Dynamic VM Dependability Monitoring Using Hypervisor Probes

Zachary J. Estrada, Cuong Pham, Fei Deng,
Zbigniew Kalbarczyk, Ravishankar K. Iyer
University of Illinois at Urbana-Champaign, USA
{zestrad2, pham9, feideng2, kalbarcz, rkiyer}@illinois.edu

Lok Yan
Air Force Research Laboratory
Rome, NY, USA
lok.yan@us.af.mil

***Abstract*—Many current VM monitoring approaches require guest OS modifications and are also unable to perform application level monitoring, reducing their value in a cloud setting. This paper introduces hprobes, 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. To demonstrate the usefulness of this framework, we present three sample detectors: an emergency detector for a security vulnerability, an application watchdog, and an infinite-loop detector. We test our detectors on real applications and demonstrate that those detectors achieve an acceptable level of performance overhead with a high degree of flexibility.**

***Keywords*—Computer Security, Reliability, Fault diagnosis, Virtual machine monitors, Platform virtualization**

I. Introduction

Failures and attacks demand a response beyond what current virtual machine (VM) monitoring solutions offer. These monitoring systems often require guest Operating System (OS) modifications, cannot be modified at runtime, and cannot monitor the execution of user programs. This lack of flexibility and high implementation complexity make many monitoring systems unsuitable for use in most environments. In this paper, we seek an approach to remedy those issues in a dynamic VM monitoring framework using hypervisor probes or "hprobes."

The primary motivation for VM monitoring is quite simple: VMs are everywhere. Whether in enterprise computing, or as the key building block in a cloud, most environments employ VMs to some extent. Furthermore, cloud providers are investigating stripped down guest OSes that provide the minimum functionality to run an application [1], [2]. As a result of reduced guest OS functionality, these "library OS" environments are heavily impaired in their ability to perform OS-level monitoring. As a hypervisor is used in these stripped down OSes, flexible hypervisor-based VM monitoring can take the place of traditional guest OS functionality.

Virtual machines provide strong isolation that can be used to enhance reliability and security monitoring [3], [4], [5], [6]. Previous VM monitoring systems require setup and configuration as part of the boot process or modification of guest OS internals. In either case, the effect on the guest is the same: a VM reboot is necessary to adapt the system. Operationally, this is undesirable for a number of reasons, e.g. due to increased downtime (discussed further in Section II). By using a dynamic monitoring system that requires no guest OS modifications or reboots, we can allow for users to respond to new threats and failure modes quickly and effectively.

Monitoring systems can generally be split into two classes: those that perform *passive* monitoring and those that perform *active* monitoring [7]. Passive monitoring systems are polling-based systems that periodically inspect the system's state. These systems are vulnerable to *transient attacks* that occur between monitoring checks [6]. Furthermore, constantly polling a system can be a source of unnecessary performance overhead. Active monitoring systems overcome these weaknesses since they are triggered only when events of interest occur. However, it is essential to ensure that an active monitoring system's event generation mechanism cannot be circumvented.

One class of active monitoring systems is a hook based system, where the monitor places hooks inside the target application or OS [8]. 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 transfered 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 paper, 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

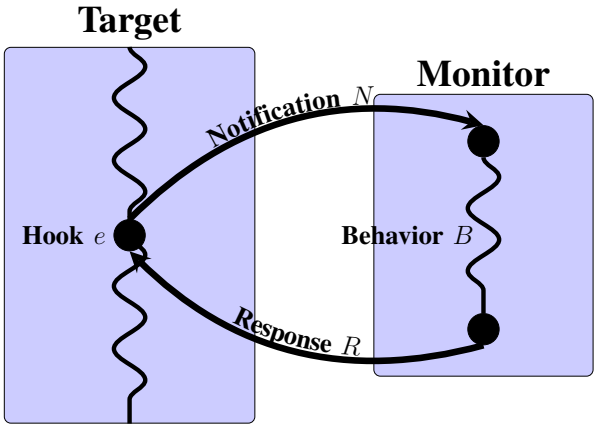

Fig. 1. Hook-based monitoring. A hook triggers based on event e and control is transferred to the monitor through notification N. The monitor processes e with a behavior B and returns control to the target with a response R.

production system. We have built a prototype implementation using Hardware-Assisted Virtualization that is integrated with the KVM hypervisor [9]. 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 (Section VII), these detectors could not have been implemented on any existing platform. All of these detectors were tested using real applications and exhibit low overhead ($\leq 5\%$).

II. Design

A. Hook-Based Monitoring

An illustration of a hook-based monitoring system adapted from the formal model presented in Lares [8] is shown in Fig. 1. Hook based monitoring involves a monitor takes 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 [8], or in the same VM [11]. 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 state 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 [3], [6], [8], [11], 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 statically configured monitoring: one can monitor only event of interest vs. a general class of events (e.g., a single system call vs. 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 [12].

B. 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:

1) **Protection:** Monitoring should be impervious to attacks (e.g., hook circumvention) inside the VM. The authors of Lares [8] outline a formal model with potential attacks and security requirements for a hook-based monitoring system. Those requirements using the notation in Fig. 1 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.
2) **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.
3) **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.
4) **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 a convenient interface for isolating monitoring from the VM while maintaining full access to the target VM. The proposed framework

[1] https://tools.ietf.org/html/rfc2535
[2] https://www.nsa.gov/public_info/press_room/2001/se-linux.shtml

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 build 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.

III. Background

This section gives the required background information necessary to understand the implementation details for the hprobe prototype presented in Section IV.

A. Debugging with `int3`

The x86 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 (`0xcc`) that raises a breakpoint exception (`#BP`). A debugger uses OS provided functionality (e.g., a system call like `ptrace()` [13] 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. More details can be found in Chapter 17 of the Intel Software Developer's Manual [14].

B. Hardware-Assisted Virtualization in x86

Extensions to the original x86 instruction set allow for a simpler and potentially more secure hypervisor when compared to software-only techniques [15]. We give a summary of Intel's Hardware-Assisted Virtualization (HAV) virtual machine extensions (VMX) to the x86 instruction set. See Chapter 23 of the Intel Software Developer's Manual [14] for a more detailed introduction to VMX (more commonly referred to as VT-x or "Vanderpool Technology").

