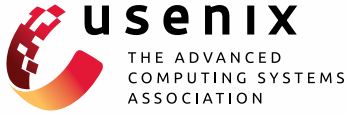

Hyperprobe: Towards Virtual Machine Extrospection

Jidong Xiao, ***College of William and Mary;*** **Lei Lu,** ***VMware Inc.;***
Hai Huang, ***IBM T. J. Watson Research Center;*** **Haining Wang,** ***University of Delaware***

https://www.usenix.org/conference/lisa15/conference-program/presentation/xiao

This paper is included in the Proceedings of the 29th Large Installation System Administration Conference (LISA15).

November 8–13, 2015 • Washington, D.C.

ISBN 978-1-931971-270

Open access to the Proceedings of the 29th Large Installation System Administration Conference (LISA15) is sponsored by USENIX

Hyperprobe: Towards Virtual Machine Extrospection

Jidong Xiao
College of William and Mary
jxiao@email.wm.edu

Lei Lu
VMware Inc.
llei@vmware.com

Hai Huang
IBM T.J. Watson Research Center
haih@us.ibm.com

Haining Wang
University of Delaware
hnw@udel.edu

Abstract

In a virtualized environment, it is not difficult to retrieve guest OS information from its hypervisor. However, it is very challenging to retrieve information in the reverse direction, i.e., retrieve the hypervisor information from within a guest OS, which remains an open problem and has not yet been comprehensively studied before. In this paper, we take the initiative and study this reverse information retrieval problem. In particular, we investigate how to determine the host OS kernel version from within a guest OS. We observe that modern commodity hypervisors introduce new features and bug fixes in almost every new release. Thus, by carefully analyzing the seven-year evolution of Linux KVM development (including 3485 patches), we can identify 19 features and 20 bugs in the hypervisor detectable from within a guest OS. Building on our detection of these features and bugs, we present a novel framework called Hyperprobe that for the first time enables users in a guest OS to automatically detect the underlying host OS kernel version in a few minutes. We implement a prototype of Hyperprobe and evaluate its effectiveness in five real world clouds, including Google Compute Engine (a.k.a. Google Cloud), HP Helion Public Cloud, ElasticHosts, Joyent Cloud, and CloudSigma, as well as in a controlled testbed environment, all yielding promising results.

1 Introduction

As virtualization technology becomes more prevalent, a variety of security methodologies have been developed at the hypervisor level, including intrusion and malware detection [26, 30], honeypots [48, 31], kernel rootkit defense [42, 40], and detection of covertly executing binaries [36]. These security services depend on the key factor that the hypervisor is isolated from its guest OSes. As the hypervisor runs at a more privileged level than its guest OSes, at this level, one can control physical resources, monitor their access, and be isolated from tampering against attackers from the guest OS. Monitoring of fine-grained information of the guest OSes from the underlying hypervisor is called virtual machine introspection (VMI) [26]. However, at the guest OS level retrieving information about the underlying hypervisor becomes very challenging, if not impossible. In this paper, we label the reverse information retrieval with the coined term virtual machine extrospection (VME). While VMI has been widely used for security purposes during the past decade, the reverse direction VME—the procedure that retrieves the hypervisor information from the guest OS level—is a new topic and has not been comprehensively studied before.

VME can be critically important for both malicious attackers and regular users. On one hand, from the attackers' perspective, when an attacker is in control of a virtual machine (VM), either as a legal resident or after a successful compromise of the victim's VM, the underlying hypervisor becomes its attacking target. This threat has been demonstrated in [35, 21], where an attacker is able to mount a privilege escalation attack from within a VMware virtual machine and a KVM-based virtual machine, respectively, and then gains some control of the host machine. Although these works demonstrate the possibility of such a threat, successful escape attacks from the guest to the host are rare. The primary reason is that most hypervisors are, by design, invisible to the VMs. Therefore, even if an attacker gains full control of a VM, a successful attempt to break out of the VM and break into the hypervisor requires an in-depth knowledge of the underlying hypervisor, e.g., type and version of the hypervisor. However, there is no straightforward way for attackers to obtain such knowledge.

On the other hand, benign cloud users may also need to know the underlying hypervisor information. It is commonly known that hardware and software systems both have various bugs and vulnerabilities, and different hardware/software may exhibit different vulnerabilities. Cloud customers, when making decisions on the choice of a cloud provider, may want to know more informa-

tion about the underlying hardware or software. This will help customers determine whether the underlying hardware/software can be trusted, and thus help them decide whether or not to use this cloud service. However, for security reasons, cloud providers usually do not release such sensitive information to the public or customers.

Whereas research efforts have been made to detect the existence of a hypervisor [25, 22, 24, 50], from a guest OS, to the best of our knowledge, there is no literature describing how to retrieve more detailed information about the hypervisor, e.g., the kernel version of the host OS, the distribution of the host OS (Fedora, SuSE, or Ubuntu?), the CPU type, the memory type, or any hardware information. In this paper, we make an attempt to investigate this problem. More specifically, as a first step towards VME, we study the problem of detecting/inferring the host OS kernel version from within a guest OS, and we expect our work will inspire more attention on mining the information of a hypervisor. The major research contributions of our work are summarized as follows:

- We are the first to study the problem of detecting/inferring the host OS kernel version from within a VM. Exploring the evolution of Linux KVM hypervisors, we analyze various features and bugs introduced in the KVM hypervisor; and then we explain how these features and bugs can be used to detect/infer the hypervisor kernel version.
- We design and implement a novel, practical, automatic, and extensible framework, called Hyperprobe, for conducting the reverse information retrieval. Hyperprobe can help users in a VM to automatically detect/infer the underlying host OS kernel version in less than five minutes with high accuracy.
- We perform our experiments in five real world clouds, including Google Compute Engine [3], HP Helion Public Cloud [29], ElasticHosts [20], Joyent Cloud [8], and CloudSigma [19], and our experimental results are very promising. To further validate the accuracy of Hyperprobe, we perform experiments in a controlled testbed environment. For 11 of the 35 kernel versions we studied, Hyperprobe can correctly infer the exact version number; for the rest, Hyperprobe can narrow it down to within 2 to 5 versions.

2 Background

Hypervisor, also named as virtual machine monitor, is a piece of software that creates and manages VMs. Traditionally, hypervisors such as VMware and Virtual PC use the technique of binary translation to implement virtualization. Recently, x86 processor vendors including Intel and AMD released their new architecture extensions to support virtualization. Those hypervisors that use binary translation are called software-only hypervisors, and recent hypervisors that take advantage of these processor extensions are called hardware assisted hypervisors [12]. In this paper, we focus on a popular hardware assisted commodity hypervisor, Linux KVM. We develop our framework and perform experiments on a physical machine with Linux OS as the host, which runs a KVM hypervisor, and a VM is running on top of the hypervisor. Our study covers Linux kernel versions from 2.6.20 to 3.14. While 2.6.20, released in February 2007, is the first kernel version that includes KVM, 3.14, released in March 2014, is the latest stable kernel at the time of this study. More specifically, we study the evolution of KVM over the past seven years and make three major observations. In this section, we briefly describe Linux KVM and report our observations.

