.e., this is a synchronous mechanism). The processor enters SMM when the SMM interrupt pin (SMI) is raised, generally by the Advanced

Programmable Interrupt Controller (APIC). The hypervisor is required to create SMI interrupts to switch the CPU to SMM mode. Upon entering SMM, the processor will launch a program stored in system management RAM (SMRAM). SMRAM is either a locked region of system DRAM, or a separate chip, and ranges in size from 32 KB to 4 GB [6] Outside of SMM, SMRAM may not be read or written. Within SMM, the integrity checking agent has unfettered access to all RAM and devices, and is not limited by a IOMMU or other attacks discussed previously. Unfortunately, SMM mode also has the limitation that Windows and Linux will hang if any software spends too much time in SMM, bounding the time introspection code can take.

HyperSentry [22] further refines this model by triggering an SMI handler from an Intelligent Platform Management Interface (IPMI) device. IPMI devices generally execute system management code, such as powering the system on or off over the network, on a device hidden from the system software.

A limitation of any SMM-based solution, including the ones above, is that a malicious hypervisor could block SMI interrupts on every CPU in the APIC, effectively starving the introspection tool. For VMI, trusting the hypervisor is not a problem, but the hardware isolation from the hypervisor is incomplete.

Each of these systems focus on measuring the integrity of system software—e.g., checking that the executable pages have a known-good hash value. At least in SMM mode, more computationally expensive introspection may be impractical. Because all of these operations operate on periodic snapshots, which may visibly perturb memory access timings, a concern is that an adversary could predict the snapshotting interval and race with the introspection agent. In order to ensure that transient attacks cannot race with the snapshot creation, more recent systems have turned to snooping, which can continuously monitor memory changes.

2) Snooping: A number of recent projects have developed prototype security hardware that snoops on the memory bus [60, 67, 71]. These systems have the useful function of efficiently monitoring writes to sensitive code regions; unlike page protections, snooping systems can monitor writes at the finer granularity of cache lines, reducing the number of needless checks triggered by memory accesses adjacent to the structure being monitored. These systems can also detect updates to memory from a malicious device or driver by DMA, which page-level protections cannot detect.

Although most prototypes have focused on detecting modifications to the kernel binary itself, KI-Mon also watches for updates to object hooks [60], and there is likely no fundamental reason other solutions could not implement this.

Because these snooping devices aim to be very lightweight, they cannot then check data structure invariants or code integrity, but instead signal a companion snapshotting device (as discussed above) to do these checks. However, a specific memory event triggering asynchronous checks is a clear improvement over periodic snapshots, in both efficiency and risk of races with the attacker. A small complication with snooping-triggered introspection is that invariants often span multiple cache lines, such as `next.prev == next` in a doubly-linked list. If an invariant check is triggered on the first write in a critical section, the system will see many false positives. KI-Mon addresses this by waiting until the system quiesces.

We note that these systems do not use commodity hardware, but are implemented in simulators or FPGAs. Section VI-B argues that this is a promising area of research that deserves more attention, but more work has to be done to demonstrate the utility of the approach before it will be widely available. Similarly, these systems have initially focused on attack detection, but it would be interesting to extend these systems to recovering from a detected attack.

Snooping is useful for finer-grained memory monitoring.

C. Memory Protection: A necessary property for prevention

We end this section by observing that all prevention systems employ some form of memory protection to synchronously interpose on sensitive data writes. For example, HookSafe [89] and Lares [73] use memory protection to guard against unexpected updates to function pointers. In contrast, it isn't clear how to convert an asynchronous memory search from a detection into a prevention tool. The most likely candidate is with selective, fine-grained hardware memory bus snooping, described above. Thus, if attack prevention is a more desirable goal than detection after-the-fact, the community should focus more effort on discovering lightweight, synchronous monitoring mechanisms.

All current prevention systems rely on synchronous memory protection.

V. Attacks, Defense, and Trust

This section explains the three major classes of attacks against VMI, known defenses against those attacks, and explains how these attacks relate to an underlying trust placed in the guest OS. These issues are summarized in Table III.