All major Operating Systems executing on the x86 platform run in protected mode or long mode, and effectively use two privilege levels: ring 0 ("system") and ring 3 ("user"). The OS runs in ring 0 and the user applications run in the less-privileged ring 3. Similarly, Intel VT-x uses two modes of operation: *root mode* and *non-root mode*, which are layered below ring 0 and ring 3. The hypervisor executes in root mode and the virtual machines (VMs) execute in non-root mode (often referred to as guest mode). The hypervisor runs a virtual machine (VM) by executing a *VM Entry* (via the `vmlaunch` instruction). When the VM is executing in non-root mode and certain operations are performed (e.g., a privileged instruction is executed), control is transferred to the hypervisor (the CPU transitions to root mode) via a *VM Exit* event. This allows a guest mode OS to run in privilege ring 0 and is an implementation of the "trap-and-emulate" style of virtualization [16]. VT-x introduced the Virtual Machine Control Structure (VMCS) that is used to manage each virtual CPU (vCPU) of a VM. A hypercall is a request (similar to a system call in an OS) that can be issued by a VM so that the hypervisor can perform an operation on its behalf.

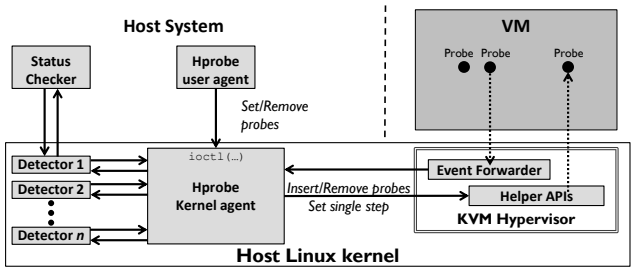

Fig. 2. Hprobes integrated with the KVM hypervisor. The Event Forwarder has been added to KVM and communicates with a separate kernel agent through Helper APIs. Detectors can either be implemented as kernel modules in the Host OS or in user space by communicating with the kernel agent through `ioctl` functions.

In addition to virtualizing the CPU, one also needs to be concerned with the virtualization of the Memory Management Unit (MMU). Virtual memory is the cornerstone of process isolation in every modern OS, and therefore is a necessary feature for VMs. Earlier implementations of x86 hypervisors used Shadow Page Tables, which are data structures that contain mappings from *guest virtual addresses* to *host physical addresses*. Shadow Page Tables use a costly VM Exit synchronization technique to match the shadow structures with the hardware page tables. To avoid this overhead, CPU vendors added a feature, *Second-Level Address Translation* (SLAT) or *Two-Dimensional Paging*, to their virtualization extensions. This technology is called Extended Page Tables (EPT) in the Intel Architecture and Nested Page Tables (NPT) in AMD. With EPT, the hardware uses a second set of page tables to translate from *guest physical addresses* to *host physical addresses*. Handling this translation in hardware eliminates the majority of VM Exits used to synchronize guest page tables. Therefore, EPT provides performance benefits in most cases, but results in very costly TLB misses as an additional set of page tables must be traversed [17]. See Chapter 28 of the Intel Software's Developer Manual [14] for more details.

While the remainder of this paper and the prototype implementation use Intel VT-x, the discussion and concepts map very well to AMD's AMD-V (see Chapter 15 of the AMD Software Developer's Manual [18]).

IV. Prototype Implementation

A. Integration with KVM

The hprobe prototype was inspired by the Linux kernel profiling feature kprobes [19], which has been used for real-time system analysis [20]. 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 [9]. The prototype's architecture is shown in Fig. 2. 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

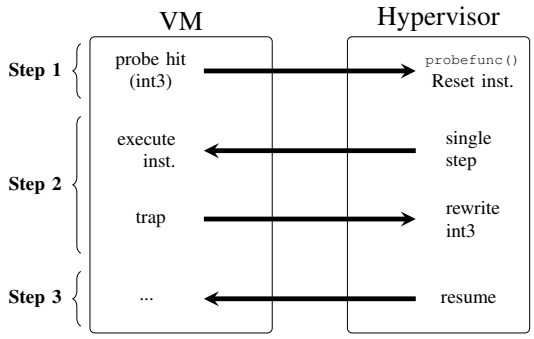

Fig. 3. A probe hit in the hprobe protoype. Right-facing arrows are VM Exits and left-facing arrows are VM Entries. When `int3` is executed, the hypervisor takes control. The hypervisor optionally executes a probe handler (`probefunc()`) and places the CPU into single-step mode. It then executes the 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.

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. 3 and Fig 4. 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 Section III-A, the `int3` instruction generates an exception when executed. With HAV properly configured, this exception generates a VM Exit event, 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 put the CPU in single-step mode. Once the original instruction executes, a single-step (`#DB`) exception is generated, causing another VM Exit event [14] (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

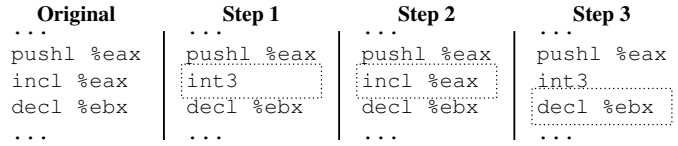

Fig. 4. Assembly pseudocode demonstrating what an hprobe looks like in the VM's memory before adding a probe (left frame) and during a probe hit (right three frames). The dashed box indicates the VM's current instruction.

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.

B. Sample API

Pseudocode for adding and removing probes using the Helper APIs in the prototype is shown below:

```
int HPROBE_add_probe(addr_info, vmid,
                     (*)probe_func(vcpu_type *VCPU));
int HPROBE_remove_probe(addr_info, vmid);
```

The `addr_info` structure contains the guest virtual address of the probe location as well as any paging information needed for user space probes (described in the next subsection), `vmid` is a unique identifier for the target VM, and `probe_func` is a function pointer to the (optional) probe handler that has an argument containing the vCPU state.

C. 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 x86) 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 [4]. 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 Section III-B, recent versions of x86 Hardware Assisted Virtualzation (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. 5. 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.

Often times 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 (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.

