Trustworthy Services Built on Event-Based Probing for Layered Defense

Read Sprabery*, Zachary J. Estrada*, Zbigniew Kalbarczyk*, Ravishankar Iyer*, Rakesh B. Bobba†, Roy Campbell*
*University of Illinois at Urbana-Champaign, †Oregon State University
*{spraber2, zestrad2, kalbarcz, rkiyer, rhc} - @illinois.edu, †rakesh.bobba@oregonstate.edu

Abstract—**Numerous event-based probing methods exist for cloud computing environments allowing a hypervisor to gain insight into guest activities. Such event-based probing has been shown to be useful for detecting attacks, system hangs through watchdogs, and for inserting exploit detectors before a system can be patched, among others. Here, we illustrate how to use such probing for trustworthy logging and highlight some of the challenges that existing event-based probing mechanisms do not address. Challenges include ensuring a probe inserted at given address is trustworthy despite the lack of attestation available for probes that have been inserted dynamically. We show how probes can be inserted to ensure proper logging of every invocation of a probed instruction. When combined with attested boot of the hypervisor and guest machines, we can ensure the output stream of monitored events is trustworthy. Using these techniques we build a trustworthy log of certain guest-system-call events. The log powers a cloud-tuned Intrusion Detection System (IDS). New event types are identified that must be added to existing probing systems to ensure attempts to circumvent probes within the guest appear in the log. We highlight the overhead penalties paid by guests to increase guarantees of log completeness when faced with attacks on the guest kernel. Promising results (less that 10% for guests) are shown when a guest relaxes the trade-off between log completeness and overhead. Our demonstrative IDS detects common attack scenarios with simple policies built using our guest behavior recording system.**

I. INTRODUCTION

Cloud computing enables service oriented architectures as one can more efficiently manage services that run as separate virtual machines. This approach has led to virtual machine images being sold in marketplaces as so called Virtual appliances (VAs) meant to run single services, such as in micro-service based architectures [1]. In addition the clearer separation of concerns, the second aspect of cloud computing that makes it more amenable to service-oriented IDS is the hypervisor's ability to inspect guest memory to provide new services [2], [3], [4], [5].

In previous work, we used the hypervisor as a basis for detecting rootkits [6]. Examples from literature have used the hypervisor to detect malware that hides itself from process-listing tools using a variety of approaches [3], [2]. Many of these approaches either require running a second VM [7], [8], rely heavily on knowledge about kernel data structures[6], or focus on specific types of intrusions or malware[9], [3], [2], [10]. Our approach aims to log malicious activity so higher level services can detect the presence of abnormal behavior and take action before a malicious actor has had the chance to carry out a meaningful attack. Our previous work in this space focused on probing techniques [11] or detecting specific rootkits [6]; here, we focus on probe placement and a logging system built using the probing method previously introduced.

Our logging technique utilizes hypervisor level probes such as those presented by Estrada et al. [11] and Lengyel et al. [12]. These probing techniques utilize Hardware Assisted Virtualization (HAV) to allow the hypervisor to insert probes into guest memory by replacing an instruction with another that causes control to transfer back to the hypervisor (through a `VMExit`). The hypervisor can then inspect guest memory before transferring control back to the guest. Our IDS is built around logs gathered using only probes placed at specifically chosen system calls. The system call interface tends to be very stable and only requires knowledge regarding which arguments are passed in which CPU registers. We build probes that hook two system calls in Linux, `sys_exec` and `sys_open`, highlight the trade-off between performance and the attack surface of the logging system, and show an example service built on top of such a log in the form of an IDS. By limiting hooked system calls, we aim to lower the performance penalty paid by guests (by logging less information) while increasing the costs to execute an undetected attack against the guest (by limiting the actions that can be taken without being logged). We focus on protection of the logging system as it serves as a basis on which audits, IDS, and other services can be built.

We have implemented our system on Linux 3.13 (Ubuntu 14.04 LTS) as the KVM hypervisor host. We explore how event-based probing systems can be loaded before the probed instruction(s) have the chance to execute even once. Probe insertion before execution is guaranteed by inducing a unique sequence of page faults in the hardware accelerated guest-physical-address to host-physical-address translation available in modern processors. By combining existing trusted boot techniques for both the hypervisor [13] and guests [14], [15], write protecting the probed instruction, and monitoring specific hardware registers, we can increase the cost of compromising log completeness. In the context of this paper, we define "log completeness" as indicating that every invocation of an event under inspection must appear in the log. This log completeness definition implies that probes must be inserted before the instruction at the probed location has executed even once. The approach requires trust be placed in the hypervisor. Beyond attested boot, additional methods from literature, such as HyperSentry [16] and HyperSafe [17] can be used for additional guarantees that the hypervisor can be trusted. Note that currently other hardware generated events are not monitored, such as those stemming from hardware modules that allow for remote power management of machines, such as the Base Management Controller (BMC). We show that the integrity of the probe cannot be fully guaranteed by existing probe based monitoring systems and that two more traps which transfer control back to the hypervisor must be added to provide more logging coverage of the attack space on a system call based log.

Our contributions include:

- An algorithm for inserting monitoring probes at time of boot before the probed instruction can be called.
- Methods to protect against a number of attacks that attempt to circumvent the hypervisor-based trusted logging system.
- An IDS built on top of the trusted logging mechanism which is more effective as it can be tuned to single service oriented VAs.
- A semi-automated policy-recording mechanism that reduces the burden of generating and managing white-listing polices for the proposed IDS.
- An evaluation of the performance both in terms of impact to applications running within guests and in terms of detection capabilities of the IDS.

To the best of our knowledge this paper is the first work that considers issues surrounding probe insertion time in relationship to completeness of guest kernel-based events logged using trustworthy probes.

II. Goals of a Hypervisor-Based Trusted Log

We set forward five requirements that must be met to guarantee the integrity of a trusted log meant to monitor guest Virtual Machines (VM's). Increasing the integrity and completeness of our trusted log provides better guarantees for higher level services built using such a log. The requirements are as follows:

R1 Information provided by the guest cannot alter the logging entity's control flow. Information is simply logged and higher level services can respond to logged data appropriately,

R2 Guests cannot modify or remove an event from the log after the fact,

R3 In-guest modifications to instrumented locations should be logged,

R4 Modifications to functions invoking the hooked instruction should similarly be logged,

R5 The event log must contain every event ϵ_T of type T if there exists any probe P_T in the set of probes which produces output corresponding to events of type T, up to and including a malicious action within the guest.

We also have three design goals that drive the engineering choices behind the architecture proposed here. These are:

D1 Minimize the performance impact on guests,

D2 Minimize additions to the trusted compute base,

D3 Require no modification of guests (i.e., transparent).

