Reliability and Security Monitoring of Virtual Machines Using Hardware Architectural Invariants

Cuong Pham, Zachary Estrada, Phuong Cao, Zbigniew Kalbarczyk, Ravishankar Iyer
University of Illinois at Urbana-Champaign
{pham9, zestrad2, pcao3, kalbarcz, rkiyer}@illinois.edu

Abstract—**This paper presents a solution that simultaneously addresses both reliability and security (RnS) in a monitoring framework. We identify the commonalities between reliability and security to guide the design of HyperTap, a hypervisor-level framework that efficiently supports both types of monitoring in virtualization environments. In HyperTap, the logging of system events and states is common across monitors and constitutes the core of the framework. The audit phase of each monitor is implemented and operated independently. In addition, HyperTap relies on hardware invariants to provide a strongly isolated root of trust. HyperTap uses active monitoring, which can be adapted to enforce a wide spectrum of RnS policies. We validate HyperTap by introducing three example monitors: Guest OS Hang Detection (GOSHD), Hidden RootKit Detection (HRKD), and Privilege Escalation Detection (PED). Our experiments with fault injection and real rootkits/exploits demonstrate that HyperTap provides robust monitoring with low performance overhead.**

I. Introduction

Reliability and security (RnS) are two essential aspects of modern highly connected computing systems. Traditionally, reliability and security tend to be treated separately because of their orthogonal nature: while reliability deals with accidental failures, security copes with intentional attacks against a system. As a result, mechanisms/algorithms addressing the two problems are designed independently, and it is difficult to integrate them under a common monitoring framework.

Addressing RnS aspects separately may lead to unforeseen consequences. For example, a reliability monitor (e.g., a heartbeat server) may have a vulnerability that allows remote attackers to exploit the system. On the other hand, a security monitor may introduce a new failure mode that the current system is not designed to handle. Furthermore, different modules' design and implementation may not be compatible. For instance, suppose two monitors both require exclusive access to a resource, e.g., a performance register. Such monitors cannot co-exist in the same system. This situation places system designers in a difficult position, in which they must trade-off one essential quality for another. In a milder scenario, the system has to pay a combinational cost, e.g., development, deployment, and runtime performance costs, of both solutions.

In this paper, we identify the commonalities between reliability and security monitoring to guide the development of suitable frameworks for combining both uses of monitoring. We apply our observations in the design and implementation of the HyperTap framework for virtualization environments.

A monitoring process can be divided into two tightly coupled phases: *logging* and *auditing* [1]. In the *logging* phase, relevant system events (e.g., a system call) and state (e.g., system call parameters) are captured. In the *auditing* phase, these events and states are analyzed, based on a set of policies that classify the state of the system, e.g., normal or faulty. Based on that model, we observe that although RnS monitors may apply different policies during the auditing phase, they can utilize the same event- and state- logging capability. This observation suggests that the logging phases of multiple RnS monitors need to be combined into a common framework. Unification of logging phases brings further benefits, namely, it avoids potential conflict between different monitors that track the same event or state, and reduces the overall performance overhead of monitoring.

A unified logging framework for RnS must be founded on an *isolated root of trust* and have support for *active monitoring*. Current virtual machine monitoring techniques, e.g., Virtual Machine Introspection (VMI), either exhibit neither of those two properties, or offer only one at time. An isolated root of trust asserts that the source of captured events and state cannot be tampered with by actors inside target systems. Traditional VMI techniques fail on that condition, as they choose to rely solely on the guest Operating System (OS) to report its own state. An example of that violation is presented in [2] (the issue is further discussed in Section IV-B). For RnS monitoring, active monitoring (or *event-driven monitoring*) has been shown to be more advantageous than passive monitoring (or *state polling*), as the former can capture operational events in addition to the system's state[3]. Furthermore, active monitoring can overcome the time sensitivity of passive monitoring, e.g., it can detect short latency failures and transient attacks [4], as further illustrated in Section IV-C.

In order to fulfill the requirements stated above, we present a framework implemented at the hypervisor level called *HyperTap*, that provides an event logging infrastructure suitable for implementing various types of RnS policies for Virtual Machines (VMs). In HyperTap, the logging phase is common for all monitors and constitutes the core of the framework. The auditing phase of each monitor is implemented and operated independently. To achieve an isolated root of trust, HyperTap employs hardware architectural invariants, which cannot be modified by attackers and failures inside VMs. These invariants hold under assumptions about the trustworthiness of the hypervisor and hardware stated in Section V-A. In order to support active monitoring and intercept a wide range of system events, HyperTap utilizes the Hardware Assisted Virtualization (HAV) event generation mechanism. The events are then delivered to registered auditors which realize a variety of RnS monitoring policies.

In order to demonstrate the feasibility of HyperTap as a framework that unifies RnS monitoring for virtualized environments, we describe the design and evaluation of three practical lightweight auditors: Guest Operating System Hang Detection (GOSHD), Hidden Rootkit Detection (HRKD), and Privilege Escalation Detection (PED). The GOSHD and HRKD auditors are chosen to show that a common event, e.g., context switching, can be simultaneously used for both reliability and security monitoring. The PED auditor is chosen to show the advantages of active monitoring over passive monitoring. In addition to facilitating the unification of RnS monitors, HyperTap's dependable hardware invariants and active monitoring mechanism enable auditors with high detection coverage. GOSHD can detect 99.8% of injected hang failures, including partial hang failures in multiprocessor VMs – a new failure mode revealed by GOSHD. HRKD can detect both hidden processes and kernel threads regardless of their hiding mechanisms. And PED can detect all four types of proposed attacks that defeat Ninja [5], a real-world privilege escalation detector that uses passive monitoring.

II. Related Work

Previous research [6], [7], [8] has recognized the importance of addressing RnS under a common framework. Frequently, the approach has been to extend existing hardware [6], [7] with support for RnS monitoring. While hardware implementations have performance and accessibility advantages over software in the context of fine-grained monitoring; their extensibility and customizability after deployment can be quite limited. HyperTap extends the concept of Virtual Machine Introspection (VMI), which takes advantage of the hypervisor software layer to provide monitoring support for the upper VM layer.

Traditional VMI techniques, such as VMWatcher [9] and XenAccess [10], extract knowledge from the internal data structures of the guest operating system (OS). That information is then used to detect security attacks [9], [10], [11], [12], [13]. However, that approach is vulnerable to attacks that can manipulate the data structures used by VMI, as demonstrated in [2], [14], [15]. Another limitation of traditional VMI is that it only supports *passive monitoring*, i.e., monitoring that performs system inspection in a *polling manner*. Passive monitoring is not suitable for enforcing many security policies [3]. Moreover, it is vulnerable to *transient attacks* [4], which are attacks that occur between logging phases.

In order to address the limitations of passive monitoring, event-driven, or *active*, monitoring has been proposed for out-of-VM security enforcement [3], [16]. Lares [3], for example, is an architecture that securely places hooks in protected VMs and intercepts their events. However, this hook placement mechanism is intrusive to the guest system. To reduce the amount of manual intervention in the process, the authors of [16] propose a method to automatically identify locations to place useful application-aware hooks.

Previous studies [17], [18], [19], [20] show how hardware architectural state can be used to interpret a guest OS's operations. For example, Antfarm [17] and its extension Lycosid [18] describe a guest user process counting technique based on monitoring virtual memory (i.e., tracking `CR3` in x86). Ether [19] utilizes the VM Exit mechanism provided by HAV to record traces of guest VM execution for offline malware analysis. HyperTap builds on those concepts to provide robust online monitoring for both reliability and security.

Out-of-VM failure detection has also been a subject of study in previous research. The study in [21] uses supervised machine learning on a set of hypervisor-level counters, such as guest CPU usage and I/O count, to detect guest OS failures. Such approaches can benefit greatly from HyperTap's common logging infrastructure and the counters it provides (e.g., different types of events and states, which directly reflect the operations of guest VMs).