2.1 Linux KVM

KVM refers to kernel-based virtual machine. Since Linux kernel version 2.6.20, KVM is merged into the Linux mainline kernel as a couple of kernel modules: an architecture independent module called kvm.ko, and an architecture dependent module called either kvm-intel.ko or kvm-amd.ko. As a hardware assisted virtualization technology, KVM relies heavily on the support of the underlying CPUs and requires different implementations for different CPU vendors, such as Intel VT-x and AMD SVM. Figure 1 illustrates the basic architecture of KVM. KVM works inside a host kernel and turns the host kernel into a hypervisor. On top of the hypervisor, there can be multiple VMs. Usually KVM requires a user-level tool called Qemu to emulate various devices, and they communicate using predefined ioctl commands.

Over the years, KVM has changed significantly. The original version in 2.6.20 consists of less than 20,000 lines of code (LOC); but in the latest 3.14 version, KVM modules consist of about 50,000 LOC. The reason of such growth is that 3485 KVM related patches have been released by Linux mainline kernel[1]. By carefully analyzing these patches, we make a few important observations about the evolution of the KVM development process.

First, while ideally hypervisors should be transparent to guest OSes, this is not realistic. In particular, during its development process, on the one hand, KVM exposes more and more processor features to a guest OS; on the other hand, KVM has been provided with many paravirtualization features. These changes improve performance but at the cost of less transparency.

Second, for the sake of better resource utilization, KVM has also included several virtualization-specific features, e.g., nested virtualization [16] and kernel same

[1] KVM has recently started supporting non-x86 platform, such as ARM and PPC; however, in this study, we only consider patches for x86 platforms, i.e., the number 3485 does not include the patches for the non-x86 platforms.

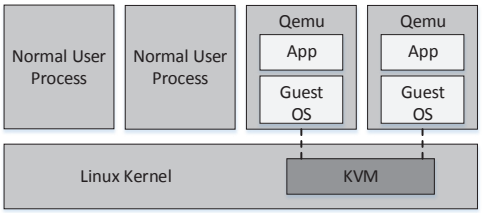

Figure 1: KVM Overview

page merging (KSM) [15], many of which can be detected from within the guest OS.

Third, similar to all other large projects, KVM have bugs. Among the 3485 patches, about 30% of them are bug fixes. In particular, we notice that a common type of bugs in KVM is related to registers. This reflects the fact that emulating a CPU is hard. Since a modern CPU defines hundreds of registers, emulating the behaviors of various registers correctly is challenging. Failing to do so usually causes various unexpected results. In fact, register related bugs have been reported on a regular basis.

During our study, we discover that these features and bugs can help us determine the underlying hypervisor kernel version. A more detailed description of our design approach is presented in Section 3.

2.2 Intel VT-x Extension

As a hardware assisted hypervisor, KVM relies on the virtualization extensions of the underlying processors. In 2006, both Intel (VT-x) and AMD (AMD-SVM) introduced hardware virtualization extensions in their x86 processors. According to their respective manuals, these two technologies are very similar to each other. Because our current implementation of Hyperprobe is based on the Intel processors, we will briefly describe Intel VT-x.

The key concept of Intel VT-x is that the CPU is split into the root mode and the non-root mode. Generally, the hypervisor runs in the root mode and its guests run in the non-root mode. Transitions from the root mode to the non-root mode are called VM entries, and transitions from the non-root mode to the root mode are called VM exits. The hypervisor can specify which instructions and events cause VM exits. These VM exits actually allow the hypervisor to retain control of the underlying physical resources. An example of a VM exit is, when a guest OS attempts to access sensitive registers, such as control registers or debug registers, it would cause a VM exit. A handler defined by the hypervisor will then be invoked, and the hypervisor will try to emulate the behavior of the registers. As mentioned above, given the large number of registers, register emulation is hard and error-prone.

The first generation of Intel VT-x processors mainly simplifies the design of hypervisors. But since then, more and more features have been included in their later processor models. To name a few, Extended Page Table (EPT), which aims to reduce the overhead of address translation, is introduced by Intel since Nehalem processors, and VMCS shadow, which aims to accelerate nested virtualization, is introduced since Haswell. Once these new hardware features are released, modern hypervisors such as KVM and Xen, provide their support for these new features on the software side.

3 Design

Hyperprobe framework has the following goals:

- Practical: The framework should detect the underlying hypervisor kernel version within a reasonable amount of time with high accuracy and precision. As more test cases are added to provide more vantage points of different kernel versions, its accuracy and precision should also be improved.
- Automatic: The framework should run test cases, collect and analyze results automatically without manual intervention. To this end, the test cases should not crash the guest or host OS.[2]
- Extensible: The framework should be easily extended to detect/infer future Linux kernel versions and to add more vantage points to previously released kernel versions. To this end, the whole framework should be modular, and adding modules to the framework should be easy.[3]

3.1 Technical Challenges

To meet these design goals, we faced several challenges: even though the hypervisor introduces new features frequently, how many of them are detectable from within the guest OS? Similarly, how many hypervisor bugs are detectable from within the guest OS?

After manually analyzing the aforementioned 3485 patches, we found a sufficient number of features and bugs that meet our requirements. Tables 1 and 2 illustrate the features and bugs we have selected for our framework. To exploit each, it would require an in-depth knowledge of the kernel and also a good understanding of the particular feature/bug. Due to limited space, we are not able to explain each of the features/bugs, but we will choose some of the more interesting ones and explain them in the next section as case studies. In this section, we elaborate on how we use these features and bugs to infer the underlying hypervisor kernel version.

3.2 KVM Features

KVM releases new features regularly. One may infer the underlying hypervisor kernel version using the following

[2] Kernel bugs that cause guest or host OS to crash are very common, but we purposely avoided using them in our test cases. One could utilize these bugs to gain more vantage points, but they should be used with great caution.

[3] We plan to make Hyperprobe an open source project so that everyone can contribute, making it more robust and accurate.

Table 1: Features We Use in Current Implementation of Hyperprobe

Kernel Major Version	Features	Description
2.6.20		KVM first merged into Linux mainline kernel
2.6.21	Support MSR_KVM_API_MAGIC	Custom MSR register support
2.6.23	SMP support	Support multiple processors for guest OS
2.6.25	Expose KVM CPUID to guest	KVM_CPUID_SIGNATURE
2.6.26	EPT/NPT support	Extended/Nested Page Table
2.6.27	MTRR support	Support the memory type range registers for guest OS
2.6.30	Debug register virtualization	Add support for guest debug
2.6.31	POPCNT support	Support POPCNT instruction in guest OS
2.6.32	KSM support	Kernel Same Page Merging
2.6.34	RDTSCP support, Microsoft Enlightenment	Support RDTSCP instruction and Microsoft Enlightenment
2.6.35	New kvmclock interface	Support paravirtualized clock for the guest
2.6.38	Support MSR_KVM_ASYNC_PF_EN	Enable asynchronous page faults delivery
3.1	Add "steal time" guest/host interface	Enable steal time
3.2	Support HV_X64_MSR_APIC_ASSIST_PAGE	Support for Hyper-V lazy EOI processing
3.3	PMU v2 support	Expose a version 2 of Performance Monitor Units to guest
3.6	Support MSR_KVM_PV_EOI_EN	Support End of Interrupt Paravirtualization
3.10	Support preemption timer for guest	Support preemption timer for guest
3.12	Nested EPT	Expose Nested Extended Page Table to guest OS
3.13	Support Nested EPT 2MB pages	Expose 2MB EPT page to guest
3.14	Support HV_X64_MSR_TIME_REF_COUNT	Support for Hyper-V reference time counter