A. Kernel Object Hooking

A Kernel Object Hooking (KOH) attack [8] attempts to modify function pointers (hooks) located in the kernel text or data sections. An attacker overwrites a function pointer with the address of a function provided by the attacker, which will then allow the attacker to interpose on a desired set of kernel operations. In some sense, Linux Security Modules provide similar hooks for security enhancements [91]; the primary difference is that KOH repurposes other hooks used for purposes such as implementing an extensible virtual file

system (VFS) model. The defenses against KOH attacks generally depend on whether the hook is located in the text or data segment.

1) Text section hooks: The primary text section hooks are the system call table and interrupt descriptor table. For instance, an attacker could interpose on all file `open` calls simply by replacing the pointer to the `sys_open()` function in the system call table.

In older OSes, these hooks were in the data segment despite not being dynamically changed by most OSes. In order to prevent malware from overwriting these hooks, most kernels now place these hooks in the read-only text segment. As discussed in §III-D1, a sufficient defense is hypervisor-imposed, page-level Write $\oplus$ Execute permissions.

2) Data section hooks: Kernel data section hooks are more difficult to protect than text section hooks. Data section hooks place function pointers in objects, facilitating extensibility. For instance, Linux implements a range of different socket types behind a generic API; each instantiation overrides certain hooks in the file descriptor for a given socket handle.

The fundamental challenge is that, although these hooks generally do not change during the lifetime of the object, they are often placed in the same page or even cache line with fields that do change. Because most kernels mix hooks which should be immutable with changing data, most hardware-based protection mechanisms are thwarted.

In practice, these hooks are very useful for rootkits to hide themselves from anti-malware tools inside the VM. For instance, the Adore-ng [36] rootkit overrides the `lookup()` and `readdir()` functions on the `/proc` file system directory. Process listing utilities work by reading the subdirectories for each running process under `/proc`; a rootkit that overrides these functions can filter itself from the `readdir()` system call issued by `ps`.

In order to defend against such attacks, the function pointers need to 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 which can be write-protected [89], or augment hooks in the kernel with checks against a whitelist of trusted functions [75].

Trust: Protecting the kernel code from unexpected modifications at runtime is clearly sensible. Underlying these defenses is the assumption that the kernel is initially trusted, but may be compromised later. The more subtle point, however, is that all of the VMI tools discussed in §III assume that the kernel text will not change. Thus, preventing text section modification is effectively a prerequisite for current VMI techniques.

Defenses against KOH on data hooks generally posit trust in the ability of an administrator to correctly identify trustworthy and untrustworthy kernel modules. As explained in Section III-D, KOH defenses assume that kernel modules are benign in order to provide some meaningful protections without solving the significantly harder problem of kernel control flow integrity in the presence of untrusted modules.

KOH defenses generally assume benign kernel modules.

Finally, we note that some published solutions to the KOH data section problem are based on best-effort dynamic analysis, which can miss hooks that are not exercised. There is no fundamental reason this analysis should be dynamic, other than the unavailability of source code. In fact, some systems do use static analysis to identify code hooks [51], which can identify all possible data section hooks.

B. Dynamic Kernel Object Manipulation

Manipulating the kernel text and code hooks are the easiest attack vector against VMI; once KOH defenses were developed, attackers turned their attention to attacks on the kernel heap. Dynamic Kernel Object Manipulation (DKOM) [28] attacks modify the kernel heap through a loaded module or an application accessing `/dev/mem` or `/proc/kcore` on Linux. DKOM attacks only modify data values, and thus are distinct from modifying the control flow through function hooks (KOH).

A DKOM attack works by invalidating latent assumptions in unmodified kernel code. A classic example of a DKOM attack is hiding a malicious process from a process listing tools, such as `ps`. The Linux kernel tracks processes in two separate data structures: a linked list for process listing and a tree for scheduling. A rootkit can hide malicious processes by taking the process out of the linked list, but leaving the malicious process in the scheduler tree. The interesting property is that loading a module can be sufficient to alter the behavior of unrelated, unmodified kernel code.