R1 implies that probes must not trust memory read from guests. In particular, probes must not trust the guest to inform the logging function of appropriate bound lengths. This requires that thorough bounds checking is performed on memory inspected from the guest and requires reasonable stopping points for data structures that need the size parameter to be inferred from guest memory. Setting limits also protects against maliciously linked recursive data structures. This ensures that a compromised guest cannot affect probe behavior.

R2 requires that a malicious guest can neither modify nor prevent the logging of an event ϵ_T occurring at time t_x in any time t_{x+n} the (i.e.: future actions in the guests cannot alter previously logged operations). Event-based logging ensures that the executions of probed instructions in the guest are captured in the log right away. Preventing log modifications by guests ensures that events captured cannot be deleted or modified by guests even if the guest reformats media or terminates.

R3 allows services built on top of the log to decide how much trust to place in events captured in the log after a modification event. For instance, if an administrator observes a modification event ϵ_{mod} at time t_x she can decide to trust or not trust the events logged after time t_x depending on other available information. For example, non-malicious performance profiling within the guest may have caused the modification event. We only guarantee that the event will appear in the log and leave any event classification up to higher-level services. The probes' only job is logging, event processing is done in a separate process. Recording potential attempts to circumvent logging ensures that higher-level applications have sufficient information to classify events. **R4** guarantees that an attacker cannot circumvent logging by simply redirecting away from the instruction(s) under inspection. In our case, that means write protecting the preceding call stack. The call stack includes the `sys_call_table` indicating the memory locations of the specific system call handlers, the general system call handler `system_call` and the hardware registers indicating which block of code to execute after performing an interrupt. In the case of hooking only the general system call handler, only modifications to `system_call` and the hardware registers (such as the `idt` register) must be monitored. We elaborate more on monitoring events and the specific registers that need to be monitored in Section VI.

R5 can help ensure log completeness. In later sections we discuss the conditions under which this can be met. Log completeness, as defined above, is desirable because the trust placed in the events occurring at some time point t_x relies upon the integrity of every event logged between when logging can begin, time t_0, and the time of the event immediately preceding event x at time t_{x-1}. If any event before event x is determined to be malicious, then we may decide that the details of an event occurring at time t_x cannot be trusted. Thus, if a service cannot review the events occurring between time t_0

and when a probe is inserted, the service cannot determine the integrity of any event. The situation of missing data can occur during guest boot; existing tools built on event-based probing insert probes at some time t_{boot+n}. If a malicious action occurs between when the system boots, t_{boot}, and when the probe is inserted, then it will go unlogged by naïvely applying the instruction replacement technique present in literature. Existing work assumes that the guest is trusted until some time after boot, t_{boot+n}. We remove this assumption, only trusting that the system boots into a known state at time t_{boot}. We present a method to guarantee a probe is inserted before any invocation of the instruction it is replacing. However, we do not currently support the ability to log dynamically generated code that modifies itself after boot.

Apart from the requirements **R1** – **R5**, an additional goal of our system is to impose low overheads to remain practical. **D1** dictates that any attempts at logging must pose minimal performance impact on guests while also being transparent (**D3**). These two design goals ensure that any ensuing architecture remain feasible for production workloads. **D2** requires that we keep probing functions to the minimal required for logging in order to minimize additional attack surface. Below we discuss how we can achieve these goals through the use of probing techniques, novel guest boot sequence analysis, and well placed probes.

III. Background

Below we describe the technologies being used to help meet the requirements listed above. By utilizing and extending these existing techniques, we meet our design goals and increase resilience against attacks.

A. Hardware Assisted Virtualization

The x86 architecture was not originally designed with virtualization in mind, but as VM's became popular, hardware manufacturers looked at ways to improve their performance and robustness. Both AMD and Intel have released support for HAV in the form of extensions to the x86 instruction set.

HAV allows a VM to execute instructions natively on the hypervisor's CPU(s). However, the hypervisor must maintain control of the VM's execution. When the CPU is executing a VM's instructions, `VMExit` events are generated for any privileged operations that the VM attempts. A `VMExit` transfers control from the VM to the hypervisor allowing the hypervisor to perform any necessary operations before returning control back to the VM.

While allowing for robust and simplified hypervisor software, `VMExit`'s do incur performance overhead. Historically, one of the major causes of overhead in HAV was due to page faults in the VM. In earlier HAV implementations, every page fault would result in a `VMExit` since the guest could not control its own page tables. To alleviate this, vendors introduced a technique called two-dimensional page tables (TDP). In this paper we utilize Intel's TDP implementation, known as Extended Page Tables (EPT). The techniques apply to AMD's equivalent Nested Page Tables (NPT).

EPT allows VM's to manage their own page tables by managing guest-physical to host-physical address translations in hardware, effectively eliminating `VMExit`'s on page faults. Similar to conventional x86 page tables, EPT also provides a set of access flags that can be set at the page level: execute enable, write enable, and read enable. A `VMExit` is triggered on accesses that violate the access flags due to an EPT violation. We later show how EPT access flags can be used to guarantee that probing systems do not miss events of interest occurring within the guest immediately after guest boot, helping to fulfill **R5**. For more information on EPT we refer the reader to Volume 3 of the Intel Software Development Manuals [18]; for AMD's equivalent NPT the reader can refer to Volume 2 of AMD's Programmer Manual [19].

B. Virtual Machine Monitor based Probing

The research community has shown that timer based guest introspection (passive monitoring) can be circumvented by a malicious or compromised guest [20], [21]. We avoid this issue by using event-based monitoring. Here, we highlight the mechanism used to enable such logging. Event-based probing using debugging techniques has been proposed and applied in a number of different contexts [12], [22], [11], [23]. Lengyel et al. use event-based probing for dynamic analysis of malware with the goal of remaining undetected during monitoring [12]. Estrada et. al. show the effectiveness of similar techniques for reliability and security monitoring [11], [24]. XenProbes uses the technique for profiling performance inside guests [22] and Spider uses it for stealthy debugging [23].

All of these approaches utilize HAV to invoke `VMExits` upon execution of `int3` (`0xCC`) instructions in the guest. The key feature of event-based probing is that an instruction within an untrusted environment can be replaced by an instruction (`int3` in this case) that causes a hardware enforced trap (i.e., a `VMExit`) to transfer control flow to a trusted environment. After guest inspection is done, the original instruction is executed within the guest and the breakpoint is re-inserted before guest execution resumes. Because probes cause a `VMExit`, which is an expensive operation, one must carefully design services built on such probes to reduce the number of exit events while also ensuring enough information is available to ensure meaningful services can be developed utilizing the logged data. We do not consider the event-based approached used by LibVMI [25] as it invokes a `VMExit` on every single instruction in the target page for the logged event. Such an approach causes high overhead and is intractable due to our performance requirement **D1**. Our approach gives users the flexibility to determine the overhead paid based on the level of protection deemed necessary for a given application.