Table 2: Bugs We Use in Current Implementation of Hyperprobe

Fixed	Bug Description	Intro'ed
2.6.22	MSR_IA32_MCG_STATUS not writable	2.6.20
2.6.23	MSR_IA32_EBL_CR_POWERON not readable	2.6.20
2.6.25	MSR_IA32_MCG_CTL not readable	2.6.20
2.6.26	MSR_IA32_PERF_STATUS wrong return value upon read	2.6.20
2.6.28	MSR_IA32_MC0_MISC+20 not readable	2.6.20
2.6.30	MSR_VM_HSAVE_PA not readable	2.6.20
2.6.31	MSR_K7_EVNTSEL0 not readable	2.6.20
2.6.32	DR register unchecked access	2.6.20
2.6.34	No support for clear bit 10 of msr register MSR_IA32_MC0_CTL	2.6.20
2.6.35	No support for write 0x100 to MSR_K7_HWCR	2.6.20
2.6.37	MSR_EBC_FREQUENCY_ID not readable	2.6.20
2.6.39	MSR_IA32_BBL_CR_CTL3 not readable	2.6.20
3.2	MSR_IA32_UCODE_REV returns invalid value upon read	2.6.20
3.4	Write 0x8 to MSR_K7_HWCR is buggy	2.6.20
3.5	CPUID returns incorrect value for KVM leaf 0x4000000	2.6.25
3.8	MSR_IA32_TSC_ADJUST not readable	2.6.20
3.9	MSR_AMD64_BU_CFG2 not readable	2.6.20
3.10	MSR_IA32_VMX_ENTRY_CTLS is not set properly as per spec	3.1
3.12	MSR_IA32_FEATURE_CONTROL behave weirdly	3.1
3.14	MSR_IA32_APICBASE reserve bit is writable	2.6.20

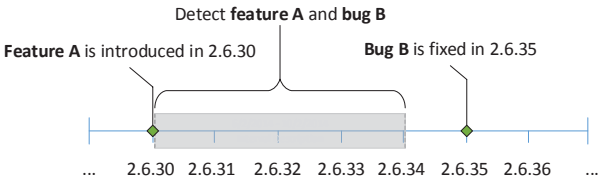

Figure 2: An Inferring Example of The Hyperprobe

logic: if feature *A* is introduced in 2.6.30 and feature *B* is introduced in 2.6.35, then if one can detect feature *A* but not *B*, one may infer that the underlying host kernel version is between 2.6.30 and 2.6.34. However, this may lead to inaccuracies. Since even if feature B is introduced into the Linux mainline kernel on a particular release, the feature could be disabled by system administrators. Therefore, even if feature *B* is not detected, it does not mean the underlying hypervisor kernel version is older than 2.6.35. Such customizations could impact precision.

To avoid such inaccuracies, Hyperprobe uses the following strategy to handle the existence or non-existence of a kernel feature: if we detect a feature exists, we assert that the underlying hypervisor kernel version is no older than the version in which this feature was first introduced. By designing test cases that detect these features, we report a minimum version number. *This number can be viewed as the lower bound of the underlying hypervisor kernel version.*

3.3 KVM Bugs and Bug Fixes

KVM has bugs and bug fixes like any other software. If bugs can be detected from within the guest OS, then one may infer the underlying hypervisor kernel version using the following logic: assuming bug *A* is fixed in kernel version 2.6.30, and bug *B* is fixed in kernel version 2.6.35. If one detects that bug *A* does not exist but bug *B* does, one may infer that the underlying hypervisor kernel is between 2.6.30 and 2.6.34. Similarly, this may lead to inaccuracies, as a bug could be manually fixed in an older kernel without updating the entire kernel. Therefore, the non-existence of a bug does not necessarily mean the kernel is newer than a particular version.

To avoid such inaccuracies, Hyperprobe uses the following strategy to handle the existence or non-existence of a kernel bug: if a bug is detected, we assert that the underlying kernel version is older than the kernel version where this bug is fixed. By creating test cases that detect kernel bugs, we report a maximum version num-

ber. *This number can be viewed as the upper bound of the underlying hypervisor kernel version.* Along with the test cases that detect kernel features, which can report a lower bound, we can then narrow down the hypervisor kernel to a range of versions. Figure 2 illustrates an example: upon the detection of feature A and bug B, we report that the hypervisor has kernel version 2.6.30 as the lower bound and 2.6.34 as the upper bound.

4 Implementation

Our framework implementation consists of 3530 lines of C code (including comments). To meet the extensible goal, we implement the framework of Hyperprobe in a very modular fashion. More specifically, we design 19 test cases for feature detection and 20 test cases for bug detection. Each test case is designed for detecting a specific feature or bug, and is therefore independent of any other test cases. On average, each test case consists of 80 lines of C code. Such a design model makes Hyperprobe fairly extensible. If we identify any other detectable features or bugs later, they can be easily added.

We define two linked lists, named *kvm_feature_testers* and *kvm_bug_testers*. The former includes all the feature test cases, and the latter includes all the bug test cases. Each feature test case corresponds to a kernel version number, which represents the kernel in which the feature is introduced. The feature test cases are sorted using this number and the bug test cases are organized similarly.

Hyperprobe executes as follows. The detection algorithm involves two steps. First, we call the feature test cases in a descending order. As soon as a feature test case returns true, which suggests the feature exists, we stop the loop and report the corresponding number as the lower bound. Second, we call the bug test cases in an ascending order. As soon as a bug test case returns true, which suggests the bug exists, we stop the loop and report the corresponding number as the upper bound.

Most hypervisor kernel features and bugs that we have chosen in this study can be easily detected within the guest OS. In what follows, we describe some of the more interesting ones as case studies.

4.1 Case Studies: Kernel Features

4.1.1 Kernel Samepage Merging

Kernel samepage merging (KSM) [15], introduced in Linux kernel 2.6.32, is a mechanism to save memory, allowing memory overcommitment. This is a crucial feature in a virtualized environment, where there could be a large number of similar VMs running on top of one hypervisor. Other popular hypervisors, such as Xen and VMware, have also implemented similar features [49, 27]. Consequently, if we can detect KSM is enabled we can ascertain that the underlying hypervisor kernel is newer than or equal to version 2.6.32.

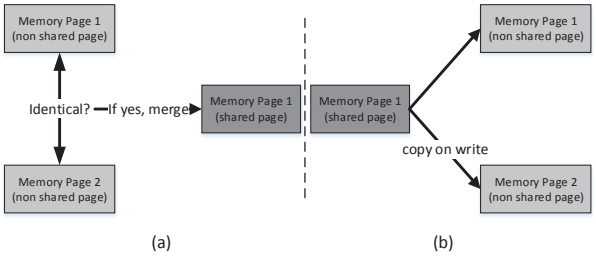

Figure 3: Kernel Same Page Merging
(a) merging identical pages (b) a copy-on-write technique is used when a shared page is modified

KSM scans memory pages and merges those that are identical. Merged pages are set to be copy-on-write, illustrated in Figure 3. This technique is widely used, and it has been proven to be effective in saving memory. However, due to copy-on-write, a write to a shared page incurs more time than a write to a non-shared page. Existing research [46, 50] has shown that this timing difference is large enough to tell if KSM is enabled.

