Using OS Design Patterns to Provide Reliability and Security as-a-Service for VM-based Clouds

Zachary J. Estrada
University of Illinois, Rose-Hulman Institute of Technology
zak.estrada@ieee.org

Read Sprabery
University of Illinois
rspraber2@illinois.edu

Lok Yan
Air Force Research Laboratory
lok.yan@us.af.mil

Zhongzhi Yu
University of Illinois
zyu19@illinois.edu

Roy Campbell
University of Illinois
rhc@illinois.edu

Zbigniew Kalbarczyk
University of Illinois
kalbarcz@illinois.edu

Ravishankar K. Iyer
University of Illinois
rkiyer@illinois.edu

Abstract

This paper extends the concepts behind cloud services to offer hypervisor-based reliability and security monitors for cloud virtual machines. Cloud VMs can be heterogeneous and as such guest OS parameters needed for monitoring can vary across different VMs and must be obtained in some way. Past work involves running code inside the VM, which is unacceptable for a cloud environment. We solve this problem by recognizing that there are common OS design patterns that can be used to infer monitoring parameters from the guest OS. We extract information about the cloud user's guest OS with the user's existing VM image and knowledge of OS design patterns as the only inputs to analysis. To demonstrate the range of monitoring functionality possible with this technique, we implemented four sample monitors: a guest OS process tracer, an OS hang detector, a return-to-user attack detector, and a process-based keylogger detector.

DISTRIBUTION A. Approved for public release: distribution unlimited. Case Number: 88ABW-2017-0936.

DOI: 10.1145/3050748.3050759

Keywords Virtualization; Security; Reliability; VM monitoring; OS design patterns; Dynamic Analysis

1. Introduction

Cloud computing allows users to obtain scalable computing resources, but with a rapidly changing landscape of attack and failure modes, the effort to protect these complex systems is increasing. What is needed is a method for alleviating the amount of effort and skill cloud users need in order to protect their systems.

Cloud computing environments are often built with virtual machines (VMs) running on top of a hypervisor, and VM monitoring could be used to offer as-a-Service protection for those systems. However, existing VM monitoring systems are unsuitable for cloud environments as those monitoring systems require extensive user involvement when handling multiple operating system (OS) versions. In this work, we present a VM monitoring system that is suitable for cloud systems, as its monitoring is driven by abstractions that are common across multiples versions of an OS.

There has been significant research on virtual machine monitoring (Garfinkel et al. 2003; Payne et al. 2007, 2008; Jones et al. 2008; Sharif et al. 2009; Pham et al. 2014; Suneja et al. 2015). By running monitoring outside the VM, the monitoring software is protected against attacks and failures inside the VM. Still, hypervisors can only look at the raw physical memory, and we cannot interpret OS data structures without knowing the OS specifications. This problem is known as the semantic gap.

A highlight of VM monitoring research is virtual machine introspection (VMI). The goal of VMI is to obtain object layouts such as the offset of a particular field (e.g., process name) so that the monitor can interpret the data structures at the hypervisor layer. In traditional VMI implementations (e.g., libVMI (Payne 2012) on Linux), semantic information about the guest OS was gathered by running a privileged program (e.g., a kernel module) inside the VM that calculates data structures offsets. Researchers have automated this process by extracting OS state using instruction traces of unprivileged application execution and automatically generating out-of-VM introspection tools (Dolan-Gavitt et al. 2011; Fu and Lin 2012). However, even those approaches that utilize user-level programs have a clear disadvantage in cloud systems: the cloud provider must run code inside the user's VM, breaking the cloud provider/user abstraction.

Monitors developed using our technique are based on higher-level OS concepts and are not dependent on version-specific data structure layouts or variable names. In contrast, while VMI techniques are effective for extracting a wealth of information from the guest OS, they are often dependent on not only the OS family, but also the version of the OS. If the data structure layout or structure member names change, the VMI system will need to be updated. This means that a VMI system for a cloud-based service would need to include every possible variation of data structure layout and types (e.g., structure members) for its customer VMs. Furthermore, for every set of offsets and types, there will also be a set of values that vary across more VMs (e.g., for each minor kernel version). In our proposed approach the monitors still interact using low-level operations (e.g., instructions, registers, memory reads and writes) as that is the layer at which hypervisor-based systems operate. The detectors we design, however, do not need to understand the low-level information such as which offset into a data structure is the process ID, necessary low-level parameters are automatically inferred through the use of OS design patterns.

Our prototype implementation requires the VM's virtual disk image as the only input to each monitor. In the prototype, we run the user's VM image in an emulator-based dynamic analysis framework (DAF). The DAF is an instrumented version of a full-system emulator that allows us to analyze guest OS activity. For each reliability and security monitor, a plugin for the DAF is written to recognize the necessary OS design pattern(s) and infer monitoring parameters (e.g., function addresses).

While the DAF allows us to analyze the guest OS, VMs in a DAF can run orders of magnitude slower than physical hardware because of the combined overheads of emulation and instrumentation. In order to take advantage of the DAF's robust capabilities but maintain runtime performance acceptable to cloud users, we use a hook-based VM monitoring system for the runtime monitoring component. In our monitoring framework, the DAF identifies a set of addresses, and the hook-based VM monitoring system performs runtime monitoring at those addresses.

The key contributions of this paper are:

1. A technique for event-based VM monitoring that is based on OS design patterns that are common across multiple versions of an OS,
2. A reliability and security monitoring framework that does not modify a cloud user's VM or require input from the user other than a VM image containing his or her specific OS and applications,
3. Four monitors that demonstrate the range of monitoring one could deploy using the proposed Reliability and Security as-a-Service (RSaaS) framework: a guest OS process creation logger, a guest OS hang detector, a privilege escalation detector for return-to-user attacks, and a keylogger detector.

2. Approach

The approach presented in this paper is built on two observations: **(1)** VM monitoring systems often offer more information than is needed for actionable monitoring and **(2)** there exist similarities in OS designs that we can use to identify the information needed by VM monitors without requiring guest OS data structure layouts.

Monitors built on the technique in this paper detect key events of interest that are identified using *OS design patterns* common across OS versions. These design patterns allow us to identify events of interest for monitoring (e.g., a function call) based on hardware events visible to the hypervisor (e.g., a system call is represented by a `sysenter` instruction). Our monitors are parametrized based on guest OS details that change with version. To identify parameter values for a particular guest OS version, we look for changes in the hardware state: e.g., an interrupt/exception, a register access, or the execution of a specific instruction. These hardware interactions map directly to OS design patterns and allow us to develop VM monitoring tools using those patterns. The threat model covered by these detectors is an attack or failure inside the guest OS, assuming the hypervisor/guest interface is protected. If an attacker has already compromised the guest OS, then

Designing a monitor requires answering two questions: **(Q1):** what is the OS design pattern that is needed for monitoring and **(Q2):** how can that design pattern be identified? The workflow for building a monitor based on those questions is shown in Fig. 1. We choose an OS design pattern that can be used to provide information based on the monitoring specification. After choosing a design pattern, we determine the monitor's parameters. We define a *parameter* as an aspect of the OS design pattern needed for the monitor that is expected to change with the OS version (e.g., a function address, interrupt number, etc...). The next step is to determine a hardware event that is associated with that OS design

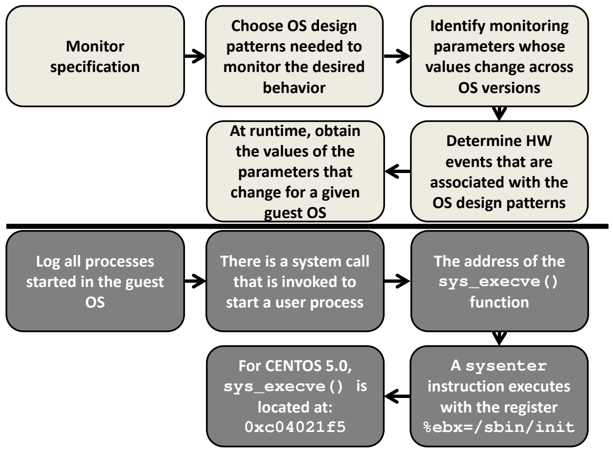
Monitor specification
Choose OS design patterns needed to monitor the desired behavior
Identify monitoring parameters whose values change across OS versions
Determine HW events that are associated with the OS design patterns
At runtime, obtain the values of the parameters that change for a given guest OS
Log all processes started in the guest OS
There is a system call that is invoked to start a user process
The address of the sys_execve() function
A sysenter instruction executes with the register %ebx=/sbin/init
For CENTOS 5.0, sys_execve() is located at: 0xc04021f5

Figure 1: The workflow used when building a detector for the Reliability and Security as-a-Service framework. The light colored boxes on the top describe the workflow and the bottom darker boxes show the workflow applied to process logging example in Section 2.1.

pattern. After identifying a hardware event associated with the parameter, one can find the value of the parameter by observing that hardware event.