D. 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`.

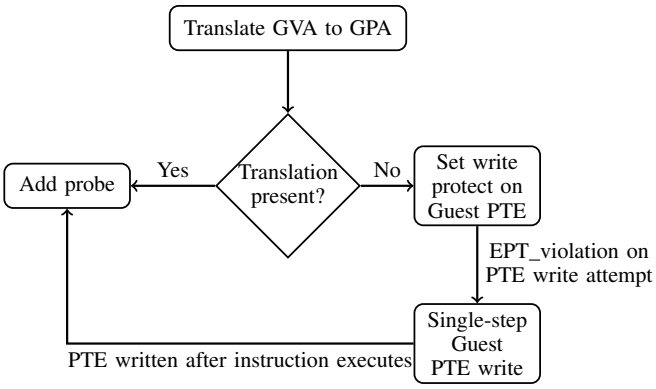

Fig. 5. How a user space probe is added. A guest virtual address (GVA) for the probe's location must be translated into a guest physical address (GPA). If the translation fails because the page is not present, we write protect the EPT page containing the guest page table entry (PTE) for that GVA. When 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.

When reflecting on the requirements set forth in Section II, we observe that the hprobe framework satisfies those requirements:

1) **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).
2) **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 Section V required only two hours of programming to be fully functional. Hprobes can be used on an unmodified guest OS.
3) **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.
4) **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. See Section VI for analysis and details.

This protoype satisfies the protection requirements adapted from Lares [8] in Section II-B. 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`).

E. Limitations

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

1) 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 [3], but with the flexibility of hprobes. We are considering this in future work.
2) Hprobes leverage VM Exits, resulting in non-optimal performance. This tradeoff is worth the simpler, more robust implementation with its trust rooted in HAV.
3) 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.

V. Detectors

In this section, we present sample reliability and security detectors built upon the hprobe prototype framework. These detectors are unique to the hprobe framework and cannot be implemented on any other current VM monitoring system.

A. Emergency Exploit Detector

Most systems operators fear zero-day vulnerabilities as there is little that can be done about them until the vendor/-maintainer of the software releases a fix. Furthermore, even after a vulnerability is made public, a patch takes time to be developed and must be put through a QA cycle. This can further be exacerbated in environments with high availability concerns and stringent change control requirements: even if a patch is available, many times it is not possible to restart the system or service until a regular maintenance window. This leaves operators with a difficult decision: risk damage from restarting a system with a new patch or risk damage from running an unpatched system.

Consider the CVE-2008-0600 vulnerability that was resulted in a local root exploit through the `vmsplice()` system call [21], [22]. This example represents a highly dangerous buffer overflow since a successful exploit allows one to arbitrarily execute code in ring 0 using a program that is publicly available on the Internet. Since this exploit involves the base kernel code (i.e., not a loadable module), patching it would require installing a new kernel followed by a system reboot (or without a reboot using techniques discussed in Section VII). As discussed earlier, in many operational cases a system reboot or OS patch can only be conducted during a predetermined maintenance window. Furthermore, many organizations would be hesitant to run a fresh kernel image on production systems without having gone through a proper testing cycle first.

The `vmsplice()` system call is used to perform a zero-copy map of user memory into a pipe. At a high level, the CVE-2008-0600 `vmsplice()` constructs specially crafted compound page structures in userspace. A compound page is a structure that allows one to treat a set of pages as a single data structure. Every compound page structure has a pointer to a destructor function that handles the cleanup of those underlying pages. The exploit works by using an integer overflow to corrupt the kernel stack such that it references the compound page structures crafted in userspace. Before calling `vmsplice()`, the exploit closes the pipe, so that when the system call runs it deallocates the pages, resulting in calling the compound pages' destructor function. The destructor is set to privilege escalation shellcode that allows an attacker to hijack the system.

The CVE-2008-0600 exploit hinges on an integer overflow in one of the system call arguments - a pointer to a `struct iovec` that contains the member `iov_len`, which is set to `ULONG_MAX` by the exploit. Since Linux uses registers to hold the system call number as well as arguments for system calls [23], we could use classical system call monitoring/-tracing tools to detect this exploit [24], [25]. We can watch whenever a system call is invoked and check for the correct system call number and parse arguments to detect an integer overflow attempt. However, since hprobes are dynamic, we can set a probe to trigger only on the `sys_vmsplice()` function (that is called after the system call assembly linkage). This ensures that only the execution path of the `vmsplice()` system call is inspected as opposed to all system calls (as in traditional system call tracing). At this point in the system call invocation, the function just uses the regular compiler function calling convention (in most instances of the Linux kernel, the gcc convention) and the arguments are on the stack. Either way, we can use hprobes to obtain these arguments. Pseudocode describing how the detector is implemented is shown in Fig. 6. Essentially, one needs to ensure that `iov_len` will not cause overflow. Depending on the environment, the

[3]https://pypi.python.org/pypi/diff-match-patch/

```
1: procedure VMSPLICE_HANDLER(vcpu)
2:     if 32-bit guest then
3:         arg_offset = 8              ▷ 2nd arg on stack 32bit
4:         max ← ULONG_MAX_32 - PAGE_SIZE
5:     else
6:         arg_offset = 16             ▷ 2nd arg on stack 64bit
7:         max ← ULONG_MAX_64 - PAGE_SIZE
8:     end if
9:     ▷ The read function checks for a valid address
10:    iov_pointer ← read_guest(vcpu.esp+arg_offset)
11:    iov_len ← read_guest_virt(iov_pointer)
12:    if iov_len ≥ max then
13:        HANDLE_EXPLOIT_ATTEMPT(vcpu)
14:    end if
15: end procedure
```

Fig. 6. Pseudocode for an hprobe based CVE-2008-0600 Detector. This handler will is executed when the `vmsplice()` system call is used. The overflow occurs when a struct member in the second argument is ULONG_MAX. The code protects against the integer overflow by ensuring that if a ULONG_MAX argument that would cause an overflow is used, the exploit is caught.

operator can choose how to handle the detected exploit. One could send an alert, simply modify `iov_len` to a benign value that causes `vmsplice()` to fail, or take a more drastic action (such as killing the process or VM) if desired.

