Virtual Machine Extrospection: A Reverse Information Retrieval in Clouds

1P. Vijay Bhaskar Reddy 2N. Venkateswarlu

2 Scholar, Dept. of Master of Computer Applications, Narayana Engineering College, Gudur.

1Assistant Professor, Dept. of Master of Computer Applications, Narayana Engineering College, Gudur.

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 six real-world clouds, including Google Compute Engine (a.k.a. Google Cloud), HP Helion Public Cloud, ElasticHosts, Joyent Cloud, CloudSigma, and VULTR, as well as in a controlled testbed environment, all yielding promising results.

Index Terms—Virtualization, Hypervisor, Extrospection, Linux, KVM.

1. INTRODUCTION

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 com-

http://indusedu.org
Page 239

This work is licensed under a Creative Commons Attribution 4.0 International License
monly 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 information 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 [10], [11], [12], [13], 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 six real world clouds, including Google Compute Engine [14], HP Helion Public Cloud [15], ElasticHosts [16], JoyentCloud [17], CloudSigma [18], and VULTR [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 2 to 5 versions.

The remainder of the paper is organized as follows. Section 2 describes the background of our work. Section 3 presents the design of Hyperprobe. Section 4 details the implementation of Hyperprobe with several case studies. Section 5 presents experimental results on virtual machines in the cloud and our controlled testbed. Section 6 discusses some potential extensions to the framework. Section 7 surveys related work, and finally, Section 8 concludes the paper.

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. Later on, x86 processor vendors including Intel and AMD released their 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 [20]. 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 LinuxKVM

KVM refers to kernel-based virtual machine. Since Linux kernel version 2.6.20, KVM is merged into the Linux main-line kernel as a couple of kernel modules: an architecture
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 [21] and kernel same page merging (KSM) [22], 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.

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.

### 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

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 improve.

- 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.

- Extensible: The framework should support for these new features on hypervisors such as KVM and Xen to gain more vantage points, but they should be added modules to the framework to handle the concurrency of many features and bugs to infer the underlying hypervisor version.

KVM Features

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

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.
This work is licensed under a Creative Commons Attribution 4.0 International License
Algorithm 2: Detecting KSM

```
1 Procedure test_ksm()
2   load_file_once_into_memory(file);
3     // record the clock time before we
4     // write to each page of the file
5     time1 ← clock.getting();
6     foreach page of file in memory do
7         write to that page;
8     // record the clock time after we write
9     // to each page of the file
10    time2 ← clock.getting();
11    t1 ← diff(time1, time2);
12    load_file_twice_into_memory(file);
13    // sleep and hope the two copies will
14    // be merged
15    sleep (NUM_OF_SECONDS);
16    // record the clock time before we
17    // write to each page of the file
18    time1 ← clock.getting();
19    foreach page of file in memory do
20         write to that page;
21    // record the clock time after we write
22    // to each page of the file
23    time2 ← clock.getting();
24    t2 ← diff(time1, time2);
25    ratio ← t2/t1;
26    if ratio > KSM_THRESHOLD then
27       return 1;
28    else
29       return 0;
```

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

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 \texttt{kvm\_feature\_testers} and \texttt{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 (Algorithm 1) involves two steps. First, we use a loop and inside the loop we call the feature test cases in a descending order. In other words, we first run the test case for kernel version 3.14, and then if necessary, we run the test case for kernel version 3.13. 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.

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 [13], [24] has shown that this timing difference is large enough to tell if KSM is enabled.
Case Studies: Kernel Features

Kernel Same page Merging

Memory over commitment, a mechanism to save memory, was originally implemented in VMware [23]. Later on, this mechanism is introduced into Linux kernel since version 2.6.32, and it is called kernel same page merging (KSM) [22]. 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. 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.

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 maintain 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 3 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 [20], [25], [26], to evaluate virtualization performance. The main part of this benchmark repeats the operation of process creation and destruction very aggressively. Similar to [26], 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.

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.
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. Windows systems released after 2008, such as Windows Server 2008, and Windows Vista, are fully enlightened [27], [28], which means 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 [29], [30], 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.