III. Hardware-Assisted Virtualization Review

In order to support RnS monitoring, HyperTap takes advantage of features used by HAV, particularly the VM Exit mechanism. This section reviews the basic concepts of HAV to provide the context for the discussion in subsequent sections.

In 1974, Popek and Goldberg described the "*trap-and-emulate*" model of virtualization [22]. "Trapping" prevents the VM from taking privileged control, and "emulating" ensures that the semantics of the control are done without violating the VM's expectations.

The trap-and-emulate can be done either (i) entirely in software via *binary translation* and/or *para-virtualization*, or (ii) using Hardware-Assisted Virtualization (e.g., Intel VT-x and AMD-V). The latter design, HAV, supports an unmodified guest OS with small performance overhead and significantly simplifies the implementation of hypervisors. Although here we focus on the x86 architecture and Intel's VT-x, the techniques could be mapped to AMD-V and PowerPC, since these provide a similar mechanism that traps privileged instructions.

A. VM Exits

In addition to x86's privilege rings, HAV defines guest mode and host mode execution. Certain operations (e.g. privileged instructions) are restricted in guest mode. If a guest attempts to execute a restricted operation, the processor relinquishes control to the hypervisor. If that happens, the processor fires a *VM Exit* event and transitions from guest mode to host mode. After the host has finished handling the exception, it resumes guest execution via a *VM Entry* event.

Each type of restricted operation triggers a different type of VM Exit event. For example, if the guest attempts to modify the contents of a Control Register (CR), the processor fires a `CR_ACCESS` VM Exit event. In addition to the event, control fields and the state of the suspended VM are saved into a data structure (*VMCS* in Intel VT-x and *VMCB* in AMD-V).

B. Extended Page Tables (EPT)

Extended Page Tables (EPT) is a hardware-supported mechanism for virtualizing the Memory Management Unit (MMU). When EPT is enabled, each VM accesses its private memory via a *guest-physical address (GPA)*. The processor translates GPAs to physical addresses by traversing the *EPT paging structures*, which are transparent to the guest OS. *Guest virtual address (GVA)* is the term for the virtual addresses used by the guest system. EPT also allows specification of access permissions for guest memory pages, namely 'read,' 'write,' and 'execute.' Guest attempts at unauthorized accesses cause `EPT_VIOLATION` VM Exits.

C. Notation

Virtual CPU (vCPU): a VM's virtual processor. With HAV, each vCPU occupies one physical CPU core until the next VM Exit event. At a VM Entry transition, the hypervisor assigns an available physical CPU core to the suspended vCPU, unless CPU affinity is used.

`A.B`: reference to field `B` of data structure `A` at the *host* layer, e.g., `vcpu.CR3`: `vcpu` contains a field that stores the value of the guest's `CR3` register.

`C`→`D`: reference to field `D` of data structure `C` at the *guest* layer. In other words, `C` is a guest virtual address, e.g., `TSS`→`RSP0` references the field `RSP0` of the `TSS` structure managed by the guest kernel.

IV. Designing Reliability and Security Monitoring for Virtual Machines

This section discusses the benefits of (i) having a unified logging channel for all monitors, (ii) using active monitoring instead of passive monitoring, and (iii) placing the root of trust at hardware invariants for virtual machine monitoring.

A. Unified Logging

It is not uncommon for co-deployed logging mechanisms to conflict. For instance, two monitors relying on a certain counter that only allows exclusive access cannot uses it simultaneously. A concrete example would be to deploy both the failure detection technique proposed in [23] and the malware detection technique proposed in [24] in the same system, as they both use hardware performance counters. In addition, one monitor may become a source of noise for other monitors. For example, intrusive logging could generate an excessive number of events.

The problem can be solved by unifying logging for co-located monitors. Unified logging is responsible for (i) retrieving common target system events and states, and then (ii) streaming them in a timely manner to customizable auditors, which enforce RnS policies.

Aside from avoiding potential conflicts, the combination of logging phases yields additional benefits. It can reduce the overall performance overhead of combined monitors. To ensure the consistency of captured states and events, logging is often a blocking operation. Once the event and state have been logged, an audit can be performed in parallel with execution of the target system. Therefore, combining blocking logging phases boosts performance, even in cases where the captured states differ. Furthermore, this approach inherits other benefits of the well-known divide-and-conquer strategy: it allows one to focus on hardening the core logging engine, and enables incremental development and deployment of auditing policies.

B. Achieving Isolation via Architectural Invariants

An *OS invariant* is a property defined and enforced by the design and implementation of a specific OS, so that the software stack above it, e.g., user programs and device drivers, can operate correctly. In the context of VMI, OS invariants allow the internal state of a VM to be monitored from the outside by decoding the VM's memory [9], [10], [12], [11], [13]. No user inside a VM can interfere with the execution of outside monitoring tools. However, monitoring tools still share input, e.g., a VMs' memory, with the other software inside VMs. Therefore, those monitoring tools are vulnerable to attacks at the guest system level, as demonstrated in [2], [14], [15].

An *architectural invariant* is a property defined and enforced by the hardware architecture, so that the entire software stack, e.g., hypervisors, OSes, and user applications, can operate correctly. For example, the x86 architecture requires that the `CR3` and `TR` registers always point to the running process's Page Directory Base Address (`PDBA`) and Task State Segment (`TSS`), respectively. Hardware invariants and HAV features have been studied in the context of security monitoring [17] and offline malware analysis [19].

We find that architectural invariants, particularly the ones defined by HAV, provide an outside view with desirable features for VM RnS monitoring. The behaviors enforced by HAV involve primitive building blocks of essential OS operations, such as context switches, privilege level (or ring) transfers, and interrupt delivery. Furthermore, strong isolation between VMs and the physical hardware ensures the integrity of architectural invariants against attacks inside VMs. Software inside VMs cannot tamper with the hardware as it can with the OS. In this study, we explore the full potential of HAV for online enforcement of RnS policies.

However, relying solely on architectural invariants and ignoring OS invariants would widen the semantic gap separating the target VM and the hypervisor. The reason is that many OS concepts, such as user management (e.g., processes owned by different users), are not defined at the architectural level. In this study, we propose to *use architectural invariants as the root of trust when deriving OS state.* For example, the `thread_info` data structure in the Linux kernel containing thread-level information can be derived from the `TSS` data structure, a data structure defined by the x86 architecture.

In order to circumvent our OS state derivation, an attack would need to change the layout of OS-defined data structures (e.g., by adding fields to an existing structure that point to tainted data). Changing data structure layout, as opposed to changing values, is difficult for attackers, because (i) it involves significant changes to the kernel code that references the altered fields, and (ii) it would need to relocate all relevant kernel data objects. Not only are those attacks difficult to perform on-the-fly, but since malware always tries to minimize

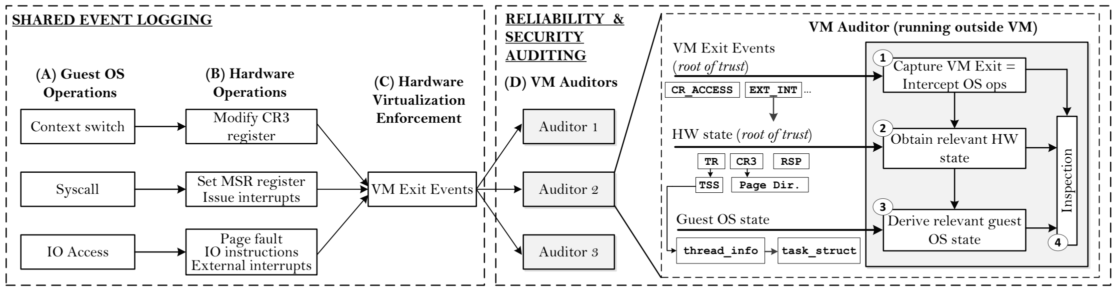

Fig. 1: HyperTap Monitoring Framework: (A) Guest OS operations that are subjects of the monitors; (B) Hardware operations that are required to perform each guest OS operation; (C) VM Exit events that are generated before logged operations are performed; (D) The captured events are delivered to auditors running outside the VM.

its footprint, our approach significantly impedes would-be attackers.

