Multi-layered Monitoring for Virtual Machines

Cuong Pham

Abstract This chapter describes monitoring methods to achieve both security and reliability in virtualized computer systems. We show how to perform continuous monitoring and leverage information across different layers of a virtualized computer system to detect malicious attacks and accidental failures.

1 Motivation

When a system is deployed at scale, the efficient automation of monitoring is key to achieving resilience against accidental failures and malicious attacks. This chapter specifically focuses on monitoring virtualized computer systems, which is an enabling technology of modern data centers.

Why monitoring? Computer systems fail regardless of how carefully they are constructed. A failure is either a reliability incident or a security incident. While reliability incidents are primarily caused by the increasing complexity of computer systems, security threats increase as data stored and processed by computers carry greater value.

It is a well-established design principle to treat reliability and security incidents as the norm, rather than the exception [1]. A system operates under the assumption that it can accidentally fail or be attacked at any point in time. Therefore, to produce steady and useful progress, the system needs to be monitored so that adverse incidents are detected and mitigated as quickly as possible. This is the principle that embraces high-fidelity monitoring as essential to achieve resiliency in computer systems.

Our research shares the same core proposition with this design principle: using monitoring as the main vehicle to cope with attacks and failures. We focus on the design and construction of efficient monitoring methods that can capture high-fidelity views of target systems.

C. Pham (✉)
2 Nguyen Van Tuong street, district 7, Ho Chi Minh city, Vietnam
e-mail: phammanhcuong@gmail.com

L. Wang et al. (eds.), *System Dependability and Analytics*, Springer Series in Reliability Engineering, https://doi.org/10.1007/978-3-031-02063-6_6

Why virtualized computer systems? Virtualization is the means to enable sharing and to achieve high utilization in modern data centers. In 2012, 51% of ×86 servers were virtualized, a 13% increase from 2011 [2]. In addition to virtualized servers being more prevalent than non-virtualized ones, the density of VMs on each server is also increasing [3].

The primary driving force of this trend is cloud computing, which leverages virtualization on commodity hardware as the core technology to facilitate sharing. Not unlike other types of utilities, cloud computing benefits from the economies of sharing and scaling. This is because sharing greatly decreases the cost of computing resources, which in turn attracts more users and providers to join the flow.

Given the abundance of VMs, an improvement in the security and reliability of this technology will have a large impact.

2 Target System Model

In this chapter, the target of monitoring is a virtualized system as depicted in Fig. 1. The bottom layers, including Hardware, Firmware/Bios, and Hypervisor/OS, constitute the host machine. The layers on the top, including Application and OS, constitute the virtual machines. The host machine can accommodate multiple VMs running at the same time. From user perspectives, VMs operate independently of each other.

We use the term VM monitoring to indicate any monitoring method that has the protection target (or target for short) in a layer of the virtualization software stack, including software running on a VM and the hypervisor. When the context is unclear,

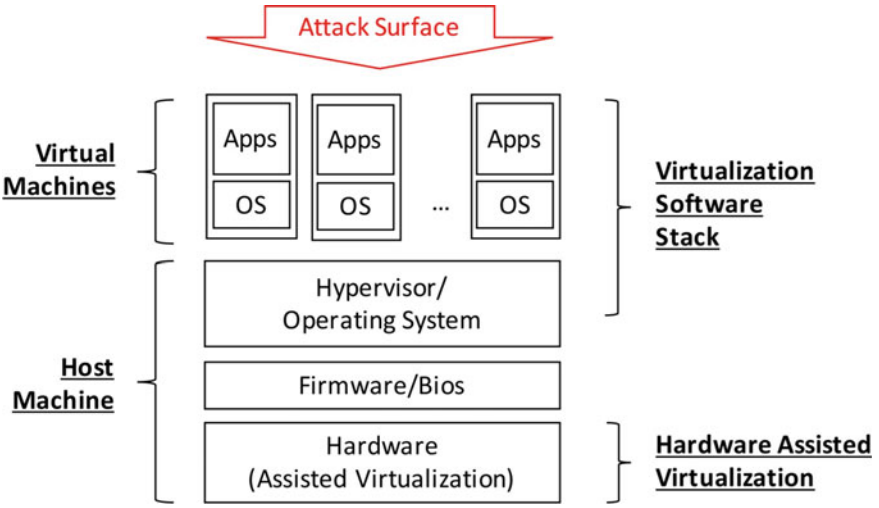

Fig. 1 A typical virtualized computer system. The virtualization software stack is the target of our monitoring

we use a more descriptive term to indicate the target of monitoring. Specifically, we use guest operating system monitoring, or guest application monitoring to indicate that monitoring targets are an operating system (OS) and applications running inside VMs, respectively. Similarly, we use hypervisor monitoring to indicate that the hypervisor is the target of monitoring. In addition, we use out-of-VM monitoring to indicate monitoring techniques deployed outside of target VMs to monitor software running inside VMs (e.g., monitoring is done from the hypervisor or from other VMs).

In the designs of our monitoring, we assume that Hardware Assisted Virtualization (HAV), such as Intel VT-x [4] or AMD-V [5]), is an integral component of the system, and is utilized by hypervisors to implement virtualization. At the moment, all server-grade $\times 86$ processors on the market support HAV. Furthermore, all popular hypervisor implementations, such as the VMWare hypervisor family, KVM [6], XEN [7], and Virtual Box, can utilize HAV to execute VMs.

With regard to security monitoring, our threat model assumes that VM share the attack surface of the target virtualized system. This assumption is derived from the model of data centers that rely on virtualization to serve users and process workloads. Infrastructure as a Service (IaaS) in cloud computing is a typical example of this model. In such a system, a user can execute arbitrary software, from user applications to their own OSs, inside VMs. Meanwhile, they do not have direct access to the host machine, except via the VM-hypervisor interface provided by HAV. Furthermore, we explicitly trust the underlying hardware. We also do not consider physical tampering and inside attackers (e.g., malicious administrators who already have remote access to the host machine).

In this threat model, we consider two broad scenarios: attacking a VM and attacking a hypervisor. The first scenario refers to attacks that aim at compromising software running inside a VM. Since in a typical data center setup most VMs must expose some remote access via the Internet to be used, they are constantly at risk of being targeted by attackers. The second scenario assumes the attacker has full access to a VM and exploits the VM-hypervisor interface to launch attacks against the underlying hypervisor (and other co-located VMs). For example, a public IaaS cloud allows any user to launch their own VMs at a very small cost. Those VMs can be used as an attack entry point to the hypervisor. Or a successful attack described in the previous scenario may grant the attacker administrative access to the exploited VM, which in turn can lead to an attack against the hypervisor.

3 Limitations of State-of-the-Art VM Monitoring

Despite significant research effort that has been invested, state-of-the-art VM monitoring techniques still experience some fundamental limitations that dwarf their practicality. Those are limitations that leave critical gaps for failures and attacks to escape detection. Here we present limitations in regard to security and reliability monitoring.

3.1 Polling-and-Scanning Monitoring Paradigm

Most VM monitoring techniques, e.g., [8–12], follow the polling-and-scanning paradigm. In this paradigm, monitoring is done by scanning the target system at a specific polling interval. This paradigm is also known as *passive monitoring* [13].

There are two major limitations of the polling-and-scanning method. First, it leaves vulnerable time gaps between consecutive polling intervals. During those temporal gaps, a transient attack, which completely removes its footprint after completing, cannot be detected. We have demonstrated in [14, 15] that transient attacks can be crafted to evade VM monitors with a high chance of success. Next, this monitoring method can only scan the static state of the target system, e.g., the state that is stored in SRAM or persistent storage. What it misses is operational data about the activities of the target system, which is necessary to enforce many security and reliability monitoring policies.

3.2 Untrustworthy Input

The goal of monitoring is to capture and present a trusted view of target systems. This view is used at a later phase in a system's operational pipeline, e.g., enforcing a security or reliability policy. Thus, the input of monitoring must be carefully selected to faithfully represent the target system. This requirement is particularly imperative in the context of security monitoring, because attackers always proactively seek opportunities like this, which let them manipulate input to falsify monitoring views.

However, many out-of-VM monitoring techniques [8–12] fail to satisfy this requirement, as they rely on untrustworthy input. These monitoring techniques exclusively rely on data structures maintained by software inside a target VM to derive views of the VM itself. It has been demonstrated that if the guest software is compromised, those data structures can be manipulated by attackers to circumvent such out-of-VM monitors [16, 17].

3.3 Inflexible Monitor Placement

Target systems and attacks are both moving targets. For example, the target system can be reconfigured or updated, or a new vulnerability or bug can be discovered. Many of these events require a corresponding update in the monitoring system. In addition, attacks are often carried out in multiple stages [18], with each stage requiring a different set of monitors to fully cover the trace of the attack.

For these reasons, monitoring systems need to be made ready for changes. Moreover, changes in a monitoring system should not be a source of downtime to target

systems. This is however not the case for existing VM monitoring techniques, which require monitoring setup and configuration as a part of the target system boot process.

3.4 Incompatible Reliability and Security Monitoring

Reliability and security tend to be treated separately because they appear orthogonal: reliability focuses on accidental failures, security on intentional attacks. Because of the apparent dissimilarity between the two, tools to detect and recover from the different classes of failures and attacks are usually designed and implemented differently. So, integrating support for reliability and security in a single framework is a significant challenge.

Current VM monitoring techniques are no exception. While there is a substantial body of VM monitoring research dedicated to security monitoring, and some work dedicated to reliability, we are not aware of any previous effort toward combining these two subjects of monitoring.

The above four identified issues in VM monitoring hinder its adoption in production systems. Our research aims at (i) raising the awareness of those issues via demonstrations of real attacks and failures, and (ii) exploring new monitoring paradigms and methods that can resolve all of the four issues.

4 HyperTap: Virtual Machine Monitoring Using Hardware Architectural Invariants

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.

In this section, 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.

4.1 Monitoring Principles

A monitoring process can be divided into two tightly coupled phases: logging and auditing [41]. 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 Reliability and security 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 reliability and security 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.

4.1.1 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 use it simultaneously. A concrete example would be to deploy both the failure detection technique proposed in [43] and the malware detection technique proposed in [44] 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.

4.1.2 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 [8–12]. 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 [16, 17, 45].

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 ×86 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 [28] and offline malware analysis [33].

We find that architectural invariants, particularly the ones defined by HAV, provide an outside view with desirable features for VM reliability and security 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 ×86 architecture.

In order to circumvent the 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 its footprint, our approach significantly impedes would-be attackers.

4.1.3 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) [46]. 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 Sect. 4.3.2, a transient attack can be combined with other techniques to create a stealthy attack that can defeat passive monitoring. Active monitoring, or event-driven 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.

4.2 Framework and Implementation

4.2.1 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.

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 addressed by the proposed monitors in hShield (Sect. 6).

4.2.2 Monitoring Workflow

Figure 2 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,

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.

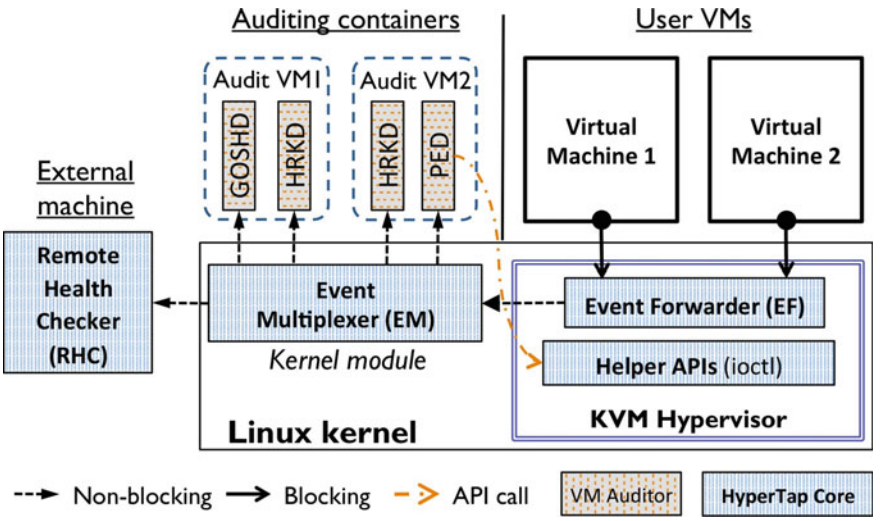

Fig. 2 Implementation of HyperTap in the KVM hypervisor. The hypervisor is modified to forward VM Exit events to the Event Multiplexer (EM), which is implemented as a separate kernel module. The EM forwards events to registered auditors running as user processes inside auditing containers. The Remote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

HyperTap delivers captured events to registered auditors, which implement specific monitoring 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 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.