2.1 Example: Process Logging

We demonstrate the construction of a monitor based on our technique with process logging as an example. Logging which processes are executed is a common VMI task and serves as a VM monitoring example (Payne et al. 2008; Sharif et al. 2009). Intrusion detection systems can use information about which processes were executed to infer whether or not users are behaving in a malicious manner (Provos 2003; Jiang and Wang 2007).

The OS design pattern used for process logging in Linux is that the `execve` system call is invoked when a process is started. After a `execve` system call, the `sys_execve` internal kernel function is called. While we know that every version of Linux has a `sys_execve` function, the address of `sys_execve` changes across kernel versions. In order to identify the address of the `sys_exec` function, we first need to know when the system call for `execve` was issued. As a reminder, system calls are invoked using a special instruction (e.g., `sysenter`) and the system call number is stored in a general purpose register (i.e., `%eax`).

We can identify a system call by observing that a particular instruction was executed, but we need to apply constraints to identify that system call as `execve`. **C1-C5** describe constraints that can be used to identify `execve` after a system call is executed.

C1 The system call number is stored in `%eax`

C2 A system call is invoked using either the `sysenter` or the `int $0x80` instruction

C3 The first argument for a system call is stored in `%ebx`

C4 The `execve` system call is system call number 11

C5 The first process started by a system is `init`

Each constraint applies to a set of guest OSes. For example, both Linux and Windows use the `%eax` register to store the system call number, so **C1** holds for those OSes on x86. **C2** holds for modern OSes using the "fast syscall" `sysenter` instruction (`syscall` on AMD), and also for legacy implementations of Linux that use `int $0x80`. Linux uses `%ebx` to hold the first argument for a system call, whereas other OSes may use different registers, so **C3** is valid for Linux. Linux system call numbers are not formally defined in any standard, and **C4** was true for 32-bit Linux, but it changed for 64-bit. If we wish to support a wider variety of guest OSes, we can use an additional constraint. The first process executed in any Unix system is the `init` process, therefore we can determine that a system call that has the string `init` as its first argument is the `execve` system call. We represent this constraint in **C5** and it can be used whenever we cannot expect **C4** to hold (e.g., on macOS one would use `launchd` instead of `init`).[1] We could also view the constraints for identifying `execve` as a logical formula: $C1 \wedge C2 \wedge C3 \wedge (C4 \vee C5)$.

We present a pseudocode algorithm used for locating `sys_execve` in Fig. 2. For the sake of brevity, we do not present a full evaluation of the process creation logger, but we have implemented it on the platform described in Section 4 and tested it on CENTOS 5.0, Ubuntu 12, and Ubuntu 13. Following the same performance testing that is used in Section 5.3, we observed an overhead of 0.1% for a kernel compile, 1.7% for a kernel extract, 1.3% for the disk benchmarking suite Postmark, and 0.7% for Apache Bench.

2.2 Summary of Guest OS Information Inferred in Example Detectors

The insight behind our observations on how to extract useful information from fundamental OS operations is best demonstrated through examples. The examples presented in this paper are summarized in Table 1. Note that we do not necessarily assume a stable ABI, but look for a higher-level abstraction in the form of an OS design pattern. Contrast this to VMI, where even changes in the kernel API require updates to the monitoring system. The assumptions we use were valid for multiple generations of OSes, but we cannot prove they will work for all future OS versions.

3. Background

3.1 Hardware Assisted Virtualization

Our prototype implementation uses Intel's Hardware-Assisted Virtualization (HAV) so we summarize Intel's virtualization technology (VT-x) (Intel Corporation 2014), but similar con-

[1] Note that for a more robust address identification algorithm, one can add the names of other processes that are expected to be executed at boot

Table 1

Monitor	OS Design Pattern(s)	Parameters	How Parameters are Inferred
Process Logging	processes are created using a system call	the address of the process creation function; the register containing the process name argument	record the address of the `call` instruction after an instruction ends; search for the string `init` in possible system call argument registers
OS Hang Detection	during correct OS execution, the scheduler will run periodically	the scheduler address; the maximum expected time between calls to the scheduler	find the function that changes process address spaces; record the maximum time observed between process changes when the system is idle
Return2User Attack Detection	code in userspace is never executed with system permissions; the transition points between userspace and kernelspace are finite;	the addresses for the entry and exit points of the OS kernel	observe the changes in permission levels and record all addresses at which those changes occur
Keylogger Detection	a keyboard interrupt will cause processes that handle keyboard input to be scheduled	interrupt number for the keystroke; number of processes expected to respond to a keystroke	use virtual hardware to send a keystroke and record the interrupt number that responds; observe scheduling behavior after a keystroke

```
procedure ON_INSTRUCTION_END(cpu_state)
    if instruction == (sysenter $\vee$ int $0x80) then
        for all r $\in$ {GPR} do
            if r contains "/sbin/init" then
                arg_register $\leftarrow$ r
                found_execve $\leftarrow$ true
                return /*execute next instruction*/
            end if
        end for
    end if
    if found_execve $\wedge$ (instruction == call) then
        sys_execve_addr $\leftarrow$ EIP
        OUTPUT(sys_execve_addr, arg_register)
        EXIT
    end if
end procedure
```

Figure 2: Pseudocode algorithm for identifying the address and argument register for the execve system call. Recall that in x86 the program counter is stored in the `EIP` register. GPR is the set of general purpose registers: {`%eax`, `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`}.

cepts are used in AMD's AMD-V(Advanced Micro Devices Inc 2013).

A conventional x86 OS runs its OS kernel in kernel mode (ring 0) and applications in user mode (ring 3). HAV maintains these modes but introduces a new guest and host mode, each with their own set of privilege rings. The hypervisor or Virtual Machine Monitor (VMM) has full access to hardware and is tasked with mediating access to all guests. This means certain privileged instructions in guest ring 0 must transition to the hypervisor for processing, this occurs via a hardware generated event called a *VM Exit*.

Paging is a common technique for memory virtualization: the kernel maintains page tables to give user processes a virtual address space (the active set of page tables is indicated by the `CR3` register). In the earliest implementations of x86 HAV, guest page faults would cause VM Exits and the hypervisor would manage translations via shadow page table structures. To reduce the number of VM Exits, two-dimensional paging (TDP) was added. Intel's TDP is called Extended Page Tables (EPT). In EPT, the hardware traverses two sets of page tables: the guest page tables (stored in the VM's address space) are walked to translate from guest virtual to host physical address, and then the EPTs (stored in the hypervisor's address space) are walked to translate from guest physical to host physical address.