C. Robust Active Monitoring

Passive monitoring is suitable for persistent failures and attacks, because it assumes the corrupted or compromised state remains in the system sufficiently longer than the polling interval. That assumption does not hold in many RnS problems. For example, the majority of crash and hang failures in Linux systems have short failure latencies (the time for faults to manifest into failures) [25]. An unnecessarily long detection latency, e.g., caused by polling monitoring, would result in subsequent failure propagation or inefficient recovery (e.g., multiple roll-backs).

As we demonstrate in Section VIII-C, a transient attack can be combined with other techniques to create a stealthy attack that can defeat passive monitoring.

Active monitoring, on the other hand, possesses many attractive features. Since it is event-driven, there is no time dependence that can be exploited. Furthermore, active monitoring can capture system activities in addition to the system state, which passive monitoring provides. System activities are the operations that transition a system from one state to another. Invoking a system call is an example of a system activity. In many cases, information about system activities is crucial to enforcing RnS policies.

Active monitoring is not foolproof, as it can suffer from *event bypass* attacks. If an attack can prevent or avoid generation of events that trigger logging, it can bypass the monitor. To make active monitoring robust, we propose to use hardware invariants, specifically the VM Exit feature provided by HAV, to generate events. Section VI presents the hardware invariants used to ensure the trustworthiness of generated events.

V. HyperTap Framework and Implementation

Following the principles presented in the previous section, here we describe the design and implementation of HyperTap.

A. Scope and Assumptions

HyperTap integrates with existing hypervisors to safeguard VMs against failures and attacks. It aims to make this protection transparent to VMs by utilizing existing hardware features. Thus, HyperTap does not require modification of either the existing hardware or the guest OS's software stack.

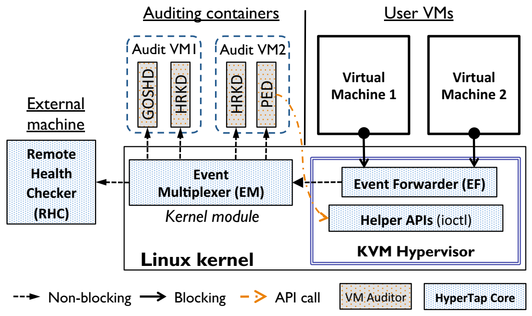

Fig. 2: Implementation of HyperTap in the KVM hypervisor. The hypervisor is modified to forward VM Exit events to the Event Multiplexer (EM), which is implemented as a separate kernel module. The EM forwards events to registered auditors running as user processes inside auditing containers. The Remote Health Checker (RHC) monitors the hypervisor's liveness.

HyperTap's implementation assumes that the underlying hardware and hypervisor are trusted. Although extra validation and protection for the hardware and hypervisor could address concerns about the robustness of different hypervisors against failures and attacks, these issues are beyond the scope of this work.

B. Monitoring Workflow

Fig. 1 depicts the overall workflow of HyperTap. The left side of the figure illustrates how the shared event logging mechanism works and the right side describes the auditing phase. HyperTap utilizes HAV to intercept the desired guest OS operations through VM Exit events generated by corresponding hardware operations. Since the HAV VM Exit mechanism is not designed to intercept all desired operations, e.g., system calls, Section VI presents algorithms to generate VM Exit events for such operations.

HyperTap supports a wide range of events, from coarse-grained events, such as process context switches, to finer-grained events, such as system calls, and very fine grained events, such as instruction execution and memory accesses. That variable granularity ensures that HyperTap can be adopted for a broad range of RnS policies.

HyperTap delivers captured events to registered auditors, which implement specific RnS policies. An auditor starts by registering for a set of events needed to enforce its policy. Upon the arrival of each event, the auditor analyzes the state information associated with the event. Auditors are associated with VMs and each VM can have multiple auditors.

HyperTap also provides an interface that allows auditors to control to target VMs. For example, the auditing phase is non-blocking by default, but an auditor may pause its target VM during analysis in order to stop the VM during an attack, or roll-back the VM when it detects a non-recoverable failure.

C. Implementation

This subsection presents the integration of HyperTap with KVM [26], hypervisor built with HAV as a Linux kernel module. Fig. 2 depicts the deployment of HyperTap's components.

HyperTap's unified logging channel is implemented through two components: an *Event Forwarder (EF)* and an *Event Multiplexer (EM)*. The EF is integrated into the KVM module, and forwards VM Exit events and relevant guest hardware state to the EM. By default, events are sent non-blocking to minimize overhead. The EM, which is implemented as another Linux kernel module in the host OS, buffers input events from the EF and delivers them to the appropriate auditors.

The EM is also responsible for sampling VM Exit events that are sent to a Remote Health Checker (RHC) running in a separate machine. The RHC server acts as a heartbeat server to measure the intervals between received events. If no events are received after a certain amount of time, it raises an alert about the liveness of the monitoring system.

Auditors are implemented as user processes inside auditing containers[1] running on the host OS. Compared to the dedicated auditing VM used in previous work [12] [3], this approach offers multiple benefits. First, it provides lightweight attack and failure isolation among different VMs' auditors, and between auditors and the host OS. Second, it simplifies implementation and reduces the performance overhead of event delivery from the EM module. Finally, it allows the integration of auditors into existing systems, since containers are robust and compatible with most current Linux distributions.

We needed to add less than 100 lines of code to KVM to implement the EF component and export Helper APIs.

VI. Hardware Invariants for VM Logging

This section describes events that can be monitored via hardware invariants and VM Exit events, the core mechanism of HyperTap's shared logging channel. Table I summarizes guest systems' internal operations, the hardware invariants, and the types of VM Exit events associated with them. The following sub-sections detail the use of these invariants.

[1]We use Linux containers (LXC) http://linuxcontainers.org/

A. Context Switch Interception

1) Process Switch Interception: **Architectural Invariant**. Process switches can be observed by monitoring `CR_ACCESS` VM Exit events. In x86, the `CR3` register, or Page Directory Base Register (`PDBR`) contains the Page Directory Base Address (`PDBA`) for the virtual address space of the running process. As this base address is unique for each user process, we can use it as a process identifier.

Process Counting Algorithm. We can count the number of processes running on a guest VM by monitoring `CR_ACCESS` events. This algorithm is independent of any data structure the guest OS uses to manage its processes.

Fig. 3A shows the pseudo-code for the process counting algorithm. The set of PDBAs (`PDBA_set`) is empty when the guest OS boots up. At each `CR_ACCESS` event in which `CR3` is modified (`CR3 <- PDBA`), the algorithm updates `PDBA_set` with the value that will be written to `CR3`.

2) Thread Switch Interception: Monitoring of thread[2] switches requires more effort than tracking `CR_ACCESS` events, as threads can share the same virtual address space. In addition, a thread can reuse the virtual address space of another process (e.g., Linux kernel threads).[3]

Architectural Invariant. In order to manage threads, the x86 processor uses the Task Register (`TR`) and Task-State Segment (`TSS`) structures. The `TSS`, stored in main memory, holds the stack pointers of a task for different privilege levels, and the `TR` points to the `TSS` structure of the current task. The `TSS` is also used to support privilege protection. Each time execution transfers from user level (3) to kernel level (0), the kernel stack pointer is automatically loaded from the `TSS` by the CPU (e.g., `RSP <- TSS`$\rightarrow$`RSP0`). Since all kernel threads share the same virtual address range, each has a separate address range for its stack. Therefore, the kernel stack pointer (`RSP0`) stored in the `TSS` can be used as a thread identifier.