**Algorithm 3: Detecting EPT**

```plaintext
Algorithm 3: Detecting EPT

Global Var: forward one process avg, access one byte avg

1. Procedure forward one process()
   // read time stamp counter before we run the forward benchmark
   counter1 ← rdtscl();
   for i = 0 to NUM_OF_PROCESS do
     if i = 0 then // child process
       exit (0);
     else // parent process, wait until child process exits
       wait (status);
     counter2 ← rdtscl();
   // read time stamp counter when the forward benchmark is finished
   counter2 ← rdtscl();
   compute average time for forward one process
   forward one process avg ← counter2 − counter1;
   cycles ← counter2 − counter1;
   // compute average time for accessing one byte
   access one process avg ← cycles/NUM_OF_PROCESS;

2. Procedure access one byte (iterations)
   offset ← 0;
   page ← alloca size(PAGE_SIZE);
   // read time stamp counter before we access memory bytes
   counter1 ← rdtscl();
   for i = 0 to iterations do
     page[offset] ← (page[offset] + 1) mod 256;
     offset ← (offset + 1) mod PAGE_SIZE;
   // read time stamp counter after we access memory bytes
   counter2 ← rdtscl();
   cycles ← counter2 − counter1;
   // compute average time for accessing one byte
   access one byte avg ← cycles/iterations;

3. Procedure one time run()
   access one byte(num_of_iterations);
   forward one process();
   value ← forward one process avg/access one byte avg;
   if value < EPT_THRESHOLD then
     return 1;
   else
     return 0;

4. Procedure test exit()
   for i = 0 to LOOP_NUMBER do
     if one time run() = 1 then
       return 1;
   return 0;
```

---

![Diagram](http://indusedu.org)
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.

a. 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.