3.2 Hook-based VM Monitoring

In order to provide as-a-Service functionality, our technique requires a monitoring system that is dynamic (can be enabled/disabled without disrupting the VM) and flexible (allows for monitoring of arbitrary operations). Hook-based VM monitoring is a technique in which one causes the VM to transfer control to the hypervisor when a hook inside the VM is executed. This is similar in operation to a debugger that briefly pauses a program to inspect its state. In fact, hook-based monitoring can be implemented using HAV with hardware debugging features that cause VM Exits, an approach we use in this paper. The research community has produced a variety of techniques for hook-based VM monitoring (Quynh and Suzaki 2007; Payne et al. 2008; Sharif

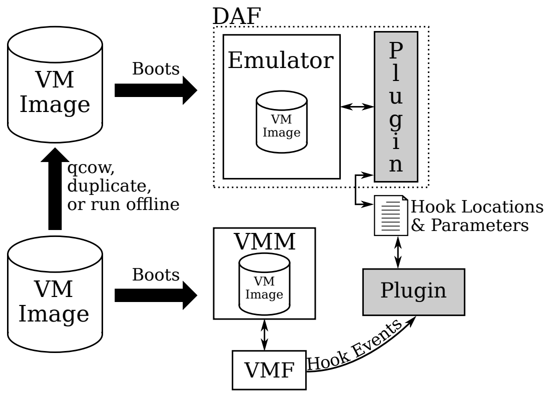

Figure 3: Architecture proposed in this work. Each monitor is defined by a Dynamic Analysis Framework (DAF) plugin and a VM Monitoring Framework (VMF) plugin (gray boxes).

et al. 2009; Deng et al. 2013; Carbone et al. 2014; Estrada et al. 2015).

4. Prototype Platform Implementation

We envision a system where the user can select the reliability and security plugins from a list of options. Once selected, the parameter inference step will automatically identify the pertinent information from the user's VM image and transfer the knowledge to the runtime monitoring environment. As a research implementation, we limit the scope of our prototype to the components that perform monitoring. The rest of the cloud API and interface (e.g., integration with a framework like OpenStack[2]) will be engineered as future work.

4.1 Example Architecture

The prototype combines both a Dynamic Analysis Framework (DAF) and a VM Monitoring Framework (VMF). The DAF performs the parameter inference step specific to each cloud user's VM (e.g., finding guest OS system call addresses). The VMF transfers control to the hypervisor based on hooks added to the VM at runtime. A diagram of how these components interact is shown in Fig. 3. Note that the proposed approach is quite general and other implementations may choose to use different techniques for parameter inference (e.g., static binary analysis of the guest OS kernel image) or runtime monitoring (e.g., kprobes for containers (Krishnakumar 2005)).

4.2 Dynamic Analysis Framework

In the parameter inference stage, we use a gray-box approach where the only input needed from the cloud user is their VM image. For our DAF, we use the open-source Dynamic Executable Code Analysis Framework or DECAF (Henderson et al. 2014).[3] We chose DECAF because it is based on QEMU (Bellard 2005) and therefore supports a VM image format that is compatible with the popular Xen and KVM hypervisors.[4]

[2] http://openstack.org

[3] DECAF is available at: https://github.com/sycurelab/DECAF

4.3 VM Monitoring Framework

We use the open-source KVM hypervisor integrated into the Linux kernel (Kivity et al. 2007). We add hooks by overwriting instructions of interest (as identified by the DAF) with an `int3` instruction (Quynh and Suzaki 2007; Carbone et al. 2014; Estrada et al. 2015). The hooks can be protected from guest modification by using EPT write-protection.

In addition to hook-based monitoring functionality, we also add a set of callbacks to the KVM hypervisor to receive information about certain events of interest (e.g, on a VM Exit, an EPT page fault, etc...). To keep our implementation portable, we have kept the modifications to KVM at a minimum, requiring only 133 additional lines of C code to support the functionality presented in this paper. All of the monitoring API and monitors presented later in the paper are implemented as external modules totaling 3007 lines of C code (including boilerplate code common across different monitors). We confirmed that the KVM unit tests[5] still pass on our modified version of KVM.

4.4 Machine Configuration

Unless otherwise noted, all performance benchmarks were performed on a machine with an Intel® Core™ i7-4790K CPU. The CPU has a clock frequency of 4.00GHz, the machine has 32GiB of DDR3 1333 MHz of RAM with a Hitachi HUA723020ALA640 7200RPM 6.0Gb/s SATA hard disk drive. This machine ran Ubuntu 14.04 LTS, though development also occurred on machines with various hardware running CENTOS7 and Ubuntu 12.04 LTS.

4.5 Discussion

Our main target for testing is the Linux OS (various distributions). While Linux is open-source, the cloud provider cannot use a white-box approach since each distribution or even user can configure the OS differently. We maintain our gray-box approach and only use OS semantics that can be obtained from our dynamic analysis or from version agnostic OS proprieties (e.g., paging, published ABI, and privilege levels) and do not rely on or use the source code. Linux is a natural choice for a target OS in IaaS cloud protection as data from Amazon's EC2 shows that there are an order of magnitude more Linux servers running in EC2 (the most popular public cloud) than the next most-popular OS (Vaughan-Nichols 2015). Nevertheless, to demonstrate the versatility of our technique, we also present a keylogger detection example using Windows 7 in Section 7.

[4] Note that DECAF uses QEMU in full emulation, whereas QEMU+KVM will be later used to run the VM

[5] http://www.linux-kvm.org/page/KVM-unit-tests

We can evaluate this prototype system partially in terms of the cloud computing aspects discussed in the NIST definition of cloud computing (Mell and Grance 2011):

- **On-demand self-service:** This system operates with the existing VM image as input and monitors can be added/removed at runtime.
- **Broad network access:** The assumption of broad network access is necessary for the deployment and transfer of VM images.
- **Resource pooling:** Once developed, monitors can be shared across multiple customers. Dynamic analysis only needs to be performed once per customers' OS kernel.
- **Rapid elasticity:** Our monitoring is elastic by its on-demand nature. Monitors can be added/remove and enabled/disabled at runtime without disrupting a running VM. This aspect was tested for the examples presented in this paper.
- **Measured service:** Differing levels of service can be measured by the type and amount of monitors the user enables.

The intended use of RSaaS is for the cloud provider to develop trusted plugins. Based on this prototype, we do not expect providers to open this interface to users without additional features to isolate hypervisor-level development from affecting other users' VMs. The skill required to develop DAF and VMF plugins is roughly the same as required for kernel module development (in that one must have an understanding of OS concepts and principles), and this effort can be amortized by reusing plugins for different customers' VM instances. Since cloud providers run extremely large systems and have administrators with expert OS experience, we do not view the skill requirement as detrimental to the adoption of our technique.

5. OS Hang Detection

One of the largest limitations of cloud computing is the lack of physical access. Since users must access the resources through a network interface, a lack of responsiveness can either be due to a network failure or system failure. To help isolate the possibility of network failures, we introduce an OS hang detector. This hang detector also demonstrates the concept of dynamic monitoring to increase the performance of a monitor. Some hypervisors provide a watchdog device,[6] but that device requires a guest OS driver. Our approach requires no drivers or configuration from the guest OS and can be added at runtime.

In a properly functioning OS, we expect the scheduler to schedule processes. If the scheduler does not run after an expected amount of time, we can declare that the guest OS is hanging. The OS design pattern for OS Hang detection is that the scheduler runs regularly to schedule processes and

[6] https://libvirt.org/formatdomain.html#elementsWatchdog

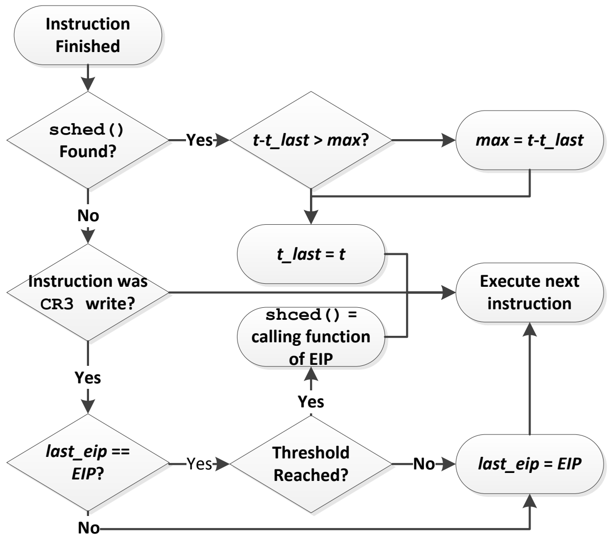

Figure 4: Flowchart for the inferring parameters for OS Hang Detection.

we monitor this at runtime by adding a hook to the guest OS scheduler address.