Algorithm 1 describes the procedure of testing KSM. The basic idea of this algorithm is as follows. We first load a random file into memory and write to each page of this file (in memory), then we record the accumulated write access time and call this time *t1*. Next, we load this file again into two separate memory regions, and wait for some time. If KSM is enabled, the identical pages between these two files will be merged. We then write into each page of this file (in memory), and record the accumulated write access time as *t2*. If *t2* is significantly larger than *t1*, namely, the ratio t2/t1 is greater than a predefined threshold, we assume KSM is enabled; otherwise, we assume it is not enabled. In fact, in our testbed, we observe that *t2* is as much as 10 times larger than *t1*. Even in five real cloud environments, we observe that *t2* is still 2 to 5 times larger than *t1*. Thus, we choose 2 as the threshold to detect if KSM is enabled or not.

4.1.2 Extended Page Table (EPT)

Traditionally, commercial hypervisors including KVM, Xen, and VMware, all use the shadow page table technique to manage VM memory. The shadow page table is maintained by the hypervisor and stores the mapping between guest virtual address and machine address. This mechanism requires a serious synchronization effort to make the shadow page table consistent with the guest page table. In particular, when a workload in the guest OS requires frequent updates to the guest page tables, this synchronization overhead can cause very poor performance. To address this problem, recent architecture evolution in x86 processors presents the extended/nested page table technology (Intel EPT and AMD NPT). With this new technology, hypervisors do not need to main-

Algorithm 1: Detecting KSM

```
Global Var: file
Procedure test_ksm()
    load_file_once_into_memory (file);
    // record the clock time before we write
       to each page of the file
    time1 ← clock_gettime();
    foreach page of file in memory do
        write to that page;
    // record the clock time before we write
       to each page of the file
    time2 ← clock_gettime();
    t1 ← diff(time1,time2);
    load_file_twice_into_memory (file);
    // sleep and hope the two copies will be
       merged
    sleep (NUM_OF_SECONDS);
    // record the clock time before we write
       to each page of the file
    time1 ← clock_gettime();
    foreach page of file in memory do
        write to that page;
    // record the clock time after we write
       to each page of the file
    time2 ← clock_gettime();
    t2 ← diff(time1,time2);
    ratio ← t2/t1;
    if ratio > KSM_THRESHOLD then
        return 1;
    else
        return 0;
```

tain shadow page tables for the VMs, and hence avoid the synchronization costs of the shadow page table scenario. The difference between shadow page table and extended page table is illustrated in Figure 4.

Before kernel 2.6.26, KVM uses shadow page table to virtualize memory. Since kernel 2.6.26, KVM starts to support Intel EPT and enable it by default. Therefore, if we can detect the existence of EPT from within the guest OS, we can assume the underlying hypervisor kernel is newer than or equal to version 2.6.26. Algorithm 2 describes the EPT detection mechanism, and we derive this algorithm from the following observations:

- On a specific VM, no matter whether the underlying hypervisor is using shadow page table or EPT, the average time to access one byte in memory is very stable. We have measured this across 30 virtual machines (with different hardware and software configurations). Note that although the time cost may vary across different machines, it remains nearly the same when we switch from EPT to shadow page table, or from shadow page table to EPT.
- When running a benchmark that requires frequent memory mapping changes, EPT offers significant performance improvements over shadow page table. Particularly, we choose the classic forkwait microbenchmark, which has been widely employed [12, 13, 17],to evaluate virtualization performance. The main part of this benchmark repeats the operation of process creation and destruction very aggressively. Similar to [17], we have tested the forkwait microbenchmark across 30 VMs (with different hardware and software configurations), and have consistently observed that EPT offers approximately 600% performance gains over shadow page table.

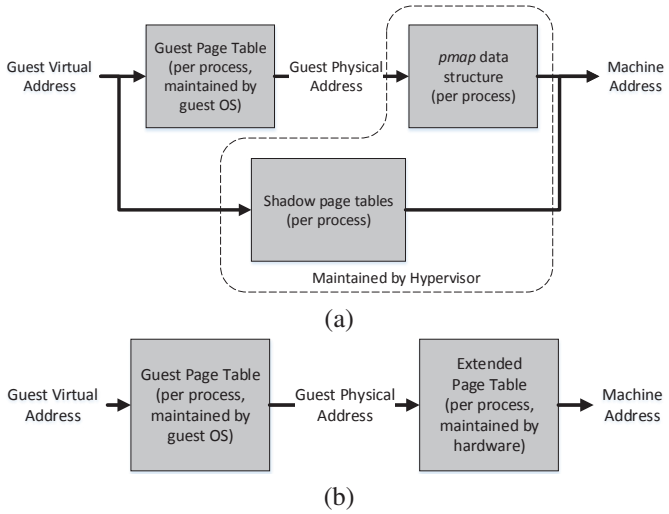

Figure 4: Shadow Page Table and Extended Page Table (a) shadow page table (b) extended page table

Therefore, our algorithm can be elaborated as follows. First we allocate a memory page, compute the average time to access one byte of the memory page, and use this average time as a baseline. Next, we run the forkwait microbenchmark, compute the average time to fork-wait one process, and record the ratio between these two average times (average time to fork-wait one process divided by average time to access one byte of memory page). On all VMs we have tested, this ratio is larger than 100,000 when the hypervisor is using shadow page table, and it is usually between 10,000 to 20,000 when the hypervisor is using EPT. Therefore, we can choose a threshold, and if the ratio is less than that threshold, we assume the underlying hypervisor is using EPT; otherwise, we assume it is using shadow page table. Our current implementation uses 30,000 as the threshold.

4.1.3 Emulating Hyper-V and Support Microsoft Enlightenment

Microsoft Enlightenment is an optimization made by Microsoft to Windows systems when running in a virtualized environment. The key idea is to let the guest OS be aware of the virtualized environment, and therefore tune its behavior for performance improvement. Recent Windows systems, such as Windows Server 2008, and Windows Vista, are fully enlightened [45, 43], which means

Algorithm 2: Detecting EPT

```
   Global Var: forkwait_one_process_avg, access_one_byte_avg
1  Procedure forkwait_one_process ()
       // read time stamp counter before we run
          the forkwait benchmark
2      counter1 ← rdtsc();
3      for i ← 0 to NUM_OF_PROCESS do
4          pid ← fork();
5          if pid = 0 then                    // child process
6              exit (0);
7          else
               // parent process, wait until
                  child process exits
8              wait (&status);
       // read time stamp counter when the
          forkwait benchmark is finished
9      counter2 ← rdtsc();
10     cycles ← counter2 − counter1;
       // compute average time for fork-waiting
          one process
11     forkwait_one_process_avg ← cycles/NUM_OF_PROCESS;
12 Procedure access_one_byte (iterations)
13     offset ← 0;
14     page ← malloc(sizeof(PAGE_SIZE));
       // read time stamp counter before we
          access memory bytes
15     counter1 ← rdtsc();
16     for i ← 0 to iterations do
17         page[offset] ← (page[offset] + 1) mod 256;
18         offset ← (offset + 1) mod PAGE_SIZE;
       // read time stamp counter after we
          access memory bytes
19     counter2 ← rdtsc();
20     cycles ← counter2 − counter1;
       // compute average time for accessing
          one byte
21     access_one_byte_avg ← cycles/iterations;
22 Procedure one_time_run()
23     access_one_byte(num_of_iterations);
24     forkwait_one_process();
25     ratio ← forkwait_one_process_avg/access_one_byte_avg;
26     if ratio < EPT_THRESHOLD then
27         return 1;
28     else
29         return 0;
30 Procedure test_ept()
31     for i ← 0 to LOOP_NUMBER do
32         if one_time_run() = 1 then
33             return 1 ;
34     return 0;
```

they take full advantage of the possible enlightenments.