Thread Switch Interception Algorithm. Each thread switch modifies the `TSS` stored in memory. Therefore, we can track thread switches by setting memory access permissions. Specifically, on a guest system with EPT, a write to an EPT write-protected address triggers an `EPT_VIOLATION` VM Exit. We use this mechanism to track the kernel stack pointer.

Fig. 3B shows the pseudo-code for this algorithm. After the guest OS finishes setting up its data structures (e.g., the `CR3` register gets written for the first time), the algorithm sets all pages that contain `TSS` structures (one per vCPU) as write-protected. Each time a `TSS` structure is modified, the hypervisor gets notified by an `EPT_VIOLATION` event.

B. System Call Interception

System calls allow user mode processes to invoke kernel mode functions. At the hardware level, a system call transfers the CPU from user to kernel mode. That transfer from a lower

[2]A thread is equivalent to a task in the x86 architecture.

[3]*kthreads* reuse the virtual address space of the previously scheduled process. All processes in Linux have the same kernel address range. Windows does not have standalone kernel threads.

TABLE I: Summary of guest internal events and related VM Exit types

Monitoring Category	Guest event	Related VM Exit	Architectural Invariant
Context switch interception (§VI-A)	Process context switch (§VI-A1)	`CR_ACCESS`	The `CR3` register always points to the `PDBA` of the running process Writes to CR registers cause `CR_ACCESS` VM Exits
	Thread switch (§VI-A2)	`EPT_VIOLATION`	The `TR` register always points to the `TSS` structure of the running process `TSS.RSP0` is unique for each thread
System call interception (§VI-B)	Interrupt-based system call (§VI-B1)	`EXCEPTION`	Software interrupts cause `EXCEPTION` VM Exits
	Fast system call (§VI-B2)	`WRMSR`, `EPT_VIOLATION`	`SYSENTER`'s target instruction is stored in an MSR register Write to MSR registers causes `WRMSR` VM Exit
I/O access interception (§VI-C)	Programmed I/O	`IO_INST`	Execution of I/O instructions (e.g., `IN, INS, OUT, OUTS`)
	Memory mapped I/O	`EPT_VIOLATION`	Access to memory mapped I/O areas, which are set as protected
	Hardware interrupt	`EXTERNAL_INT`	Hardware interrupt delivery causes `EXTERNAL_INT` VM Exits
	I/O APIC access	`APIC_ACCESS`	I/O Advance Programmable Interrupt Controller (APIC) events
Low-level interception (§VI-D)	Memory access	`EPT_VIOLATION`	Accesses to memory regions with proper permissions cause `EPT_VIOLATION` VM Exits
	Instruction execution	`EPT_VIOLATION`	Execution of instructions from non-executable regions causes `EPT_VIOLATION` VM Exits

to higher privilege is strictly checked by the processor: it must be done through pre-defined *gates*. This section describes techniques to intercept two types of system calls: *interrupt-based system calls* and *fast system calls*.

1) Interrupt-based System Calls: The legacy method for issuing a system call in x86 is to raise a software interrupt. For example, Linux uses `INT $0x80` and Windows uses `INT $0x2E` to issue system calls. The interrupt handler routine is the common gate for all system calls, and parameters of system calls are passed through general-purpose registers.

Architectural Invariant. In a VM, each software interrupt triggers an `EXCEPTION` VM Exit.[4]

Interrupt-based System Call Interception Algorithm. We developed an algorithm that intercepts interrupt-based system calls, shown in Fig. 3D. If the type and number of the interrupt indicate a system call, the algorithm records all the registers that could carry the system call's parameters and then generates a notification regarding the system call.

2) Fast System Calls: A fast system call mechanism was added to x86 with the `SYSENTER/SYSEXIT` instruction pair for Intel processors and the `SYSCALL/SYSRET` instructions for AMD processors.

Architectural Invariant. The `SYSENTER` instruction takes input from Model Specific Registers (MSRs) and general-purpose registers. For example, `SYSENTER`'s target instruction address is stored in the `IA32_SYSENTER_EIP` MSR. An MSR can only be modified via a `WRMSR` instruction, a privileged instruction that causes `WRMSR` VM Exits.

Fast system call interception algorithm. Fig. 3E contains pseudo-code for fast system call interception. The algorithm uses `WRMSR` events to identify the address of the system call entry point in the guest VM. The address is set to execute-protect so that a guest's attempt to execute the system call entry point will generate an `EPT_VIOLATION` VM Exit.

C. I/O Access Interception

A primary function of the hypervisor is to multiplex I/O devices for its VMs, except when a VM is given exclusive access via an I/O pass-through mode. HAV provides several VM Exits that the hypervisor can use to capture IO accesses from guest VMs. We categorize I/O accesses into three types:

[4]Intel VT-x allows selection of which interrupts cause `EXCEPTION` VM Exits via an `EXCEPTION_BITMAP`.

Programmed I/O (PIO) is performed through I/O instructions, such as `IN` and `OUT`. These instructions trigger `IO_ACCESS` events when executed in guest mode.

Memory Mapped I/O (MMIO) is performed through instructions that manipulate memory (e.g., `MOV`, `AND`, `OR`). In order to trap MMIO, the hypervisor sets memory protection for the allocated MMIO area so that accesses to this area will trigger `EPT_VIOLATION` events.

I/O interrupts are interrupts raised by physical devices to notify guest VM about I/O-related events (e.g., an incoming network packet). The presence of a pending interrupt causes either an `EXTERNAL_INT` or `APIC_ACCESS` VM Exit event.

Because of the diversity of I/O devices, details for each type of device are not covered, and it is up to implementers to choose an appropriate mechanism.

D. Fine-grained Interception

The EPT feature presented in Section III-B makes it possible to track a guest's execution at the single instruction and memory access level by setting appropriate access permissions. However, that fine-grained interception incurs a significant performance cost. To minimize its impact, an auditor should make use of that feature only for selective critical protection.

VII. Examples of Auditors

We expand on the techniques presented in the previous section to demonstrate how to build auditors using HyperTap. We present two examples that showcase how RnS monitoring can be combined (GOSHD and HRKD) and one example that demonstrates the effectiveness of active monitoring (PED).

A. Guest OS Hang Detection

1) Failure Model: We consider an OS as being in a hang state if it ceases to schedule tasks. This failure model is similar to the one introduced in [23]. In multiprocessor systems, it is possible for the OS to experience a hang on a proper subset of available CPUs. If that happens, we say that OS is in a *partial hang state*, as opposed to a full hang state, in which the OS is hung on all CPUs.

An example of a software bug that causes hangs in the OS kernel is a missing unlock (i.e., release) of a spinlock in an exit path of a kernel critical section. All threads that try to acquire this lock after the buggy exit path has been executed end up in a hung state. If the hung kernel thread is in a non-preemptible code section (e.g., either the kernel itself is non-preemptible, A

```
At VM Start:
  PDBA_set = {}
  Monitor CR_ACCESS events

At each CR_ACCESS event (CR3 <- PDBA):
  if (PDBA not in PDBA_set)
    PDBA_set += PDBA

Count the Virtual Address Spaces:
  // save current PDBA
  Saved_CR3 = vcpu.CR3
  // Remove invalid PDBA
  for_each PDBA in PDBA_set {
    // Step 1: Change Page Directory
    vcpu.CR3 = PDBA
    // Step 2: Test Page Directory
    gpa = gva_to_gpa(known_gva)
    if (gpa == UNMAPPED_GVA)
      remove(PDBA_set, PDBA)
  }
  // restore the PDBA
  vcpu.CR3 = Save_CR3
  return size_of(PDBA_set)
```