5.1 Hang Detector Parameter Inference

In order to locate the address of the scheduler, we observe that the scheduler is the OS component that switches processes. Each process has a unique set of page tables, and a process switch will write to the `CR3` register. While other functions write to `CR3`, we have observed that the scheduler consistently writes to `CR3` over time. This leads to a simple heuristic: the scheduler is the function that writes to CR3 the most.

Note that the scheduling interval may not be a constant value. In earlier versions of the Linux kernel, the scheduler was invoked at a regular interval based on a "tick." The tick was configurable and defaulted to 1000HZ. Recent kernels, however, have moved to a "tickless" approach to reduce power (Siddha et al. 2007; Corbet 2013). With a tickless kernel, the scheduler no longer runs at a fixed frequency, so the maximum measured scheduler interval depends on OS configuration and the software installed. (i.e., systems running a variety of applications will have more scheduler invocations).

We interpret the measured scheduling interval to be an upper bound on the scheduling interval. This is because the guest OS will enter an idle state after boot. At run time, we expect more activity due to input and therefore more scheduling events. Furthermore, we introduce overhead from emulation and dynamic analysis, which also inflates the measured scheduling interval.

A flowchart for the parameter analysis is presented in Fig 4. While we did not encounter Linux with in-kernel Address Space Layout Randomization (ASLR) in our experi-

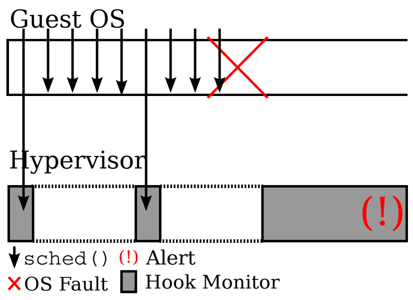

Figure 5: Dynamic monitoring example: the hypervisor is notified when the scheduler runs. During a hang the hook is still added but the scheduler does not run.

ments, if the system is using in-kernel ASLR, an offset from fixed location in the kernel text section (e.g., `SYSENTER_EIP`) as opposed to the scheduler's address could be used since both the scheduler address and system call handler are in the text section of the main kernel.

Table 2 summarizes the results of running the DAF plugin for parameter inference on various kernel versions. Note that for Fedora 11 the plugin did not identify the scheduler. However, the hang detector will detect a kernel hang because the `switch_mm` function is called when processes are changed. However, the frequency at which `switch_mm` is called is lower than `schedule` and the detection latency is higher with `switch_mm` than with `schedule`.

5.2 Hang Detector Runtime Monitor

If one generates a VM Exit event on every scheduler event, there could be significant overhead. However, we do not need to hook every call to the scheduler. Instead, we can take a dynamic monitoring approach. We add a hook to the scheduler and after the scheduler executes we remove the hook. We then queue another hook to be added after the expected scheduling interval. This is illustrated in Fig. 5.

5.3 Hang Detector Evaluation

In order to evaluate the effectiveness of the hang detector, we performed fault injections using double spinlocks and NULL pointer dereferences to hang the kernel. To measure the detection latency (the time from when a fault is injected to when that fault is detected) while ensuring the robustness of our detector against race conditions, we repeated both injections 1000 times each. For both fault types tested, the detection coverage was 100% with 0 false positives and 0 false negatives. The cumulative probability distribution for the detection latency is plotted in Fig. 6.

We evaluated the performance benefits of dynamic monitoring with a context switch microbenchmark.[7] The benchmark measures the time the OS takes to switch between two threads. Switching two threads will invoke the scheduler but almost nothing else. We ran the benchmark on the VM

[7] https://github.com/tsuna/contextswitch

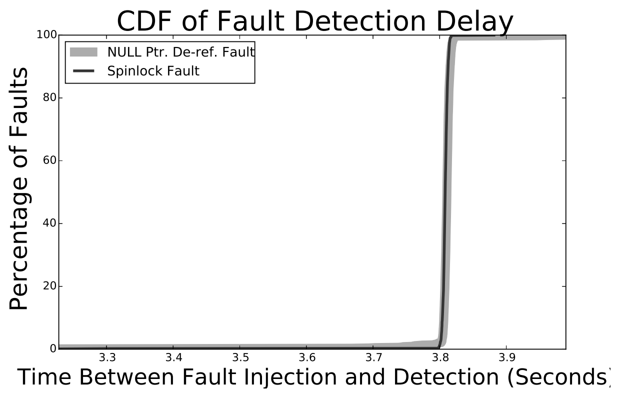

Figure 6: CDF for detection latency of a system hang for both a deadlock and NULL pointer de-reference.

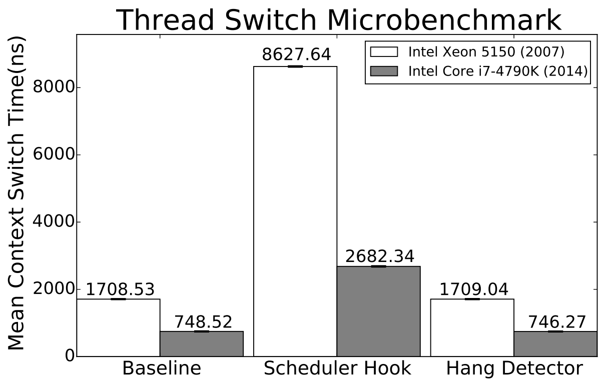

Figure 7: Context switch microbenchmark. The baseline is a VM without the hang detection monitor. The scheduler hook represents a naïve approach where a hook was always added to the scheduler. The last data set represents the dynamic monitoring approach.

without any hooks, with hooks always added to the scheduler (naïve approach), and with the dynamic monitoring approach. From Fig. 7, we can see that the dynamic monitoring approach has negligible overhead, even in a microbenchmark.

To gauge the performance impact of this detector on cloud applications, we run three application benchmarks: a compile of Linux Kernel 2.6.35, Apache Bench, and PostMark. Apache Bench and PostMark were both configured and run using the Phoronix Test Suite[8] and all three were run 30 times. Apache Bench is used to represent a traditional webserver workload and is evaluated in terms of requests per second. PostMark is used to measure disk performance and is evaluated in terms of transactions per second. All of these experiments were performed on an Ubuntu 10.10 VM. Fig. 8 shows the results of this evaluation with error bars indicating the 95% confidence interval of the mean.

[8] http://www.phoronix-test-suite.com/

Table 2: Functions Identified as the Scheduler

OS	Inferred Scheduler Address	Function Name	Measured Interval (s)	Kernel Version
CENTOS 5.0	0xc05fa3ac	`schedule`	3.5193	2.6.18-8.el5
CENTOS 5.4	0xc062628c	`schedule`	0.2507	2.6.18-398.el5PAE
Fedora 11	0xc0428565	`switch_mm`	20.0120	2.6.29.4-167.fc11.i686.PAE
Ubuntu 10.10	0xc05f1620	`schedule`	1.0077	2.6.35-32-generic-pae
Arch Linux	0xc146baa0	`__schedule`	1.7563	3.17.6-ARCH

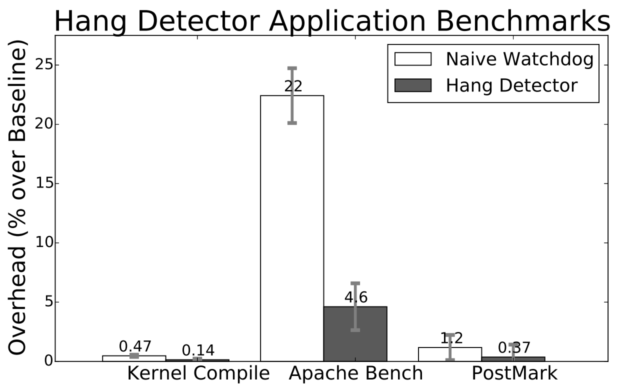

Figure 8: Application overhead comparing hang detection methods. The naïve approach hooks every schedule call and the last column uses dynamic monitoring. Lower is better.