Microsoft Enlightenment was originally designed for Hyper-V, but Microsoft provides APIs for other hypervisors to utilize this optimization. Since kernel 2.6.34, KVM has started utilizing these APIs and supporting Microsoft Enlightenment. According to the Hyper-V specification [6, 7], several synthetic registers are defined, including HV_X64_GUEST_OS_ID, HV_X64_HYPERCALL, HV_X64_VP_INDEX, as well as the EOI/TPR/ICR APIC registers. Details of these registers are shown in Table 3. Before kernel 2.6.34, accessing these registers would generate a general protection fault, but since kernel 2.6.34, they should be accessible whether accessing from a Windows or Linux guest OS. Thus, we attempt to access these registers. If they are accessible, we assume the kernel version is newer than or equal to version 2.6.34; otherwise, the feature may not be present, but we do not make any assertion regarding the underlying kernel version. In addition, in some later kernel versions, more Hyper-V defined synthetic registers are emulated by KVM. For example, HV_X64_MSR_TIME_REF_COUNT is emulated in kernel 3.14. Thus, successful access to the register suggests that the underlying hypervisor kernel should be as new as 3.14.

4.2 Case Studies: Kernel Bugs

4.2.1 Debug Register Unchecked Access

Debug registers are protected registers. They should only be accessed by ring 0 code, namely kernel code. However, before kernel 2.6.32, KVM does not check the privilege of the guest code that accesses the debug registers. Therefore, any process, regardless of its current privilege level (CPL), is able to read from and write to debug registers. This leads to a security issue in the guest OS, as attackers might be able to implement the infamous DR rootkit [14, 28] without installing a kernel module, thus making the rootkit more difficult to detect even from the hypervisor level.

On kernel 2.6.32, KVM maintainer, Avi Kivity, submitted a patch that fixed this bug. The patch would check the CPL before accessing debug registers, and would generate a fault if the CPL is greater than zero. We built a simple test case based on this bug. The basic idea is to use the fork system call to create a child process, and let the child process try to access a debug register. If the bug is fixed, the child process should be terminated by a segmentation fault signal. But if the bug has not yet been fixed, the child process will continue to run and eventually exit normally. Therefore, we let the parent process wait until the child process exits, and check the exit status of the child process. If it exits normally, we report the bug still exists; otherwise, we report the bug is fixed.

4.2.2 Model Specific Register (MSR) Bugs

CPU vendors such as Intel and AMD define hundreds of model specific registers on their processors. Some of these registers are common across different types of processors, while others might only exist in a specific processor. Due to the large variety of such registers, over the years, emulating the behavior of these registers has always been a painful task in modern hypervisors. Because of this, Andi Kleen, a key maintainer of Linux kernels, who used to be in charge of the x86_64 and i386 architectures, believes that it is impossible to emulate a

Table 3: Hyper-V Defined Synthetic Registers

Register Name	Address	Description	Supported in Linux Kernel Since
HV_X64_MSR_GUEST_OS_ID	0x40000000	Used to identify guest OS	2.6.34
HV_X64_MSR_HYPERCALL	0x40000001	Used to enable/disable Hypercall	2.6.34
HV_X64_MSR_VP_INDEX	0x40000002	Used to identify virtual processor	2.6.34
HV_X64_MSR_EOI	0x40000070	Fast access to APIC EOI register	2.6.34
HV_X64_MSR_ICR	0x40000071	Fast access to APIC ICR register	2.6.34
HV_X64_MSR_TPR	0x40000072	Fast access to APIC TPR register	2.6.34
HV_X64_MSR_APIC_ASSIST_PAGE	0x40000073	Used to enable/disable lazy EOI processing	3.2
HV_X64_MSR_TIME_REF_COUNT	0x40000020	Time reference counter	3.14

particular CPU 100% correctly [33].

However, incorrect emulation of these registers could cause problems in the guest OS. For example, to fix their hardware defects, Intel defines a capability in its Pentium 4, Intel Xeon, and P6 family processors called microcode update facility. This allows microcode to be updated if needed to fix critical defects. After microcode is updated, its revision number is also updated. BIOS or OS can extract this revision number via reading the MSR register IA32_UCODE_REV, whose address is 0x8BH. Previously, in Linux kernel, when the guest tries to read this register, KVM would return an invalid value, which is 0, and this would cause Microsoft Windows 2008 SP2 server to exhibit the blue screen of death (BSOD). To fix this problem, since kernel 3.2, KVM reports a non-zero value when reading from IA32_UCODE_REV. Details of this bug fix can be found in [47].

Our detection is also straightforward: Linux kernel provides a kernel module called *msr* that exports an interface through file /dev/cpu/cpuN/msr, where N refers to the CPU number. This interface allows a user level program to access MSR registers. Therefore, we can detect the bug by accessing this file with the address of IA32_UCDOE_REV, which is 0x0000008b according to Intel's manual. If a read to this register returns 0, we can assert that the bug exists.

5 Evaluation

To demonstrate how Hyperprobe performs in the wild, we ran its test suite on VMs provisioned from different public cloud providers to detect their hypervisor kernel versions. In most cases, we were able to narrow the suspected hypervisor kernel versions down to a few; in one case, we even had an exact match. However, as public cloud providers do not disclose detailed information about the hypervisors they are using (for obvious security reasons), we had to find other means to confirm these results, such as user forums and white papers. Our results do coincide with what are being reported via these side channels. To more rigorously verify the accuracy of Hyperprobe, we also evaluated it in a controlled testbed environment across 35 different kernel versions with very encouraging results.

5.1 Results in Real World Clouds

The public cloud providers we selected in this study include Google Compute Engine, HP Helion Public Cloud, ElasticHosts, Joyent Cloud, and CloudSigma. (all KVM-based) In our experiments, we intentionally created VMs with different configurations to test the detection robustness and accuracy of our framework. The results are shown in Tables 4, 5, 6, 7, and 8. Running the test suite and analyzing the collected results take less than 5 minutes to complete, which is fairly reasonable from a practical point of view. In fact, we observe that the running time is mainly dominated by those test cases that require sleeping or running some microbenchmarks. In what follows, we detail our findings for each cloud provider.

5.1.1 Google Compute Engine

Google Compute Engine is hosted in data centers located in Asia, Europe, and America. One can choose the number of VCPUs per VM ranging from 1 to 16. Hyperprobe shows that Google is using a kernel version between 3.2 and 3.3 in its hypervisors. According to a recent work [37] and some online communications written by Google engineers [32, 1], Debian 7 is most likely used in its hypervisors as this Linux distribution is widely used in its production environments. The default kernel of Debian 7 is 3.2.0-4, agreeing with our findings.