C. The Semantic Gap

Any Virtual Machine Introspection (VMI) application must cross the "Semantic Gap" - the gap faced by developers of code running within the Virtual Machine Monitor (VMM) that must inspect guest memory with no knowledge of the kernel data structures or memory layout of the guest. Much research has been done in this area, and we point the reader to the overview done by Hebbal et al. for a more thorough discussion of the issue and many of the proposed solutions [26]. For this work, we assume that the address of the `sys_exec` and `sys_open` calls in Linux, along with the offset at which the Linux kernel `.text` addressing begins are provided (this memory mapping is well documented [27]). The latter is needed in order to identify the guest physical locations of the above functions (which are loaded into memory before paging is enabled in the guest). In Section V we discuss in more detail why this is necessary.

We favor the approach of querying `System.Map` for the location of relevant functions due to ease of access; this approach has shown to be successful in the literature for providing a low cost method for crossing the semantic gap [5], [22], [8]. We limit our discussion to Linux guests as the open source nature of Linux lends itself to easier distribution of VAs, the focus of our IDS, but a similar approach of querying the debug symbols for the Windows kernel has also been met with success [12], [8].

D. Virtual Appliances

VAs are a popular method for deploying cloud services. One can simply choose an appliance from a list of images made available on a cloud provider's marketplace and immediately deploy services such as databases or web servers with minimal configuration. The tuned nature of these appliances makes their behavior more predicable than a VM used for general purpose computation. In this paper we present an IDS that leverages the "appliance" nature of cloud based deployments instantiated using VAs. The IDS is built using guest event driven hypervisor-level probes to deliver relevant information to the policy compliance layer.

A typical deployment of a cloud based web application may have a reverse proxy routing requests to an application processing layer, each of which communicate with a database before returning a response. Each of these services would be a different VA. We envision a system for which different policies protect each kind of VA. In our example, there there would be three main policies, one for the reverse proxy VA, one for the database VA and one for the VAs which serve as the application server(s). Policies can share layers if VAs are built using the same base distribution as a single distribution will have the same cron binaries running for example. While policies are stackable, the main advantages reside in the policies for each that differ, allowing for good coverage while limiting false positives. For example, a database server running MySQL should never execute a shell outside of configuration events; our monitoring system would detect such an operation as a violation. The event log can then also be used as compliance monitoring during configuration periods, and could serve as a method to detect insider threats attempting to re-configure applications in an attempt to cause unstable behavior.

IV. Attack Model Against the Logging System

Keeping the above technologies in mind, we motivate our design choices using the attack model and assumptions presented in this section to improve the resilience of the logging system. We assume that the hypervisor is a trusted entity and that the hypervisor side of the logging framework is secure. For the log file itself, a simple way to provide guarantees is to use remote logging, or approaches used in literature [28]. Here, we focus on the elements of logging that must be in place to facilitate proper logging of a guest that may become malicious at some point after boot.

We assume that the hypervisor is using trusted boot, thus the integrity can be attested. Additionally, we assume that guests running on the hypervisor are also using attested boot mechanisms, such as those presented in [15] and [14], and that guest kernels are known, non-malicious builds of Linux. This allows the hypervisor to guarantee the integrity of any guest kernel before the guest boots. We assume that the guest kernel is not malicious until after the first user-space program runs. This is a reasonable assumption as attempts to exploit a kernel will come from software loaded after boot (either malicious software will be loaded or vulnerable software exploited).

Attacks can include loading kernel modules, modification of the kernel in place, or attempts to circumvent logging through process tampering (more details in Attack **A2**). An attacker may try and copy the page of memory with the replaced instruction, fix said instruction, and redirect system calls to this new page. Such a redirect would either require modification of the Interrupt Descriptor Table in memory that is referenced to by the general system call handler or may come as a write to a hardware register in an effort to circumvent the code block executed after an interrupt.

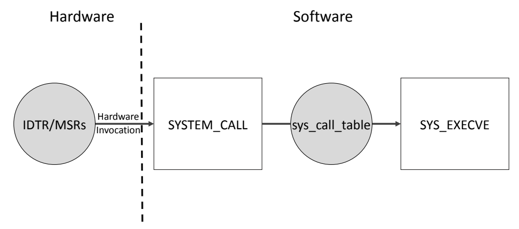
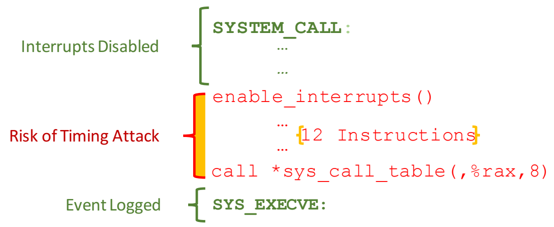


Fig. 1. Invocation Process for a Specific System Call Handler

To protect the integrity of an event placed at a given location, we must protect the entire stack trace leading up to the execution of that event. In this paper, we are primarily concerned with logging the specific handlers of select system calls. Our attack model against the logging system concerns protecting against any modification of the steps leading up to the invocation of the monitored system call handler as highlighted by Figure 1. The figure shows the hardware invocation of the `system_call` code block, the location of which is designated by values stored in hardware registers. During execution of the general system call handler, interrupts are re-enabled (shown in Figure 2). The general system call handlers transfers control to the specific handler through the



```
SYSTEM_CALL:
    …
    …
enable_interrupts()
    … {12 Instructions}
    …
call *sys_call_table(,%rax,8)
SYS_EXECVE:
```

Fig. 2. Timing Constraints for Interrupt Attack (**A2**)

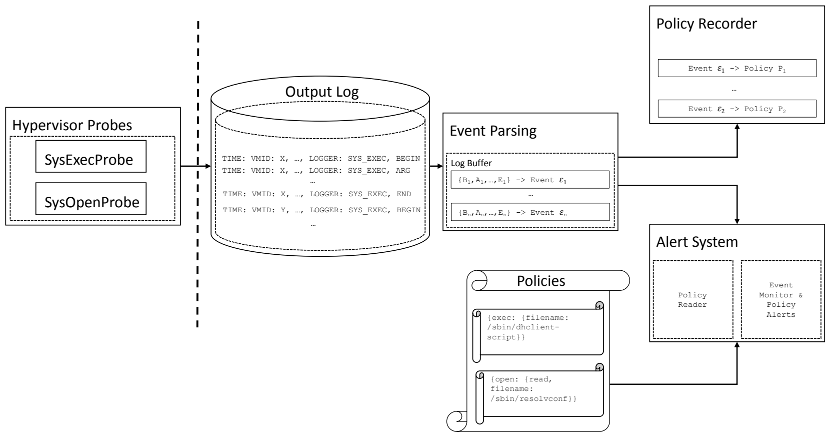

Hypervisor Probes
SysExecProbe
SysOpenProbe
Output Log
TIME: VMID: X, …, LOGGER: SYS_EXEC, BEGIN
TIME: VMID: X, …, LOGGER: SYS_EXEC, ARG
TIME: VMID: X, …, LOGGER: SYS_EXEC, END
TIME: VMID: Y, …, LOGGER: SYS_EXEC, BEGIN
Event Parsing
Log Buffer
{B1,A1,…,E1} -> Event ε1
{Bn,An,…,En} -> Event εn
Policy Recorder
Event ε1 -> Policy P1
Event ε2 -> Policy P2
Alert System
Policy Reader
Event Monitor & Policy Alerts
Policies
{exec: {filename: /sbin/dhclient-script}}
{open: {read, filename: /sbin/resolvconf}}