The emergency detector works by checking the arguments of a system call for a potential integer overflow. This differs in functionality from the upstream patch,[4] which checks if the memory region (specified by the `struct iovec` argument) is accessible to the user program. One could write a probe handler that performs a similar function by checking if all of the region referred to by the `struct iovec` pointer + `iov_len` is in the appropriate range (e.g., by walking the page tables belonging to that process). However, a temporary measure to protect against an attack should be as lightweight and simple as possible to avoid unpredictable side effects. One major benefit of using an hprobe handler is that developing this detector does not require a deep understanding of the vulnerability: the developer of the emergency detector only needs to understand that there is an integer overflow in an argument. This is far simpler than developing and maintaining a patch for a core kernel function (a system call), especially when reasoning about the risk of running a home-patched kernel (a process that would void most enterprise support agreements).

Our solution uses a monitoring system that resides outside of the VM and relies on a hardware-enforced `int3` event. A would-be attacker cannot circumvent this event without having first compromised the hypervisor or having modified the guest's kernel code. This could be done with a code injection attack that causes a different `sys_vmsplice()` system call handler to be invoked. However, it is unlikely that an attacker who already has the privileges necessary for code injection into the kernel would have anything to gain by exploiting a local privilege escalation vulnerability. While this detector cannot defeat an attacker that has previously obtained root access, its ease of rapid deployment sufficiently mitigates this risk. Since no reboot is required and the detector can be used in a "read-only" monitoring mode (only reporting the attack vs. taking an action), the risk of using this detector on a running production system is minimal. To test the CVE-2008-0600 detector, we used a CENTOS5 VM (the exploit was discovered while the source-equivalent Red Hat Enterprise Linux 5.0 OS was in production) and the publicly available exploit. As an unprivileged user, we ran an exploit script on the unpatched OS and were able to obtain root access. With the monitor in place, all attempts to obtain root access using the exploit code were detected.

B. Application Heartbeat Detector

One of the most basic reliability techniques used to monitor computing system liveness is a heartbeat detector. In that class of detector, a periodic signal is sent to an external monitor to indicate that the system is functioning properly. A heartbeat serves as an illustrative example for how an hprobe-based reliability detector can be implemented. Using hprobes, we can construct a monitor that directly measures the application's execution. That is, since probes are triggered by application execution itself, they can be viewed as a mechanism for direct validation that the application is functioning correctly. Many applications execute a repetitive code block that is periodically reentered (e.g., a Monte Carlo simulation that runs with a main loop, or an http server that constantly listens for new connections). If one profiles the application, it is possible to determine a period (in units of time or using a counter like the number of instructions) at which this code block is reentered. During correct operation of the application, one can expect that the code block will be executed at the profiled interval.

The hprobe-based application heartbeat detector is built on the principle described in the previous paragraph and illustrated in Fig 7. This test detector is a kernel module that is installed in the Host OS (i.e., one of the detectors on the left side of Fig. 2). An hprobe is inserted at the start of the code block that is expected to be periodically reentered. When the hprobe is inserted, a delayed workqueue[5] is scheduled for the timeout corresponding to the reentry period for the code block. When the timeout expires, the workqueue function is executed and declares failure (if the user desires a more aggressive watchdog style detector, it is possible to have the hprobe handler perform an action such as restart the application or VM). During correct operation (i.e., when the hprobe is hit), the workqueue is canceled and a new workqueue is scheduled for the same interval, starting a new timeout period. This continues until the application finishes or the user no longer desires to monitor it and removes the hprobe. If having an hprobe hit on every iteration of the main loop is too costly, one can ensure that the probe active for an acceptable time interval and it can be added/removed until desirable performance is achieved (the detection latency would still be low as a tight loop would have a small timeout value).

We use the open-source Path Integral Quantum Monte Carlo (pi-qmc) simulator [26] as a test application.[6] This application represents a long-running scientific program that can take many hours or days to complete. As is typical with

[4] https://gitorious.org/kernel-linux/linux-stable/commit/af395d8632d0524be27d8774a1607e68bdb4dd7f

[5] http://www.makelinux.net/ldd3/chp-7-sect-6

[6] available at: http://phys-tools.github.com/pi-qmc/

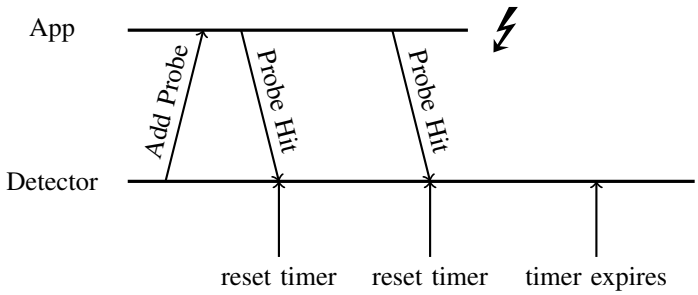

Fig. 7. Application Heartbeat Detector. A probe is inserted in a critical periodic section of the application (e.g., the main loop). During normal execution, a timer is continuously reset. In the presence of a failure (such as an I/O hang), the timer expires and failure is declared.

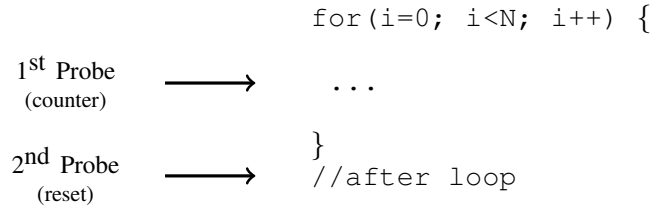

Fig. 8. Probe locations for the infinite loop detector (ILD). The ILD has two modes of operation, both utilizing the same set of probes. In the first mode, failure is declared when the loop executes more times than a set threshold. The second mode of operation tracks the register state. If more than a specified set of registers remains static for N iterations, failure is declared.

scientific computing applications, pi-qmc has a large main loop. Since Monte Carlo simulation involves repeated sampling and therefore repeated execution of the same functions, we only need to run the main loop a handful of times to determine the time per iteration. After determining the expected duration of each iteration, we set the heartbeat to timeout to the twice the expected value, set the detector to a statement at the end of the main loop, and injected hangs (e.g., SIGSTOP) and crashed the application (e.g., SIGKILL). All crashes (including VM crashes since the timer executes in the hypervisor) were detected.