4.2.3 Implementation

This subsection presents the integration of HyperTap with KVM [6], hypervisor built with HAV as a Linux kernel module. Figure 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 4 running on the host OS. Compared to the dedicated auditing VM used in previous work [11, 11], 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.

4.3 Performance Evaluation

We conducted experiments to measure the performance overhead of individual HyperTap auditors as well as the combined overhead of running multiple auditors. We measured the runtime of the UnixBench 3 performance benchmark when (i) each auditor was enabled, and (ii) all three auditors were 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 3.07 GHz processor and 8 GiB of RAM.

The results were illustrated in Fig. 3. 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

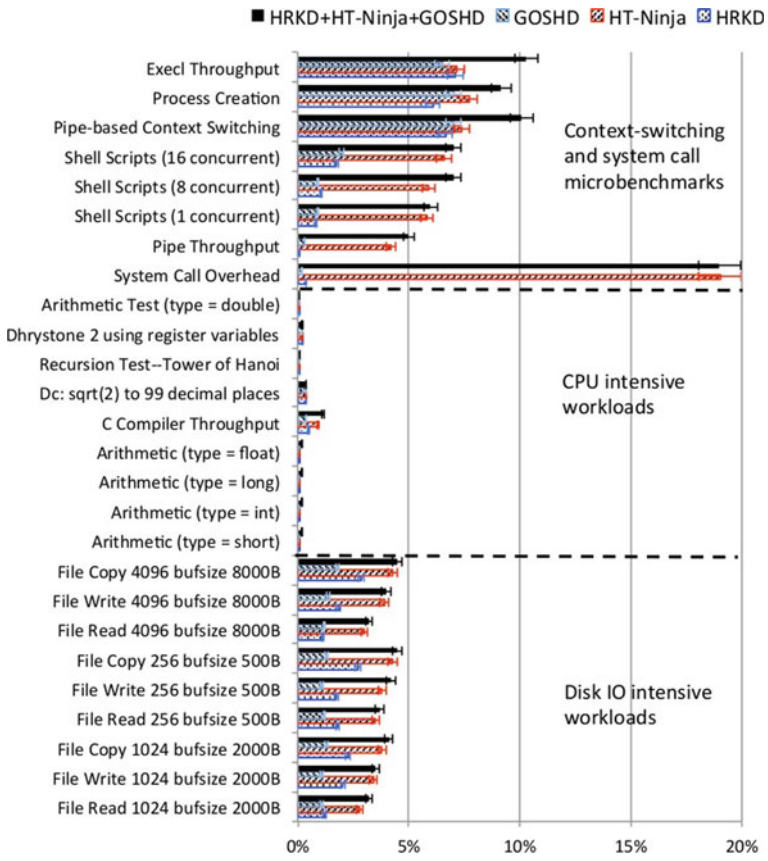

Fig. 3 Measured performance overhead of HyperTap sample monitors. The workloads are run with three different configurations: (1) both HRKD and HT-Ninja, (2) only HT-Ninja, and (3) only HRKD. Error bars indicate one standard deviation

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.

5 Hprobes: Dynamic Virtual Machine Monitoring Using Hypervisor Probes

This section introduces Hprobe, a framework that allows one to dynamically monitor applications and operating systems inside a VM. The Hprobe framework does not require any changes to the guest OS, which avoids the tight coupling of monitoring with its target.

Furthermore, the monitors can be customized and enabled/disabled while the VM is running.

5.1 Introduction

The HyperTap framework introduced in the previous chapters provides an efficient and hard-to-bypass event-driven monitoring mechanism. The key design of HyperTap is the reliance on a fixed set of hardware architectural invariants to capture guest OS's activities. While we have shown that this monitoring capability is effective to support an important set of reliability and security monitoring policies (see Chap. 4 for examples of the evaluated policies), there are still many cases in which monitors requires a more flexible means to place monitoring points, or hooks, to capture specific guest OS and applications' operational activities.

One class of active monitoring systems is a hook based system, where the monitor places hooks inside the target application or OS [13]. A hook is a mechanism used to generate an event when the target executes a particular instruction. When the target's execution reaches the hook, control is transferred to the monitoring system where it can record the event and/or inspect the system's state. Once the monitor has finished processing the event, it returns control to the target system and execution continues until the next event. Hook based techniques are robust against failures and attacks inside the target when the monitoring system is properly isolated from the target system.

We find *dynamic* hook-based systems attractive for dependability monitoring as they can be easily adapted: once the hook delivery mechanism is functional, implementing a new monitor involves adding a hook location and deciding how to process the event. In this case, dynamic refers to the ability to add and remove hooks without disrupting the control flow of the target. This is particularly important in real-world use, where monitoring needs to be configured for multiple applications and operational environments. In addition to supporting a variety of environments, monitoring must also be responsive to changes in those environments.

In this section, we present the Hprobe framework, a dynamic hook-based VM reliability and security monitoring solution. The key contributions of the Hprobe framework are that it: is loosely coupled from the target VM, can inspect both the OS and user applications, and it supports runtime insertion/removal of hooks. All of these aspects result in a VM monitoring solution that is suitable for running on an actual production system. We have built a prototype implementation using Hardware-Assisted Virtualization that is integrated with the KVM hypervisor [6]. From our experiments, the overhead for an individual probe (the time between hook invocation and when control is returned to the VM) is 2.6 μs on a modern server-class CPU. To demonstrate monitoring using the Hprobe framework, we have constructed an emergency security vulnerability detector, a heartbeat detector, and an infinite loop detector. While our prototype framework shares some similarities and builds on previous monitoring systems, these detectors could not have been implemented on

any existing platform. All of these detectors were tested using real applications and exhibit low overhead ($\leq$5).

5.2 *Design*

5.2.1 Hook-Based Monitoring

An illustration of a hook-based monitoring system adapted from the formal model presented in Lares [13] is shown in Fig. 4. Hook based monitoring involves a monitor taking control of the target after the target reaches a hook. In the case of hypervisor-based VM monitoring, the target is a virtual machine and the monitor can run in either the hypervisor [10], in a separate security VM [13], or in the same VM [30]. Regardless of the separation mechanism used, one must ensure that the monitor is resilient to tampering from within the target VM and the monitor has access to all relevant states of that VM (e.g., hardware, memory, etc.). Furthermore, a VM monitoring system should be able to trigger on the execution of any instruction, be it in the guest OS or in an application.

If a monitoring system can capture all relevant events, it also follows that the monitoring system should be dynamic. This is important in the fast-changing landscape of IT security and reliability. As new vulnerabilities and bugs are discovered, one will inevitably need to account for them.

The value of a static monitoring system decreases drastically over time unless periodic software updates are issued. However, in many VM monitoring solutions [8, 13, 14, 30], such software updates would require a hypervisor reboot or at the very least a guest OS reboot. These reboots result in system downtime whenever the monitor needs to be adapted. In many production systems, this additional downtime is unacceptable, particularly when the schedule is unpredictable (e.g., security vulnerabilities). Dynamic monitors can also provide performance improvement over

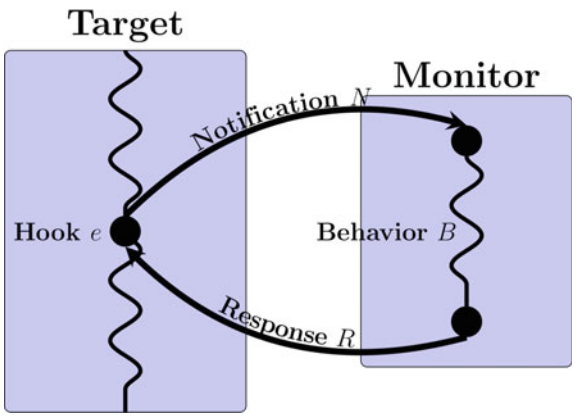

Fig. 4 Hook-based monitoring. A hook triggers based on event e and control is transferred to the monitor through notification N. The monitor processes e with a behavior B and returns control to the target with a response R

statically configured monitoring: one can monitor only events of interest vs. a general class of events (e.g., a single system call versus all system calls). Furthermore, it is possible to construct dynamic detectors that change during execution (e.g., a hook can be used to add or remove other hooks). Static monitoring systems also present a subtle design flaw: a configuration change in the monitoring system can affect the control flow of the target system (e.g., by requiring a restart).

In line with dynamism and loose coupling with the target system, the detector must also be simple in its implementation. If a system is overly complex and difficult to extend, the value of that system is drastically reduced as much effort needs to be expended to use that system. In fact, such a system will simply not be used. DNSSEC[1] and SELinux[2] can serve as instructive examples: while they provide valuable security features (e.g., authentication and access control), both of these systems were released around the year 2000 and to this day are still disabled in many environments. Furthermore, a simpler implementation should yield a smaller attack surface [58].

5.2.2 Design Principles

In light of the observation made in the previous section, we set the following design principles for a dynamic VM active monitoring system:

- **Protection**: Monitoring should be impervious to attacks (e.g., hook circumvention) inside the VM. The authors of Lares [13] outline a formal model with potential attacks and security requirements for a hook-based monitoring system. Those requirements using the notation in Fig. 5 are: the notification N should only be triggered on legitimate events, the state of the target should not change during monitoring, an attacker cannot modify the behavior B of the monitor, and the response R cannot be avoided by the target.
- **Simplicity**: The monitoring system should be simple to implement and extend. In order to ease adoption and support cloud environments, it should not require any modification of the guest OS.
- **Dynamism**: The monitoring system should be loosely coupled with the target. The target itself should be protected from changes in the monitoring system: reconfiguration can be expected to affect execution time, but it should not disrupt the control flow of the target (e.g., require a reboot or application restart). Furthermore, it should be possible to insert the hooks into both the target OS and its applications.
- **Performance**: The monitoring system should have acceptable overhead for use in a production system.

We use these requirements as a guide to design a hook-based hypervisor monitoring framework that we call hypervisor probes or hprobes. The hypervisor provides

[1] https://tools.ietf.org/html/rfc2535.

[2] https://www.nsa.gov/publicinfo/pressroom/2001/se-linux.shtml.

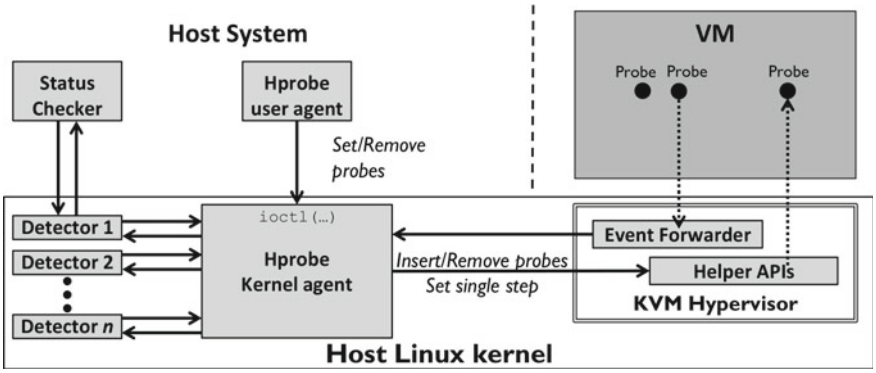

Fig. 5 Hprobes integrated with the KVM hypervisor. The Event Forwarder has been added to KVM and communicates with a separate kernel agent through Helper APIs. Detectors can either be implemented as kernel modules in the Host OS or in user space by communicating with the kernel agent through ioctl functions

a convenient interface for isolating monitoring from the VM while maintaining full access to the target VM. The proposed framework allows one to insert and remove hooks into arbitrary locations inside the guest's memory (i.e., both the guest OS and user applications) at runtime. To demonstrate the effectiveness of our framework, we built a prototype and three monitors. Two of the monitors implement reliability techniques, and the third illustrates the simplicity of using hprobes to rapidly produce a monitor that protects against a security vulnerability.