All benchmarks were run with no monitor loaded, a naïve monitor, and with our dynamic hang detector. In Fig. 8 the impact of our hang detector over the baseline is negligible in all three cases, reducing the mean performance by 0.38%, 4.41%, and 0.34% for the kernel compile, Apache Bench, and PostMark, respectively.

6. Return-to-User Detection

Return-to-user (ret2user) attacks are attacks where userspace code is executed from a kernel context. Ret2user is a common mechanism by which kernel vulnerabilities are exploited to escalate privileges, often using a NULL pointer dereference or by overwriting the target of an indirect function call (Keil and Kolbitsch 2007). Ret2user is simpler for attackers than using pure-kernel techniques like Return Oriented Programming (ROP) since the attacker has full control over their shellcode in userspace, and only needs to trick the kernel into executing that shellcode (as opposed to deriving kernel addresses or figuring out a way to copy shellcode into kernel memory). If a ret2user vulnerability cannot be used to escalate privileges, it can be used to crash a system via a Denial-of-Service (DoS) attack by causing a kernel-mode exception. We use the ret2user attack as an example of how to build a security detector for the RSaaS framework that is based on OS design patterns that apply to multiple vulnerabilities.

```
procedure ON_INSTRUCTION_END(cpu_state)
    if last_cpl == 3 && cpu_state.CS.sel == 0 then
        /*Transition from user to kernel*/
        KERNEL_ENTRIES ∪ cpu_state.EIP
    else if last_cpl == 0 && cpu_state.CS.sel == 3 then
        /*Transition from kernel to user*/
        /*The EIP of the previous instruction is a kernel address*/
        KERNEL_EXITS ∪ last_eip
    end if
    last_cpl ← cpu_state.CS.sel
    last_eip ← cpu_state.EIP
end procedure
```

Figure 9: Identifying kernel entry and exit points. The processor's current privilege level (`CPL`) is stored in the selector of the `CS` segment register.

In Linux, the kernel's pages are mapped into every process's address space. While the OS is expected to copy data to/from user-level pages in memory, the kernel should never execute code inside user pages. The ret2user detector detects when the kernel attempts to execute code in a user page. The OS design patterns used by the ret2user detector are: (1) the kernel runs in ring 0 and the user applications run in ring 3 and (2) the kernel entry/exit points are finite and will not change across reboots (though we did not encounter this, if needed our approach could be adapted to a system where ASLR is present as was discussed for OS hang detection).

6.1 Return-to-User Parameter Inference

The parameters for the ret2user detector are the entry and exit points to and from the kernel. We identify those entry and exit points by tracking the `CPL` after each instruction was executed and recording the value of the `EIP` register when the CPL transitioned from 0→3 or 3→0. The pseudocode is shown in Fig. 9.

6.2 Return-to-User Runtime Monitor

The monitor for the ret2user detector adds hooks to the kernel entry and exit points obtained during parameter inference. After the VM boots, we scan the guest page tables to identify which guest virtual pages belong to the kernel. After obtaining the virtual addresses for the kernel's code, we

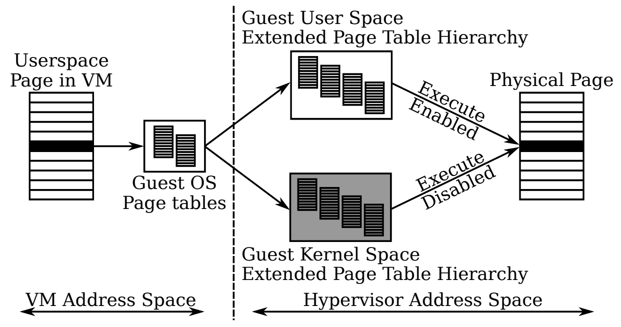

Figure 10: Ret2user example detector. When the VM transitions from guest user to guest kernel space the hypervisor switches EPTs. In the guest kernel address space, EPT entries for guest user pages have execution disabled to prevent ret2user attacks. The VM controls its own page tables, but is isolated from editing the EPTs.

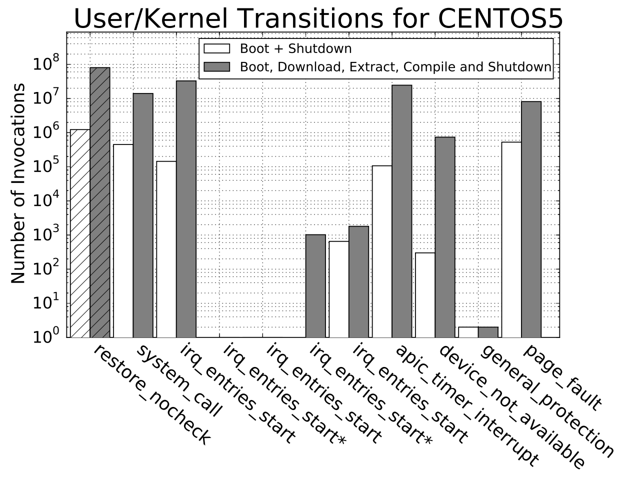

Figure 11: The diagonal hatched/solid bars represent guest kernel exit/entry points, respectively. The vertical axis indicates the number of times the transition point was invoked and the horizontal axis indicates the function containing the point. `irq_entries_start` appears multiple times as each IRQ line represents a unique kernel entry point ($*$ denotes transitions unique to the kernel compile workload).

create a second set of EPTs. We then copy the last-level EPT entries to the new tables so that the last level still correctly points to the host pages containing the VM's data. When copying the last-level entries, we remove execute permissions. We switch the set of active EPT tables at each transition: we use the original tables while the guest is executing in user mode and the duplicated tables while the guest is executing in kernel mode. Fig. 10 illustrates the ret2user detector.

Table 3: Ret2user Vulnerabilities Detected

OS	Vulnerability	# Entries	# Exits
CENTOS 5.0	CVE-2008-0600	7	1
CENTOS 5.0	CVE-2009-2692	7	1
CENTOS 5.0	CVE-2009-3547	7	1
Fedora 11	CVE-2009-2692	7	1
Ubuntu 10.10	CVE-2010-4258	6	1

6.3 Return-to-User Evaluation

To test the coverage of observed Kernel entry/exit points, we profiled a VM running CENTOS 5.0 (chosen because it contains multiple vulnerabilities). First, we collected the entry/exit points from a bootup and shutdown sequence. To test whether a bootup/shutdown sequence is sufficient, we also measured the entry/exit points with a Linux kernel source archive download, extraction, and compilation. This workload exercises the kernel entry/exit points one would expect to see during a VM's lifetime: downloading a file exercises the network and disk, extracting and compiling are mixed cpu/disk/memory workloads.

The results of the kernel entry/exit tests are shown in Fig. 11. The only entry points that were observed during the kernel workload and not during bootup/shutdown were entries in the IRQ handler. If needed, one could obtain those entries directly using the interrupt tables. All ret2user exploits we studied use the system call entry point, even exploits involving vulnerabilities in the kernel's interrupt handling code.[9] To measure the effectiveness of the ret2user detector, we tested it against public vulnerabilities as shown in Table 3. We observe that in the kernels tested, we only identified one common exit point.