Fig. 3. Trustworthy-Log Driven IDS Architecture

`sys_call_table`. Our goal is to place probes on the first instruction of the specific handler. With that in mind, consider the following list of attacks that could circumvent logging:

- **A1** Write to either the `IDTR` register (for legacy `int $80` based system calls) or various Model Specific Registers (MSRs) for so called "fast" system calls to force the hardware to invoke a malicious code block after interrupts (See Section VI for a more detailed discussion of the specific registers).
- **A2** Coordinate an interrupt after a system call (that is being logged) is made and interrupts have been re-enabled, but before the specific system call handler has been invoked. Upon interruption, modify the `thread_struct` of the system call invoking process to point to a different system call handler upon being re-scheduled. The timing constraint of this attack is highlighted by Figure 2.
- **A3** Rewrite the general system call handler to reference a new, attacker supplied, Interrupt Descriptor Table.
- **A4** Rewrite the entry for the specific system call being hooked in the Interrupt Descriptor Table to point to an attacker supplied handler for the system call.
- **A5** Rewrite replaced instruction(s) with the original instruction.
- **A6** Simulate a system call interface using an alternative means of communication between userspace and a rootkit.

In section VI we highlight how attacks can be accounted for through hardware enforced events. It is worth noting here that **A2** has tight timing constraints (an interrupt would have to occur within a 12 instruction window). We later discuss how removing the protection guarantees for **A2** greatly reduces the performance impact and we believe it has minimal effects on the overall trustworthyness of our logging architecture. While we do not currently defend against **A6** style attacks, we believe our system greatly increases the cost of attack while providing good detection coverage of many of the attacks that might try and circumvent logging.

V. Trustworthy Log Acquisition

With our attack model in place, we now discuss the specifics of the logging mechanism. In particular, we discuss how and where probes are inserted. The algorithm for guaranteeing a probe is inserted before the execution of the probed instruction is introduced and the relationship between the implementation and design requirements is described.

A. Probing Mechanism

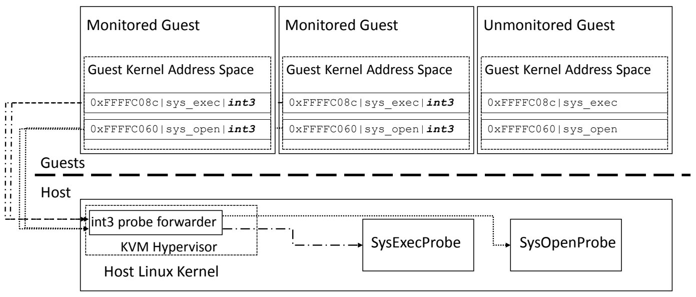


Fig. 4. Event Driven Probe Architecture

Figure 4 highlights the mechanism to probe the Linux kernel system calls `sys_exec` and `sys_open`. An event-based probing mechanism is utilized to replace instructions in the guest kernel [11], ensuring information is logged anytime the affected functions are called, fulfilling **R2**.

To ensure that any attempt to modify a probe is logged (**R3**) we use EPTs to remove write permissions for the affected page, register a callback to handle these EPT violations, and within the callback handler only log attempts to modify the affected page if the violation occurs for the guest virtual address on which we inserted the probe. While performance monitoring within the guest might cause non-malicious writes to locations of logging probes, an administrator would know the event is benign. Event classification is left to higher level services, we guarantee only that modification events do appear in the log. Logging code remains small, making formal verification more feasible. There are only 72 and 41 lines of code for our `sys_exec` logger and `sys_open` logger respectively (not including the code required to insert the probes), keeping in line with **D2**.

B. Log Completeness

We have defined log completeness to mean that our logging service guarantees that every invocation of a probed event be present in the log. In order to ensure log completeness and fulfill **R5** we must place probes in their respective locations *before* the instructions at those locations are executed. While we can guarantee this, it may be possible for an attacker to perform system call like actions, all together bypassing probed instructions (**A6**). The system calls being probed will be loaded at a predictable location within the guest physical memory (as noted in Linux's memory mapping documentation [27]). The knowledge of these locations allows us to determine the page number indicating the page containing the target instruction, which we use to watch for EPT violations of any guest physical address that occurs on the same page as an instruction of interest during the guest boot sequence. We are able to watch for such violations by utilizing a callback handler that gets invoked after we have allowed KVM to perform any necessary actions to handle the violation. We have chosen to highlight our approach using KVM, but the general technique applies to any VMM. Upon observing the first write violation for any address within the page of interest, we remove the execute bit from that page, allowing our callback handler to be invoked if any instruction on the page is executed. Subsequently, upon observation of any instruction execution on the page of interest, we know that the remaining code for that page must be loaded and can safely insert the probe. Having inserted the probe, we restore EPT permissions to allow execution and remove our checks for EPT violations due to execution exceptions on the page in which the probe is inserted as the checks are only required as the final step before probe insertion. By inserting probes in this manner during boot of guests, we are able to ensure log completeness and log every call to these two system calls, even while the first userspace applications are being started.

C. Implications of an Untrusted Guest

Finally, we must ensure that the actions taken within the probe do not place unwarranted trust in data obtained from the guest (**R1**). For example, our `sys_exec` logger logs two variable length string arrays. While these strings are typically `\0` terminated, the guest could point the probe to a location with an arbitrarily large number of bytes before a `\0` is encountered. To protect against copying strings from guest memory, we only copy 500 bytes and place a `\0` at the 500th byte. While we may log garbage data in cases of an intentionally malicious guest and may truncate binary names in the case of exceptionally long, but legitimate, calls to `sys_exec`, this is a necessary trade off to ensure the probing interface remains resilient. Potential for truncating can be seen again when iterating through variable length arrays, which should be `NULL` terminated. We only iterate over up to 50 entries and exit iteration if `NULL` is encountered (in a legitimate case) and stop at 50 in the case of a malicious guest pointing the probe to a random memory location. Again, this has the side effect of potentially truncating logged arguments. In our experiments, we never truncated any legitimate data. The length decisions did not impact the ability to log meaningful data. Regardless of whether or not arguments are truncated, we are able to protect the logging facility from the attacks listed above.