5.3 Prototype Implementation

5.3.1 Review Debugging with Software Interrupt Int3

The $\times 86$ architecture offers multiple methods for inserting breakpoints, which are used in our prototype framework. We focus on the int3 instruction as it is flexible and is not limited in the number of breakpoints that can be set. The int3 instruction is a single byte opcode ($0 \times cc$) that raises a breakpoint exception (#BP). A debugger uses OS provided functionality (e.g., a system call like ptrace() [59] in Linux) to control and inspect the process being debugged. In order to insert a breakpoint, a debugger overwrites the instruction at the desired location with int3, and then saves the original instruction. When the breakpoint is hit and the #BP exception is generated, the OS catches the exception and notifies the debugger. At this point, the debugger has control of the process and can inspect the process's memory or control its execution, e.g., by single-stepping over subsequent instructions.

5.3.2 Integration with KVM

The hprobe prototype was inspired by the Linux kernel profiling feature kprobes [60], which has been used for real-time system analysis [61]. The operating principle behind our prototype is to use VM Exits to trap the VM's execution and transfer control to monitoring functionality in the hypervisor. This implementation leverages Hardware-Assisted Virtualization (HAV), and the prototype framework is built on the KVM hypervisor [6]. The prototype's architecture is shown in Fig. 6. The modifications to KVM itself make up the Event Forwarder, which is a set of callbacks inserted into KVM's VM Exit handlers. The Event Forwarder communicates with a separate hprobe kernel agent using Helper APIs. The hprobe kernel agent is a loadable kernel module that is the workhorse of the framework. The kernel agent provides an interface to detectors for inserting and removing probes. This interface is accessible by kernel modules through a kernel API in the host OS (which is also the hypervisor since KVM itself is a kernel module) or by user programs via an ioctl interface.

The execution of an hprobe based detector is illustrated in Figs. 6 and 7. A probe is added by rewriting the instruction in memory at the target address with int3, saving the original instruction, and adding the target address to a doubly-linked list of active probes. This process happens at runtime and requires no application or guest OS restart. As explained in Sect. 5.3.1, the int3 instruction generates an exception when executed. With HAV properly configured, this exception generates a VM Exit event,

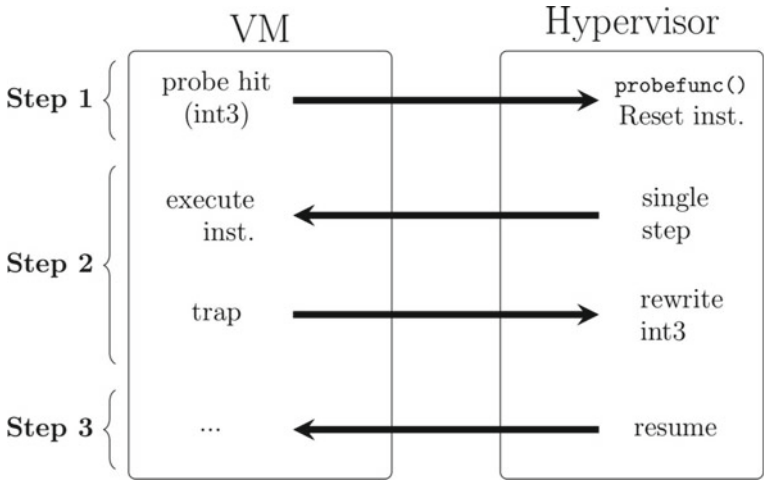

Fig. 6 A probe hit in the hprobe protoype. Right-facing arrows are VM Exits and left-facing arrows are VM Entries. When int3 is executed, the hypervisor takes control. The hypervisor optionally executes a probe handler (probefunc()) and places the CPU into single-step mode. It then executes the original instruction and does a VM Entry to resume the VM. After the guest executes the original instruction, it traps back into the hypervisor and the hypervisor will write the int3 before allowing the VM to continue as usual

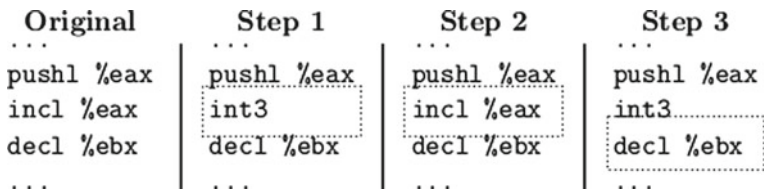

Fig. 7 Assembly pseudocode demonstrating what an hprobe looks like in the VM's memory before adding a probe (left frame) and during a probe hit (right three frames). The dashed box indicates the VM's current instruction

at which point the hypervisor intervenes (Step 1). The hypervisor uses the Event Forwarder to pass the exception to the hprobe kernel agent, which traverses the list of active probes and verifies that the int3 was generated by an hprobe. If so, the hprobe kernel agent reports the event and optionally calls an hprobe handler function that can be associated with the probe. If the exception does not belong to an hprobe (e.g., it was generated by running gdb or kprobes inside the VM), the int3 is passed back to KVM to be handled as usual. Each hprobe handler performs a user-defined monitoring function and runs in the Host OS. When the handler returns (a deferred work mechanism can also be used to support non-blocking probes, if desired), the hypervisor replaces the int3 instruction with the original opcode and puts the CPU in single-step mode. Once the original instruction executes, a single-step (#DB) exception is generated, causing another VM Exit event [4] (Step 2). At this point, the hprobe kernel agent rewrites the int3, performs a VM Entry, and the VM resumes its execution (Step 3). This single-step and instruction rewrite process ensures that the probe is always caught. If one wishes to protect the probes from being overwritten by the guest, the page containing the probe can be write-protected. Although this prototype was implemented using KVM, the concept will extend to any hypervisor that can trap on similar exceptions. Note that instead of int3, we could use any other instruction that generates VM Exits (e.g., hypercall, illegal instruction, etc.). We chose int3 since it is well supported and has a single-byte opcode.

5.3.3 Building Detectors

As mentioned in the previous section, hprobes can be controlled via an ioctl interface or a kernel API. Both interfaces distinguish between probes that are inserted into guest kernel space and guest user space. That is because while the OS always maps the kernel space pages at the same address for all virtual address spaces, each user program has its own set of pages. User space probes require the Page Directory Base Address (from the CR3 register on $\times 86$) to translate a guest virtual address into a guest physical address. Once we know the guest physical address, we can overwrite the instruction at that address and insert probes into the address space of a particular process. However, the mapping of an OS-level construct like a running process to hardware paging structures is not readily available from the hypervisor

due to the semantic gap between the VM and the hypervisor. Therefore, we use libVMI to obtain the value of the CR3 register corresponding to the target process's virtual address space [62]. This allows us to translate the virtual address of a probe location (which can be obtained from dynamic/static analysis, or by inspecting the application's symbol table) to a guest physical address that can be used to add a probe.

If one wishes to insert a probe into a user application, however, there exists another challenge. Unlike the guest OS, the pages of a running application's code may not be resident in memory at all times. That is, during an application's lifetime, some of its code may reside on disk. When execution reaches a page that is not resident, the OS will bring that page into memory. This means that the hypervisor may not be able to insert probes directly into all locations of the program at all times (i.e., it would have to wait for the OS to bring certain pages into memory). This situation arises particularly during application startup. In this case, the OS uses a demand paging mechanism in which the pages belonging to the application reside on disk until the application attempts to access one of those pages. Therefore, if the page containing the target location for a probe has not yet been accessed, a translation for guest physical address to guest virtual address will not exist. In order to support probes for user programs, this situation must be resolved so that the hprobe framework can guarantee that once a probe has been added through the APIs, it will get called on the next invocation of the instruction at the probe's desired location.

One approach to solving the problem of having target code paged out is to wait until the OS naturally brings the necessary page into memory. As mentioned in Sect. 2.2, recent versions of $\times 86$ Hardware Assisted Virtualization (HAV) use two-dimensional page tables, and do not require VM Exits for all page table updates. Therefore, in order to trap a page table update when using EPT, one must remove access permissions from EPT entries to induce an EPT VIOLATION VM Exit event. In this case, we remove write permissions from the guest physical page corresponding to the guest page table entry that refers to the guest virtual page for the intended probe location. We remind the reader that in this case the page itself is not yet present in the guest OS, and therefore a translation from guest virtual address to guest physical address does not exist in the guest OS paging structures. When an EPT violation corresponding to our protected guest page table entry occurs (indicating that the page containing the probe location is now in memory), we put the CPU into single-step mode. After the instruction writing to the guest page table executes, we can insert the probe by performing the usual translations and traversing the guest paging structures. This process of using page protection to insert probes into non-resident locations is described in Fig. 8. Note that we could improve performance slightly by avoiding the single-step and decoding the trapped instruction that caused the EPT VIOLATION. In practice, however, this paged-out situation only occurs once during the lifetime of the program (unless a page is swapped out, in which case disk latency would dominate VM Exit latency) and the performance gain would be negligible.

Oftentimes when monitoring, it is necessary to not only be aware of events in the VM (e.g., an instruction at a particular address was executed), but also the state of the VM (e.g., registers, flags, etc.). When inserting an hprobe from within the hypervisor

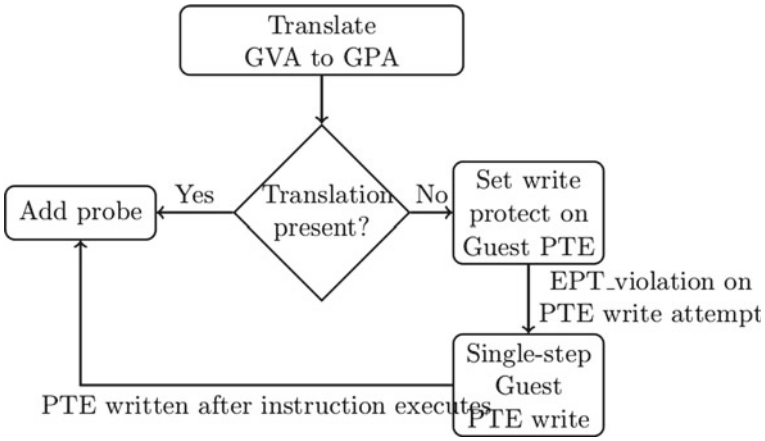

Fig. 8 How a user space probe is added. A guest virtual address (GVA) for the probe's location must be translated into a guest physical address (GPA). If the translation fails because the page is not present, we write protect the EPT page containing the guest page table entry (PTE) for that GVA. When the guest OS attempts to update the guest page table, the hprobe kernel agent is notified via an EPT violation and sets single step mode. After the single-step, the translation succeeds, and the probe is added

(i.e., using a kernel module in the Host OS), the hprobe kernel agent passes a pointer to a structure containing vCPU state to the hprobe handler. These privileged probe handlers can use this structure to decode additional information or possibly modify the state of the VM to mitigate a failure or vulnerability.

5.3.4 Discussion

Our use of int3 to generate an exception utilizes hardware enforcement of event generation: there is no dependence on any functionality inside the guest OS. This allows the hprobe hooking mechanism to be used on any guest OS supported by the hypervisor. Since the majority of the work is done outside of the hypervisor modifications (i.e., all of the heavy lifting is done inside of the kernel agent), the system can be ported to other hypervisors that support trapping on int3.