DKOM attacks are hard to prevent because they are a metaphorical needle in a haystack of expected kernel heap writes. As a result, most practical defenses attempt to identify data structure invariants, either by hand, static, or dynamic analysis, and then detect data structure invariant violations asynchronously. Because an attacker can create objects from any memory, not just the kernel heap allocator, data structure detection is also a salient issue for detecting DKOM attacks (§III-A). Thus, a robust, asynchronous DKOM detector must search all guest memory, increasing overheads, and tolerate attempts to obfuscate a structure.

Trust: DKOM defenses introduce additional trust in the guest beyond a KOH defense, and make several assumptions which an attacker can could be violated by an attacker. Most DKOM defenses work by identifying security-related data structure invariants. Because it is difficult for the defender to ever have confidence that all security-relevant invariants have been identified, this approach will generally be best-effort and reactive in nature. Deeper source analysis tools could yield more comprehensive invariant results, but more research is needed on this topic. Many papers on the

topic focus on a few troublesome data structures, such as the `task_struct`, yet Linux has several hundred data structure types. It is unclear whether any automated analysis will scale to the number of hiding places afforded to rootkits by monolithic kernels, or whether detection tools will always be one step behind attackers. That said, even a best-effort defense has value in making rootkits harder to write.

Another problematic assumption is that all security-sensitive fields of kernel data structures have invariants that can be easily checked in a memory snapshot. For instance, one might assume that any outgoing packets come from a socket that appears in the output of a tool such as netstat (or a VMI-based equivalent). Yet a malicious Linux kernel module could copy packets from the heap of an application to the outgoing IP queue—a point in the networking stack which doesn't maintain any information about the originating socket or process. Thus, memory snapshots alone couldn't easily identify an inconsistency between outgoing packets and open sockets, especially if the packet could have been sent by a different process, such as a process with an open raw socket. Although the problem in this example could be mitigated with continuous monitoring, such monitoring would substantially increase runtime overheads; in contrast, most DKOM defenses rely on infrequent scanning to minimize overheads. In this example, the data structure invariant spans a sequence of operations, which can't be captured with one snapshot.

> A single snapshot cannot capture data structure invariants that span multiple operations.

Third, 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 only demonstrate desirable behavior during the training phase. Although we hasten to note that this assumption may be generally reasonable, it is not beyond question that an OS vendor might include a backdoor that such a classifier would learn to treat as expected behavior.

In order to ensure that the guest kernel is benign, DKOM defenses generally posit a KOH defense. Learning code invariants is of little use when an attacker can effectively replace the code. The interesting contrast between KOH and DKOM defenses is that DKOM defenses can detect invalid data modifications *even 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 is robust to untrusted modules as well, which may require substantially stronger control flow integrity protection.

> KOH defenses are a building block for DKOM defenses, but often make different trust assumptions about modules.

Finally, these detection systems explicitly assume *malware will leave persistent, detectable modifications* and implicitly assume *malware cannot 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 using a read-only view of memory. Because DKOM detectors run in increments of seconds, short-lived malware can easily evade detection. Even for persistent rootkits, a reasonably strong adversary may also have access to similar data structure classifiers and aggressively search for invariants missed by the classifier.

If a rootkit can reliably predict when a DKOM detector will view kernel memory, the rootkit has the opportunity to temporarily repair data structure invariants—racing with the detector. Reading a substantial portion of guest memory can be is highly disruptive to cache timings—stalling subsequent writes on coherence misses. Similarly, solutions based on preempting the guest OS will leave telltale "lost ticks" on the system clock. Even proposed hardware solutions can be probed by making benign writes to potentially sensitive addresses and then observing disruptions to unrelated I/O timings. Given the long history of TOCTTOU and other concurrency-based attacks [29, 92], combined with a likely timing channel induced by the search mechanism and recent successes exploiting VM-level side channels [94], the risk of an attacker successfully racing with a detector is concerning.

> DKOM defenses are potentially vulnerable to race conditions within their threat model.

C. Direct Kernel Structure Manipulation

Direct Kernel Structure Manipulation (DKSM) attacks [23] change the interpretation of a data structure between training a VMI tool and its application to classify memory regions into data structures. Simple examples of a DKSM attack include swapping two fields within a data structure or padding the structure with garbage fields so that relative offsets differ from the expectation of the VMI tool.