5.1.2 HP Helion Public Cloud

HP Helion Public Cloud is hosted in data centers in U.S. East and West regions. One can choose the number of VCPUs per VM ranging from 1 to 4. Hyperprobe detected that the HP cloud is using a kernel version between 3.2 and 3.7 in its hypervisors. According to some unofficial online documents and web pages [4, 5], HP is most likely using Ubuntu 12.04 LTS server as its host OS. The default kernel of Ubuntu 12.04 LTS is 3.2, falling within the range reported by our framework.

5.1.3 ElasticHosts

ElasticHosts is the first public cloud service provider to use Linux-KVM as its hypervisors [2]. Its data centers are located in Los Angeles, CA and San Antonio, TX. For free trial users, a VM with only 1 VCPU and 1GB of memory is given. Hyperprobe reported that the underlying hypervisor kernel version should be 3.6 to 3.8.

Table 4: Inferring Host Kernel Version in Google Compute Engine

VM Name	Zone	Machine Type	Image	VCPU	VCPU Frequency	RAM	Disk	Min	Max
gg-test1	asia-east1-a	n1-standard-1	SUSE SLES 11 SP3	1	2.50GHZ	3.8GB	10G	3.2	3.3
gg-test2	asia-east1-b	n1-highcpu-16	SUSE SLES 11 SP3	16	2.50GHZ	14.4GB	10G	3.2	3.3
gg-test3	us-central1-a	n1-highmem-16	Debian 7 wheezy	16	2.60GHZ	8GB	104G	3.2	3.3
gg-test4	us-central1-b	f1-micro	backports Debian 7 wheezy	1	2.60GHZ	4GB	0.6G	3.2	3.3
gg-test5	europe-west1-a	n1-highmem-4	backports Debian 7 wheezy	4	2.60GHZ	26GB	10G	3.2	3.3
gg-test6	europe-west1-b	n1-standard-4	Debian 7 wheezy	4	2.60GHZ	15GB	10G	3.2	3.3

Table 5: Inferring Host Kernel Version in HP Helion Cloud (3 Month Free Trial)

VM Name	Region	Zone	Size	Image	VCPU	VCPU Frequency	RAM	Disk	Min	Max
hp-test1	US East	az2	standard xsmall	SUSE SLES 11 SP3	1	2.4GHZ	1GB	20G	3.2	3.7
hp-test2	US East	az2	standard xlarge	SUSE SLES 11 SP3	4	2.4GHZ	15GB	300G	3.2	3.7
hp-test3	US East	az3	standard large	SUSE SLES 11 SP3	4	2.4GHZ	8GB	160G	3.2	3.7
hp-test4	US East	az1	standard medium	SUSE SLES 11 SP3	2	2.4GHZ	4GB	80G	3.2	3.7
hp-test5	US West	az1	standard medium	Ubuntu 10.04	2	2.4GHZ	4GB	80G	3.2	3.7
hp-test6	US West	az3	standard xlarge	Debian Wheezy 7	4	2.4GHZ	15GB	300G	3.2	3.7

Table 6: Inferring Host Kernel Version in ElasticHosts Cloud (5 Day Free Trial)

VM Name	Location	Image	VCPU (Only 1 allowed for free trial)	RAM	Disk	Min	Max
eh-test1	Los Angeles	Ubuntu 13.10	2.8GHz	1GB	10GB	3.6	3.8
eh-test2	Los Angeles	Cent OS Linux 6.5	2.8GHz	512MB	5GB SSD	3.6	3.8
eh-test3	Los Angeles	Debian Linux 7.4	2.8GHz	512MB	5GB	3.6	3.8
eh-test4	Los Angeles	Ubuntu 14.04 LTS	2.8GHz	1GB	10GB	3.6	3.8
eh-test5	San Antonio	Ubuntu 12.04.1 LTS	2.5GHz	1GB	5GB	3.6	3.8
eh-test6	San Antonio	CentOS Linux 6.5	2.5GHz	512MB	10GB	3.6	3.8

For this provider, we were not able to find information to confirm if our finding is correct.

5.1.4 Joyent Cloud

Joyent Cloud is yet another IaaS cloud service provider that uses KVM as its hypervisors [11]. Its data centers are located in U.S. East, West, and Southwest regions, as well as in Amsterdam, Netherlands. It provides a one-year free trial with very limited resources (i.e., 0.125 VCPU and 256MB of memory). Hyperprobe reported that the hypervisors hosting the free trial machines are using a rather old 2.6.34 kernel (an exact match).

Further investigation showed that Joyent runs a custom kernel called SmartOS in its hypervisors. It was created based on Open Solaris and Linux KVM, and we confirmed that Linux 2.6.34 is the version that Joyent engineers have ported into SmartOS [18].

5.1.5 CloudSigma

CloudSigma is an IaaS cloud service provider based in Zurich, Switzerland. However, its data centers are located in Washington, D.C. and Las Vegas, NV. For free trial users, only one VCPU with 2GB of memory can be obtained. Hyperprobe reported that the underlying hypervisor kernel version should be between 3.6 and 3.13.

The main reason that CloudSigma's result spans a wider range than others is its usage of AMD processors in its data centers. Our current implementation of Hyperprobe is optimized only for Intel processors. KVM includes an architecture dependent module, namely *kvm-intel.ko* and *kvm-amd.ko*, for Intel and AMD, respectively. Although some features and bugs are common in both architectures, others may not be. And these architecture-specific features and bugs can further improve the accuracy of Hyperprobe's reported results. The result for CloudSigma was mainly based on the common features and bugs, and thus, Hyperprobe was not able to narrow down the kernel versions as much as it could for the Intel-based cloud providers.

5.1.6 Summarizing Findings in Public Clouds

We found several interesting facts about these clouds:

- Even if a cloud provider has multiple data centers spread across various geographic locations, it is very likely that they are using the same kernel version and distribution. This confirms the conventional wisdom that standardization and automation are critical to the maintainability of an IT environment as it grows more complex. Modern cloud providers' data centers are as complicated as they can get.
- Cloud providers usually do not use the latest kernel. At the time of our study, the latest stable Linux kernel is version 3.14, which was released in March 2014, and our experiments were performed in June 2014. However, we can see cloud providers like HP and ElasticHosts are still using kernels older than version 3.8, which was released in February 2013. Google and Joyent Cloud are using even older kernels. This is understandable as newer kernels might not have been extensively tested, and therefore, it could be risky to use them for production workloads.

5.2 Results in a Controlled Testbed

To better observe if what Hyperprobe detects is really what is deployed, we ran the same test suite in a con-

Table 7: Inferring Host Kernel Version in Joyent Cloud (1 Year Free Trial)

VM Name	Location	Image	VCPU (Only 1 allowed for free trial)	RAM	Disk	Min	Max
jy-test1	US-East	CentOS 6.5	3.07GHz	250MB	16GB	2.6.34	2.6.34
jy-test2	US-SouthWest	Ubuntu Certified 14.04	2.40GHz	250MB	16GB	2.6.34	2.6.34
jy-test3	US-West	CentOS 6.5	2.40GHz	250MB	16GB	2.6.34	2.6.34
jy-test4	EU-Amsterdam	Ubuntu Certified 14.04	2.40GHz	250MB	16GB	2.6.34	2.6.34

Table 8: Inferring Host Kernel Version in CloudSigma (7 Day Free Trial)

VM Name	Location	Image	VCPU (Only 1 allowed for free trial)	RAM	Disk	Min	Max
cs-test1	Washington DC	CentOS 6.5 Server	2.5GHz	2GB	10GB SSD	3.6	3.13
cs-test2	Washington DC	Fedora 20 Desktop	2.5GHz	1GB	10GB SSD	3.6	3.13
cs-test3	Washington DC	Debian 7.3 Server	2.5GHz	512MB	10GB SSD	3.6	3.13
cs-test4	Washington DC	SUSE SLES 11 SP3	2.5GHz	2GB	10GB SSD	3.6	3.13

trolled testbed environment across all the 35 major Linux kernel releases (2.6.20 to 3.14) since KVM was first introduced. The testbed is a Dell Desktop (with Intel Xeon 2.93GHz Quad-Core CPU and 2GB memory) running OpenSuSE 11.4. We used OpenSuSE 11.4 as the guest OS running a 3.14 Linux kernel. We manually compiled each of the 35 kernels and deployed it as the kernel used in our hypervisor. After each set of experiments, we shut down the guest OS and rebooted the host OS.