When reflecting on the requirements set forth in Sect. 5.2, we observe that the hprobe framework satisfies those requirements:

- **Protection**: By using an out-of-VM approach that is enforced by HAV, our hooks cannot be circumvented. Furthermore, we can use memory protection in the hypervisor to prevent probes from being modified (or hide them by read protecting them).
- **Simplicity**: Modifications to introduce the Event Forwarder and Helper APIs to KVM add only 117 source-lines-of-code (SLOC) and the kernel agent is 703

SLOC. The simple API allows monitors to be developed quickly and most detectors can be based on a common template (e.g., build one detector by reusing a majority of the code from a previous one). As an anecdotal example, most of the example detectors presented in Sect. 5.4 required only two hours of programming to be fully functional. Hprobes can be used on an unmodified guest OS.

- **Dynamism**: Our API allows for the insertion and removal of probes at runtime without disrupting the control flow of the target VM. Furthermore, unique to hook-based VM monitoring systems, we support application level monitoring through user space probes.
- **Performance**: While we require multiple VM Exits, we find that for our test applications and use cases, the performance is acceptable and worth the value added in the previous two dimensions.

This prototype satisfies the protection requirements adapted from Lares [13] in Sect. 5.2.2. The notification N is only delivered if events occur legitimately (spurious int3 s are ignored by the kernel agent). The context information of the event (the VM's state at event e) cannot be modified during hprobe processing since the hypervisor is in control. The security application (e.g., a probefunc()) runs inside the hypervisor and therefore, its behavior B cannot be altered by the VM. Additionally, the effects of any response R from the hypervisor are enforced since the hypervisor has full control over the target VM. Since hprobes configure VM Exits to occur on int3, one could imagine a Denial-of-Service (DOS) attack based on causing VM Exits using spurious int3 instructions. We note that hprobes do not present a new DOS threat and that if an attacker were interested in such an attack, he or she can perform it using existing functionality (e.g., using the vmcall instruction).

While using the hprobe framework does require modifications to the hypervisor, these modifications are small and robust across multiple versions of KVM and the Linux kernel. During the course of this project, we used the diff-match-patch libraries 3 to migrate the Event Forwarder and Helper APIs between KVM versions. We have tested hprobes on OpenSUSE 11.2, CENTOS7, Gentoo with kernel version 3.18.7, Ubuntu 12.04, and Ubuntu 14.04. The hprobe kernel agent is written to be version agnostic (e.g., with #ifdef macros for kernel version specific constructs like unlocked_ioctl).

5.3.5 Limitations

This prototype is useful for a large class of monitoring use cases, however it does have a few limitations. Namely,

- Hprobes only trigger on instruction execution. If one is interested in monitoring data access events (e.g., trigger every time a particular address is read from/written to), hprobes do not provide a clean way to do so. One would need to place a probe at every instruction that modifies the data (potentially every instruction that modifies any data if addresses are affected by user input). More cleanly, one could use an hprobe at the beginning and end of a critical section to turn on and

off page protection for data relevant to that critical section, capturing the events in a manner similar to livewire [8], but with the flexibility of hprobes. We are considering this in future work.
- Hprobes leverage VM Exits, resulting in non-optimal performance. This tradeoff is worth the simpler, more robust implementation with its trust rooted in HAV.
- Probes cannot be fully hidden from the VM. Even with clever EPT tricks to hide the existence of a probe when reading from its location, a timing side channel would still exist since an attacker could observe that the probed instruction takes longer than expected to complete.

6 hShield: Monitoring Hypervisor Integrity

6.1 Introduction

HyperTap and HProbes, introduced in the previous chapters, rely on the trustworthy of the underlying hypervisor to deploy their monitoring mechanisms. In this chapter, we turn the table around and validate this assumption. Particularly, we investigated VM-escape attacks, which are attacks that compromise hypervisor executions via the VM-hypervisor interface provided by Hardware Assisted Virtualization (HAV). Based on the analysis of this threat model, we introduce a new monitoring technique that detects VM-escape attacks.

In a virtualized system, the hypervisor is a single-point-of-failure. It is the centralized component that manages interactions between VMs and the underlying physical resources, such as computing, networking, and storage. Most components in hypervisor are granted high-privilege to permit access to the shared resources. If one of those components is compromised, the entire virtualized system, including physical resources and other co-located VMs, is potentially compromised as well. When an attack works on one instance of hypervisor, the attack might be extended to affect other instances, which have the same version as the exploited hypervisor.

In order to detect VM-escape attacks, we introduce a monitoring framework called hShield. The core of hShield is the incorporation of an efficient Control-Flow Integrity (CFI) enforcement method, which is specifically designed based on our analysis of HAV-based hypervisors. In addition, our CFI method addresses two fundamental limitations of state-of-the-art CFI techniques [76, 77], namely imprecise Control-Flow Graph (CFG) construction and the overhead of runtime CFI enforcement.

The design of hShield aims to provide the following features to hypervisor security monitoring:

- **Resistance to VM escape attacks that subvirt the control-flow of the hypervisor**. Many of the attacks in this class can be classified into a zero-day attack—attackers exploit an undiscovered vulnerability in the implementation of a hypervisor, which allows them to execute malicious codes together with the normal execution of the hypervisor. hShield aims at detecting this class of attacks when they are being executed without knowing the vulnerability in advance.
- **Negligible performance penalty in attack-free executions**. Similar to HyperTap and HProbes, hShield employs the principle of event-driven monitoring, which is effective in detecting both transient and persistent attacks. Additionally, we analyzed the hypervisor execution model to extract events that hShield can efficiently monitor without incurring noticeable performance overhead when the system is in an attack-free state.

In order to evaluate hShield, we compared the result of our CFI technique with that of BinCFI [77], a state-of-the-art CFI implementation. Our experiments show that the CFG constructed using our method is more precise, thus, more secure in terms of CFI enforcement. More specifically, we showed that the approximation of BinCFI's static analysis leaves dangerous paths in CFGs that can be exploited by attacks to perform a VM-escape. In addition, we showed that hShield can detect a real VM-escape attack that we crafted from a published vulnerability.

6.2 Assumptions and Threat Model

6.2.1 Assumptions

Our design targets at hypervisors that utilize Hardware Virtualization (e.g., Intel VT-x and AMD SVM) to manage VMs' executions. We make the following assumptions about the system.

The underlying hardware virtualization is implemented correctly, meaning that the only way to change from the VM privilege into the hypervisor privilege is to going through the VM-exit interface, as described in Sect. 2.2. We do not handle attacks that exploit hardware vulnerabilities.

The target host system is secured from physical tampering (e.g., secured in a server room) and there is no insider-attacker (e.g., malicious administrators who already have remote access to the host system).

The host system itself has limited direct open access from the outside world. Preventing misuse of administrative credentials, e.g., through social engineering methods to illegally obtain an administrative credential and use it against the host system, is out of the scope of this work.

The target host system is equipped with a trusted boot technology, such as Trusted Platform Module (TPM) [78], or Intel Trusted eXecution Technology (TXT) [79], which ensures the integrity of the host system, including the hypervisor, at load-time. Note that, we focus on ensuring the integrity of the hypervisor at runtime, given the

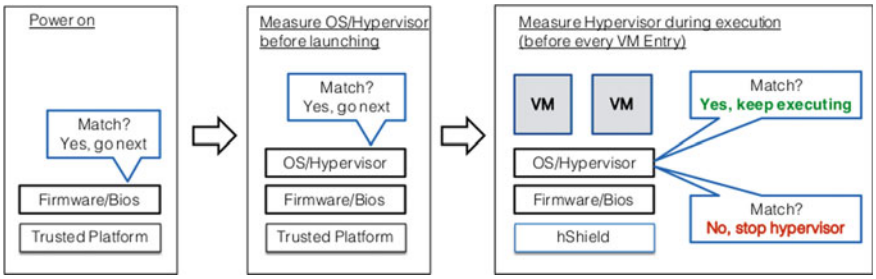

Fig. 9 hShield protects hypervisor during execution. It assumes the integrity of the platform is guaranteed at load-time by a Trusted Platform, such as TPM or Intel

integrity at load-time is guaranteed. Figure 9 shows how hShield works in tandem with trusted platform technologies.

6.2.2 Threat Model

Virtualization creates an isolated environment for each VM, so that multiple VMs can share common physical resources. The isolation is enforced so that a VM cannot access resources of the host system, or other co-located VMs.

The primary threat model that we consider is classified as VM escape attacks. A VM escape attack is an attack that breaks the isolation wall created by hypervisor to allow programs running inside a VM to violate the integrity (i.e., alter the execution) of the hypervisor. In particular, an attacker originally has full control over a VM. During the execution of the VM, the attacker is able to exploit unknown or unpatched vulnerabilities of the hypervisor software in an attempt to compromise the hypervisor. The exploit allows the attacker to redirect control flow to execute malicious code. The malicious code can be either injected by the attacker or salvaged from existing code, e.g., through a return-oriented attack. The malicious code is executed at the privilege of the hypervisor, thus it has permissions to interfere and/or access secrets stored in the hypervisor and other co-located VMs. This is a powerful class of attack. Figure 10 demonstrates the VM escape attack via VM-exit interface.

The assumption about attackers having full control over a VM is based on practical settings of virtualized computing platforms. In a public IaaS environment, such as Amazon AWS EC2, Microsoft Azure, or IBM SmartCloud, users can create a VM to run custom software with very small cost. In other virtualized environments, in which users have no direct access to a VM, attackers may gain access to a VM through exploiting vulnerabilities in the VM's software (e.g., database or web service). Once having full control over a VM, an attacker can use the VM as an entry point to start attacking the underlying hypervisor.

We further breakdown VM escape attacks into transient attacks and permanent attacks. Transient attacks are attacks that occur stealthy fast in order to bypass periodic integrity measurements [80]. Meanwhile, permanent attacks once performed

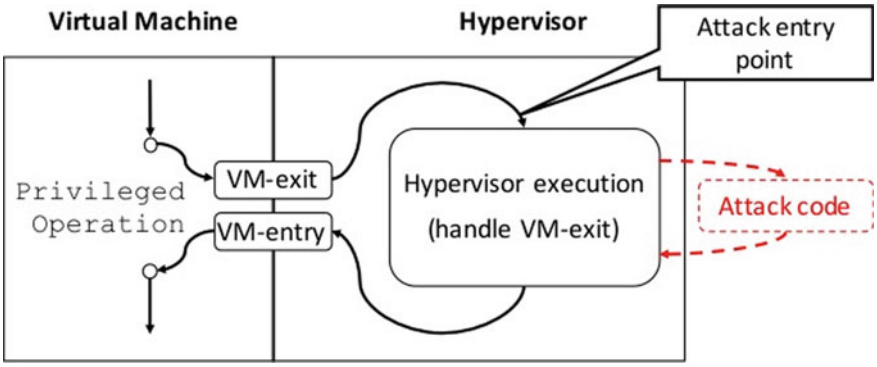

Fig. 10 Illustration of a VM escape attack in a hardware virtualization-based hypervisor. The attack entry point is the interface the hypervisor created to handle VM-exit events. The attack diverts the execution of the hypervisor (represented by the red box) from the normal execution

stay persistently in the target system. Majority of integrity measurement techniques are designed to cope with persistent attacks, leaving a gap for transient attacks to exploit [80]. Previous work [14, 15] has demonstrated the high effectiveness of transient attacks against periodic, or polling-based, monitoring. Our threat model includes both transient and permanent VM escape attacks.

6.3 hShield Approach Overview

This section describes the approach of our system, called hShield, to achieve the goals established in the previous section.

6.3.1 Limitations of Existing Control Flow Integrity Monitoring

CFI enforcement [76] is a common method used to prevent attacks relying on subverting executions of target systems (e.g., via exploiting buffer overflow vulnerabilities). In this method, valid execution paths of a program are represented as a Control-Flow Graph (CFG). The CFI runtime enforcement ensures that the target program must follow a valid path in a predetermined CFG.

A CFG is a directed graph, in which a node represents a basic block[3] in the program, and a directed edge represents a transfer in the control-flow (e.g., a jump, call, or return instruction) from a source node, where the transfer is invoked, to the target node, where the transfer lands at. Figure 9 is an example of a CFG.

[3] A basic block a consecutive sequence of instruction with no jump target except the entry and no jump source except the exit.