B

```
At VM Start:
  Monitor CR_ACCESS events

On the first CR_ACCESS event (write to CR3):
  for_each vcpu[i]
    Set write-protect for vcpu[i].TR
    Monitor EPT_VIOLATION events

At each EPT_VIOLATION event on vcpu ([Addr] <- V):
  if (Addr == &vcpu.TR->RSP0) // vcpu.TR = TSS
    // V = Kernel_stack_base
    Forward_switch_evt(V)
```

C

```
On the first CR_ACCESS event (write to CR3):
  for_each vcpu[i]
  saved_TR[i] = vcpu[i].TR;

Integrity checking (e.g., on every VM Exit):
for_each vcpu[i]
  if (save_TR[i] != vcpu[i].TR)
    // TSS has been relocated
    raise_alert();
```

D

```
At VM Start:
  Monitor EXCEPTION events

At each EXCEPTION event (ex_type, int_num):
if ((ex_type == SOFTWARE_INT) &&
     (int_num == 0x80) || (int_num == 0x2e)) {
  invoked_syscall = read_register(EAX);
  para1 = read_register(EBX);
  ...
  Forward_syscall(invoked_syscall, para1, ...);
}
```

E

```
At VM Start:
  Monitor WRMSR events

On the WRMSR event (IA32_SYSENTER_EIP <- addr):
  syscall_entry = addr;
  Each VCPU: Set execute-protected for
             the page containing syscall_entry
  Monitor EPT_VIOLATION events

At each EPT_VIOLATION event on vcpu (execute [Addr]):
  if (Addr == syscall_entry) {
    invoked_syscall = read_register(EAX);
    para1 = read_register(EBX);
    ...
    Forward_syscall(invoked_syscall, para1, ...);
  }
```

Fig. 3: Pseudo-code for each algorithm. **(A)**: Process Counting Algorithm, **(B)**: Thread switch interception, **(C)**: TSS integrity checking, **(D)**: Interrupt-based system call interception, **(E)**: Fast system call interception

or the thread has purposely disabled preemption), the kernel hangs on the CPU that is executing the hung thread. The hung thread may also be holding other locks, which can cascade into hanging of more threads. In a multiprocessor system a partial hang usually results in a full hang. The kernel stays in a partial hang state until the hang propagates to all available CPUs. However, if the kernel has no other lock dependencies with the hung threads, it can stay in the partial hang state until it gets shut down or rebooted.

Distinguishing between OS partial and full hang is important for two reasons. (i) Previous OS hang detection approaches use external probes, e.g., heartbeats, to detect OS hangs. In a multiprocessor system, mechanisms to generate heartbeats may not be affected by a partial hang, and would continue to report error-free conditions. (ii) Detecting partial hangs results in a shorter detection latency, as all full hangs are preceded by a partial hang. The Guest OS hang detection (GOSHD) module discussed in this section detects both partial and full OS hangs.

2) GOSHD Mechanism: GOSHD uses the thread dispatching mechanism discussed in Section VI-A2 to monitor the VM's OS scheduler. The `EPT_VIOLATION` and `CR_ACCESS` mechanisms in HAV guarantee that GOSHD can capture all context switch events. If a vCPU does not generate any switching events for a predefined threshold time, GOSHD declares that the guest OS is hung on that vCPU. Because the vCPUs are monitored independently of each other, GOSHD can detect both partial hangs and full hangs. From GOSHD's perspective, guest tasks are scheduled independently on each vCPU. Since GOSHD monitors the absence of context switching events to detect hangs, it is important to properly determine the threshold after which it is safe to conclude that the OS is hung on a vCPU. If this threshold is shorter than the time between two consecutive context switches, GOSHD generates false alarms. In order to be safe and fairly conservative, we profiled the guest OS to determine the maximum scheduling time slice, and set the threshold to be twice the profiled time. The numbers are usually on the order of milliseconds, or at most seconds, and are quicker compared to other hang detection techniques, such as heartbeat, or timer watchdogs, which frequently have detection times on the order of tens of seconds or minutes.

B. Hidden Rootkit Detection (HRKD)

1) Threat Model: Rootkits are malicious computer programs created to hide other programs from system administrators and security monitoring tools. For example, users cannot see a hidden process or thread via common administrative tools, such as Task Manager, PS, or TOP. Autonomic security scanning tools can also be bypassed simply because their inspection lists do not contain the hidden programs.

There are many existing techniques to hide a process, such as Direct Kernel Object Manipulation (DKOM) [27], physical memory manipulation [28], and dynamic kernel code manipulation [15]. For example, using those techniques, a rootkit can stealthily detach the data objects belonging to the malicious programs from their usual lists (e.g., remove a `task_struct` object from Linux's `task_list`). Therefore, a normal list traversal cannot reveal the detached object. As exemplified by previous studies [2], [15], [14], well-crafted rootkits can escape the detection of guest OS invariant-based scanning tools.

2) Detection Technique: Our HRKD module employs the context switch monitoring (Section VI-A) methods to inspect every process/thread that uses the vCPU, regardless of how kernel objects are manipulated. Each time a process or a thread is scheduled to use CPUs, it is intercepted by the module for further inspection. This interception defeats hidden malware; it puts malicious programs back on the inspection list.

In order to detect a hidden user process or thread, the *process counting algorithm* (Fig. 3A) or *thread switch interception algorithm* (Fig. 3B) can be used. These algorithms are independent of the method by which the guest OS manages process-related data structures, because they rely only on architectural invariants. Inspection starts from the `CR3` or `TR` registers. Therefore, the observed number of processes always reflects the exact number of running processes. This is a trusted view that can be cross-validated against other views, e.g., a view from existing VMI tools or views from in-guest utilities, which may be the target of rootkits. Discrepancies between these views reveal the presence of hidden user processes and threads.

3) How Can a Rootkit Hide from HRKD?: A rootkit can hide from our HRKD by suppressing `CR3` access (for user-level rootkits) or `RSP0` access (for kernel-level rootkits) VM Exits. It can do so by *reusing the `CR3` (virtual address space) or `RSP0` (kernel stack)* of an existing process or kernel thread. Such attacks are called code injection attacks, which are not actually rootkits. Nevertheless, our HRKD is not designed to detect this class of attack.

C. Privilege Escalation Detection (PED)

Ninja [5] is a real-world privilege escalation detection system that uses passive monitoring. Ninja is included in the mainline repository for major Linux distributions, including Debian variants like Ubuntu. Ninja periodically scans the process list to identify if a root process has a parent process that is not from an authorized user (i.e., not defined in Ninja's "magic" group). If so, the root process is flagged as privilege-escalated. Ninja optionally terminates such processes to prevent further damage to the system. In order to avoid mistakenly killing setuid/setgid processes, Ninja allows users to create a "white list" of legitimate executables that are not subjected to its checking rules. The interval between checks is configurable (1s by default).

We implement HT-Ninja, which utilizes HyperTap for detecting privilege escalation attacks. We reuse the OS-level Ninja's checking rules when looking for unauthorized processes and make the following changes:

Transform passive monitoring to active monitoring. We define the following events at which a process is checked: (i) *first context switch of each process*; and (ii) *every I/O-related system call* (e.g., open, read, write, and lseek). That ensures that we check before any unauthorized actions, e.g., file or network, are conducted.

Using architectural invariants. The original Ninja uses Linux's `/proc` filesystem to obtain information about running processes. HT-Ninja uses only hardware state, such as the `TR` and `CR3` registers, to identify current running processes. HT-Ninja derives OS-specific information, such as User ID (uid) and Effective User ID (euid), from the `TSS` structure and `RSP` register, which can be combined to obtain the exact `thread_info` and `task_struct` objects of each process.

D. Other Uses of HyperTap

The logging capabilities presented in Section VI can also be used to implement a wide variety of RnS monitors. For example, there is a class of security tools that depend on system call interception [29], [30], [31]. Failure detection based on machine learning, e.g. [21], can be applied to the events and states logged by HyperTap.