Because most VMI tools assume a benign kernel, a successful DKSM attack hinges on changing kernel control flow. The two previously proposed mechanisms are KOH attacks and return-oriented programming [66]. As discussed above, a number of successful countermeasures for KOH attacks have been developed, as have effective countermeasures to return-oriented programming, including G-Free [72], "Return-Less" kernels [62], and STIR [90].

Trust: DKSM is somewhat of an oddity in the literature because it is 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 to "split-personality" malware that behaves differently when it detects that it is under analysis.

> DKSM is a reasonable concern obviated by generous threat models.

Attack	Defense	Trust Assumption
Write text Segment	Hypervisor-enforced W $\oplus$ X.	Initial text segment benign.
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.
DKOM (heap)	Identify data structure invariants, 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. • Attackers cannot win races with the monitor.
DKSM	Prevent Bootstrapping through KOH or ROP.	OS is benign; behaves identically during training and classification.

Table III
VMI ATTACKS, DEFENSES, AND UNDERLYING TRUST ASSUMPTIONS.

D. The semantic gap is really two problems

Under a stronger threat model, the DKSM attack effectively leverages the semantic gap to thwart security measures. Under DKSM, a malicious OS actively misleads VMI tools in order to violate a security policy.

In the literature on VM introspection, the semantic gap problem evolved to refer to two distinct issues: (1) the engineering challenges of generating introspection tools, possibly without source code [40, 44, 80], and (2) the ability of a malicious or compromised OS to exploit fragile assumptions underlying many introspection designs in order to evade a security measure [51, 64, 77, 83, 89]. These assumptions include:

- Trusting that the guest OS is benign during the training phase, and will not behave differently under monitoring.
- All security-sensitive invariants and hooks can be automatically learned.
- Attacks will persist long enough to be detected by periodic searches.
- Administrators can whitelist trustworthy kernel modules.

Most papers on introspection focus on the first problem, which has arguably been solved [40, 44, 80], yet interesting attacks leverage the second issue, which is still an open problem, as is reliable introspection under stronger threat models.

Unfortunately, the literature has not clearly distinguished these problem variations, and only a close reading will indicate which one a given paper is addressing. This confusion is only exacerbated when one attempts to place these papers next to each other in the context of attacks and defenses. That said, we do believe that the overall path of starting with a weak attacker and iteratively strengthening the threat model is a pragmatic approach to research in this area; the issue is ambiguous nomenclature.

We therefore suggest a clearer nomenclature for the two sub-problems: the **weak** and **strong** semantic gap problems. The weak semantic gap is the largely solved engineering challenge of generating VMI tools, and the strong semantic gap refers to the challenge of defending against an adversarial, untrusted guest OS. A solution to the open strong semantic gap problem would not make any assumptions about the guest OS being benign during a training phase or accept inferences from guest source code as reliable without runtime validation. The strong semantic gap problem is, to our knowledge, unsolved, and the ability to review future work in this space relies on clearer delineation of the level of trust placed in the guest OS. A solution to the strong semantic gap problem would also prevent or detect DKSM attacks.

> The weak semantic gap is a solved engineering problem.
> The strong semantic gap is an open security problem.

VI. TOWARD AN UNTRUSTED OS

Any solution to the strong semantic gap problem may need to remove assumptions that the guest OS can be trusted to help train an introspection tool. As illustrated in Section III, most existing introspection tools rely on the assumption that the guest OS begins in a benign state and its source code or initial state can be trusted. Over time, several designs have reduced the degree to which they rely on the guest OS. It is not clear, however, that continued iterative refinement will converge on techniques that eliminate trust in the guest.

Table IV illustrates the space of reasonable trust models in virtualization-based security. Although a lot of effort in VMI has gone into the first row (the weak semantic gap), the community should focus on new directions likely to bridge the strong semantic gap (second row), as well as adopt useful techniques from research into the other rows.

This section identifies promising approaches to the strong semantic gap, based on insights from the literature.

A. Paraverification

Many VMI systems have the implicit design goal of working with an unmodified OS, or limiting modifications to the module loader and hooks. The goal of introspecting on an unmodified guest OS often induces trust in the guest OS to simplify this difficult problem. Specifically, most VMI tools assume the guest OS is not actively malicious and adheres to the behavior exhibited during the learning phase.