Runtime enforcing CFI aims at protecting target programs against unknown attacks based on the validity of CFG. A predetermined CFG is essentially a white-list of valid execution paths that are allowed to be executed. Hence, this white-list-based monitoring approach can detect attacks that divert the target program to execute an invalid path according to the determined CFG. As opposed to a black-list-based monitoring approach which can only detect previously identified attacks.

The first challenge of CFI enforcement is to obtain a precise CFG of the target program. The existing approach to CFG construction is to use static analysis [76, 77]—analyzing the source code or binary of target programs. However, static analysis cannot determine indirect control flow transfer—the control-flow targets that are computed at runtime, e.g., function pointers or return addresses. In order to address this limitation, current CFI techniques employ approximations to statically determine such dynamic targets [77].

This imprecision is a potential source for attack to by-pass CFI security runtime enforcement. For example, an attacker can use a jump-to-libc attack to invoke functions that are dynamically-incorrect, but statically-approximated.

The second challenge of CFI enforcement is to minimize the runtime overhead caused by runtime validation. The approach used by state-of-the-art CFI techniques is to perform target validation, e.g., validate whether the current jump follows a valid edge in the CFG, at the end of every basic block. The main challenge of this approach is to keep the performance overhead of the validation small due to the high frequency of basic block jumps.

6.3.2 hShield CFG Construction

hShield addresses the approximated CFG issue mentioned above by combining static analysis and profiling to construct a CFG. More specifically, we use static analysis to construct an initial CFG, which contains basic blocks (nodes in the CFG) and direct jumps (edges in the CFG), extracted from the target program binary. To derive indirect control flow information, we analyze the profiled traces of the target program execution under a set of representative workloads.

A trace records sequences of basic blocks visited during the execution of the target program. The order of basic blocks in a trace can be used to construct a CFG. For instance, two consecutive basic blocks B1 and B2 in a trace indicates that there is an edge from node B1 to node B2 in the CFG. A CFG constructed based on profiled traces contains both direct and indirect control flow information. However, the constructed CFG may not cover all possible valid paths that the target program may execute. The path coverage of the CFG is determined by the workloads used to execute and record the traces of the target program. All the collected traces are used to construct a CFG.

The initial CFG constructed using static analysis is merged with the CFG constructed based on profiled execution traces to produce a single CFG. That CFG contains both direct and indirect control flow information. This approach combines the advantages of both methods: static analysis can extract direct control flows,

and execution traces contain indirect control flows which can only be accurately determined at runtime.

For the purpose of detecting VM escape attacks, the constructed CFG of a hypervisor needs to cover all valid execution paths from a VM Exit to the corresponding VM Entry. According to our threat model, this is the only attack vector that an attacker inside a VM can penetrate the hypervisor.

Figures 11 and 12 show the result of the CFG construction for the KVM-QEMU hypervisor. Figure 11 indicates that IO INSTRUCTIONs are the most frequent type of VM Exits: 82% of VM Exits triggered during the execution of a VM under CentOS booting and the set of utilities in the UnixBench benchmark are IO-related events.

Figure 12 shows the detailed CFG construction results for QEMU using various types of workloads. In a KVM-QEMU hypervisor, all IO-related VM Exits are handled by QEMU, thus the collected events presented in the graph are IO-related events. The CFG was incrementally constructed using the traces collected by executing the workloads in order listed in the x-axis.

Each of the workloads was run three times. The graph shows that neither new nodes nor edges were discovered after the PostMark benchmark, meaning that the CFG constructed by a subset of benchmarks is able to cover all paths to execute all the selected benchmarks.

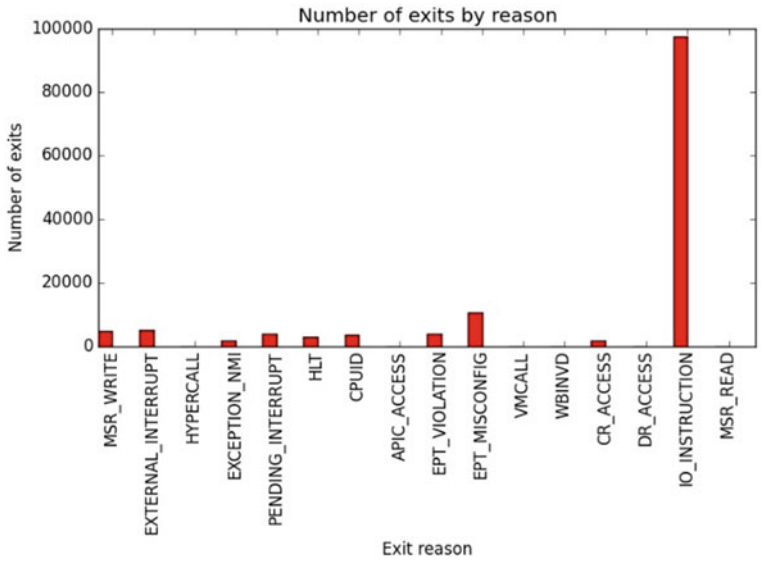

Fig. 11 The distribution of VM Exit reasons profiled during the execution of a VM under CentOS Linux booting and UnixBench workloads

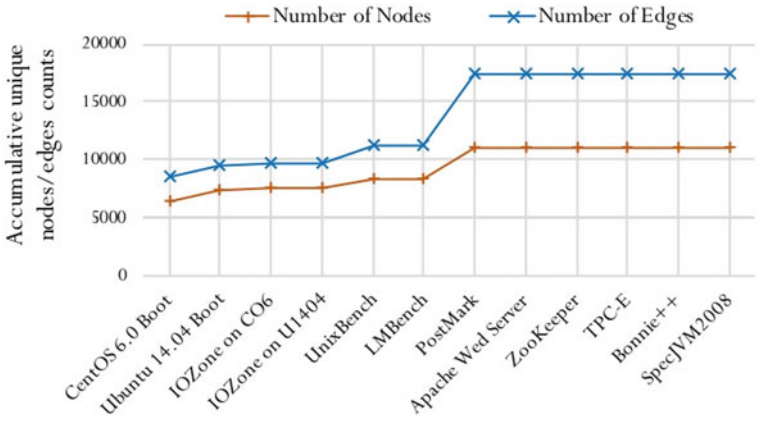

Fig. 12 Profiling QEMU (IO and MMIO exits only) under different VM workloads

6.3.3 hShield Runtime Enforcement

hShield proposes a novel technique to improve the performance overhead of CFI runtime enforcement. This technique is particularly designed for the HAV-based hypervisor execution model. Existing CFI enforcement performs validation at every control flow transfer. This validation is the major source of performance degradation occurring while executing protected programs. hShield's solution to this issue is to reduce the validation frequency by delaying it until a VM Entry is about to execute. Per our measurement, on average the frequency of executing a VM Entry is three orders of magnitude smaller than the frequency of a control flow transfer in the KVM-QEMU hypervisor.

hShield implements a hardware counter to compute a hashed value of hypervisor execution on-the-fly. Figure 13 describes how a hash is computed for each VM Exit handling. At the end of a VM Exit handling, triggered by a VM Entry event, hShield compared the computed hash against a pre-constructed HashSet. The pre-constructed HashSet represents the constructed CFG of the hypervisor. In other

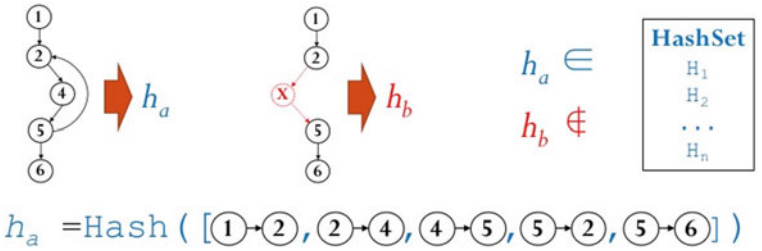

Fig. 13 HashSet construction

words, the HashSet is a white-list of valid hypervisor execution paths. If an execution path is not listed in this white-list, hShield flags it as an offended execution.

This approach of delaying CFI validation to the end of each VM Exit handling makes an important trade-off comparing the existing CFI enforcement: reducing performance overhead with the cost of longer detection latency. Since current techniques check for CFI at every control flow transfer, a CFI violation can be detected right before the execution of a malicious code. In hShield, the detection happens at the end of the violated VM Exit handling.

Section 6.4 details the hash function that hShield uses, and Sect. 6.5 describes the architectural support to hShield.

6.4 Execution Hashing

The function of execution hashing is to map an arbitrarily long execution pattern input to a fixed length output hash value. An execution pattern is a stream of machine instructions executed by the processor.

6.4.1 Requirements

The hash function needs to be collision resistant. This property is to ensure that it is computationally infeasible to find a collision—an outside execution pattern that has the same hash as one of the white-listing members. Most standard cryptographic functions, such as MD5 or the SHA family, have this property.

The hardware implementation poses several extra constraints. First, the function needs to be interactive, that is a hash can be continuously evaluated at runtime as input instructions coming, instead of storing the whole history of instructions and perform calculation at the end.

In addition, the hash function needs to facilitate the implementation of loop rerolling. hShield's loop rerolling involves frequent comparisons of basic blocks. Thus, hashing individual basic blocks should be an intermediate operation of the entire hashing scheme. Furthermore, loop rerolling requires re-evaluation of the final hashing output at runtime. For example, the hashing output changes when a loop iteration is removed. The ability to efficiently re-evaluate outputs at runtime is a necessity to enable hShield to cope with various issues, such as ones caused by hardware speculative executions. With speculative execution, a conditional branch may be predictively evaluated in advance, and unrolled and re-executed if the prediction was wrong.

6.4.2 Incremental Collision-Free Hashing

The hashing function we select is a variation of the MuHASH function in the family of incremental collision-free hashing functions proposed in [81]. The key property which makes this family of hashing functions suitable to our usage is incremental. This property allows a hash value to be updated when a portion of the input is changed without caching or re-computing the value from scratch. We leverage this feature to facilitate loop rerolling implementation and cope with speculative execution.

This family of hashing functions splits hashing into two phases: randomize and combine. Each input is broken into a sequence of blocks, and each block is randomized independently using a standard hashing function (e.g., a SHA function). The output of randomization is combined using an inexpensive commutative operation, e.g., modular multiplication in the case of MuHASH. Thanks to the communicative property of the combining operation, a hashed value can be updated by re-evaluating the randomized value of the modified input block.

Besides incrementality, MuHASH offers other properties that is suitable to hShield requirements:

- **Collision-resistance**: Based on an assumed-perfect standard hashing function (e.g., a SHA function), the security strength—the hardness of finding a collision—of the MuHASH is proven to be equivalent to the hardness of the discrete logarithm problem [81].
- **Parallel construction**: The randomization phase can be performed in parallel for each block. Note that property is stronger than interactive construction. We leverage this property to perform randomization per basic block with a small memory footprint.
- **Efficiency**: The construction uses only standard hashing function and inexpensive modular operation (as opposed to using exponentiation). The efficiency of this hashing function family is the same as using a standard hashing function on the entire input [81].

6.4.3 Runtime Construction

Essentially, the counter operates as a hash function f:

$$f : \text{Exe} \times \text{Salts} \rightarrow \text{Range}$$

The hash function f maps from the space of finite variable-length instruction streams Exe and a space of salt values $Salts$ to the space of fixed length output value $Range$.

An execution $E \in Exe$ is a finite length stream of *basic block* $B_1 B_2 ... B_n$, each basic block is a sequence of instructions $I_1 I_2 ... I_m$. Each instruction I_i is a valid $\times$ 86 instruction represented in its binary form.

A salt $salt \in Salts$ is a unique value for each system, thus it individualizes each system's counter table. A salt value is generated for a counter table when the profiling

mode is executed. Note that for a salt to be effective, it does not need to be random. Thanks to the uniqueness property of salts, the work of crafting exploit code must be redone for each every system.