C. Infinite Loop Detector

Infinite loops are a common failure that can cause process hangs. When considering proper execution of a loop in a program (that is not the main loop), the number of instructions executed in a given block of code usually falls into a fixed range, with the upper bound being the worst case execution time (WCET) [27]. Determining the WCET is a well studied problem in real-time systems, and solving it is beyond the scope of our work. Similarly, if one can identify a block of code or function that is executed repeatedly, the number of times that block is executed before the end is reached should also fall into a fixed range. One can use an automated system to infer loop invariants and bound the number of times the loop should execute [28].

Given a block of code and the WCET (either in units of time or the number of executions of that block of code), one can build a detector using a pair of hprobes. When one knows the wall clock time, one can insert one probe at the inside the block and another probe after the block. At the first probe, a timer is started (using the same technique as the heartbeat detector in Section V-B). If the timer expires before the second probe (at the end) is reached, the detector reports a failure. If there is concern that the hypervisor or guest OS is over-provisioned and significant time sharing is taking place, one can use architectural invariants [5], [6] to only count the time when the application under consideration is being executed by monitoring context switch events using the CR3 register. For the case where a bound on the number of executions of the block of code is known, one can place one probe at the beginning of the loop and one immediately after the loop. If the probe inside the loop is executed more times than expected without the block being exited, then the detector can report failure (i.e., a range violation [29]). Depending on the needs of the user, the detector can either reset its state or remove itself when the exit probe is hit.

In addition to using the WCET, one can also observe the state of the system to detect an infinite loop [30]. When using KVM, the register state of the VM is saved in KVM data structures to be reloaded upon the next VM entry. As mentioned in Section IV-C, probes inserted by a kernel module in the host OS pass a structure describing the vCPU that generated the `int3` exception. This structure contains another structure with architecture specific information, including the register state at the time of the VM Exit. The detector can check this state at every loop iteration. If the registers remain constant across a large number of iterations, this static state can be attributed to an infinite loop in many applications.

In order to test the infinite loop detector, we used the same example as presented in Jolt [30]. That example is a bug found in a development branch of the Exuberant Ctags source code indexer.[7] In that bug, a string parsing loop would get stuck due to two variable names being transposed in the source code. The example input for the ctags indexer used in Jolt is the python scientific computing package numpy.[8] Specifically, the `_import_tools.py` file contains comments that are formatted in such a way that the bug is activated. In the fixed version of the code the loop executes only one iteration each of the twelve times it is entered, meaning a small threshold could also be used. Regardless of whether the threshold or register change method is used, this loop was easily detected in all experiments since it executes at a rate of thousands of times per second.

VI. Performance

A. Methodology

All of our microbenchmarks and detector performance evaluations were conducted on a Dell PowerEdge R720 server with dual-socket Intel Xeon E5-2660 "Sandy Bridge" 2.20GHz CPUs (3.0 GHz turbo boost). To obtain runtime measurements, we have added an extra hypercall to KVM that starts and stops a timer inside the host OS. This allows us to obtain measurements independent of VM clock jitter. To ensure consistency among measurements, the test VMs were rebooted between each sample.

[7] http://ctags.sourceforge.net/
[8] http://www.numpy.org

B. Microbenchmarks

We perform microbenchmarks that estimate the latency of a single hprobe, which is the time from when the VM executes `int3` until the VM is resumed (Steps 1–3 in Fig. 3). We run these microbenchmarks without a probe handler function to determine the lower bound of hprobe-based detector overhead. Since the round-trip latency of an individual VM Exit on Sandy Bridge CPUs has been estimated to take roughly 290ns [31] and our hypercall measurement scheme induces additional VM Exits, it would be difficult to accurately measure the individual probe latency. Instead, we obtain a mean round-trip latency by repeatedly executing a probed function a large number of times (one million) and dividing by the total time taken for those executions. This helps remove jitter due to timer inaccuracies as well as the actual latency of the hypercall measurement system itself. For the test probe function we have added a no-op kernel module to the Guest OS that creates a dummy `noop` device with an `ioctl` that calls a `noop_func()` kernel function that performs no useful work (`return 0`). First, we insert an hprobe at the `noop_func()`'s location. Our microbenchmarking application starts by issuing a hypercall to start the timer and then an `ioctl` against the noop device. When the noop module in the guest OS receives the `ioctl`, it calls `noop_func()` one million times. Afterwards, another hypercall is issued from the benchmarking application to read the timer value.

For the microbenchmarking experiment, we used a 32bit Ubuntu 14.04 guest and measured 1000 samples. The mean latency (across samples) was found to be 2.6 μs. In addition to the Sandy Bridge CPU, we have also included data for an older generation 2.66GHz Xeon E5430 "Harpertown" processor (running the same kernel, KVM version, and VM image), which had a mean latency of 4.1 μs. The distribution of latencies for these experiments is shown in Fig. 9. The remainder of benchmarks presented use the Sandy Bridge E5-2660. The hprobe prototype requires multiple VM Exits per probe hit. However, in many practical cases the flexibility of dynamic monitoring and lower maintenance due to a simple implementation outweigh this cost. This flexibility can increase performance in many practical cases by allowing one to add and remove probes throughout the VM's lifetime, as will be demonstrated later. Furthermore, CPU manufacturers are constantly working to reduce the impact of VM Exits, as Intel's VT-x saw an 80% reduction in VM Exit latency over its first six years [31].

C. Detector performance

In addition to microbenchmarking individual probes, we measure the overhead of the example hprobe-based detectors presented in Section V. All measurements in this section were obtained using the hypercall-based timer.