VI. Logged Events

Choosing which events to log is critical to increasing the resilience against the attacks listed in Section IV. Here, we show which events should be logged and how each event type can be used to improve resilience.

In addition to the information listed for each event type as defined below, all events also include the `hostname` of the KVM hypervisor on which the event occurs, a `timestamp` for the event, and the `vmid` (the qemu-kvm process id of the VM on the host on which the event is logged).

The five event types currently in our system, and information collected unique to each type, are as follows:

- T_{mod} - Probe modification events containing: `gva`

The next two events log activity of interest and are useful for facilitating detection of abnormal actions.

- T_{se} - `sys_exec` events containing: `filename, argv,` and `envp`
- T_{so} - `sys_open` events containing: `filename, mode` and `flags`.

The following two events are unique to logging system calls and increase the cost of circumventing the logging mechanism. These require additional callbacks be provided by the underlying probing framework. To prove viability, we have implemented the `wrmsr` event for modern system calls. We will implement the `lidt` event in future work. These two events are not currently provided by any event-based monitoring framework discussed in the literature [11], [22], [12] as they only become necessary when providing defenses for system call monitoring.

- T_{lidt} - `lidt` event. Triggered on execution of the `lidt` (Load interrupt descriptor table) x86 instruction.
- T_{wrmsr} - `wrmsr` event. Triggered on execution of the `wrmsr` (Write Model Specific Regsiter) x86 instruction.

These two events are hardware enforced; once the hypervisor has configured the processor to trap these calls, their execution will always force a `VMExit`. The `lidt` trap can be configured by setting bit 2 (Descriptor Table Exiting) of the `IA32_VMX_PROCBASED_CTLS2` model specific register to 1 within the hypervisor before VM's are started. Similarly, writes to model specific registers within the guests can be trapped by ensuring bit 28 of the same model specific register is 1 and then configuring the MSR bitmap field in the Virtual Machine Control Structure (VMCS) to only cause `VMExits` on writes to the specific registers that need monitoring. This ensures that performance overhead remains low by not inducing `VMExits` for writes to every

MSR. For `int $80` based system calls, the `lidt` trap is sufficient. For `sysenter` invoked system calls, the three MSRs `IA32_SYSENTER_{ES, EIP, ESP}` must be monitored through the `wrmsr` trap. Finally, for `syscall` invoked system calls, the MSR `IA32_LSTAR` must be monitored with the `wrmsr` trap. The registers listed above are used to register Interrupt Service Routines (ISRs) with the processor. In Linux, these point to the general system call handler. The performance impact of these two events should be negligible under normal operation as these events occur only during boot of the guest kernel and during configuration of MSRs.

A. Detection of Attacks on the Logging System

Let us now consider how these event types can facilitate detection of the attacks against the logging facility listed above. We protect against attacks **A5**, **A3**, and **A4** by properly removing the write enable bits for the pages containing the instruction modified, the general system call handler, and the interrupt descriptor table and listening to events of type T_{mod}. The event T_{mod} is hardware enforced by EPT. Attempts to modify pages for which the write enable bit has been removed will trigger a `VMExit` through an EPT violation. Attacks that try to change the ISR for system calls (**A1** above) can be logged with events of type T_{lidt} and T_{wrmsr}. Finally, careful placement of probes can ensure that logging occurs before interrupts have been re-enabled by placing the probe on the general system call handler, mitigating attack **A2**. Mitigating **A2** does have high performance impact as we discuss in Section VII; we believe placing the probe at the specific system call handlers is a reasonable trade-off as attacks of this kind must meet a tight timing constraint. Finally, if an attacker can compromise the guest kernel it would be possible to recreate a separate system call interface (**A6**). Consider a rootkit that finds the `task_struct` of the userspace process it is hiding. It could poll the memory of the process for system call like arguments and then execute a separate code blocking performing the same actions as a system call. Such an attack may be possible to detect by monitoring timing interrupts. In future work, we will explore ways to use event-based probing to further protect the guest kernel against attacks of this kind.

Note that many more event types are possible as event-based probing provides a trusted mechanism with which to hook any kernel function. But in keeping with **D1** and **D2**, we choose to keep this number small.

B. Event Logging Format

All probe output is placed into the VMM's `/var/log/kern.log`. Output is processed by a user-space application that builds and processes events. This design is shown in Figure 3. In order to allow for easier processing by higher level applications, we adhere to a JSON like format when doing logging within the host kernel.

A log sample for a `touch text.log` event is shown in Listing 1.

Listing 1. Example `sys_open` Probe Output

```
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER", "KIND": "BEGIN"}
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER",
   "KIND": "ARG","ARG_NAME": "filename", "VALUE": "test.text"}
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER",
   "KIND": "ARG", "ARG_NAME": "flags", "VALUE": "0x941"}
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER",
   "KIND": "ARG", "ARG_NAME": "mode", "VALUE": "0x1b6"}
{"VMID": 1884, "LOGGER": "SYS_OPEN_LOGGER", "KIND": "END"}
```

The `TIMESTAMP, HOSTNAME` and `LOG_ID` are also included and are set by the `printk` function within the hypervisor. We trust these fields to be accurate when read by higher level tools. The accuracy of these fields is important as will be discussed in the next section on the development of higher level tools.

For each event, we have a `BEGIN` statement and an `END` statement. Everything in between those statements make up the body of the event and are used to log parameters read from the guest.

VII. Intrusion Detection System for Virtual Appliances

To highlight our approach to services built on top of an event based log, we have developed an IDS which triggers alerts on violations of filename white-lists. The checks are performed on filenames passed to guest `sys_exec` and `sys_open` calls. To enable ease of use, we have also built a policy recorder that translates guest events to white-list policies during the recording or learning phase.

A. IDS Architecture

The architecture of our intrusion detection system is shown in Figure 3. Raw probe logs are transferred from kernel to user space using the `/var/log/ kern.log` interface. From there, the logs are placed in a buffer as they are read from the file. An `ioctl` interface to `/var/log/kern.log` is used to ensure updates are pushed to the user space application as soon as probes write to the file. Within the user space event parser, buffers must be used to ensure that output from a probe P^1_{so} into guest G_1 do not become integrated into an event ϵ_2 from the output of the probe P^2_{so} placed into guest G_2, as the arrival of such logs may be intermingled within `/var/log/kern.log`. This is ensured by placing all logs from a given probe into a unique buffer identified by the `LOGGER_TYPE,HOSTNAME,VMID` sequence. Since the buffer being used is determined by this sequence of values, these values must be set by the hypervisor. No value read from the guest is used to identify a probed event or which buffer in which to place a logged statement, ensuring the guest can not impact actions taken by the logging system, . For now, we do not consider multiple vCPU guests, thus only need to worry about intermingling between guests. In the case of multiple vCPUs, the vCPU id would also need to be used as a unique identifier as it would be possible that a probed location be called from multiple vCPUs simultaneously. This limitation is partly due to the chosen probing framework, other frameworks [12] would support multiple vCPU guests. Extending this approach to multiple vCPU based guests will be done in future work.