A hashing session starts on a VM-exit event, and ends on the corresponding VM-entry event. The continuous construction of the hash function during a session is as follows:

- Step 1: Session starts with resetting basic block counter to i = 1:
- Step 2: For each incoming basic block B_i, concatenate a 32-bit binary encoding $<i>$ of the basic block counter, and the salt value:

$$B_i' = \langle i \rangle \cdot \langle \text{salt} \rangle \cdot B_i$$

- Step 3: (Randomization) Compute a hash value for the incoming basic block:

$$h_i = \text{shal}\left(B_i'\right)$$

- Step 4: (Combination) Combine h i using a combining operation current hash value of the execution chunk:

$$f_i = \begin{cases} h_1, & i = 1 \\ f_{i-1} \odot h_i, & i > 1 \end{cases}$$

  As recommended by [81], we use the arithmetic operation multiplication modulo for combining operator to achieve collision-resistance.
- Step 5: Continue going back to step 2 until the session is ended.

Assuming that there are n basic blocks in the evaluated execution chunk E, the final construction can be summarized in Fig. 14, and as the equation follows:

$$f(E, salt) = \odot_{i=1}^{n} \text{sha1}(\langle i \rangle \cdot \langle \text{ salt } \rangle \cdot B_i)$$

6.5 hShield Architectural Design

hShield is a security assisted hardware extension to the existing HAV to perform whitelist-based continuous monitoring of hypervisor executions. This section describes an example architectural design of hShield (Fig. 15).

6.5.1 hShield Components

Each physical host is equipped with one hShield unit. An hShield unit consists of multiple per-core hShield Counters and one per-host hShield Auditor. Each hShield

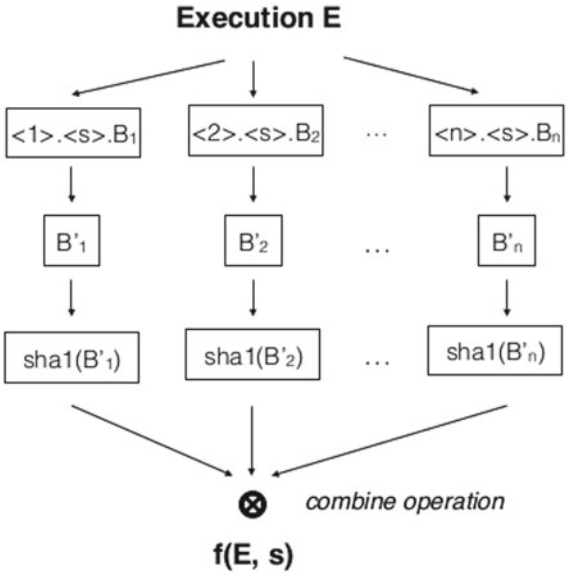

Fig. 14 The construction of the incremental hashing function

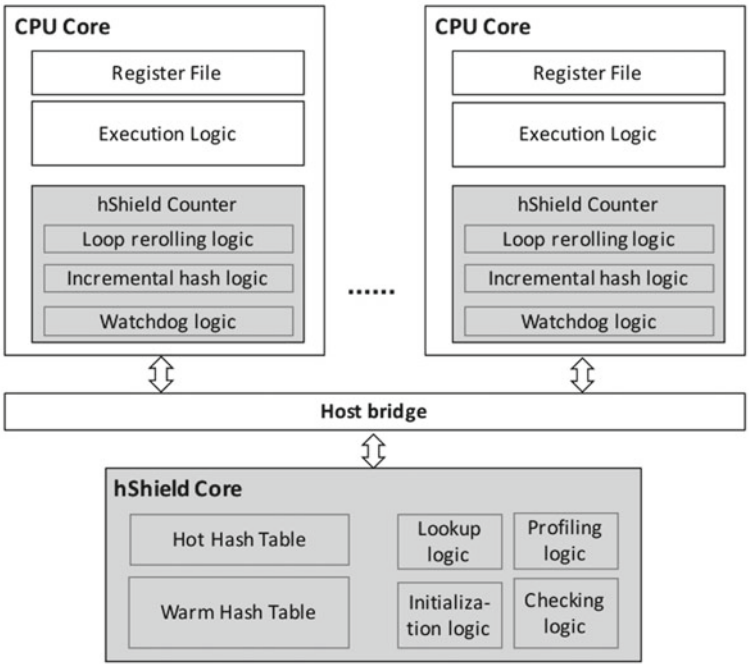

Fig. 15 hShield architecture. Each CPU core has its own hShield counter to measure hypervisor execution at runtime. After a measurement is complete, the result is sent to the hShield core, which is a dedicated core per host system, to verify the measurement.

counter is built-in into a processor core, called the counter's host core. Each counter independently carries out the measurement of VM-exit handler executing on its host core. At the end of each measurement, the result, i.e., the hash represents the VM-exit handler execution, is sent to the auditor for whitelist member checking. The hShield auditor, implemented as a dedicated co-processor in this design, is responsible for securely loading and storing the whitelist, and efficiently executing whitelist updating and membership checking. Figure 15 illustrates this architecture.

hShield is designed to facilitate both whitelist construction and runtime checking. hShield auditor has two operational modes: profiling and checking. The profiling mode is used to support whitelist construction. In this mode, the auditor records hashes sent by counters to its hash tables. Meanwhile, the checking mode is used to validate hypervisor's executions during regular runs (e.g., with arbitrary clients' VMs). In this mode, the auditor validates an execution by comparing the hash sent by a counter against in the whitelist loaded in its hash tables.

hShield architectural design follows the separation of concerns principle. After being the initialized by the centralized auditor, the operation of each counter are independent from each other, and also independent from the auditor. An hShield counter operates the same way whether the auditor is in the profiling or checking mode. The only component that stores the whitelist is the auditor. During runtime, there is only one type of unidirectional interaction between a counter and the auditor, which is sending-receiving a hash. There is no other interface that can leak information about the whitelist from the auditor to any of the processing cores.

Table 1 shows the interface of hShield Counters and hShield Auditor via the commands they process. The next subsections describe in details hShield counters and auditors.

6.5.2 hShield Counters

Figure 16 depicts the finite state machine (FSM) of an hShield counter's operation. Each node of the FSM represents an operational state of a counter, and each edge represents an event that triggers a state transition. Note that the FSM can be terminated when it is in any state, and the "End" state is not shown in the figure for readability purposes. Besides the "End" state, an hShield counter can be in one of the following operational states:

"Init": At boot time, all hShield counters are initialized by the hShield auditor. Particularly, the hShield auditor instructs each of the hShield counters to load two common salt and proof values. When the initialization is done, represented by the "Done initialization" edge, the hShield counter transits to the "Ready" state.

"Ready": When an hShield counter is in this state, the processor is executing either in the guest mode (i.e., a VM is executing), or other tasks that do not belong to the hypervisor. Upon a "VM-exit" event, the counter transits to the "Reset counter" state. Meanwhile, upon an event that indicates "Hypervisor resumed" (e.g., a task switch event that the to-be-executed task belongs to the hypervisor), the counter transits to the "Reload counter" state.

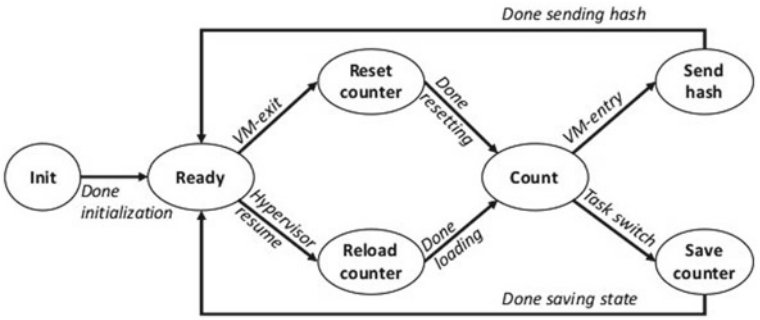

Fig. 16 Finite state machine of an hShield counter operation. A node is a state of the counter, an edge is an event that triggers a state transition. All state can transit to the "End" state, which is not shown in this figure

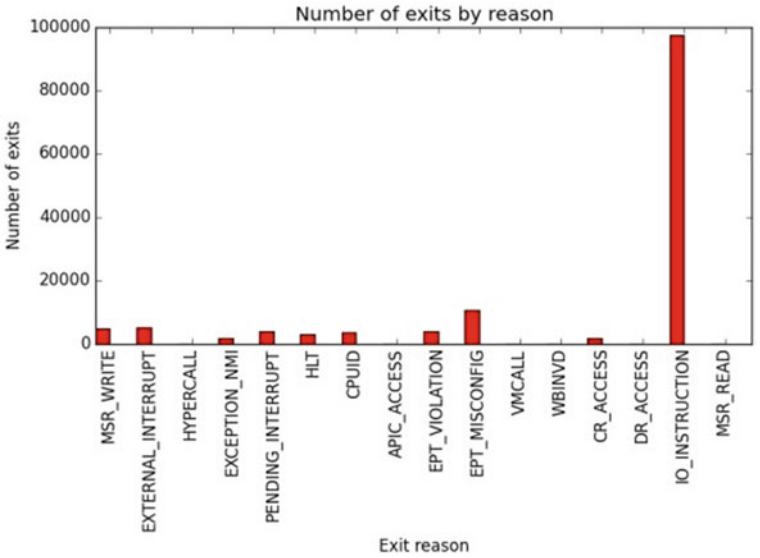

Fig. 17 The number of unique paths per each type of exit

"Reset counter": An hShield counter in this state is to respond to a VM-exit event issued by its host core. In this state, the counter resets all its internal state, e.g., the basic block counter, to get ready for a new hashing session. Upon completing the resetting, the counter transits to the "Count" state.

"Reload counter": In this state, the hShield counter loads an on-going hashing session context from memory to its internal state. The counter only loads the context which was properly signed using its hShield Proof. Upon completing the loading, the counter transits to the "Counter" state.

"Count": When an hShield counter is in this state, the host core is executing a hypervisor task that handles a VM-exit. In this state, the counter executes a hashing session, which implements the execution inference techniques and incremental hashing scheme. In the event of a task switching, the counter suspends the on-going hashing session, and then moves to the "Save counter" state. In the event of an VM-entry, which signifies the end of the on-going hashing session, the counter compute the final hash of the hypervisor execution, and then transits to the "Send Hash" state.

"Save counter": In this state, the hShield counters save the context of the on-going hashing session to main memory. The saved data is signed with the hShield Proof to prevent tampering. Upon completing the saving, the counter transits back to the "Ready" state.

"Send Hash": This state marks the end of a hashing session by sending its result to the hShield auditor. Upon completing the sending, the counter transits back to the "Ready" state.

6.5.3 hShield Auditor

An hShield Auditor is a centralized component that manages the whitelist for a host system. An hShield Auditor operates in either of the two modes: profiling and checking. Setting which mode hShield Auditor operates on is done through the BIOS.

Profiling Mode

The profiling mode is used to facilitate the construction of the target hypervisor whitelist. This mode is also considered the unsafe mode of hShield Auditor, because its whitelist can be read and updated. Thus, the profiling mode must be run in a strictly controlled environment with known-good VM workloads. In this mode, an hShield Auditor performs the following tasks.

At boot time, the following tasks are performed in a sequence:

1. Generates a new *salt* value.
2. Generates a new *proof* value.
3. Broadcasts the `HS_COUNTER_INIT` command together with the salt and proof values to all hShield Counters in the host to trigger their initialization process.

During runtime, the following tasks are performed in response to specific events:

- Upon receiving a hash from a Counter, the Auditor updates its hashing tables.
- Upon receiving a `HS_WL_COUNT` instruction, the Auditor returns the number of whitelist members.
- Upon receiving a `HS_WL_READ` instruction, the Auditor returns the hash corresponding to the specified whitelist member.

- Upon receiving a `HS_SALT_READ` instruction, the Auditor returns the value of the generated salt.

The `HS_WL_READ` and `HS_SALT_READ` instructions are used at the end of the profiling process to fetch the whitelist from the hShield Auditor to persist to the host's storage.

Checking Mode

The checking mode is used for runtime monitoring of the target hypervisor, given that the whitelist has been properly constructed. In this mode, an hShield Auditor performs the following tasks.