i. 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 [35] and some online communications written by Google engineers [36], [37], 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.
<table>
<thead>
<tr>
<th>VM Name</th>
<th>Zone</th>
<th>Machine Type</th>
<th>Image</th>
<th>VCPU</th>
<th>VCPU Frequency</th>
<th>RAM</th>
<th>Disk</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>ase-test1</td>
<td>as2</td>
<td>n1-standard-1</td>
<td>SUSE SLES 11 SP3</td>
<td>1</td>
<td>2.50GHz</td>
<td>3.8GB</td>
<td>10G</td>
<td>3.2</td>
<td>3.3</td>
</tr>
<tr>
<td>ase-test2</td>
<td>as2</td>
<td>n1-highcpu-10</td>
<td>SUSE SLES 11 SP3</td>
<td>6</td>
<td>2.50GHz</td>
<td>14.4GB</td>
<td>10G</td>
<td>3.2</td>
<td>3.3</td>
</tr>
<tr>
<td>ase-test3</td>
<td>us-central1-a</td>
<td>n1-highmem-16</td>
<td>Debian / wheezy</td>
<td>16</td>
<td>2.00GHz</td>
<td>8GB</td>
<td>10G</td>
<td>3.2</td>
<td>3.3</td>
</tr>
<tr>
<td>ase-test4</td>
<td>us-central1-b</td>
<td>n1-highmem-16</td>
<td>Debian / wheezy</td>
<td>16</td>
<td>2.00GHz</td>
<td>8GB</td>
<td>10G</td>
<td>3.2</td>
<td>3.3</td>
</tr>
<tr>
<td>ase-test5</td>
<td>europe-west-a</td>
<td>n1-highmem-4</td>
<td>Debian / wheezy</td>
<td>4</td>
<td>2.00GHz</td>
<td>15GB</td>
<td>10G</td>
<td>3.2</td>
<td>3.3</td>
</tr>
<tr>
<td>ase-test6</td>
<td>europe-west-b</td>
<td>n1-standard-4</td>
<td>Debian / wheezy</td>
<td>4</td>
<td>2.00GHz</td>
<td>15GB</td>
<td>10G</td>
<td>3.2</td>
<td>3.3</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>VM Name</th>
<th>Region</th>
<th>Zone</th>
<th>Size</th>
<th>Image</th>
<th>VCPU</th>
<th>VCPU Frequency</th>
<th>RAM</th>
<th>Disk</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>hp-test1</td>
<td>US East</td>
<td>az2</td>
<td>standard small</td>
<td>SUSE SLES 11 SP3</td>
<td>1</td>
<td>2.4GHz</td>
<td>1GB</td>
<td>2GB</td>
<td>3.2</td>
<td>3.7</td>
</tr>
<tr>
<td>hp-test2</td>
<td>US East</td>
<td>az2</td>
<td>standard large</td>
<td>SUSE SLES 11 SP3</td>
<td>4</td>
<td>2.4GHz</td>
<td>15GB</td>
<td>300G</td>
<td>3.2</td>
<td>3.7</td>
</tr>
<tr>
<td>hp-test3</td>
<td>US East</td>
<td>az2</td>
<td>standard large</td>
<td>SUSE SLES 11 SP3</td>
<td>4</td>
<td>2.4GHz</td>
<td>8GB</td>
<td>16G</td>
<td>3.2</td>
<td>3.7</td>
</tr>
<tr>
<td>hp-test4</td>
<td>US East</td>
<td>az2</td>
<td>standard medium</td>
<td>SUSE SLES 11 SP3</td>
<td>2</td>
<td>2.4GHz</td>
<td>4GB</td>
<td>8G</td>
<td>3.2</td>
<td>3.7</td>
</tr>
<tr>
<td>hp-test5</td>
<td>US East</td>
<td>az2</td>
<td>standard medium</td>
<td>Linux 10.04</td>
<td>2</td>
<td>2.4GHz</td>
<td>1GB</td>
<td>8G</td>
<td>3.2</td>
<td>3.7</td>
</tr>
<tr>
<td>hp-test6</td>
<td>US East</td>
<td>az2</td>
<td>standard large</td>
<td>Debian / Wheezy</td>
<td>4</td>
<td>2.4GHz</td>
<td>15GB</td>
<td>300G</td>
<td>3.2</td>
<td>3.7</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>VM Name</th>
<th>Location</th>
<th>Image</th>
<th>VCPU (Only 1 allowed for free trial)</th>
<th>RAM</th>
<th>Disk</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>eh-test1</td>
<td>Los Angeles</td>
<td>Ubuntu 13.10</td>
<td>2.8GHz</td>
<td>1GB</td>
<td>10GB</td>
<td>3.6</td>
<td>3.8</td>
</tr>
<tr>
<td>eh-test2</td>
<td>Los Angeles</td>
<td>CentOS Linux 6.5</td>
<td>2.8GHz</td>
<td>512MB</td>
<td>5GB SSD</td>
<td>3.6</td>
<td>3.8</td>
</tr>
<tr>
<td>eh-test3</td>
<td>Los Angeles</td>
<td>Debian 7.4</td>
<td>2.8GHz</td>
<td>512MB</td>
<td>5GB</td>
<td>3.6</td>
<td>3.8</td>
</tr>
<tr>
<td>eh-test4</td>
<td>Los Angeles</td>
<td>Ubuntu 14.04 LTS</td>
<td>2.8GHz</td>
<td>1GB</td>
<td>10GB</td>
<td>3.6</td>
<td>3.8</td>
</tr>
<tr>
<td>eh-test5</td>
<td>San Antonio</td>
<td>Ubuntu 12.04.1 LTS</td>
<td>2.5GHz</td>
<td>1GB</td>
<td>9G</td>
<td>3.6</td>
<td>3.8</td>
</tr>
<tr>
<td>eh-test6</td>
<td>San Antonio</td>
<td>CentOS Linux 6.5</td>
<td>2.5GHz</td>
<td>512MB</td>
<td>10GB</td>
<td>3.6</td>
<td>3.8</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>VM Name</th>
<th>Location</th>
<th>Image</th>
<th>VCPU (Only 1 allowed for free trial)</th>
<th>RAM</th>
<th>Disk</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>j-test1</td>
<td>US-East</td>
<td>CentOS 6.5</td>
<td>3.07GHz</td>
<td>250MB</td>
<td>16GB</td>
<td>2.634</td>
<td>2.634</td>
</tr>
<tr>
<td>j-test2</td>
<td>US-SouthWest</td>
<td>Ubuntu Certified 14.04</td>
<td>2.40GHz</td>
<td>250MB</td>
<td>16GB</td>
<td>2.634</td>
<td>2.634</td>
</tr>
<tr>
<td>j-test3</td>
<td>US-West</td>
<td>CentOS 6.5</td>
<td>2.40GHz</td>
<td>250MB</td>
<td>16GB</td>
<td>2.634</td>
<td>2.634</td>
</tr>
<tr>
<td>j-test4</td>
<td>EU-Amsterdam</td>
<td>Ubuntu Certified 14.04</td>
<td>2.40GHz</td>
<td>250MB</td>
<td>16GB</td>
<td>2.634</td>
<td>2.634</td>
</tr>
</tbody>
</table>

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

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

This work is licensed under a Creative Commons Attribution 4.0 International License
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.