The results are listed in Table 9. To sum up, from Table 9, it can be seen that, among the 35 host OS kernel versions, we can find an exact match for 11 of them; for 15 of them, we can narrow down to 2 versions; for 4 of them, we can narrow down to 3 versions; for 4 of them, we can narrow down to 4 versions; and for 1 of them, we can narrow down to 5 versions.

Table 9: Inferring Results in a Controlled Testbed

Kernel	Reported Min	Reported Max	Accuracy
2.6.20	2.6.20	2.6.21	2 versions
2.6.21	2.6.21	2.6.21	exact match
2.6.22	2.6.21	2.6.22	2 versions
2.6.23	2.6.23	2.6.24	2 versions
2.6.24	2.6.23	2.6.24	2 versions
2.6.25	2.6.25	2.6.25	exact match
2.6.26	2.6.25	2.6.27	3 versions
2.6.27	2.6.27	2.6.27	exact match
2.6.28	2.6.27	2.6.29	3 versions
2.6.29	2.6.27	2.6.29	3 versions
2.6.30	2.6.30	2.6.30	exact match
2.6.31	2.6.31	2.6.31	exact match
2.6.32	2.6.32	2.6.33	2 versions
2.6.33	2.6.32	2.6.33	2 versions
2.6.34	2.6.34	2.6.34	exact match
2.6.35	2.6.35	2.6.36	2 versions
2.6.36	2.6.35	2.6.36	2 versions
2.6.37	2.6.35	2.6.38	4 versions
2.6.38	2.6.38	2.6.38	exact match
2.6.39	2.6.38	3.1	4 versions
3.0	2.6.38	3.1	4 versions
3.1	3.1	3.1	exact match
3.2	3.2	3.3	2 versions
3.3	3.3	3.3	exact match
3.4	3.3	3.4	2 versions
3.5	3.3	3.7	5 versions
3.6	3.6	3.7	2 versions
3.7	3.6	3.7	2 versions
3.8	3.6	3.8	3 versions
3.9	3.6	3.9	4 versions
3.10	3.10	3.11	2 versions
3.11	3.10	3.11	2 versions
3.12	3.12	3.13	2 versions
3.13	3.13	3.13	exact match
3.14	3.14	3.14	exact match

6 Discussion

In this section, we discuss some potential enhancements.

6.1 Other Hypervisors

Our current framework is developed for KVM, but the approach we propose should certainly work for other popular hypervisors such as Xen. In fact, we notice that KVM and Xen share many of the same features and bugs. For instance, They both support the Microsoft enlightenment feature, and we also notice that some MSR register bugs exist in both KVM and Xen. Therefore, we plan to include the support for Xen hypervisors in our framework.

Meanwhile, we are also trying to enhance our framework for closed-source hypervisors, such as VMware and Hyper-V. Even though their source codes are not available, the vendors provide a release note for each major release, which clearly states their new features. And the bugs of these hypervisors are also publicly available.

6.2 Open Source

We have implemented Hyperprobe as a framework, which includes different test cases, but each test case is totally separated from all the other test cases. In other words, each test case can be developed separately. Such a key property allows it to meet one of our design goals: extensible. In fact, we plan to make it open source, so that we can rely on a community of users to use it and contribute additional test cases. The more vantage points (i.e., test cases) we have, the better precision our detection result can achieve. And this will certainly accelerate our development process and our support for the other hypervisors.

7 Related Work

We survey related work in two categories: detection of a specific hypervisor and attacks against hypervisors.

7.1 Detection of Hypervisors

Since virtualization has been widely used for deploying defensive solutions, it is critical for attackers to be able to detect virtualization, i.e., detect the existence of a hypervisor. To this end, several approaches have been proposed for detecting the underlying hypervisors and are briefly described as follows.

RedPill [44] and Scooby Doo [34] are two techniques proposed to detect VMware, and they both work because VMware relocates some sensitive data structures such as Interrupt Descriptor Table (IDT), Global Descriptor Table (GDT), and Local Descriptor Table (LDT). Therefore, one can examine the value of the IDT base, if it exceeds a certain value or equals a specific hard-coded value, then one assumes that VMware is being used. However, these two techniques are both limited to VMware detection and are not reliable on machines with multi-cores [41]. By contrast, the detection technique proposed in [41] is more reliable but only works on Windows guest OSes. Their key observation is that because LDT is not used by Windows, the LDT base would be zero in a conventional Windows system but non-zero in a virtual machine environment. Therefore, one can simply check for a non-zero LDT base on Windows and determine if it is running in VMware environment.

A variety of detection techniques based on timing analysis have also been proposed in [25, 23].The basic idea is that some instructions (e.g., RDMSR) are intercepted by hypervisors and hence their execution time is longer than that on a real machine. One can detect the existence of a hypervisor by measuring the time taken to execute these instructions. Note that all these previous works can only detect the presence of a hypervisor and/or its type, but none are able to retrieve more detailed information about the underlying hypervisor, such as its kernel version.

7.2 Attacks against Hypervisors

Modern hypervisors often have a large code base, and thus, are also prone to bugs and vulnerabilities. Considering a hypervisor's critical role in virtualized environments, it has been a particularly attractive target for attackers. Vulnerabilities in hypervisors have been exploited by attackers, as demonstrated in prior work [35, 21]. Perez-Botero et al. [39] characterized various hypervisor vulnerabilities by analyzing vulnerability databases, including SecurityFocus [10] and NIST's Vulnerability Database [9]. Their observation is that almost every part of a hypervisor could have vulnerabilities. Ormandy [38] classified the security threats against hypervisors into three categories: total compromise, partial compromise, and abnormal termination. A total compromise means a privilege escalation attack from a guest OS to the hypervisor/host. A partial compromise refers to information leakage. An abnormal termination denotes the shut down of a hypervisor caused by attackers. According to the definition above, gaining hypervisor information by Hyperprobe belongs to a partial compromise.

8 Conclusion

In this paper, we investigated the reverse information retrieval problem in a virtualized environment. More specifically, we coined the term virtual machine extrospection (VME) to describe the procedure of retrieving the hypervisor information from within a guest OS. As a first step towards VME, we presented the design and development of the Hyperprobe framework. After analyzing the seven-year evolution of Linux KVM development, including 35 kernel versions and approximately 3485 KVM related patches, we implemented test cases based on 19 hypervisor features and 20 bugs. Hyperprobe is able to detect the underlying hypervisor kernel version in less than five minutes with a high accuracy. To the best of our knowledge, we are the first to study the problem of detecting host OS kernel version from within a VM. Our framework generates promising results in five real clouds, as well as in our own testbed.