B. Policy Generation

After event parsing is complete, processed events are passed to either a policy recording layer or an alert system for our IDS. The policy recording system allows an administrator to record standard behavior for a VA in terms of white-listing the actions taken during policy recording. Listing 2 shows an example policy built using our policy recorder while executing the `which` command on a guest under inspection. Currently, white-lists are separated from attackers executing in the guest by the VMM. In future work we will investigate using attestation mechanisms for the white-list and while-list enforcing mechanism. Continuous integration test suites could be used to generate polices through this policy recording mechanism.

Listing 2. Example `which.policy` file

```
{ "policies": [
{"exec": {"type": "whitelist","filename": "/usr/bin/which"}},
{"open": {"type": "whitelist","access_type":
   "read", "filename": "/etc/ld.so.cache"}},
{"open": {"type": "whitelist","access_type":
   "read", "filename":
      "/lib/x86_64-linux-gnu/libc.so.6"}},
{"open": {"type": "whitelist","access_type":
   "read", "filename": "/usr/bin/which"}}]}
```

C. Threat Analysis

We note that there are certain limitations to our approach that would allow an attacker to commit a malicious action without being logged. Consider a vulnerable binary running on a system that is compromised through a buffer overflow attack. Assuming the attacker does not crash the binary, it world be possible to run code under the guise of an already executing process. As long as the payload never opened a file or executed another binary, it would go unlogged. While any such process hijacking will go unlogged, our approach substantially reduces the actions that can be taken by an attacker. Adding a separate event for system calls dealing with network access would further mitigate the possibility that a malicious payload is able to do any useful work without being logged. Our approach is complimentary to and can be combined with other defense-in-depth approaches such as ASLR, non-executable heaps and other defenses to increase the cost of implementing a successful attack.

VIII. Evaluation

In this section we evaluate both the impact of the probes on the performance of the guest and on the ability of the IDS to detect attacks on applications running in VAs. The IDS is evaluated against a real world attack on a popular cloud based web application.

A. Performance

To evaluate the overhead of our probing mechanisms driven by guest events, we run three benchmarks that are representative of cloud workloads. These include:

- Apache Bench - a web serving benchmark for the Apache web server, [29],
- Redis Bench - a benchmark for the in memory data store [30],
- OpenSSL Profiling - used to understand the impact on encrypted communication within guests.

These tests were chosen because they represent a disk-read heavy workload (Apache), network heavy workload (Redis, Apache), and a CPU heavy workload (OpenSSL). Cloud applications will often call in-memory caches before sending a response using Apache configured with OpenSSL. All tests are configured using the Phoronix Test Suite and are run 90 times each. The first 30 runs are performed with our trusted probes loaded and then run 30 times without. The last 30 runs are done while having probes loaded at the general system call handler, before interrupts have been re-enabled in the guest to highlight the performance penalty paid while protecting against **A2**. Figure 5 shows the results for both Apache Bench and for OpenSSL. Apache bench results are in terms of requests served per second and those for OpenSSL are in terms of signatures generated per second, but here both have been normalized to highlight the percentage decrease in performance caused by probing. Looking at the first two bars in Figure 5, it is easy to see the performance implications of placing probes at the generic system call handler. In the case of the specific handler (the first bar), we see less than 10% overhead. But Apache has about a 55% overhead when placing the probe at the general handlers. The next two bars highlight how probing has little impact on the performance of OpenSSL, regardless of probe location. This is because OpenSSL does not have to interact with the kernel as much as Apache and Redis to complete its workload. OpenSSL works by loading a key in memory and then generates signatures using that key. It is up to another process, Apache for example, to write out any information to the network.

Figure 6 is for the Redis benchmark, which runs five separate request types against the in memory data store. As seen from the "B" bars in the figure, the performance overhead of probing specific handlers when compared to no probing (the "A" bars) is negligible. The "C" bars for each query type show the high performance impact of defending against **A2**, which is about 75%.

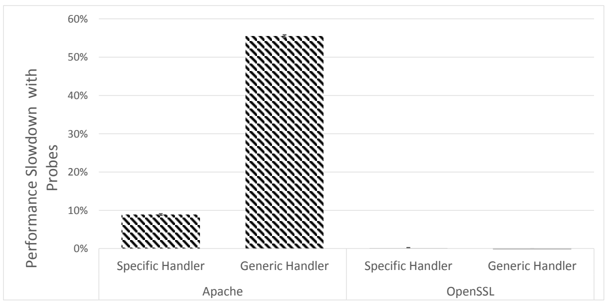


Fig. 5. Apache Bench and OpenSSL Overhead Relative to Running with no Probing.

In the case of hooking specific system call handlers, it is clear to see that overheads remain tolerable (less than 10%), because we are only probing two guest kernel functions. The overheads are large when protecting against **A2** though, around

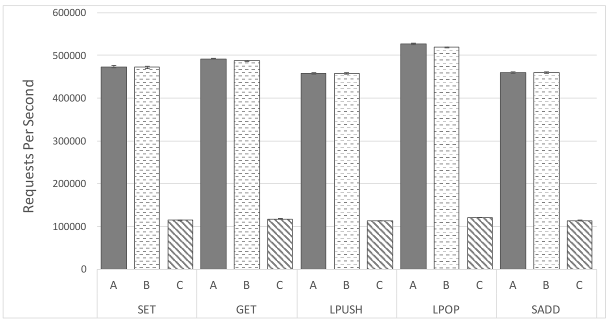


Fig. 6. Redis Benchmark Overhead for 5 Redis Operations. (A) without probing the guest, (B) probing only the specific system call handlers, and (C) probing the general system call handler.

55% for Apache and 75% for Redis. Apache and Redis are both opening sockets and sending data over the network, which is why we see a much higher penalty being paid when hooking the generic system call handler. We feel that the protections against **A5**, **A3**, **A4**, **A1** (requirements **D3** and **R4**) go a long way in protecting the specific system call handler, substantially reducing the unloggable attack space when hooking only the specific system call handlers. In future work, we will investigate preventing rootkits to provide more guarantees for the output log.

B. IDS Evaluation

We evaluate the efficacy of the IDS built on top of our trusted logging platform by looking at real world exploits for motivation. In a recent attack on the website for the Linux distribution Linux Mint [31], attackers were able to gain shell access as the `www-data` user, the user typically reserved for only running the `httpd` process [32]. The attack exploited a vulnerability in the popular blogging framework, Wordpress. Wordpress is representative of a typical cloud application as it can be deployed on many VAs to enable horizontal scalability. To see how our system would have handled such an attack, we installed a copy of Wordpress with a typical plugin and attacked the setup using Wordpress Vulnerability Database ID #8209 [33].