1) Emergency Exploit Detector: Our integer overflow detector that protects against the CVE-2008-0600 `vmsplice()` vulnerability is extremely lightweight. Unless `vmsplice()` is used, the overhead of the detector is zero since the probe will not be executed. The `vmsplice()` system call is rare (at least in open source repositories that we searched), so this zero overhead is overwhelmingly the common case. Keeping in mind that security vulnerabilities are often found in "cold" regions of code [32], we believe this low-overhead to extend beyond our simple example. One application that does use `vmsplice()` is Checkpoint/Restart in Userspace (CRIU),[9]. CRIU uses `vmsplice()` to capture the state of open file descriptors referring to pipes. We used the Folding@Home molecular dynamics simulator[33] and the pi-qmc Monte Carlo simulator from earlier as test programs. We ran these applications in a 64-bit Ubuntu 14.04 VM. At each sample, we allowed the application to warm up (load input data and start the main simulation) and then checkpointed it. The timing hypercalls were inserted into CRIU to measure how long it takes to dump the application. This was repeated 100 times for each case with and without the detector and the results are tabulated in Table I. From the table, we can see that there is a slight difference in the mean checkpoint time (roughly 3.3% for F@H and 1.7% for pi-qmc) and that the variance in the experiment with the detector active is higher for the Folding@Home case. When checkpointing Folding@Home, `sys_vmsplice()` was called 28 times, and 11 times for pi-qmc. We can attribute this to negative cache effects of the context switch when activating probes. We also measured another class of "Naïve" detector that probes the `system_call()` function (the entry point for all system calls) during the checkpoint as opposed to `sys_vmsplice()`. In the case where we probe on all system calls, we can see that there is a significant performance penalty (and the number of probe invocations increases to $\sim$ 3000). We remind the reader that the detector only probes `sys_vmsplice()`, meaning the overhead incurred is only when taking a checkpoint.

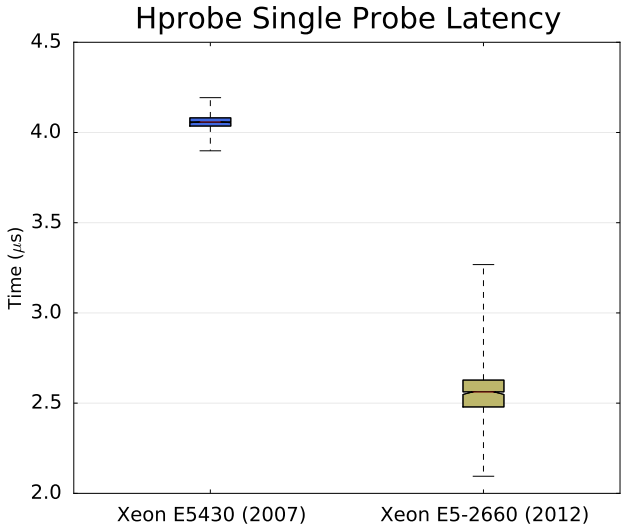

Fig. 9. Single probe latency (parentheses are the CPU's release year). The E5-2660's larger range can be attributed to "Turbo Boost," where the clock scales from 2.2 to 3.0GHz. The shaded area is the quartile range (25th percentile to 75th percentile), whiskers are minimum/maximum, center is the mean, and notches in the middle are the 95% confidence interval of the mean.

2) Application Heartbeat Detector: We the pi-qmc simulator from Section V-B to measure the performance overhead of the application watchdog detector. The pi-qmc simulator allows configuration of its internal sampling and we utilize this feature to vary the length of the main loop. In order to determine how the detector impacts performance we measure

[9]http://www.criu.org/

TABLE I. CVE-2008-0600 DETECTOR W/CRIU

Application	Runtime (s)	95% CI (s)	overhead (%)
F@H Normal	0.221	0.00922	0
F@H w/Detector	0.228	0.0122	3.30
F@H w/Naïve Detector	0.253	0.00851	14.4
pi-qmc Normal	0.137	0.00635	0
pi-qmc w/Detector	0.140	0.00736	1.73
pi-qmc w/Naïve Detector	0.152	0.00513	11.1

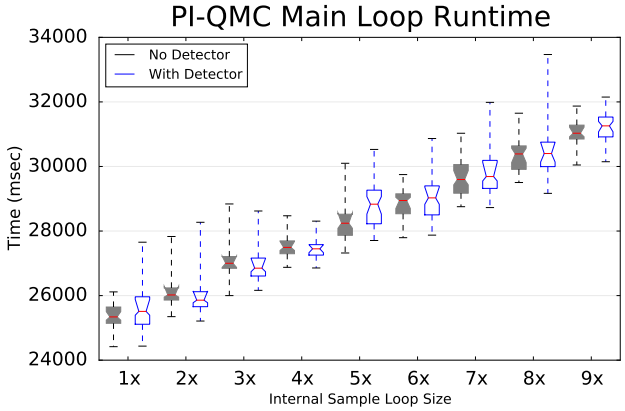

Fig. 10. Benchmarking of the application watchdog detector. The horizontal axis indicates the scaling of an internal loop in the target pi-qmc program. The vertical axis shows a distribution of the completion time for each iteration of the main loop. The boxplot characteristics are the same as in Fig. 9.

the total runtime of each iteration of the main loop when the probe is inserted and run the program for 15 minutes. The results of our experiments are shown in Fig. 10.

From Fig. 10, we show that the detector does not affect performance in a statistically significant way. This is due to the fact that pi-qmc, like many scientific computing applications, does a large amount of work in each iteration of its main loop. However, by setting the threshold of the detector to a conservative value (like twice the mean runtime), one can achieve fault detection in a far more acceptable timeframe than other methods like manual inspection. Furthermore, this detector goes beyond checking if the process is still running - it can detect any fault that causes a main loop iteration to halt (disk I/O hang, network outage when using MPI, software bug that does not lead to a crash, etc...).

3) Infinite Loop Detector: In order to measure the performance overhead of our infinite loop detector, we use a patched version of the ctags application from Section V-C. We ran ctags on the complete numpy source tree 60 times and obtained the mean completion time and 95% confidence interval. The results are tabulated in Table II. There are two implementations of the detector used in these experiments, the "Naïve" detector and the "Smart" detector. The Naïve detector is the same detector as presented in Section V-C and the Smart detector has probes that dynamically add/remove themselves (i.e., the loop exit probe is only added after the loop is entered). When starting the application, the code segment containing the target function was paged out to disk (a clean boot for each sample). The rows in Table II with "Page fix" refer to the runs where we needed to use the EPT mechanism presented in Section IV-C. We also forced the application to page in the target code block at startup, represented by the "No Page Fix"