References

[1] Bringing debian to google compute engine. http://googleappengine.blogspot.com/2013/05/bringing-debian-to-google-compute-engine_9.html.

[2] Elastichosts wiki page. http://en.wikipedia.org/wiki/ElasticHosts.

[3] Google compute engine. https://cloud.google.com/products/compute-engine/.

[4] Hp cloud os faqs. http://docs.hpcloud.com/cloudos/prepare/faqs/.

[5] Hp cloud os support matrix for hardware and software. http://docs.hpcloud.com/cloudos/prepare/supportmatrix/.

[6] Hypervisor top-level functional specification 2.0a: Windows server 2008 r2. http://www.microsoft.com/en-us/download/details.aspx?id=18673.

[7] Hypervisor top-level functional specification 3.0a: Windows server 2012. http://www.microsoft.com/en-us/download/details.aspx?id=39289.

[8] Joyent. http://www.joyent.com/.

[9] National vulnerability database. http://nvd.nist.gov/.

[10] Security focus. http://www.securityfocus.com/.

[11] Virtualization performance: Zones, kvm, xen. http://dtrace.org/blogs/brendan/2013/01/11/virtualization-performance-zones-kvm-xen/.

[12] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. *Proceedings of the 11th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) 41*, 11 (2006), 2–13.

[13] AHN, J., JIN, S., AND HUH, J. Revisiting hardware-assisted page walks for virtualized systems. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)* (2012), IEEE Computer Society, pp. 476–487.

[14] ALBERTS, B. Dr linux 2.6 rootkit released. http://lwn.net/Articles/296952/.

[15] ARCANGELI, A., EIDUS, I., AND WRIGHT, C. Increasing memory density by using ksm. In *Proceedings of the Linux Symposium* (2009), pp. 19–28.

[16] BEN-YEHUDA, M., DAY, M. D., DUBITZKY, Z., FACTOR, M., HAR'EL, N., GORDON, A., LIGUORI, A., WASSERMAN, O., AND YASSOUR, B.-A. The turtles project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation (OSDI)* (2010), vol. 10, pp. 423–436.

[17] BHATIA, N. Performance evaluation of intel ept hardware assist. *VMware, Inc* (2009).

[18] CANTRILL, B. Experiences porting kvm to smartos. *KVM Forum 2011*.

[19] Cloudsigma. https://www.cloudsigma.com/.

[20] Elastichosts. http://www.elastichosts.com/.

[21] ELHAGE, N. Virtunoid: Breaking out of kvm. *Black Hat USA* (2011).

[22] FERRIE, P. Attacks on more virtual machine emulators. *Symantec Technology Exchange* (2007).

[23] FRANKLIN, J., LUK, M., JONATHAN, M., SESHADRI, A., PERRIG, A., AND VAN DOORN, L. Towards sound detection of virtual machines. *Advances in Information Security, Botnet Detection: Countering the Largest Security Threat*, 89–116.

[24] FRANKLIN, J., LUK, M., MCCUNE, J. M., SESHADRI, A., PERRIG, A., AND VAN DOORN, L. Remote detection of virtual machine monitors with fuzzy benchmarking. *ACM SIGOPS Operating Systems Review 42*, 3 (2008), 83–92.

[25] GARFINKEL, T., ADAMS, K., WARFIELD, A., AND FRANKLIN, J. Compatibility is not transparency: Vmm detection myths and realities. In *Proceedings of the 9th USENIX workshop on Hot topics in operating systems (HotOS)* (2007).

[26] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Symposium on Network and Distributed Systems Security (NDSS)* (2003), pp. 191–206.

[27] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A., VARGHESE, G., VOELKER, G., AND VAHDAT, A. Difference engine: Harnessing memory redundancy in virtual machines. *Proceedings of the 8th symposium on Operating Systems Design and Implementation (OSDI)* (2008).

[28] HALFDEAD. Mistifying the debugger, ultimate stealthness. http://phrack.org/issues/65/8.html.

[29] Hp helion public cloud. http://www.hpcloud.com/.

[30] JIANG, X., WANG, X., AND XU, D. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)* (2007), pp. 128–138.

[31] JIANG, X., AND XU, D. Collapsar: A vm-based architecture for network attack detention center. In *USENIX Security Symposium* (2004), pp. 15–28.

[32] KAPLOWITZ, J. Debian google compute engine kernel improvements, now and future. https://lists.debian.org/debian-cloud/2013/11/msg00007.html.

[33] KLEEN, A. Kvm mailing list discussion. https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg611255.html.

[34] KLEIN, T. Scooby doo-vmware fingerprint suite. http://www.trapkit.de/research/vmm/scoopydoo/index.html, 2003.

[35] KORTCHINSKY, K. Cloudburst: A vmware guest to host escape story. *Black Hat USA* (2009).

[36] LITTY, L., LAGAR-CAVILLA, H. A., AND LIE, D. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium* (2008), pp. 243–258.

[37] MERLIN, M. Live upgrading thousands of servers from an ancient red hat distribution to 10 year newer debian based one. In *Proceedings of the 27th conference on Large Installation System Administration (LISA)* (2013), pp. 105–114.

[38] ORMANDY, T. An empirical study into the security exposure to hosts of hostile virtualized environments. http://taviso.decsystem.org/virtsec.pdf, 2007.

[39] PEREZ-BOTERO, D., SZEFER, J., AND LEE, R. B. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 international workshop on Security in cloud computing* (2013), ACM, pp. 3–10.

[40] PETRONI JR, N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)* (2007), pp. 103–115.

[41] QUIST, D., AND SMITH, V. Detecting the presence of virtual machines using the local data table. http://www.offensivecomputing.net/files/active/0/vm.pdf, 2006.

[42] RILEY, R., JIANG, X., AND XU, D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Recent Advances in Intrusion Detection (RAID)* (2008), Springer, pp. 1–20.

[43] RUSSINOVICH, M. Inside windows server 2008 kernel changes. *Microsoft TechNet Magazine* (2008).

[44] RUTKOWSKA, J. Red pill... or how to detect vmm using (almost) one cpu instruction. http://invisiblethings.org/papers/redpill.html, 2004.

[45] SMYTH, N. *Hyper-V 2008 R2 Essentials*. eBookFrenzy, 2010.

[46] SUZAKI, K., IIJIMA, K., YAGI, T., AND ARTHO, C. Software side channel attack on memory deduplication. *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 11' POSTER)* (2011).

[47] TOSATTI, M. Kvm: x86: report valid microcode update id. https://github.com/torvalds/linux/commit/742bc67042e34a9fe1fed0b46e4cb1431a72c4bf.

[48] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *ACM SIGOPS Operating Systems Review 39*, 5 (2005), 148–162.

[49] WALDSPURGER, C. Memory resource management in vmware esx server. *Proceedings of the 5th symposium on Operating Systems Design and Implementation (OSDI) 36*, SI (2002), 181–194.

[50] XIAO, J., XU, Z., HUANG, H., AND WANG, H. Security implications of memory deduplication in a virtualized environment. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2013).