We first setup a Wordpress application server and separate database server to act as our VAs. Since our IDS supports policy stacking, we are able to record a separate policy for Wordpress and use the `dhcp.policy` file common to all VAs built using the same base Ubuntu 14.04 LTS distribution. The `dhcp.policy` file was auto-generated by running our policy generation tool against the log output of a default Ubuntu install. Including that policy is necessary as it removes the chance of false positives every time a dhcp lease renewal is performed. It would not be necessary for VAs using static IP's. An abridged version of the Wordpress policy file is shown in Listing 3. Our policy recording utility auto-generated a policy that served as a starting point and then we used knowledge about proper Wordpress installs to fine tune the policy. For example, the policy recording utility produced many single `filename: /var/www/html/*.php` entries. We removed these and converted it into a single `directory: /var/www/html` entry as shown on the first line of the policy in the listing.

Listing 3. Abridged `wordpress.policy` file

```
{"open": {"type": "whitelist","access_type":  "read",
        "directory": "/var/www/html"}},

{"open": {"type": "whitelist","access_type": "create",
        "directory": "/var/www/html/wp-content/uploads"}},
{"open": {"type": "whitelist","access_type": "modification",
        "directory": "/var/www/html/wp-content/uploads"}},
{"open": {"type": "whitelist","access_type": "read",
"directory": "/var/www/html/wp-content/uploads"}},

{"open": {"type": "whitelist","access_type": "create",
"directory": "/var/www/html/wp-content/plugins"}},
```

We exploit the vulnerability using Metasploit [34] to determine if our alerting system is able to capture anomalous events. Because the exploit works by injecting arbitrary PHP code, we can only detect attacks that use PHP to access other files on the system (outside of the /var/www/html directory) or execute system binaries. We detect the exploit after the attacker performs an action anomolous to the hijacked process. In this case, the attack is detected upon attacker execution of a shell, as `/bin/sh` should never execute on the system. We could detect the exploit sooner by adding an extra probe to `sys_socket`. In future work, we will explore detection coverage and delay in relationship to the overheads paid by guests (when only hooking specific system call handlers) to determine which kernel functions to probe.

While we have demonstrated the IDS on one example, it has shown the viability of our technique. Our approach relies on the fact that many exploits require a binary to load and execute on a system. And if the exploit does not run in a separate process, as is the case in the example given above, the attacker will likely either execute a system binary or open a file, revealing malicious activity. For instance, the loading of kernel modules could be audited by looking at events of type T_{se} with `filename` equal to `insmod`. This would potentially reveal the loading of a rootkit. Payloads executed through process hijacking can explore the full system call interface and potentially exploit the running kernel. Such an event would not be logged, though any attempt to remove our probe using such an exploit would be noted in the log. This increases the burden of carrying out a successful attacks as malicious payloads will have to be carried out within a vulnerable binary or the kernel to remain undetected. In future work we will explore creating probes for the most vulnerable locations within the Linux kernel by evaluating past exploits.

IX. Related Work

Huh et al. discuss a trusted logging architecture for grid computing using Xen [28]. Their approach relies on logging events as they are intercepted by Xen device drivers. Our trusted logging is more flexible as any action within the guest can be logged on instruction execution. Additionally, the authors propose an extensive architecture for guaranteeing the log is not fabricated by the provider. We view this work as complementary. Thus far, we have focused on trust related issues related to log generation and can utilize similar techniques for improving trustworthiness.

Crawford et al. discuss a methodology for detecting insider threats that relies on scanning the memory of running virtual machines every 30 minutes [9]. As we discussed earlier, polling techniques such as this are limited in that they are easily circumvented, giving attackers a 30 minute window in which to perform malicious activities. Kienzle et al. explore using VMI techniques for endpoint configuration compliance, but require the compliance audit package run in a separate VM, increasing the resources of the monitor [35]. Their approach to compliance also relies on polling, thus can be circumvented. Our approach provides a trusted log which is guaranteed to capture every event probed. In future work, we intend to use our trusted event log to perform compliance checks of Mandatory Access Control (MAC) systems running within guests. Win et al. propose using VMI to provide additional layers of security for a similar system, but rely on information from a trusted in-guest monitoring agent to report relevant accesses to a trusted compliance layer VM [36]. Our approach places no trust in the guest after the initial kernel is loaded using an attestation technique provided by a TPM.

KvmSec is a security extension for KVM, but relies on probes running in untrusted guests [37]. Our approach places no trust in the guest. In “Space Traveling across VM” [7], the authors cross the semantic gap by relying on an additional virtual machine from which to run probes. This approach has a large overhead, thus would violate **R6**. Techniques like “Virtuoso” are complimentary to our trusted log and could be used to inform future probes of relevant locations within the guest for probing [38]. With regards to work related to IDS, Kosoresow and Hofmeyr show the effectiveness of system call traces by using temporal patterns of system calls to detect intrusions [39]. While the IDS presented here relies upon white-listing, their technique could also be applied.

HIMA [40] provides run time integrity checking of userspace programs. The authors monitor system calls to enable these integrity checks. Their approach used a much older VMM that generated `VMExits` for every interrupt, thus the authors paid minimal overheads during monitoring. Modern hardware does not exit on every interrupt, thus we utilize event-based probing to monitor specific system calls.

X. Conclusion & Future Work

In this paper we have shown the events that must be logged when probing guest instructions from within a VMM to ensure attempts to circumvent logging can be audited by higher level services. We show how existing instruction-replacement based probing mechanisms must be extended to include a mechanism to guarantee that every invocation of a probed instruction triggers an event. We do this by inducing a unique sequence of EPT violations to guarantee probes placed in a guest kernel are placed before the instruction can be invoked. We also identify two events that must be added to increase the number of attempts to circumvent logging that can be audited. These events will allow writes to MSRs and the `idt` to be audited to determine if an attacker is attempting to circumvent probes placed in system calls.

We highlight a methodology for creating trustworthy services built on top of instruction based probes. Namely, attempts to circumvent probes must be well understood and handled appropriately. This requires one consider not just protection of the instruction being replaced by the probing mechanism, but protection for every instruction that transfers control flow to the probed instruction. We highlight five requirements that drive this methodology. These include control flow considerations (**R1**), log integrity protections (**R2**), probe robustness (**R3**), calling function considerations (**R4**), and a requirement that logged data contain every event up to and including an attempt to circumvent logging (**R5**). Additionally, we try to adhere to design goals to minimize the performance impact on guests (**D1**), reduce additions to the trusted compute base (**D2**) and require no guest modifications (**D3**). We also outline potential attacks against an event-driven log and show how attacks can be audited.