The ret2user detector cannot be circumvented by a guest unless a user in the VM compromises the hypervisor or creates a new kernel entry/exit point. The EPT-based detection technique can detect exploits using yet-to-be-discovered vulnerabilities. Intel has released a similar protection in hardware called Supervisor Mode Execution Protection (SMEP) or OS Guard (see Section 4.6 of the Intel Software Developer's Manual (Intel Corporation 2014)). SMEP offers protection similar to our detector, but since it is controlled by the guest OS SMEP: (1) requires support in the guest OS, and (2) can be disabled by a vulnerability in the guest OS (Rosenberg 2011; Shishkin and Smit 2012). The ret2user detector can also be used to protect VMs which are running legacy OSes (a common virtualization use case) or on CPUs that do not support SMEP. This detector is flexible and one could change the criteria for what is protected beyond preventing kernel executions of userspace code (e.g., to restrict code from unapproved drivers or system calls (Gu et al. 2014)).

[9] https://www.exploit-db.com/exploits/36266/

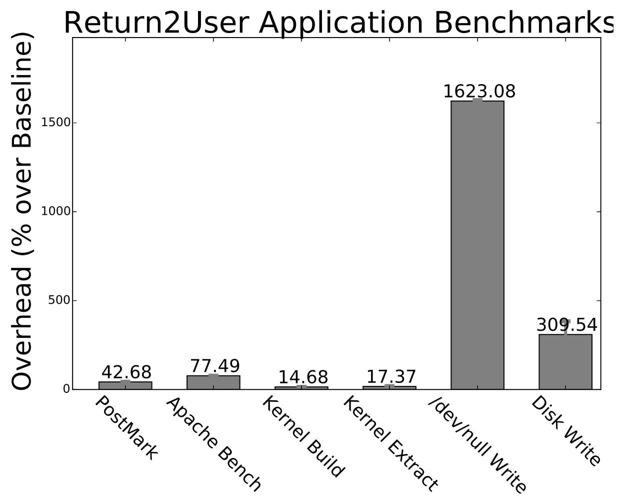

Figure 12: Benchmark overhead for Apache Bench, PostMark, a kernel source extract, build, and microbenchmarks focused on writes compared against a baseline without the ret2user monitor. Lower is better.

To measure the overhead of the detector, we ran a kernel uncompress and compile as well as a disk write and kernel entry/exit microbenchmark. The disk write is a copy of 256 MiB from `/dev/zero` to `/tmp` (the buffer cache is cleared on every iteration) and the microbenchmark is the same except it outputs to `/dev/null` to remove any disk latency and effectively exercise only kernel entry/exit paths.

The results of the performance measurements for ret2user are given in Fig. 12. The microbenchmark exhibits roughly 20x overhead, but the kernel workloads exhibit 0.15x overhead. Additionally, we reran the same filesystem and web workloads from Section 5.3. The results for Apache Bench and PostMark can be seen in Fig. 12. The ret2user detector adds 77.49% overhead for Apache Bench and 42.68% overhead for PostMark, respectively. Our technique's ability to change its monitoring functionality at runtime allows it to be an ideal platform for a future adaptive monitoring system (Hill and Lynn 2000). The adaptive system could, for example, use more expensive security monitors (e.g., ret2user) only when lower overhead monitors detect suspicious activity (e.g., process trace sees `gcc` run on a webserver) (Cao et al. 2015). We note that a less expensive hooking mechanism would significantly reduce this overhead (Sharif et al. 2009).

Vulnerabilities are common, but released infrequently on computing timescales. As of writing, 192 privilege escalation vulnerabilities have been identified in the Linux kernel since 1999.[10] Even if an organization is using vulnerable software, it is unlikely that every vulnerability discovered for that software applies to the configuration used by that organization. However, clouds by their nature are heterogeneous (many customers running various applications and configurations). Therefore, a provider can reasonably expect that any given vulnerability will apply to a subset of that provider's users and enable a detector like ret2user to mitigate risk before systems can be patched. A performance cost during this period can be preferable to either running unpatched systems or disrupting a system for patching.

[10] https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

7. Process-based Keylogger Detection

Many enterprise environments use Virtual Desktop Integration or VDI to provide workstations for their employees. In VDI, each user's desktop is hosted on a remote VM inside a datacenter or cloud. VDI offers many advantages including a simpler support model for (potentially global) IT staff and better mitigation against data loss (e.g., from unauthorized copying). While VDI provides security benefits due to the isolation offered by virtualization, VDI environments are still vulnerable to many of the same software-based attacks as traditional desktop environments. One such attack is a software based keylogger that records keystrokes.

Process based keyloggers are keyloggers that run as processes inside the victim OS. These keyloggers represent a large threat as they are widely available and easy to install due to portability. Previous work in keylogger detection is built on looking at I/O activity as keyloggers will either send data to a remote host or store the keystroke data locally until it can be retrieved (Ortolani et al. 2010). In this section, we present a new detection method for process based keyloggers that monitors for changes in the behavior of the guest OS.

The OS design pattern used to detect a process-based keylogger is that after a keystroke is passed into the guest OS, the processes that consume that keystroke will be scheduled.

7.1 Keylogger Detection Parameter Inference

In order to detect what processes respond to a keystroke, we must detect a keystroke event from the hypervisor. Just like a physical keyboard, a virtual keyboard will generate an interrupt which will then be consumed by an interrupt service routine (ISR). In x86, the ISRs are stored in the Interrupt Descriptor Table (IDT). The goal of the parameter inference step is to identify which ISR is responsible for handling keyboard interrupts as different VM instances may use different IDT entries or even different virtual devices.

In the dynamic analysis framework, we send keyboard input to the VM by sending keyboard events through software without user interaction. Using a hardware interrupt callback, we determine the IDT entry for the keyboard interrupt handler as well as the `EIP` of the keyboard interrupt handler.

7.2 Keylogger Detection Runtime Detector

The detector takes as its input the IDT entry number. When the keylogger detector is enabled, a hook is then added to the ISR for the keyboard interrupt as determined during the dynamic analysis step. Whenever a key is pressed, the detector re-enables `CR3` VM exits and the `CR3` values after the keystroke are recorded and analyzed as described below.

Each process-based keylogger uses slightly nuanced hooking methods (none that we tested overrode IDT entries), but all of the keyloggers we tested were scheduled shortly after a keystroke. This allows us to build a simple, but effective detection heuristic: the more processes (represented by `CR3` values) responding to a keystroke, the more likely it is that a keylogger is present.

We call the per-process metric we developed the *responsiveness score* for that process. The responsiveness score is computed for each process at every keystroke by exponential decay with a half-life of $t_{1/2}$ as shown in Eq. 1:

$$R_k(\texttt{CR3}) = e^{-\ln(2)\frac{t_{CR3}-t_k}{t_{1/2}}} \quad (1)$$

$R_k(\texttt{CR3})$ is summed over every CR3 change after keystroke k and before keystroke $k+1$. If that sum is ≥ 1 for any process, we count that process as responding to keystroke k. While detector is running, we measure the mean value of R_k across all processes and when it is greater than an expected threshold, we report the presence of a keylogger.

In order to measure $R_k(\texttt{CR3})$ at runtime, we add a hook to the ISR responsible for keyboard events and re-enable `CR3` VM Exits. We compute $R_k(\texttt{CR3})$ in a separate analysis program running in userspace on the hypervisor.

7.3 Keylogger Detection Evaluation

In our experiments we use a half-life of $t_{1/2} = 100$ms. This was chosen based on the perception time of the average person (Nielsen 1993). Note that this detector is only an example and not an exhaustive study on keylogger detection, so it is likely that there exist better parameters or functions for measuring responsiveness. We tested the parameter inference on Ubuntu 16.04, Windows 7, and Windows 8 and found that the IDT entries for the keyboard were 0x31, 0x91, and 0x90, respectively. For a more thorough evaluation we used a Windows 7 VM and tested the detector against four keyloggers freely available from the Internet. The keyloggers we tested were: Revelear Keylogger, Actual Keylogger, Spyrix Keylogger, and Free Keylogger Platinum. We tested each keylogger with multiple workloads: typing in notepad, browsing in Internet Explorer, browsing in Mozilla Firefox, editing a spreadsheet, editing a slideshow presentation, and running ten background processes while editing a document. We also ran the same workloads with no keylogger present and the Receiver Operating Characteristics of our experiments are plotted in Fig. 13. We see that this basic keylogger detector performs well, with a AUC of 0.95 (an ideal detector would have an AUC of 1.0). We found that the false negatives came from experiments with the Actual Keylogger in web browsing workloads. After inspection, we discovered

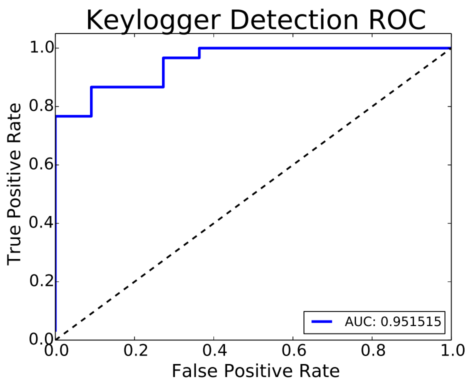

Figure 13: Receiver Operating Characteristics for the keylogger detector on Windows 7. The Area Under the Curve was 0.95 (1.0 would represent a perfect system).