This subsection observes that, rather than relax the threat model for VMI, relaxing the requirement of an unmodified

App	Guest OS	Hypervisor	Challenge	Solutions
	√	√	Weak Semantic Gap	Layered Security, VMI. Incrementally reduce trust in the guest OS.
		√	Strong Semantic Gap	Difficult to solve. Need techniques that can learn from untrusted sources and detect inconsistencies during VMI.
√		√	Untrusted guest OS	Paraverification. Application trust bridges the semantic gap.
√	√		Untrusted cloud hypervisor	Support from trusted hardware like SGX [7, 70].
√			Untrusted guest OS and hypervisor	Fine grained support from trusted hardware needed.

Table IV
TRUST MODELS. (√ INDICATES THE LAYERS THAT ARE TRUSTED.)

OS may be a more useful stepping stone toward an untrusted OS. By analogy, although initial hypervisors went through heroic efforts to virtualize unmodified legacy OSes on an ISA very unsuitable for virtualization [26], most modern OSes now implement paravirtualization support [25]. Essentially, paravirtualization makes small modifications to the guest OS that eliminate the most onerous features to emulate. For instance, Xen allowed the guest OS to observe that there were inaccessible physical pages, substantially reducing the overheads of virtualizing physical memory. The reason paravirtualization was a success is that it was easy to adopt, introduced little or no overheads when the system executes on bare metal, and dramatically improved performance in a VM.

Thus, we expect that light modifications to a guest OS to aid in introspection could be a promising direction. Specifically, we observe that the recent InkTag [52] system introduced the idea of *paraverification*, in which the guest OS provides the hypervisor with evidence that it is servicing an application's request correctly. The evidence offered by the guest OS is easily checked by the hypervisor without trusting the guest OS. For instance, a trusted application may request a memory mapping of a file, and, in addition to issuing an `mmap` system call, also reports the request to the hypervisor. When the OS modifies the application's page tables to implement the `mmap` system call, the OS also notifies the hypervisor that this modification is in response to a particular application request. The hypervisor can then do an end-to-end check that (1) the page table changes are applied to an appropriate region of the application's virtual memory, (2) that the CPU register values used to return to the application are sensible, and (3) that the contents of these pages match the expected values read from disk, using additional metadata storing hashes of file contents.

We hasten to note that the goals of InkTag are different from VMI—ensuring a trusted application can safely use functionality from a malicious OS. This problem has also been explored in a number of other papers [35, 63]. Moreover, InkTag leverages the trusted application to bridge the semantic gap—a strategy that would not be suitable for the types of problems VMI aims to solve. Nonetheless, forcing an untrusted OS to aid in its own introspection could be fruitful if the techniques were simple enough to adopt.

> Rather than relaxing the threat model for VMI, relax strict limits on guest modifications.

B. Hardware support for security

As we observe in §IV-C, Memory protection or other synchronous notification mechanisms appear to be a requirement to move from detection to prevention. Unfortunately, the coarseness of mechanisms in commodity hardware introduce substantial overheads. §IV-B summarizes recent work on memory monitoring at cache line granularity—a valuable approach meriting further research.

An interesting direction recently taken by Intel is developing a mutual distrust model for hardware memory protection, called Software Guard Extensions (SGX) [21, 50, 70]. SGX allows an OS or hypervisor to manage virtual-to-physical OS mappings for an application, but the lower-level software cannot access memory contents. SGX provides memory isolation of a trusted application from an untrustworthy software stack. Similar memory isolation has been provided by several software-only systems [35, 52], but at a substantial performance cost attributable to frequent traps to a trusted hypervisor. Finally, we note that in order for an application to safely *use* system calls on an untrusted OS, a number of other problems must be addressed [33, 52].

In the context of introspection or the strong semantic gap, hardware like SGX can also be useful for creating a finer-grained protection domain for code implanted in the guest OS III-B. More fine-grained memory protection and monitoring tools are needed from hardware manufacturers.

> Fine-grained memory protection and monitoring hardware can reduce overheads and trust.