TABLE II. CTAGS ON NUMPY SOURCE TREE

Application	Runtime (s)	95% CI (s)	% overhead
Normal	1.13	0.0325	N/A
Naïve ILD - Page Fix	1.26	0.0229	11.5
Naïve ILD - No Page fix	1.26	0.0265	11.8
Smart ILD - Page Fix	1.14	0.0267	1.15
Smart ILD - No Page Fix	1.15	0.0215	1.9

samples. From Table II we can see that the performance impact of our solution to deal with paged-out user space application code is not statistically significant (compare the "Page Fix" rows of the same detector to the "No page fix" rows). However, using dynamic probes yields large performance gains. In the Naïve approach, the overall overhead is roughly 11.5% for this input data. With the Naïve detector, the first and second probe get executed 2585 and 54308, respectively. This is due to the fact that in many cases, the loop is skipped over, but the instruction immediately after the loop (i.e., what the second probe replaces) always gets executed. In the Smart approach, the first and second probe both get executed 2585 times (in correct operation on this input data, the loop has only one iteration), yielding a nominal difference between the Smart implementations and the base case without probes. If this loop had instead a high number of internal iterations, then one could use a similar dynamic probe approach, but retain the exit probe and remove the internal probe, adding it periodically or using a timeout mechanism. Note that the capability behind the "Smart" approach is unique to the dynamism in the hprobe framework.

VII. RELATED WORK

The research community has produced a variety of techniques for hook-based virtual machine monitoring. Of particular note are the Lares [8] and SIM [11] approaches. Lares uses a memory-protected trampoline inserted by a driver in the guest VM. That trampoline issues a hypercall to notify a separate security VM that an event of interest has occurred. This approach requires modification to the guest OS (albeit in a trusted manner), so runtime adding and removing of hooks is not possible. Furthermore, a guest OS driver and trampoline is needed for every OS and version of OS supported by the monitoring infrastructure. The Secure In-VM Monitoring (SIM) approach uses a clever configuration of HAV that prevents VM Exits when switching to a protected page inside the VM that performs monitoring. Since SIM does not incur VM Exits, it achieves low overhead. However, this method involves adding special entry and exit gates to the guest OS and hooks are placed in specific kernel locations. In contrast to both of these approaches, hprobes do not require any modification of the guest OS and can be added at runtime to arbitrary locations inside guests. Note that neither work mentions the removal of hooks, which the hprobe framework supports. In addition to platforms built on top of open-source technology, similar dynamic monitoring solutions also exist in proprietary systems. For example, there are vprobes for VMware® ESXi [34]. Since the hooking concepts are similar, one could build the same high-level functionality found in vprobes using the hprobe framework.

In our prototype, the mean latency for a single hprobe was 2.6 μs (4.1 μs for a CPU from the same timeframe as

the related systems). This fits between the single-hook latency of Lares (28 μs) and SIM (0.4 μs). While the raw overhead of a probe hit may appear relatively high, it is important to remember that probe events are rare and even our application benchmarks (Section VI) intentionally exercised the probes beyond what may be typical. The dynamism of our framework allows us to remove probes that are no longer needed at runtime. It is difficult to achieve a similar flexibility with techniques that have hooks statically inserted into the guest OS. In those systems, in order to support the bare minimum flexibility of application-specific monitoring one would either have to maintain a set of guest OS kernels or use a scheme that modifies a running kernel from within the guest.

Our hooking mechanism is enforced by the hardware since it is supported by the VM Exit mechanism. As mentioned in Section IV-D, the prototype's use of `int3` with the `VMCS` exception bitmap configured accordingly can be viewed as a system rooted in hardware architectural invariants [5], [6]. In this case, the invariant is that a properly functioning virtual machine will generate VM Exits on privileged operations (an assumption that is essential for a "trap-and-emulate" VMM). To protect our hooks, we can use Intel's Extended Page Tables (EPT) or AMD's Nested Page Tables (NPT) and write protect the pages that contain active probes. This write protection satisfies the security requirement where hooks cannot be evaded by actors inside the VM and only incurs a performance impact when pages containing probes are written to (a rare event for code in memory). The hprobes framework does place the hypervisor at the root of trust, but well known techniques exist for signing hypervisor code (one should extend the trusted computing base to include the hprobe kernel agent as well) [35], [36], [37].

Previous researchers have utilized `int3` for VMs in xenprobes [38], which provides a guest OS kernel debugging interface for Xen VMs. In this work, we focus on reliability and security monitoring as opposed to debugging and provide concrete example detectors. Additionally, xenprobes can use an Out-of-line Execution Area (OEA) to execute the replaced instruction (vs. always executing in place with a single step like the hprobe prototype does). The OEA provides a performance boost, but it results in a more complex code base and carries the need to create and maintain a separate memory region for this area. The OEA requires an OS driver to allocate and configure the OEA at guest OS boot, and the number of OEAs are statically allocated at boot, placing a hard upper bound on the number of supported probes (which is acceptable for debugging, but not for dependability monitoring). In terms of code complexity, our approach is simpler (less than 1000 lines of code vs 4000 lines of code).

Ksplice [39], a rebootless kernel patching mechanism, can be used in a similar fashion as the `vmsplice()` emergency detector. The 4.0 version of the Linux kernel is also scheduled to incorporate a rebootless patching feature [40]. Ksplice allows for live kernel patching by replacing a function call with a jump to a new patched version of that function. The planned Linux 4.0 feature will use ftrace[10] to switch to a new version of the function after some safety checks. While these techniques can be useful for patches that have been properly tested and worked through a QA cycle, many operators would be uneasy with an untested patch on a live OS. This can be particularly worrisome with data structure consistency concerns [41]. When considering newly reported vulnerabilities, hprobe's simple interface allows one to quickly deploy an out-of-band monitor to detect the vulnerability without modifying the control flow of a running kernel. This temporary monitoring could even be used to provide a stopgap measure while a rebootless patch is in QA testing: one could use the monitor immediately after a vulnerability is announced and until the patch is vetted and safe to use. A technique like this would drastically reduce the vulnerable window and alleviate pressure to perform risky maintenance outside of critical windows. It should be noted that while our example focused on a kernel vulnerability, this emergency detector technique can be extended to a user space program.