At boot time, after the integrity of the host system is verified, e.g., by TPM and Intel TXT, the following task are performed in a sequence:

1. Load the whitelist and salt from the host persistent storage.
2. Generates a new proof value.
3. Broadcasts the `HS_COUNTER_INIT` command together with the salt and proof values to all hShield Counters in the host to trigger their initialization process.

During runtime:

- Upon receiving a hash from a Counter, the Auditor verifies the membership of the hash.

Regardless of the hShield Auditor's operational mode, the operation of the hShield Counters in the same host is not affected: upon each VM-entry, the corresponding hShield Counter sends a hash to the centralized Auditor.

Hash Tables

Hash tables are hShield Counters internal storage to keep the whitelist. An hShield Counter contains two hash tables: hot and warm. The two tables function in the same way, except the following differences: The hot table's size is smaller than the warm table's; the hot table stores the top popular whitelist members, in the meanwhile, the warm table stores the less popular whitelist members; a membership check operation is performed in the hot table first, it there is no hit in the hot table, the operation is then performed on the warm table.

The hot-warm hash table design is to take advantage of the observed distribution of the frequency of the hit rate of whitelist members. The hot table is smaller, but stores the most frequently hit whitelist members.

7 Conclusion

This chapter proposes three new continuous monitoring methods that address both VM attack and hypervisor attack scenarios mentioned in Sect. 2. Figure 18 summarizes our contributions organized in relation to the layers in the target system (y-axis) and system operational phase (x-axis).

7.1 Continuous Monitoring of Guest OS and Applications

For monitoring software running inside VMs, we introduce HyperTap and Hprobes, which are out-of-VM monitoring frameworks that facilitate detection of security and reliability incidents occurring inside a VM. These two frameworks can work in tandem to provide desirable monitoring features. HyperTap primarily focuses on monitoring the guest OS, while Hprobes adds guest application monitoring capability. On the one hand, HyperTap relies on fixed and well-defined hardware invariants to achieve robust and strong isolation with target VMs; on the other hand, Hprobes provides a mechanism for dynamic and flexible deployment of monitoring in the target VMs.

Both HyperTap and Hprobes employ the event-driven monitoring paradigm, which allows monitors to reactively respond to events of interest. In contrast to polling-and-scanning, event-driven monitoring exposes no temporal gap for failures and attacks to exploit. In addition, the event-driven monitoring mechanisms employed by these frameworks can capture target VMs' operational activities at

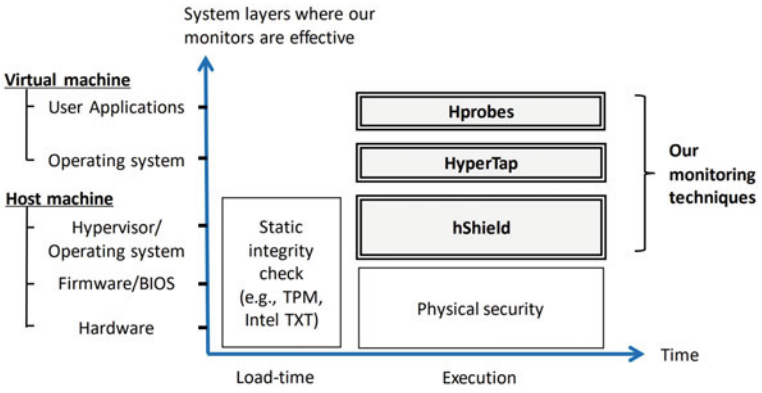

Fig. 18 An illustration of our techniques to monitor a virtualized system at runtime (e.g., during execution). The y-axis represents the system layers from hardware at the bottom to user applications in a VM at the top. The techniques are positioned at the layers where they provide monitoring: Hprobe monitors the VM's user applications, HyperTap monitors the VM's operating system, and hShield monitors the hypervisor

various granularities, e.g., system call invocations and process/task-switching events. This provides a basis of support for a broad range of security and reliability enforcement policies.

To demonstrate the capabilities of HyperTap and Hprobes in supporting security and reliability monitoring, we introduced a set of low-cost and high-coverage monitors:

HyperTap Guest OS Hang Detection (GOSHD). GOSHD detected 99.8% of injected hang failures in a guest OS. GOSHD is also able to identify partial hangs, a new failure mode in multi-processor systems.

HyperTap Hidden-Rootkit Detection (HRKD). Rootkits are malicious computer programs that hide other programs from system administrators and security-monitoring tools. HRKD guarantees discovery of hidden processes and threads regardless of their hiding techniques.

We verify the claim by testing HRKD against nine real-world rootkits in both Linux and Windows environments, with various types of hiding mechanisms.

HyperTap Privilege Escalation Detection (PED). In a privilege escalation attack, a process gains higher privileges than originally assigned to it in order to obtain unauthorized access to system resources. We demonstrate that PED can detect this class of attacks, including attacks that successfully bypassed Ninja [19], a real-world monitor, by exploiting temporal gaps created by polling-and-scanning monitoring.

Hprobes Emergency Exploit Detectors (EED). Often, a security vulnerability is discovered. After the vulnerability is made public, a patch takes time to be developed and must be put through a QA cycle. During this time, the target system is at risk of being attacked at the known vulnerability. We show that Hprobes can solve this practical problem by developing EED, a class of detectors that can prevent the exploitation of newly discovered vulnerabilities without patching the target system.

Hprobes Application Heartbeat Detector (AHD). One of the most basic reliability techniques used to monitor computing system liveness is a heartbeat detector. Using Hprobes, we constructed AHD, a monitor that directly measures the application's execution. That is, since probes are triggered by the application execution itself, they can be viewed as a mechanism for direct validation that the application is functioning correctly.

Hprobes Infinite Loop Detector (ILD). Infinite loops are a common failure that can cause process hangs. We demonstrated ILD, a monitor that uses Hprobes dynamic hook placement mechanism to measure the worst case execution time (WCET) [20] of a loop. The measure WCET is used to effectively detect infinite loops.

Table 1 hShield counter and auditor commands

Command	Callee[a]	Caller[b]	Mode[c]	Parameters	Return
`HS_COUNTER_INIT`	Counter	Auditor		()	`void`
`HS_WL_COUNT`	Auditor	Software	Profiling	()	Number of members
`HS_WL_READ`	Auditor	Software	Profiling	(`s`, `e`)	Whitelist members indexed from `s` to `e`
`HS_SALT_READ`	Auditor	Software	Profiling	()	*salt*
`HS_HASH`	Auditor	Counter	Profiling/checking	*hash*	`void`

[a]Callee is the either a Counter or Auditor, which processes the commands
[b]Caller is the component that can invoke the command. When a caller is "Software", that means this command is an instruction available for a software to use
[c]Mode is applicable for Auditor (as a callee) only. Mode specifies in which Auditor's mode ("Profiling", "Checking", or both) the command is available

7.2 Continuous Monitoring of Hypervisor

HyperTap and Hprobes rely on the trustworthiness of the underlying hypervisor to deploy their monitoring mechanisms. We demonstrate that this assumption can be violated by VM-escape attacks, which are attacks that compromise hypervisor executions via VM-exits, the VM-hypervisor interface provided by HAV. Based on the analysis of this threat model, we introduce hShield, which implements a novel Control-Flow Integrity (CFI) enforcement method to detect VM-escape attacks.

hShield continuously measures the CFI of every VM-exit handler, the basic block of hypervisor execution that handles VMs' privilege operations. The measurement is compared against a preconstructed Control-Flow Graph (CFG) to validate whether a valid path is executed. In hShield, a CFG is constructed using dynamic analysis, as opposed to the static analysis used by state-of-the-art techniques, to enhance the precision. We show that attacks can exploit the approximation of static analysis in building CFG to execute insecure paths, while our precise CFG cannot be exploited in this way.

In addition to demonstrating the strength of the constructed CFG, we show that our prototype of hShield is able to detect attacks crafted using a high-profile vulnerability in QEMU [21].

References

1. Ghemawat S, Gobioff H, Leung S-T (2003) The google file system. ACM SIGOPS Oper Syst Rev 37:29–43. ACM

2. 451 Research (2013) Theinfopro servers and virtualization study. https://451research.com/the infopro-commentator/servers-and-virtualization
3. Al Gillen, Eastwood M, Feng I, Stolarski K, Scaramella J, Chen G (2013) Worldwide virtual machine 2013–2017 forecast: virtualization buildout continues strong. IDC report
4. Intel Corporation (2014) Intel R 64 and IA-32 architectures software developer's manual volume 3 (3A, 3B & 3C): system programming guide, September 2014
5. Advanced Micro Devices Inc (2013) AMD64 architecture programmer's manual volume 2: system programming, May 2013
6. Kivity A, Kamay Y, Laor D, Lublin U, Liguori A (2007) KVM: the Linux virtual machine monitor. In: Proceedings of the Linux symposium, vol 1, pp 225–230
7. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the art of virtualization. ACM SIGOPS Oper Syst Rev 37:164–177. ACM
8. Garfinkel T, Rosenblum M (2003) A virtual machine introspection based architecture for intrusion detection. In: Proceedings of network and distributed systems security symposium, pp 191–206
9. Jiang X, Wang X, Xu D (2010) Stealthy malware detection and monitoring through VMM-based out-of-the-box semantic view reconstruction, vol 13, March 2010. ACM, New York, NY, USA, pp 12:1–12:28. https://doi.org/10.1145/1698750.1698752.
10. Payne BD, de Carbone MDP, Lee W (2007) Secure and flexible monitoring of virtual machines. In: Twenty-third annual computer security applications conference (ACSAC). IEEE, pp 385–397
11. Dolan-Gavitt B, Leek T, Zhivich M, Giffin J, Lee W (2011) Virtuoso: narrowing the semantic gap in virtual machine introspection. In: 2011 IEEE symposium on security and privacy (SP). IEEE, pp 297–312
12. Hofmann S, Dunn AM, Kim S, Roy I, Witchel E (2011) Ensuring operating system kernel integrity with osck. In: Proceedings of the sixteenth international conference on architectural support for programming languages and operating systems, ASPLOS XVI. ACM, New York, NY, USA, pp 279–290. ISBN 978-1-4503-0266-1. https://doi.org/10.1145/1950365.1950398.
13. Payne B, Carbone M, Sharif M, Lee W (2008) Lares: an architecture for secure active monitoring using virtualization. In: 2008 IEEE symposium on security and privacy (SP). IEEE, pp 233–247
14. Pham C, Estrada Z, Cao P, Kalbarczyk Z, Iyer RK (2014) Reliability and security monitoring of virtual machines using hardware architectural invariants. In: 2014 44th annual IEEE/IFIP international conference on dependable systems and networks (DSN), pp 13–24, June 2014. https://doi.org/10.1109/DSN.2014.19
15. Wang G, Estrada ZJ, Pham C, Kalbarczyk Z, Iyer RK (2015) Hypervisor introspection: a technique for evading passive virtual machine monitoring. In: 9th USENIX workshop on offensive technologies (WOOT 15), Washington, D.C., August 2015. USENIX Association. https://www.usenix.org/conference/woot15/workshop-program/presentation/wang
16. Bahram S, Jiang X, Wang Z, Grace M, Li J, Srinivasan D, Rhee J, Xu D (2010) DKSM: subverting virtual machine introspection for fun and profit. In: 29th IEEE symposium on reliable distributed systems, pp 82–91
17. Hund R, Holz T, Freiling FC (2009) Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In: Proceedings of the 18th USENIX security symposium, pp 383–398
18. Cao P, Badger E, Kalbarczyk Z, Iyer R, Slagell A (2015) Preemptive intrusion detection: theoretical framework and real-world measurements. In: Proceedings of the 2015 symposium and Bootcamp on the science of security, p 5
19. Flo TR (2005) Ninja: privilege escalation detection system for GNU/Linux. Ubuntu Manual, http://manpages.ubuntu.com/manpages/lucid/man8/ninja.8.html
20. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T et al (2008) The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans Embedded Comput Syst (TECS) 7(3):36
21. NIST (2015) Vulnerability summary for cve-2015-3456. Online. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-3456