C. Reconstruction from untrusted sources

Current tools that automatically learn data structure signatures assume the OS will behave similarly during training and classification (§V-B). Among the assumptions made in current VMI tools, this is one that potentially has the best chance of being incrementally removed. For example, one approach might train the VMI classifiers on the live OS, and continue incrementally training as the guest OS runs.

Another approach would be to detect inconsistencies between the training and classification stages of VMI. By

analogy, distributed fault tolerance systems are often built around the abstraction of a **proof of misbehavior**, where a faulty participant in the protocol generates signed messages to different participants that contradict one another [19, 61]. Similarly, one approach to assuring learning-based systems is to look for proof of misbehavior in the guest OS. For instance, Lycosid detected inconsistencies between the `cr3` register and the purported process descriptor's `cr3` value [55]. A proof of misbehavior may also include inconsistencies in code paths or data access patterns between the training and classification phases of introspection.

VMI should detect inconsistent behavior over the life of an OS, not just between training and classification.

VII. Under-explored issues

Based on our survey of the literature on VMI, we identify a few issues that deserve more consideration in future work.

A. Scalability

Many VMI designs are fairly expensive, especially designs that run a sibling VM on a dedicated core for analysis. For example, one state-of-the-art system reports overheads ranging from 9.3—500× [44]. There is a reasonable argument why high VMI overheads might be acceptable: the average desktop has idle cores anyway, which could be fruitfully employed to improve system security. However, this argument does not hold in a cloud environment, where all cores can be utilized to service additional clients. In a cloud, customers will not be pleased with doubling their bill, nor would a provider be pleased with halving revenue.

It is reasonable to expect that VMI would be particularly useful on a cloud or other multi-VM system. Thus, future work on VMI must focus not only on novel techniques or threat models, but also on managing overheads and scalability with increasing numbers of VMs.

VMI research must measure multi-tenant scalability.

Another strategy to mitigate the costs of asynchronous scanning is to adjust the frequency of the scans—trading risk for performance. For instance, a recent system measured scanning time at 50ms, and could keep overheads at 1% by only scanning every 5s [51]. Similarly, one may cache and reuse introspection results to trade risk of stale data for better scalability [80]. An interesting direction for future work is identifying techniques that minimize both overheads and risk.

B. Privacy

VMI has the potential to create new side-channels in cloud systems. For instance, after reading application binaries, Patagonix [65] queries the NSRL database with the binary hash to determine the type of binary that is running on the system. This effectively leaks information about the programs run within a VM to an outside observer, undermining user privacy.

More generally, VMI has the potential for one guest to observe different cache timings based on the behavior of another guest. Consider a VMI tool that does periodic memory scans of multiple VMs on a cloud system, one after another. The memory scan or snapshot will disrupt cache timings of the guest under observation by forcing exclusive cache lines to transition back to a shared, read-only mode §V-B. Based on its own cache timings, the VM can observe the frequency of its periodic scans. Because the length of a scan of another VM can also be a function of what the VM is doing, changes in time between scans of one VM can indicate what is happening in another VM on the same system.

Although it is unclear whether this example side channel is exploitable in practice, the example raises the larger issue that VMI projects should be cognizant of potential side channels in a multi-VM system. Richter et al. [78] present initial work on privacy-preserving introspection, but more work is needed. An ideal system would not force the user to choose between integrity or privacy risks.

VMI designs should evaluate risks of new side channels.

VIII. Conclusion

Virtual machine introspection is a relatively mature research topic that has made substantial advances over the last twelve years since the semantic gap problem was posed. However, efforts in this space should be refocused on removing trust from the guest OS in service of the larger goal of reducing the system's TCB. Moreover, future VMI solutions should balance innovative techniques and security properties with scalability and privacy concerns. We expect that the lessons from previous work will guide future efforts to adapt existing techniques or develop new techniques to bridge the strong semantic gap.

Acknowledgements

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

References

[1] Draugr. Online at https://code.google.com/p/draugr/.

[2] FatKit. Online at http://4tphi.net/fatkit/.

[3] Foriana. Online at http://hysteria.sk/~niekt0/foriana/.