HyperTap could also be incorporated into the runtime memory safety technique proposed in [32]. That technique consists of two steps: (i) compiler analysis and instrumentation, to guide (ii) runtime memory safety checking. The latter step requires OS modification to intercept privileged operations, e.g., MMIO, MMU configuration, and context switching [33]. Since HyperTap supports those interceptions without altering the guest OS, it shows promise for being integrated with runtime checking. Such incorporation would allow a variety of RnS detectors to be implemented, such as detectors for silent data corruption, buffer overflow, and code injection. We leave that integration for future work.

VIII. Functionality Evaluation

A. Guest OS Hang Detection

1) Experimental Setup: The experiments were conducted on a guest VM with two vCPUs and 1024MiB of RAM. For the guest OS, we used the default build of SUSE Enterprise Linux Server 11 SP1, with and without kernel preemption enabled. The profiled maximum scheduling timeslice in both cases was two seconds, and hence the hang detection threshold was set to four seconds.

2) Experimental Methodology: In order to assess the hang detection capabilities of GOSHD, we used the fault injection framework proposed in [34]. As indicated in [34], one of the common causes of system hangs is improper implementation and invocation of locking mechanisms (e.g., spinlocks, reader/writer locks) that protect access to shared data structures in the kernel. Based on those findings, the authors of [34] identified four causes of hang failures: missing spinlock releases, wrong spinlock orderings, missing unlock/lock pairs, and missing interrupt state restorations. We further extended that concept to inject transient and persistent faults. A transient fault is only activated once when the fault location is first executed. Conversely, a persistent fault is activated every time the fault location is executed. Fault injection was repeated with different types of workloads running on the guest system:

- **Hanoi Tower**: "Tower of Hanoi" recursive program.
- **make -j1**: serial compilation of libxml.
- **make -j2**: compilation of libxml with two tasks in parallel.
- **HTTP server**: serving of an HTTP load from ApacheBench, which ran on a separate machine.

The first step of a fault injection experiment is to identify the injection location(s). We chose to inject faults into core functions of the Linux kernel and into frequently used kernel modules, such as ext3, char, and block. By profiling the kernel using the above workloads, we identified 374 locations on the execution path of the kernel to inject faults.

For each fault location, we started from a clean VM and then injected a fault while running the workload. There were five possible outcomes from each injection:

- **Not Manifested**: The fault was injected, but no observable failure was detected.

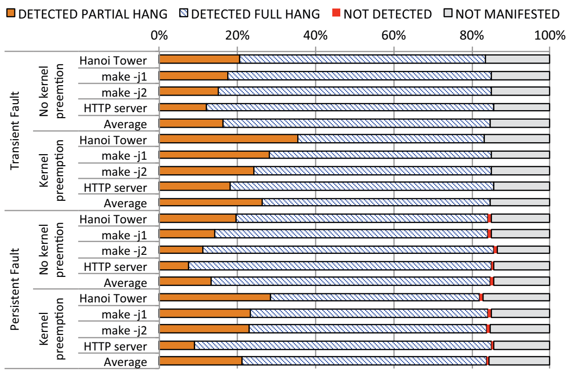

Fig. 4: Guest OS Hang Detection coverage

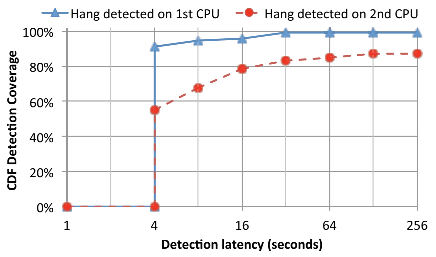

Fig. 5: Guest OS Hang Detection latency. The blue line (with triangle markers) reflects the latency of detecting the first hang of a two vCPU VM. The red dashed line (with circle markers) reflects the latency of partial hangs.

- **Not Detected**: A fault was injected, the VM was non-responsive but GOSHD did not report a vCPU hang.
- **Not Activated**: A fault was injected, but the workload did not execute the code that contained the fault.
- **Partial Hang**: At least one vCPU was still operational after 10 minutes (roughly twice the longest failure-free execution of the workloads) from the time a hang was detected on another vCPU.
- **Full Hang**: All vCPUs hung within 10 minutes after hang was detected on the first vCPU.

3) Detection Coverage Results: Fig. 4 summarizes the detection coverage and percentages of partial and full hangs detected by GOSHD. About 82% of injected faults manifested as hangs. Overall, GOSHD missed 24 failures across all experiments, which resulted in 14,720 failures (17,952 injections × 0.82 manifested faults) or *a hang detection coverage of 99.8%*.

Further analysis of the misclassified failures indicates that the failures were caused by a fault location that was repeatedly activated by the guest SSH server, which was used by our external probe to check for false alarms by GOSHD. As a result, although the SSH probe reported hangs, the kernel and other processes on the VM still executed normally.

On average, 18% to 26% of faults caused partial hangs on the non-preemptible and preemptible kernels, respectively. Those significant numbers emphasize the importance of partial hang detection. In many partial hang cases, the VM was still accessible from outside (e.g., via SSH connections). That demonstrates the ineffectiveness of hang detection methods such as heartbeats, as the process/thread responsible for generating a heartbeat can still be fully operational and will continue to report that the system is as well.

Transient faults caused slightly more partial hangs than permanent faults did in single-task workloads (Hanoi Tower and make -j1), but significantly more partial hangs in concurrent multi-tasking workloads (make -j2 and HTTP server), because persistent faults can be reactivated and cause more independent hanging threads.

Kernel preemption does not appear to help prevent a hang due to spinlocks, as most critical sections in the kernel are non-preemptible. However, preemption does reduce the number of full hangs. For example, consider two tasks T_1 and T_2 sharing a user-level lock l_u. While holding l_u, task T_1 hangs because of our injection into a kernel spinlock. Task T_1 cannot be preempted because it is executing in a non-preemptible critical section (causing a partial hang). Now let us assume that task T_2 attempts to acquire l_u. In the non-preemptible kernel, task T_2 will hang as well, thus causing a full hang. But in the preemptible kernel, task T_2 can be preempted, and therefore the kernel remains in a partial hang.

4) Detection Latency Results: Detection latency measures how quickly a detector can identify a problem. GOSHD raises an alarm when it finds that the guest OS scheduler has not scheduled processes for a predefined time. Therefore, GOSHD's minimal detection latency is that threshold (four seconds in our experiments). Specifically, detection latency represents the time between fault activation and the moment GOSHD raises an alarm. Note that the guest OS is not necessarily hung at the moment the fault is injected. Fig. 5 shows the detection latency of GOSHD for the same set of experiments described previously. Fig. 5 demonstrates how partial hang detection helps reduce full hang detection latency. The blue line (triangles) shows that GOSHD can detect more than 90% of hangs after four seconds and all hangs within 32 seconds. Meanwhile, the red line (circles) shows that only 54% of hangs result in a full hang after four seconds. Many full hangs can be detected tens of seconds ahead through the use of partial hang detection.

B. Hidden Rootkit Detection

1) HRKD Coverage: We tested HRKD on a variety of OSes and HRKD detected the presence of malware against all tested real-world rootkits.[5] On Windows, the tested rootkits included FU, HideProc, AFX, HideToolz, HE4Hook, and BH. HRKD's process counting technique showed additional processes beyond those reported by the Task Manager. On Linux, HRKD was able to discover all tested kernel-level rootkits: Ivyl's, Enyelkm 1.2, SucKIT, and PhalanX. Table II summarizes the results.

Since HRKD's process counting technique relies only on architectural invariants, it worked properly for all tested OSes,

[5]We modified some rootkits' source code so they could work properly on our tested OS versions.