22. Garfinkel S (1999) Architects of the information society: 35 years of the Laboratory for Computer Science at MIT. Mit Press
23. Spiceworks (2014) Start of SMB it report. Spiceworks report. http://www.spiceworks.com/marketing/state-of-smb-it
24. Bartels A, Rymer JR, Staten J, Kark K, Clark J, Whittaker D (2014) The public cloud market is now in hypergrowth: sizing the public cloud market, 2014 to 2020. Forrester report. https://www.forrester.com/The+Public+Cloud+Market+Is+Now+In+Hypergrowth/fulltext/-/E-RES113365?intcmp=blog:forrlink
25. Popek GJ, Goldberg RP (1973) Formal requirements for virtualizable third generation architectures, p 121. https://doi.org/10.1145/800009.808061
26. Bhatia N (2009) Performance evaluation of Intel ept hardware assist. VMware, Inc
27. Fu Y, Lin Z (2012) Space traveling across vm: automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In: 2012 IEEE symposium on security and privacy (SP). IEEE, pp 586–600
28. Jones ST, Arpaci-Dusseau AC, Arpaci-Dusseau RH (2006) Antfarm: tracking processes in a virtual machine environment. In: Proceedings of the USENIX annual technical conference, pp 1–14
29. Jones ST, Arpaci-Dusseau AC, Arpaci-Dusseau RH (2008) Vmm-based hidden process detection and identification using lycosid. In: Proceedings of the Fourth ACM SIGPLAN/SIGOPS international conference on virtual execution environments, VEE '08. ACM, New York, NY, USA, pp 91–100. ISBN 978-1-59593-796-4. https://doi.org/10.1145/1346256.1346269
30. Sharif MI, Lee W, Cui W, Lanzi A (2009) Secure in-vm monitoring using hardware virtualization. In: Proceedings of the 16th ACM conference on computer and communications security, CCS '09. ACM, New York, NY, USA, pp 477–487. ISBN 978-1-60558-894-0. https://doi.org/10.1145/1653662.1653720.
31. Liu Q, Weng C, Li M, Luo Y (2010) An in-vm measuring framework for increasing virtual machine security in clouds. IEEE Sec Privacy 8(6):56–62
32. Dolan-Gavitt B, Leek T, Hodosh J, Lee W (2013) Tappan zee (north) bridge: mining memory accesses for introspection. In: Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security, CCS '13. ACM, New York, NY, USA, pp 839–850. ISBN 978-1-4503-2477-9. https://doi.org/10.1145/2508859.2516697
33. Dinaburg A, Royal P, Sharif M, Lee W (2008) Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM conference on computer and communications security, CCS '08. ACM, New York, NY, USA, pp 51–62. ISBN 978-1-59593-810-7. https://doi.org/10.1145/1455770.1455779
34. Pfoh J, Schneider C, Eckert C (2011) Nitro: hardware-based system call tracing for virtual machines. In: Advances in information and computer security. Springer, pp 96–112
35. Liu Y, Xia Y, Guan H, Zang B, Chen H (2014) Concurrent and consistent virtual machine introspection with hardware transactional memory. In: 2014 IEEE 20th international symposium on high performance computer architecture (HPCA), February 2014, pp 416–427. https://doi.org/10.1109/HPCA.2014.6835951
36. Estrada ZJ, Pham C, Deng F, Yan L, Kalbarczyk Z, Iyer RK (2015) Dynamic vm dependability monitoring using hypervisor probes. In: European dependable computing conference (EDCC)
37. Petroni Jr NL, Hicks M (2007) Automated detection of persistent kernel control-flow attacks. In: Proceedings of the 14th ACM conference on computer and communications security, CCS '07. ACM, New York, NY, USA, pp 103–115. ISBN 978-1-59593-703-2. https://doi.org/10.1145/1315245.1315260
38. Nergal (2001) The advanced return-into-lib(c) exploits: Pax case study. Phrack #58, Article 4. http://www.phrack.org/issues.html?issue=58&id=4
39. Zhang F, Leach K, Sun K, Stavrou A (2013) Spectre: a dependable introspection framework via system management mode. In: Proceedings of the 43rd annual IEEE/IFIP international conference on dependable systems and networks (DSN'13), June 2013
40. Pelleg D, Ben-Yehuda M, Harper R, Spainhower L, Adeshiyan T (2008) Vigilant–out-of-band detection of failures in virtual machines. Oper Syst Rev 42(1):26

41. Bishop M (1989) A model of security monitoring. In: Fifth annual computer security applications conference. IEEE, pp 46–52
42. Moon H, Lee H, Lee J, Kim K, Paek Y, Kang BB (2012) Vigilare: toward snoop-based kernel integrity monitor. In: Proceedings of the 2012 ACM conference on computer and communications security, CCS '12. ACM, New York, NY, USA, pp 28–37. ISBN 978-1-4503-1651-4. https://doi.org/10.1145/2382196.2382202
43. Wang L, Kalbarczyk Z, Gu W, Iyer RK (2006) An os-level framework for providing application-aware reliability. In: PRDC'06. 12th Pacific Rim international symposium on dependable computing. IEEE, pp 55–62
44. Demme J, Maycock M, Schmitz J, Tang A, Waksman A, Sethumadhavan S, Stolfo S (2013) On the feasibility of online malware detection with performance counters. SIGARCH Comput Archit News 41(3):559–570. ISSN 0163-5964. https://doi.org/10.1145/2508148.2485970
45. Rhee J, Riley R, Xu D, Jiang X (2009) Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. In: International conference on availability, reliability and security (ARES). IEEE, pp 74–81
46. Yim KS, Kalbarczyk ZT, Iyer RK (2009) Quantitative analysis of long-latency failures in system software. In: PRDC'09. 15th IEEE Pacific Rim international symposium on dependable computing. IEEE, pp 23–30
47. Cotroneo D, Natella R, Russo S (2009) Assessment and improvement of hang detection in the linux operating system. In: SRDS'09. 28th IEEE international symposium on reliable distributed systems. IEEE, pp 288–294
48. Butler J, Hoglund G (2004) Vice–catch the hookers. Black Hat USA, p 61
49. Devik Sd. (2001) Linux on-the-fly kernel patching without LKM. Phrack Magazine #58, Article 7. http://www.phrack.org/issues.html?id=7&issue=58
50. Ormandy T (2010) The GNU C library dynamic linker expands $ORIGIN in setuid library search path. http://seclists.org/fulldisclosure/2010/Oct/257. [Online]. Accessed 29-April-2013
51. SecurityFocus (2013) Linux kernel cve-2013-1763 local privilege escalation vulnerability. http://www.securityfocus.com/bid/58137/info. [Online]. Accessed 29-April-2013
52. Jana S, Shmatikov V (2012) Memento: learning secrets from process footprints. In: 2012 IEEE symposium on security and privacy (SP), pp 143–157. https://doi.org/10.1109/SP.2012.19
53. Garfinkel T (2003) Traps and pitfalls: practical problems in system call interposition based security tools. In: Proceedings of the network and distributed systems security symposium, vol 33
54. Provos N (2003) Improving host security with system call policies. In: Proceedings of the 12th USENIX security symposium, vol 1. Washington, DC, p 10
55. Kosoresow AP, Hofmeyer SA (1997) Intrusion detection via system call traces. IEEE Softw 14(5):35–42
56. Criswell J, Geoffray N, Adve VS (2009) Memory safety for low-level software/hardware interactions. In: USENIX security symposium, pp 83–100
57. Criswell J, Lenharth A, Dhurjati D, Adve V (2007) Secure virtual architecture: a safe execution environment for commodity operating systems. In: Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles, SOSP '07. ACM, New York, NY, USA, pp 351–366. ISBN 978-1-59593-591-5. https://doi.org/10.1145/1294261.1294295
58. Manadhata PK, Wing JM (2011) An attack surface metric. IEEE Trans Softw Eng 37(3):371–386
59. Padala P (2002) Playing with ptrace, part 1. Linux J (103). http://www.linuxjournal.com/article/6100
60. Krishnakumar R (2005) Kernel korner: kprobes-a kernel debugger. Linux J 2005(133):11
61. Feng W, Vishwanath V, Leigh J, Gardner M (2007) High-fidelity monitoring in virtual computing environments. In: Proceedings of the international conference on the virtual computing initiative
62. Payne BD (2012) Simplifying virtual machine introspection using libvmi. Sandia report
63. NIST (2008) Vulnerability summary for cve-2008-0600. Online. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-0600

64. Corbet J (2008) vmsplice(): the making of a local root exploit. Online. http://lwn.net/Articles/268783/
65. Arnold J, Kaashoek MF (2009) Ksplice: automatic rebootless kernel updates. In: Proceedings of the 4th ACM European conference on computer systems. ACM, pp 187–198
66. Vaughan-Nichols SJ (2015) No reboot patching comes to linux 4.0. Online. http://www.zdnet.com/article/no-reboot-patching-comes-to-linux-4-0/
67. Bovet DP, Cesati M (2005) Understanding the Linux kernel. O'Reilly Media, Inc
68. Spinellis D (1994) Trace: a tool for logging operating system call transactions. ACM SIGOPS Oper Syst Rev 28(4):56–63
69. Gilbert MJ, Shumway J (2009) Probing quantum coherent states in bilayer graphene. J Comput Electron 8(2):51–59
70. Ernst MD, Perkins JH, Guo PJ, McCamant S, Pacheco C, Tschantz MS, Xiao C (2007) The daikon system for dynamic detection of likely invariants. Sci Comput Program 69(1):35–45
71. Pattabiraman K, Saggese GP, Chen D, Kalbarczyk Z, Iyer R (2011) Automated derivation of application-specific error detectors using dynamic analysis. IEEE Trans Depend Sec Comput 8(5):640–655
72. Carbin M, Misailovic S, Kling M, Rinard MC (2011) Detecting and escaping infinite loops with jolt. In: ECOOP 2011–object-oriented programming. Springer, pp 609–633
73. Agesen O, Mattson J, Rugina R, Sheldon J (2012) Software techniques for avoiding hardware virtualization exits. In: USENIX annual technical conference, pp 373–385
74. Wagner J, Kuznetsov V, Candea G, Kinder J (2015) High system-code security with low overhead. In: 36th IEEE symposium on security and privacy, number EPFL-CONF-205055
75. Larson SM, Snow CD, Shirts M et al (2022) Folding@home and genome@home: using distributed computing to tackle previously intractable problems in computational biology
76. Abadi M, Budiu M, Erlingsson U, Ligatti J. Control-flow in-tegrity principles, implementations, and applications. ACM Trans Inf Syst Secur 13(1):4:1–4:40, November 2009. ISSN 1094-9224. https://doi.org/10.1145/1609956.1609960
77. Zhang M, Sekar R (2013) Control flow integrity for cots binaries. Presented as part of the 22nd USENIX security symposium (USENIX Security 13), Washington, D.C. USENIX, pp 337–352. . ISBN 978-1-931971-03-4. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang
78. Trusted Computing Group (2015) Trusted computing group: trusted platform module. http://www.trustedcomputinggroup.org/developers/trusted_platform_module
79. Intel Corporation (2015) Trusted compute pools with intel(r) trusted execution technology. http://www.intel.com/txt
80. Azab AM, Ning P, Wang Z, Jiang Z, Zhang X, Skalsky NC (2010) Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In: Proceedings of the 17th ACM conference on computer and communications security, CCS '10. ACM, New York, NY, USA, pp 38–49. ISBN 978-1-4503-0245-6. https://doi.org/10.1145/1866307.1866313
81. Bellare M, Micciancio D (1997) A new paradigm for collision-free hashing: incrementality at reduced cost. In: Advances in cryptology—EUROCRYPT'97. Springer, pp 163–192
82. Weaver VM, Terpstra D, Moore S (2013) Non-determinism and overcount on modern hardware performance counter implementations. In: 2013 IEEE international symposium on performance analysis of systems and software (ISPASS), pp 215–224, April 2013. https://doi.org/10.1109/ISPASS.2013.6557172