[4] GREPEXEC: Grepping Executive Objects from Pool Memory). Online at http://uninformed.org/?v=4&a=2&t=pdf.
[5] idetect. Online at http://forensic.seccure.net/.
[6] Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3B.
[7] Intel Software Guard Extensions (Intel SGX) Programming Reference.
[8] Kernel object hooking rootkits (koh rootkits). http://my.opera.com/330205811004483jash520/blog/show.dml/314125.
[9] Kntlist. Online at http://www.dfrws.org/2005/challenge/kntlist.shtml.
[10] lsproc. Online at http://windowsir.blogspot.com/2006/04/lsproc-released.html.
[11] Memparser. Online at http://www.dfrws.org/2005/challenge/memparser.shtml.
[12] PROCENUM. Online at http://forensic.seccure.net/.
[13] Red Hat Crash Utility. Online at http://people.redhat.com/anderson/.
[14] The Linux Cross Reference. Online at http://lxr.linux.no/.
[15] The Volatility framework. Online at https://code.google.com/p/volatility/.
[16] Volatilitux. Online at https://code.google.com/p/volatilitux/.
[17] Windows Memory Forensic Toolkit. Online at http://forensic.seccure.net/.
[18] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, pages 340–353, 2005.
[19] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR Fault Tolerance for Cooperative Services. In *SOSP*, pages 45–58, 2005.
[20] AMD. AMD I/O Virtualization Technology (IOMMU) Specification Revision 1.26. White Paper, AMD: http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf, Nov 2009.
[21] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. HASP '13, 2013.
[22] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *CCS*, pages 38–49, 2010.
[23] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu. Dksm: Subverting virtual machine introspection for fun and profit. In *SRDS*, pages 82–91, 2010.
[24] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC*, pages 77–86, 2008.
[25] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.
[26] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM TOCS*, 30(4):12:1–12:51, Nov. 2012.
[27] T. W. Burger. Intel Virtualization Technology for Directed I/O (VT-d): Enhancing Intel platforms for efficient virtualization of I/O devices. http://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices/, February 2009.
[28] J. Butler and G. Hoglund. Vice - catch the hookers! In *Black Hat USA 2004*, Las Vegas, USA, 2004.
[29] X. Cai, Y. Gui, and R. Johnson. Exploiting unix file-system races via algorithmic complexity attacks. In *Oakland*, pages 27–41, 2009.
[30] M. Carbone, M. Conover, B. Montague, and W. Lee. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *RAID*, pages 22–41, 2012.
[31] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *CCS*, pages 555–565, 2009.
[32] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *SOSP*, pages 101–114, 2011.
[33] S. Checkoway and H. Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *ASPLOS*, 2013.
[34] P. M. Chen and B. D. Noble. When virtual is better than real. In *HotOS*, pages 133–, 2001.
[35] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, pages 2–13, 2008.
[36] J. Corbet. A new adore root kit. *LWN*, March 2004. http://lwn.net/Articles/75990/.
[37] A. Cristina, L. Marziale, G. G. R. Iii, and V. Roussev. Face: Automated digital evidence discovery and correlation. In *Digital Forensics*, 2005.
[38] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. In *USENIX Security*, pages 42–42, 2012.
[39] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS*, pages 51–62, 2008.
[40] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Oakland*, pages 297–312, 2011.
[41] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In

CCS, pages 566–577, 2009.

[42] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

[43] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, Dec. 2007.

[44] Y. Fu and Z. Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Oakland*, pages 586–600, 2012.

[45] Y. Fu and Z. Lin. Exterior: using a dual-vm based external shell for guest-os introspection, configuration, and recovery. In *VEE*, pages 97–110, 2013.

[46] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, pages 191–206, 2003.

[47] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process implanting: A new active introspection framework for virtualization. In *SRDS*, pages 147–156, 2011.

[48] R. T. Hall and J. Taylor. A framework for network-wide semantic event correlation, 2013.

[49] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In *PLDI*, pages 254–263, 2001.

[50] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP*, 2013.

[51] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In *ASPLOS*, pages 279–290, 2011.

[52] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: secure applications on an untrusted operating system. In *ASPLOS*, pages 265–278, 2013.

[53] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *CCS*, pages 128–138, 2007.