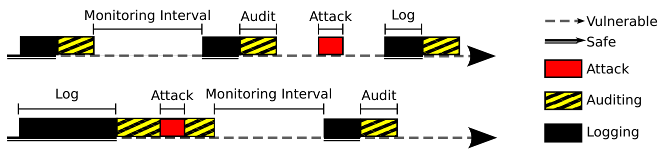

Fig. 6: *Top*: Transient attack, the attacker attacks when a passive monitor is not logging. *Bottom*: Spamming attack, the attacker causes an attack to go undetected by creating extra work for both the logger and auditor.

namely Windows XP, Vista, 7, and Server 2008, and various distributions of Linux kernel 2.6, without any adjustment. In addition, the detection capability of that technique was not affected by the implementation or strategy used by rootkits. In fact, the rootkits we evaluated employed a variety of hiding techniques, ranging from DKOM to system call hijacking (see Table II). Thus, HRKD will be able to detect future hidden rootkits, even if they use novel hiding mechanisms.

C. The Three Ninjas

1) Illustrating Attacks on Ninja: Here, we intend to use Ninja only to demonstrate the limitations of passive monitoring, and are not criticizing its checking rules. We evaluated two passive-monitoring versions of Ninja: an original in-OS version (O-Ninja) and our modified version (H-Ninja), which was implemented at the hypervisor-level using *traditional VMI*. Later on, we will compare those two implementations against our active monitoring HT-Ninja. But first, we demonstrate four attacks that can bypass passive monitoring mechanisms:

Transient attacks: We used two real privilege escalation exploits, namely a glibc vulnerability (CVE-2010-3847) [35] and a kernel out-of-bounds error (CVE-2013-1763) [36] to obtain a terminal with root privileges. Ninja can easily detect the privilege escalated terminal if it remains in the system. However, when we terminated a process right after it finished an operation (e.g., copying sensitive data), both Ninja versions were unaware of the existence of the terminal, as its lifetime was short. See the top of Fig. 6 for an illustration.

Side channel attacks: We exploited the Linux `/proc` file system as a side channel [37] to determine Ninja's monitoring interval and when each check would be performed. Specifically, the file `/proc/PID/stat` allowed us to determine the current state of a process (e.g., Sleep or Running) and the current instruction pointer (EIP). By observing Ninja's process state, we could accurately measure the monitoring inteval, i.e., how long Ninja was in the Sleep state and when the next check would be performed. With the measured interval and checking time, we could launch transient attacks that avoided detection. However, that particular strategy did not work on H-Ninja, as it does not generate a `/proc` file in the target VM. Table III shows result of a trial of this method, in which each interval was sampled 30 times.

TABLE II: Real-world rootkits evaluated with HRKD (All were detected).

Rootkit	Target OS	Hiding Technique(s)
FU	Win XP, Vista	DKOM
HideProc	Win XP, Vista	DKOM
AFX	Win XP, Vista	Hijack system calls
HideToolz	Win XP, Vista, 7	Hijack system calls
HE4Hook	Win XP	Hijack system calls
BH-Rookit-NT	Win XP, Vista	Hijack system calls
Ivyl's Rootkit	Linux >2.6.29	Hijack system calls
Enyelkm 1.2	Linux 2.6	kmem, Hijack system calls
SucKIT	Linux 2.6	kmem, DKOM
PhalanX	Linux 2.6	kmem, DKOM

TABLE III: Predicting Ninja's monitoring interval (seconds)

Ninja's Interval	Predicted Mean	Min	Max	SD
1	1.00039	1.00006	1.00367	0.00071
2	2.00023	1.99998	2.00164	0.00038
4	4.00025	3.99986	4.00189	0.00040
8	8.00025	7.99951	8.00126	0.00035

Rootkit combined attacks: In a more substantial attack, we combined a privilege escalation exploit with a rootkit, which was able to hide processes. After the terminal was escalated, we immediately ran the rootkit to prevent Ninja from discovering the presence of the terminal, bypassing both versions of Ninja.

Spamming attacks: We increased the execution time of the function that iterated over the process list by launching a large number of valid processes together with one privilege escalated process. The purpose was to increase the scanning time so that the escalated process can complete before the scan reached it. Note that a blocking H-Ninja is protected against this attack. See the bottom of Fig. 6 for an illustration.

2) Active Monitoring with HT-Ninja: To show the benefits of HyperTap's active monitoring mechanism, we compared the detectability of the three versions of Ninja (O-Ninja, H-Ninja, and HT-Ninja) against real exploits, coupled with the attack strategies described in Section VIII-C1. It is worth mentioning that both O-Ninja and H-Ninja are vulnerable to DKOM rootkits, e.g., SucKIT, because they only use OS invariants.

Our experiments showed that O-Ninja with a 0-second checking interval was quickly defeated by a privilege escalation exploit CVE-2013-1763 [36] combined with spamming and a rootkit.[6] The attack was performed as follows: (i) a number of idle processes were created; (ii) the exploit code was run to grant root privileges to the current process; and (iii) with root privileges, the rootkit was installed to remove the escalated process from the process list. We repeated the attack 300 times and have timed the attack to take $\sim$4ms on an Intel(R) Core(TM)2 Duo CPU E8400 3GHz CPU. Without creating extra processes on a system with 31 processes running, O-Ninja can detect $\sim$10% attacks. When we introduced 100 idle processes, the detection probability was reduced to single-digit percentages (between 2 and 3%). With 200 or more idle processes, O-Ninja cannot detect any attacks (with the exception of outliers $\ll 1\%$).

To test H-Ninja, we used the same privilege escalation + rootkit combination as with O-Ninja. Since the attack was

[6]https://github.com/ivyl/rootkit

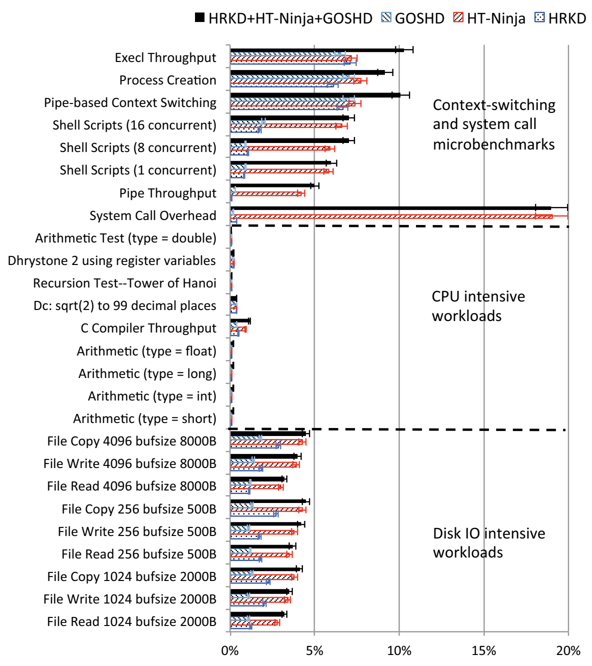

Fig. 7: Measured performance overhead of HyperTap sample monitors. The workloads are run with three different configurations: 1) Both HRKD and HT-Ninja, 2) only HT-Ninja, and 3) only HRKD. Error bars indicate one standard deviation.

quick, a small value for the checking interval was needed. With an interval of 4 ms, H-Ninja could detect 100% of the attacks, and the detection dropped to near 60% at 8 ms. With an interval $>$ 20 ms, the detection probability became $< 5\%$.

Although an attacker with no access to side channels must rely on a certain amount of luck to defeat O-Ninja and H-Ninja, his/her probability of success can be increased by spamming (O-Ninja) or by reasoning that administrators would not wish to incur the penalty of scanning the process tree with a frequency in the single-digit milliseconds. Also, the speed of the attack at 4 ms was extremely naïve; a more sophisticated attacker should be able to do better.

Since it uses active monitoring, HT-Ninja was able to detect all attacks in all tested scenarios.