A white-list based IDS and policy recording system is presented, highlighting how higher level services can be built on a trusted log. The IDS aims to provide defense-in-depth to VAs. Cloud computing and the prevalence of VAs allows for more complete white-listing with fewer false positives as VAs are commonly deployed to perform a single task. Any deviation from that task is a potential security violation. We demonstrate the effectiveness of our system using a real world vulnerability. Finally, we have shown the performance trade-offs required to protect against attack vectors when using hypervisor-based probing mechanisms. In particular, protection against the preemption attack vector requires a large overhead (55-75% in some benchmarks). Such an attack requires careful timing and relaxing our logging system to not defend against this attack still greatly increases the cost of carrying out a successful attack against the logging system while providing more tolerable performance overhead (around 10% in the worst case among evaluated benchmarks).

A6 highlights certain attacks that we do not currently log. In future work, we will look at porting systems like NICKLE [41] to modern hardware. NICKLE protects guest VM’s against rootkits, but does so without using HAV. Utilizing HAV may make the approach more tenable in terms of performance overhead on modern hardware and could be used to defense against **A6**. We have implemented our system on Linux using native virtualization. The specifics of ensuring timely probe placement in other operating systems or when using paravirtualization will be slightly different.

XI. Acknowledgments

This material is based on research funded in part by the Air Force Research Laboratory and the Air Force Office of Scientific Research, under agreement number FA8750-11-2-0084 and by the National Science Foundation Graduate Research Fellowship Program under Grant Number DGE-1144245. The views and opinions expressed in this article are the authors’ own.

REFERENCES

[1] [Online]. Available: https://aws.amazon.com/marketplace/

[2] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "VMM-based Hidden Process Detection and Identification Using Lycosid," in *International Conference on Virtual Execution Environments (VEE)*. ACM, 2008.

[3] ——, "Antfarm: Tracking processes in a virtual machine environment." in *USENIX Annual Technical Conference, General Track*, 2006, pp. 1–14.

[4] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pp. 233–247, 2008.

[5] X. Jiang, X. Wang, and D. Xu, "Stealthy Malware Detection Through VMM-based Out-of-the-Box Semantic View Reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.

[6] J. Lamps, I. Palmer, and R. Sprabery, "WinWizard: Expanding Xen with a LibVMI Intrusion Detection Tool," in *Int. Conference on Cloud Computing (CLOUD '14)*.

[7] Y. Fu and Z. Lin, "Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection," in *Symposium on Security and Privacy*. IEEE, 2012.

[8] A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy, "CloudSec: A security monitoring appliance for Virtual Machines in the IaaS cloud model," in *Network and System Security (NSS), 5th International Conference on*, 2011, pp. 113–120.

[9] M. Crawford and G. Peterson, "Insider Threat Detection using Virtual Machine Introspection," in *Hawaii International Conference on System Sciences (HICSS '13)*.

[10] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation," in *22nd ACM Conference on Computer and Communications Security (CCS '15)*.

[11] Z. J. Estrada, C. Pham, F. Deng, L. Yan, Z. Kalbarczyk, and R. K. Iyer, "Dynamic VM Dependability Monitoring Using Hypervisor Probes," in *Dependable Computing Conference (EDCC), 2015 Eleventh European*. IEEE, 2015, pp. 61–72.

[12] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system," in *30th Annual Computer Security Applications Conference on (ACSAC)*, 2014.

[13] J. Cihula, "Trusted boot: Verifying the xen launch," *Xen Summit*, vol. 7, 2007.

[14] R. Perez, R. Sailer, L. van Doorn *et al.*, "vTPM: virtualizing the trusted platform module," in *Proc. 15th Conf. on USENIX Security Symposium*, 2006, pp. 305–320.

[15] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 193–206.

[16] 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 *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 38–49.

[17] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 380–395.

[18] [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf

[19] [Online]. Available: http://support.amd.com/TechDocs/24593.pdf

[20] 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 *Reliable Distributed Systems, 2010 29th IEEE Symposium on*. IEEE, 2010, pp. 82–91.

[21] G. Wang, Z. J. Estrada, C. Pham, Z. Kalbarczyk, and R. K. Iyer, "Hypervisor Introspection: A Technique for Evading Passive Virtual Machine Monitoring," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. Washington, D.C.: USENIX Association, Aug. 2015. [Online]. Available: https://www.usenix.org/conference/woot15/workshop-program/presentation/wang

[22] N. A. Quynh and K. Suzaki, "Xenprobes, a lightweight user-space probing framework for xen virtual machine," in *USENIX Annual Technical Conference (ATC '07)*.

[23] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 289–298.

[24] C. Pham, Z. Estrada, P. Cao, Z. Kalbarczyk, and R. K. Iyer, "Reliability and security monitoring of virtual machines using hardware architectural invariants," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 13–24.

[25] B. D. Payne, M. D. P. de Carbone, and W. Lee, "Secure and Flexible Monitoring of Virtual Machines," in *Annual Computer Security Applications Conf. (ACSAC '07)*.

[26] Y. Hebbal, S. Laniepce, and J.-M. Menaud, "Virtual Machine Introspection: Techniques and Applications," *2015 10th International Conference on Availability, Reliability and Security (ARES)*, pp. 676–685, 2015.

[27] [Online]. Available: https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

[28] J. H. Huh and A. Martin, "Trusted logging for grid computing," in *Third Asia-Pacific Trusted Infrastructure Technologies Conference (APTC '08).*, 2008.

[29] [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ ab.html

[30] [Online]. Available: http://redis.io/topics/benchmarks

[31] [Online]. Available: https://www.linuxmint.com/

[32] [Online]. Available: http://thehackernews.com/2016/02/linux-mint-hack.html

[33] [Online]. Available: https://wpvulndb.com/vulnerabilities/8209

[34] [Online]. Available: https://www.rapid7.com/db/modules/exploit/unix/webapp/wp_ajax_load_more_file_upload

[35] D. Kienzle, R. Persaud, and M. Elder, "Endpoint Configuration Compliance Monitoring via Virtual Machine Introspection," *System Sciences (HICSS)*, 2010.

[36] T. Y. Win, H. Tianfield, and Q. Mair, "Virtualization security combining mandatory access control and virtual machine introspection," *IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC '14)*.

[37] F. Lombardi and R. Di Pietro, "KvmSec: a security extension for Linux kernel virtual machines," in *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 2009, pp. 2029–2034.

[38] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 297–312.

[39] A. P. Kosoresow and S. A. Hofmeyer, "Intrusion detection via system call traces," *IEEE software*, vol. 14, no. 5, pp. 35–42, Jan. 1997.

[40] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang, "Hima: A hypervisor-based integrity measurement agent," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 2009, pp. 461–470.

[41] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2008, pp. 1–20.