[10]http://elinux.org/Ftrace

TABLE III. HOOK-BASED VM MONITORING SYSTEMS

Name	Userspace hooks	Latency	Dynamic	Modifications
xenprobes	No	48μs	Yes	Hypervisor/Guest OS
Lares	No	28μs	No	Hypervisor/Guest OS
SIM	No	0.40μs	No	Hypervisor/Guest OS
hprobes	Yes	2.6μs	Yes	Hypervisor

Table III compares the hprobe framework with a selection of hook-based active VM monitoring systems. Our work extends this past research in several key ways. As far as we know, the hprobe framework is the first to perform application level hook-based monitoring from the hypervisor, paving the way for cloud-based monitoring-as-a-service (where a cloud provider could give the user a set of probe-based detectors to choose from). Our approach requires no modification of the guest OS and detectors can be added/removed at runtime. These features are indispensable in any system intended for production use. We also demonstrated concrete detectors that monitor against real failures and attacks, whereas most work in this area focuses solely on hook implementation and discussion of potential detectors. Our detectors could not have been implemented using any of the previous systems as no other framework supports user space execution hooks (livewire [3] does offer page protection for user space integrity checking) or the dynamism needed to monitor a single system call after a vulnerability is announced.

VIII. CONCLUSIONS

The hprobe framework is characterized by its simplicity, dynamism, and ability to perform application-level monitoring. Our prototype for this framework uses Hardware-Assisted Virtualization and satisfies protection requirements presented in the literature [8]. We find that compared to past work, the simplicity at which detectors can be implemented and inserted/removed at runtime allows us to quickly develop monitoring solutions. Based on our experience, this framework is appropriate for use in real-world environments. From our sample detectors, we see that the framework is suitable for providing detection for bugs, random faults, and use as a stopgap measure against vulnerabilities.

ACKNOWLEDGMENTS

The authors wish to thank Mika Latimer for help in automated migration between kernel versions. This material is

based upon work supported in part by the National Science Foundation under Grant No. CNS 10-18503 CISE, by the Army Research Office under Award No. W911NF-13-1-0086, by the National Security Agency under Award No. H98230-14-C-0141, by the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement No. FA8750-11-2-0084, by the Department of Energy under Award Number DE-OE0000097, by an IBM faculty award, and by Infosys Corporation.

REFERENCES

[1] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library os from the top down," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 291–304, 2011.

[2] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 461–472.

[3] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection." in *NDSS*, vol. 3, 2003, pp. 191–206.

[4] B. D. Payne, "Simplifying virtual machine introspection using libvmi," *Sandia Report*, 2012.

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

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

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

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

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

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

[11] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *In Proc of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 477–487.

[12] P. K. Manadhata and J. M. Wing, "An attack surface metric," *Software Engineering, IEEE Transactions on*, vol. 37, no. 3, pp. 371–386, 2011.

[13] P. Padala, "Playing with ptrace, part1," *Linux Journal*, no. 103, Nov. 2002. [Online]. Available: http://www.linuxjournal.com/article/6100

[14] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developers Manual Volume 3 (3A, 3B & 3C): System Programming Guide*, September 2014.

[15] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

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

[17] N. Bhatia, "Performance evaluation of intel ept hardware assist," *VMware, Inc*, 2009.

[18] Advanced Micro Devices Inc, *AMD64 Architecture Programmers Manual Volume 2: System Programming*, May 2013.

[19] R. Krishnakumar, "Kernel korner: kprobes-a kernel debugger," *Linux Journal*, vol. 2005, no. 133, p. 11, 2005.

[20] W. Feng, V. Vishwanath, J. Leigh, and M. Gardner, "High-fidelity monitoring in virtual computing environments," in *Proceedings of the International Conference on the Virtual Computing Initiative*, 2007.

[21] NIST, "Vulnerability summary for cve-2008-0600," Online, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-0600, USA, 2008.

[22] J. Corbet, "vmsplice(): the making of a local root exploit," Online, http://lwn.net/Articles/268783/, 2008.

[23] D. P. Bovet and M. Cesati, *Understanding the Linux kernel*. " O'Reilly Media, Inc.", 2005.

[24] D. Spinellis, "Trace: A tool for logging operating system call transactions," *ACM SIGOPS Operating Systems Review*, vol. 28, no. 4, pp. 56–63, 1994.

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

[26] M. Gilbert and J. Shumway, "Probing quantum coherent states in bilayer graphene," *Journal of computational electronics*, vol. 8, no. 2, pp. 51–59, 2009.

[27] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problemoverview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[28] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[29] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. Iyer, "Automated derivation of application-specific error detectors using dynamic analysis," *Dependable and Secure Computing, IEEE Transactions on*, vol. 8, no. 5, pp. 640–655, 2011.

[30] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard, "Detecting and escaping infinite loops with jolt," in *ECOOP 2011–Object-Oriented Programming*. Springer, 2011, pp. 609–633.

[31] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon, "Software techniques for avoiding hardware virtualization exits." in *USENIX Annual Technical Conference*, 2012, pp. 373–385.

[32] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High system-code security with low overhead," in *36th IEEE Symposium on Security and Privacy*, no. EPFL-CONF-205055, 2015.

[33] S. M. Larson, C. D. Snow, M. Shirts *et al.*, "Folding@ home and genome@ home: Using distributed computing to tackle previously intractable problems in computational biology," 2002.

[34] M. Carbone, A. Kataria, R. Rugina, and V. Thampi, "Vprobes: Deep observability into the esxi hypervisor," *vmware Technical Journal*, vol. 14, no. 5, pp. 35–42, 2014.

[35] A. M. Azab, P. Ning, and X. Zhang, "Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 375–388.

[36] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi, *Cpu transparent protection of os kernel and hypervisor integrity with programmable dram*. ACM, 2013, vol. 41, no. 3.

[37] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building verifiable trusted path on commodity x86 computers," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 616–630.

[38] N. A. Quynh and K. Suzaki, "Xenprobes, a lightweight user-space probing framework for xen virtual machine," in *USENIX Annual Technical Conference Proceedings*, 2007.

[39] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 187–198.

[40] S. J. Vaughan-Nichols, "No reboot patching comes to linux 4.0," Online, http://www.zdnet.com/article/no-reboot-patching-comes-to-linux-4-0/, 2015.

[41] J. Corbet, "A rough patch for live patching," Online, http://lwn.net/Articles/634649/, 2015.