[54] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *ASPLOS*, ASPLOS XII, pages 14–24, 2006.

[55] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based Hidden Process Detection and Identification Using Lycosid. In *VEE*, pages 91–100, 2008.

[56] D. Kienzle, N. Evans, and M. Elder. NICE: Network Introspection by Collaborating Endpoints. In *Communications and Network Security*, pages 411–412, 2013.

[57] T. Kim, R. Chandra, and N. Zeldovich. Recovering from intrusions in distributed systems with DARE. In *APSYS*, pages 10:1–10:7, 2012.

[58] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *OSDI*, pages 1–9, 2010.

[59] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security*, pages 191–206, 2002.

[60] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang. Ki-mon: a hardware-assisted event-triggered monitoring platform for mutable kernel object. In *USENIX Security*, pages 511–526, 2013.

[61] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, pages 9–9, 2004.

[62] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *EuroSys*, pages 195–208, 2010.

[63] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP*, pages 178–192, 2003.

[64] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS*, 2011.

[65] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *SS*, pages 243–258, 2008.

[66] L. Liu, J. Han, D. Gao, J. Jing, and D. Zha. Launching return-oriented programming attacks against randomized relocatable executables. In *TRUSTCOM*, pages 37–44, 2011.

[67] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi. Cpu transparent protection of os kernel and hypervisor integrity with programmable dram. In *ISCA*, pages 392–403, 2013.

[68] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Oakland*, pages 143–158, 2010.

[69] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *EuroSys*, pages 315–328, 2008.

[70] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*, 2013.

[71] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: toward snoop-based kernel integrity monitor. In *CCS*, pages 28–37, 2012.

[72] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *ACSAC*, pages 49–58, 2010.

[73] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Oakland*, pages 233–247, 2008.

[74] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime

integrity monitor. In *USENIX Security*, pages 13–13, 2004.

[75] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS*, pages 103–115, 2007.

[76] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *CACM*, 17(7):412–421, July 1974.

[77] J. Rhee, R. Riley, D. Xu, and X. Jiang. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In *RAID*, pages 178–197, 2010.

[78] W. Richter, G. Ammons, J. Harkes, A. Goode, N. Bila, E. De Lara, V. Bala, and M. Satyanarayanan. Privacy-sensitive VM Retrospection. In *HotCloud*, pages 10–10, 2011.

[79] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *RAID*, pages 1–20, 2008.

[80] A. Saberi, Y. Fu, and Z. Lin. HYBRID-BRIDGE: Efficiently Bridging the Semantic Gap in Virtual Machine Introspection via Decoupled Execution and Training Memoization. In *NDSS*, 2014.

[81] A. Schuster. Pool allocations as an information source in Windows memory forensics. In *IMF*, pages 104–115, 2006.

[82] A. Schuster. The impact of Microsoft Windows pool allocation strategies on memory forensics. *Digital Investigation*, 5:S58–S64, 2008.

[83] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP*, pages 335–350, 2007.

[84] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-VM monitoring using hardware virtualization. In *CCS*, pages 477–487, 2009.

[85] Y. Shin and L. Williams. An Empirical Model to Predict Security Vulnerabilities Using Code Complexity Metrics. In *ESEM*, pages 315–317, 2008.

[86] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu. Process out-grafting: An efficient "out-of-VM" approach for fine-grained process execution monitoring. In *CCS*, pages 363–374, 2011.

[87] V. Tarasov, D. Jain, D. Hildebrand, R. Tewari, G. Kuenning, and E. Zadok. Improving I/O performance using virtual disk introspection. In *HotStorage*, pages 11–11, 2013.

[88] J. Wang, A. Stavrou, and A. Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *RAID*, pages 158–177, 2010.

[89] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *CCS*, pages 545–554, 2009.

[90] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *CCS*, pages 157–168, 2012.

[91] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. K. Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security Symposium*, 2002.

[92] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. In *HotPar*, pages 15–15, 2012.

[93] Y. Zhang, Y. Gu, H. Wang, and D. Wang. Virtual-machine-based intrusion detection on file-aware block level storage. In *SBAC-PAD*, pages 185–192, 2006.

[94] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, pages 305–316, 2012.