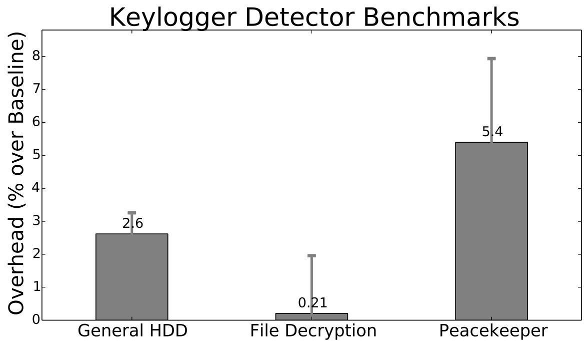

Figure 14: Benchmarks of the keylogger detector running on a Windows 7 VM.

that the Actual Keylogger does not log keystrokes from Internet Explorer.

Due to machine availability, the keylogger detection performance benchmarks were conducted on a different platform than the other monitors. The CPU was a 14 core Intel® Xeon™CPU E5-2683 v3 @ 2.00GHz and the machine had 256 GB of LR-DIMM DDR4 memory operating at 2133 MHz. To measure the performance impact of the keylogger detector, we ran two desktop benchmarks from PCMark05: the general HDD benchmark and the file decryption benchmark (Niemela 2005). To simulate a desktop application workload, we also ran the FutureMark Peacekeeper HTML5/Javascript browser benchmark with the Google Chrome web browser.[11] The mean and 95% confidence interval of the mean for the percent overhead of running 30 samples of each benchmark are plotted in Fig. 14.

From Fig. 14, we see that the overhead from the keylogger detector in those experiments is between 0-5%, with a large dispersion in the Peacekeeper browser benchmark. Even though the overhead was low, these measurements re-

[11] http://peacekeeper.futuremark.com/

flect an unrealistic worst-case scenario. The primary cause of overhead during these benchmarks was from re-enabling `CR3` VM Exits. As a performance improvement for production systems we could disable `CR3` VM Exits in between keystrokes after a certain threshold (e.g., when the contribution to R_k would be negligible). However, doing so would make benchmarking difficult as the performance impact would be more dependent on the typing rate of the user and at the time of writing, we did not have the ability to conduct performance tests involving users. We also note that for the VDI use case, performance is not critical as long as the system remains interactive.

7.4 Discussion

From Fig. 13, we see that the Keylogger detector has an optimal False Positive Rate of 0 for a threshold of mean $\bar{R} = 1.69$ yielding a True Positive Rate of 0.77. Since keyloggers are usually long-lived processes, we can take this conservative value and expect to detect the keylogger eventually. We recognize that we have only thoroughly evaluated the keylogger detection for Windows 7. For other OSes, both the threshold $\bar{R}$ and $t_{1/2}$ will likely change. One limitation of this keylogger monitor is that it only detects process-based keyloggers. There are other classes of keylogger software (e.g., kernel-based or DLL based), but process based are the most common as they are the easiest to deploy by attackers.

8. Related Work

Various hook-based VM monitoring systems are present in the literature (Quynh and Suzaki 2007; Payne et al. 2008; Sharif et al. 2009; Estrada et al. 2015). Lares and SIM (Payne et al. 2008; Sharif et al. 2009) overwrite function pointers for hooks whereas xenprobes and hprobes (Sharif et al. 2009; Estrada et al. 2015) use the same mechanism as this paper. A critical issue with hook-based VM monitoring systems is that one must know the address of where to place the hooks. If we wish to provide as-a-Service functionality, we cannot expect the user to provide addresses for function pointers or functions of interest. The framework in this paper builds on previous hook-based VM monitoring by demonstrating how one can infer hook locations based on OS design patterns.

Virtual Machine Introspection (VMI) techniques such as libVMI (Payne 2012) create a view of the guest OS by parsing its data structures after running code inside the guest OS (e.g., loading a kernel module that calculates kernel data structure addresses and offsets). Virtuoso (Dolan-Gavitt et al. 2011) and VMST (Fu and Lin 2012) run utilities, such as `ps` or `lsmod`, inside of a running VM. Virtuoso has a learning step, similar to our parameter inference, in which the binary (e.g., `ps`) is converted into a libVMI module. The cost of running libVMI modules is on the order of milliseconds or more (our hooks take microseconds). VMST runs utilities on a clone of the guest in a trusted VM while redirecting kernel memory reads to the untrusted guest. While the cloud user can replace the OS in the trusted VM with their own OS, the user still must execute those utilities. Despite some automation, all of libVMI, Virtuoso, and VMST appear to require the provider to run some code inside the user's VM.

Antfarm, Lycosid, and HyperTap all perform VM monitoring based on observing VM Exits (Jones et al. 2006, 2008; Pham et al. 2014). Those systems use VM Exits occurring during the normal course of VM operation (i.e., writes to the `CR3` register). However, it is not straightforward to extract the information needed by arbitrary detectors using only "natural" VM Exits, and the functionality of such monitoring is quite limited.

SecPod (Panneerselvam et al. 2015) is used to protect the guest kernel's page tables. Like the VM Exit techniques discussed above, SecPod ensures guest kernel space is protected by reconfiguring the hypervisor to trap on privileged operations. SecPod uses a secure address space for auditing the guest kernel's paging operations. This secure address space is transitioned to and from using special entry and exit gates that are added manually to the guest kernel. One could use the techniques presented in this paper to implement SecPod-like functionality without modifying the guest kernel. Parameter inference would discover all the paging functions and runtime monitoring would hook on those functions.

The ret2user detection in this paper can be viewed as a subset of SecVisor's functionality (Seshadri et al. 2007). SecVisor uses Nested Page Tables (NPT) from AMD to create isolated address spaces for kernel and userspace. SecVisor modifies the guest kernel by adding hypercalls to learn about guest OS operation. SecVisor was implemented standalone and is similar to Intel's kernel guard (Tseng 2015): SecVisor runs underneath a single guest OS and does not support multiple VMs. It cannot be used in an IaaS context (unless one uses performance costly nested virtualization techniques (Ben-Yehuda et al. 2010)).

9. Discussion and Limitations

Parameter inference using emulator-based dynamic analysis allows us to obtain useful monitoring information without requiring interaction with a user's VM. However, a production RSaaS solution may also take advantage of the information available from VMI and static analysis in addition to the information obtained from dynamic analysis.

The parameter inference and runtime monitoring steps are decoupled. As such, there is a certain level of trust in the guest OS. If the integrity of the VM is not protected, an attacker could avoid monitors by modifying the kernel code. In many cases, however, the monitoring hooks are triggered by expected guest OS events and the absence of events from those actions could be a sign that the system is under attack.

In our prototype, the DAF can be run offline, on a copy of the VM image, or online with a QEMU copy-on-write fork of the running VM image. If needed, the DAF's profiling can be based on the user's workload if it is configured to start at boot. Otherwise, one could live-migrate the VM after starting the workload to the emulator-based analysis environment for a brief profiling period (Wei et al. 2015). Note that using snapshotting or live migration, one can infer parameters that change across boot by performing dynamic analysis on every VM startup (e.g., in the presence of ASLR or drivers being loaded in non-deterministic order).