A main limitation of HT-Ninja, as well as O-Ninja and H-Ninja, is that they do not detect privilege escalation attacks that occur within the context of "white listed" processes. Those processes, many of which are setuid programs, are ignored by Ninja. An attacks that compromises (e.g., using buffer overflow) and executes malicious code within the context of a white listed process would not be detected.

IX. Performance Evaluation

We conducted experiments to measured the performance overhead of individual HyperTap auditors as well as the combined overhead of running multiple auditors. We measured the runtime of the UnixBench[7] performance benchmark when (i) each auditor was enabled, and (ii) all three auditors are enabled. The target VM was a SUSE 11 Linux VM with 2 vCPUs and 1GiB of RAM. The host computer ran SUSE 11 Linux and the KVM hypervisor, with an 8 core Intel i5 3.07GHz processor and 8 GiB of RAM. The results were illustrated in Fig. 7. The baseline is the execution time when running the workloads in the VM without HyperTap integrated, and the reported numbers are the average of five runs of the workloads.

In most cases, the performance overhead of running all three auditors simultaneously was (i) only slightly higher than that of running the slowest auditor, HT-Ninja, individually, and (ii) substantially lower than the summation of the individual overheads of all auditors. That result demonstrates the benefits of HyperTap's unified logging mechanism.

For the Disk I/O and CPU intensive workloads, all three auditors together produced less than 5% and 2% performance losses, respectively. The Disk I/O intensive workloads appear to have incurred more overhead than CPU intensive workloads because they generated more VM Exit events, at which point some monitoring code was triggered.

For the context switching and system call micro-benchmarks, all three auditors together induced about 10% (or less) and 19% performance losses, respectively. It is important to note that those micro-benchmarks were designed to measure the performance of individual specific operations without any useful processing; they do not necessarily represent the performance overhead of general applications. The relatively high overhead was caused by the HyperTap routines enabled for logging those benchmarked operations. Since only HT-Ninja needs to log system calls, it was the primary source of the overhead in the system call micro-benchmark case.

X. Conclusions

This paper presents principles for unifying RnS monitoring. We identify the boundary dividing the logging and auditing phases in monitoring processes. That boundary allows us to unify and develop dependable logging mechanisms. We demonstrate the need for an isolated root of trust and active monitoring to support a wide variety of RnS monitors. We applied those principles when developing HyperTap, a framework that provides unified logging, based on hardware invariants, to safeguard VM environments. The feasibility of the framework was demonstrated through the implementation and evaluation of three monitors: Guest OS Hang Detection, Hidden RootKit Detection, and Privilege Escalation Detection. In all cases, the use of architectural invariants was central to the high quality and performance observed in the experiments. We presented additional analysis of the method so that other reliability and security monitors can be built on top of the HyperTap framework.

[7]http://code.google.com/p/byte-unixbench/

XI. Acknowledgments

This material is based upon work supported in part by the National Science Foundation under Grant No. CNS 10-18503 CISE, by the Army Research Office under Award No. W911NF-13-1-0086, by the National Security Agency (NSA) under Award No. H98230-14-C-0141, by the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement No. FA8750-11-2-0084, by an IBM faculty award, and by Infosys Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation, or other organizations.

References

[1] M. Bishop, "A model of security monitoring," in *Fifth Annual Computer Security Applications Conference*. IEEE, 1989, pp. 46–52.

[2] 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 *29th IEEE Symposium onReliable Distributed Systems*, 2010, pp. 82–91.

[3] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 233–247.

[4] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: Toward snoop-based kernel integrity monitor," in *In Proc. of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 28–37.

[5] T. R. Flo, "Ninja: Privilege escalation detection system for gnu/linux," Ubuntu Manual, http://manpages.ubuntu.com/manpages/lucid/man8/ninja.8.html, 2005.

[6] R. G. Ragel and S. Parameswaran, "IMPRES: Integrated monitoring for processor reliability and security," in *In Proc. of the 43rd Annual Design Automation Conference*, ser. DAC '06. New York, NY, USA: ACM, 2006, pp. 502–505.

[7] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu, "An architectural framework for providing reliability and security support," in *Dependable Systems and Networks, 2004 International Conference on*. IEEE, 2004, pp. 585–594.

[8] K. Pattabiraman, "Automated derivation of application-aware error and attack detectors," Ph.D. dissertation, Champaign, IL, USA, 2009, aAI3363053.

[9] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection and monitoring through VMM-based out-of-the-box semantic view reconstruction," vol. 13, no. 2. New York, NY, USA: ACM, Mar. 2010, pp. 12:1–12:28.

[10] B. D. Payne, M. de Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2007, pp. 385–397.

[11] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *In Proc. Network and Distributed Systems Security Symposium*, 2003, pp. 191–206.

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

[13] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with osck," in *In Proc. of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 279–290.

[14] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *In Proc. of the 18th USENIX Security Symposium*, 2009, pp. 383–398.

[15] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring," in *International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2009, pp. 74–81.

[16] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: mining memory accesses for introspection," in *In Proc. of the 2013 ACM SIGSAC conference on Computer &#38; communications security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 839–850.

[17] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: Tracking processes in a virtual machine environment," in *In Proc. of the USENIX Annual Technical Conference*, 2006, pp. 1–14.

[18] ——, "Vmm-based hidden process detection and identification using lycosid," in *In Proc. of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. New York, NY, USA: ACM, 2008, pp. 91–100.

[19] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *In Proc. of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 51–62.

[20] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "Spectre: A dependable introspection framework via system management mode," in *In Proc. of The 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, June 2013.

[21] D. Pelleg, M. Ben-Yehuda, R. Harper, L. Spainhower, and T. Adeshiyan, "Vigilant–out-of-band detection of failures in virtual machines," *Operating systems review*, vol. 42, no. 1, p. 26, 2008.

[22] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," pp. 121–, 1973.

[23] L. Wang, Z. Kalbarczyk, W. Gu, and R. K. Iyer, "An os-level framework for providing application-aware reliability," in *Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on*. IEEE, 2006, pp. 55–62.

[24] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 559–570, Jun. 2013.

[25] K. S. Yim, Z. T. Kalbarczyk, and R. K. Iyer, "Quantitative analysis of long-latency failures in system software," in *Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on*. IEEE, 2009, pp. 23–30.

[26] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *In Proc. of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[27] J. Butler and G. Hoglund, "Vice–catch the hookers," *Black Hat USA*, vol. 61, 2004.

[28] D. Sd, "Linux on-the-fly kernel patching without lkm," *Phrack Magazine #58, Article 7, http://www.phrack.org/issues.html?id=7&issue=58*, 2001.

[29] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *In Proc. of the Network and Distributed Systems Security Symposium*, vol. 33, 2003.

[30] N. Provos, "Improving host security with system call policies," in *In Proc. of the 12th USENIX Security Symposium*, vol. 1, no. 8. Washington, DC, 2003, p. 10.

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

[32] J. Criswell, N. Geoffray, and V. S. Adve, "Memory safety for low-level software/hardware interactions." in *USENIX Security Symposium*, 2009, pp. 83–100.

[33] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," in *In Proc. of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 351–366.

[34] D. Cotroneo, R. Natella, and S. Russo, "Assessment and improvement of hang detection in the linux operating system," in *Reliable Distributed Systems, 2009. SRDS'09. 28th IEEE International Symposium on*. IEEE, 2009, pp. 288–294.

[35] T. Ormandy, "The gnu c library dynamic linker expands $origin in setuid library search path," http://seclists.org/fulldisclosure/2010/Oct/257, 2010, [Online; accessed 29-April-2013].

[36] SecurityFocus, "Linux kernel cve-2013-1763 local privilege escalation vulnerability," http://www.securityfocus.com/bid/58137/info, 2013, [Online; accessed 29-April-2013].

[37] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *Security and Privacy (SP), 2012 IEEE Symposium on*, 2012, pp. 143–157.