The VMF uses the `int3` instruction, a simple and robust hook-based VM monitoring technique that does incur VM Exit overhead. For environments that have more stringent performance requirements, one could use an in-VM hook-based monitoring platform that would require in-VM modifications (Panneerselvam et al. 2015; Sharif et al. 2009). We stress that our concept is independent of the specific hook mechanism.

While our targeted application domain was a VM-based cloud, our technique has broader applicability. In particular, large-scale virtualized environments are common in enterprise IT. Our runtime adaptable approach to monitoring is amenable to enterprise IT as those environments often have stringent uptime requirements.

10. Conclusions and Future Work

We demonstrated how one could use OS design patterns along with hook-based monitoring and emulator-based dynamic analysis tools to build a Reliability and Security as-a-Service framework. We demonstrated how OS design patterns allow us to perform VM monitoring that is less intrusive by inferring necessary parameters from the guest OS. This technique allows for a fine-grained monitoring approach where a cloud provider could enforce per-VM reliability and security policies at runtime. The range of monitoring capabilities was highlighted by the example detectors in this paper, which ranged from logging to failure and attack detection and virtual desktop workloads.

An important continuation of this work would be implementing interfaces for the provider and customer and defining Service-Level Agreements (SLAs) to address what guarantees a customer can expect from the provider. The provider could also use existing research on optimal monitor placement to offer insight on which systems to monitor in a multi-VM architecture (Chaudet et al. 2005; Jackson et al. 2007; Talele et al. 2014).

Acknowledgments

The authors wish to thank Paolo Bonzini of Red Hat for help with KVM's address space handling code and Phuong Cao for help with keylogger detection. The authors also wish to acknowledge assistance in preparation of this manuscript from Jenny Applequist and Arjun P. Athreya. This material is based upon work supported in part by by the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement No. FA8750-11-2-0084, by an IBM faculty award, by Infosys Corporation, and by the National Science Foundation Graduate Research Fellowship Program under Grant Number DGE-1144245.

References

F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The turtles project: Design and implementation of nested virtualization. In *OSDI*, volume 10, pages 423–436, 2010.

P. Cao, E. Badger, Z. Kalbarczyk, R. Iyer, and A. Slagell. Preemptive intrusion detection: Theoretical framework and real-world measurements. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, page 5. ACM, 2015.

M. Carbone, A. Kataria, R. Rugina, and V. Thampi. Vprobes: Deep observability into the ESXi hypervisor. *vmware Technical Journal*, 14(5):35–42, 2014.

C. Chaudet, E. Fleury, I. G. Lassous, H. Rivano, and M.-E. Voge. Optimal positioning of active and passive monitoring devices. In *Proceedings of the 2005 ACM conference on Emerging network experiment and technology*, pages 71–82. ACM, 2005.

J. Corbet. (nearly) full tickless operation in 3.10. Online, http://lwn.net/Articles/549580/, 2013.

Z. Deng, X. Zhang, and D. Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 289–298, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2015-3. doi: 10.1145/2523649.2523675. URL http://doi.acm.org/10.1145/2523649.2523675.

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

Z. J. Estrada, C. Pham, F. Deng, L. Yan, Z. Kalbarczyk, and R. K. Iyer. Dynamic vm dependability monitoring using hypervisor probes. In *Dependable Computing Conference (EDCC), 2015 Eleventh European*, pages 61–72. IEEE, 2015.

Y. Fu and Z. Lin. Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 586–600. IEEE, 2012.

T. Garfinkel, M. Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, volume 3, pages 191–206, 2003.

Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu. Face-change: Application-driven dynamic kernel view switching in a virtual machine. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 491–502. IEEE, 2014.

A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 248–258. ACM, 2014.

D. W. Hill and J. T. Lynn. Adaptive system and method for responding to computer network security attacks, July 11 2000. US Patent 6,088,804.

A. W. Jackson, W. Milliken, C. Santiváñez, M. Condell, W. T. Strayer, et al. A topological analysis of monitor placement. In *Network Computing and Applications, 2007. NCA 2007. Sixth IEEE International Symposium on*, pages 169–178. IEEE, 2007.

X. Jiang and X. Wang. "out-of-the-box" monitoring of VM-based high-interaction honeypots. In *Recent Advances in Intrusion Detection*, pages 198–218. Springer, 2007.

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*, pages 1–14, 2006.

S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based hidden process detection and identification using lycosid. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 91–100. ACM, 2008.

S. Keil and C. Kolbitsch. Kernel-mode exploits primer. Technical report, Technical report, International Secure Systems Lab (isecLAB), 2007.

A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *In Proc. of the Linux Symposium*, volume 1, pages 225–230, 2007.

R. Krishnakumar. Kernel korner: kprobes-a kernel debugger. *Linux Journal*, 2005(133):11, 2005.

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

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

P. Mell and T. Grance. The NIST definition of cloud computing. 2011.

J. Nielsen. Response times: The 3 important limits. *Usability Engineering*, 1993.

S. Niemela. Pcmark05 pc performance analysis. *White Paper from FutureMark Corp*, 2005.

S. Ortolani, C. Giuffrida, and B. Crispo. Bait your hook: A novel detection technique for keyloggers. In *RAID*, pages 198–217. Springer, 2010.

S. Panneerselvam, M. Swift, and N. S. Kim. Bolt: Faster reconfiguration in operating systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 511–516, Santa Clara, CA, July 2015. USENIX Association. ISBN 978-1-931971-225. URL https://www.usenix.org/conference/atc15/technical-session/presentation/panneerselvam.

B. D. Payne. Simplifying virtual machine introspection using libvmi. *Sandia Report*, 2012.

B. D. Payne, M. De Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proc. 23rd Ann. Computer Security Applications Conf. (ACSAC) 2007.*, pages 385–397. IEEE, 2007.

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*, pages 233–247. IEEE, 2008.

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*, pages 13–24. IEEE, 2014.

N. Provos. Improving host security with system call policies. In *Usenix Security*, volume 3, page 19, 2003.

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

D. Rosenberg. Smep: What is it, and how to beat it on linux. Online, http://vulnfactory.org/blog/2011/06/05/smep-what-is-it-and-how-to-beat-it-on-linux/, 2011.

A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *ACM SIGOPS Operating Systems Review*, 41(6):335–350, 2007.

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*, CCS '09, pages 477–487, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0. doi: 10.1145/1653662.1653720.

A. Shishkin and I. Smit. Bypassing intel smep on windows 8 x64 using return-oriented programming. Online, http://blog.ptsecurity.com/2012/09/bypassing-intel-smep-on-windows-8-x64.html, 2012.

S. Siddha, V. Pallipadi, and A. Ven. Getting maximum mileage out of tickless. In *Proceedings of the Linux Symposium*, volume 2, pages 201–207. Citeseer, 2007.

S. Suneja, C. Isci, E. de Lara, and V. Bala. Exploring VM introspection: Techniques and trade-offs. In *ACM SIGPLAN Notices*, volume 50, pages 133–146. ACM, 2015.

N. Talele, J. Teutsch, R. Erbacher, and T. Jaeger. Monitor placement for large-scale systems. In *Proceedings of the 19th ACM symposium on Access control models and technologies*, pages 29–40. ACM, 2014.

K.-l. Tseng. Intel kernel guard technology. Online, https://01.org/intel-kgt, 2015.

S. J. Vaughan-Nichols. Ubuntu linux continues to rule the cloud. Online, http://www.zdnet.com/article/ubuntu-linux-continues-to-rule-the-cloud/, 2015.

J. Wei, L. K. Yan, and M. A. Hakim. Mose: Live migration based on-the-fly software emulation. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 221–230, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3682-6. doi: 10.1145/2818000.2818022. URL http://doi.acm.org/10.1145/2818000.2818022.