**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.

**Cloud Evolution**

After 18 Months, i.e., in January 2016, we performed a new set of measurements in three cloud environments: Google Cloud, Elastic Hosts Cloud, and VULTR Cloud [19]. The purpose of this additional measurements is to verify whether cloud vendors do upgrade their hypervisor kernels.

For Elastic Hosts, we launched one virtual machine located in their San Jose Data Center, the virtual machine is configured with a 2000MHZ virtual CPU, 1024MB memory, and is installed with a Debian 8.2 Linux operating system. Surprisingly, we found that Elastic Hosts are still using the kernel version between 3.6 to 3.8, which is the same as what Hyperprobe reported 18 months ago. One possible reason is that we were using free trial virtual machines in both our measurements in 2014 and 2016, and these virtual machines are located at some specific servers that the cloud providers do not perform regular maintenance.

For Google Cloud and VULTR Cloud, Hyperprobe reports that their kernel versions are equal to or newer than 3.14, which is the upper limit of Hyperprobe. Thus, with the current implementation of Hyperprobe, we are just able to versions. For VULTR Cloud, we performed some manual efforts to figure out its upper bound after launching a CentOS 7 x64 system, with 1 virtual CPU, 768 MB memory, and a 15GB SSD disk. The host machine is located at New Jersey. We found that the bit 63 of the CR3 register is not writable. This fact matches with a KVM bug description [43], and the bug is fixed in the kernel version 3.19. This suggests that the underlying kernel version in VULTR Cloud is between 3.14 to 3.18.

**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 controlled 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, 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.

**6 DISCUSSION**

In this section, we discuss some potential enhancements.

**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 frame-work 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 state their new features. And the bugs of these hypervisors are also publicly available.
More Features

We describe two more interesting kernel features for which we had planned to create test cases, but have not yet done so due to the unavailability of certain processors in our lab.

Pause Loop Exiting

Lock holder preemption is a problem that commonly exists in a virtualized environment [44], [45]. This problem happens when a virtual CPU holding a lock in a virtual machine is scheduled out by the hypervisor scheduler. This can happen because the hypervisor has no semantic knowledge of guest locks. The consequence is that when the other virtual CPUs are scheduled in, they have to wait until the lock holder is scheduled in and releases the lock. This is a waste of time.

To address this problem, recent generations of x86 processors present a new feature (Intel Pause Loop Exiting and AMD Pause Filter) aimed at detecting virtual CPU busy waiting. When this virtual CPU is detected, a trap to the host occurs (this is called Pause Loop Exiting, because spinlock are usually implemented with a loop of the pause instructions), and the hypervisor can then make a better scheduling decision. Since Linux kernel 2.6.33, Pause Loop Exiting (PLE) is supported in KVM. When PLE happens, the hypervisor would let a waiting vCPU sleep for 100 microseconds hoping the lock holding vCPU is scheduled in to release the lock. This, to some extent, resolves the lock holder preemption problem.

To detect PLE, we can simply run a lot of pause instructions, and see if it traps. Because of the 100 microseconds sleeping, the execution time of those pause instructions should be far longer when PLE is enabled than when it is not. And a successful detection of PLE suggests that the underlying hypervisor kernel should be newer than or equal to 2.6.33.

This PLE feature is only available in recent Intel processors: Intel Nehalem-ex and Intel Westmere, but not in the processors we are using, thus, we were not able to develop a test case for this feature.

APIC virtualization

APIC virtualization is yet another feature that exists in recent generations of x86 processors. Traditionally, guest interrupts are virtualized by the hypervisors, but this requires a lot of exits from the guest to the hypervisor and then re-entries from the hypervisor to the guest. These context switches are time consuming and incur significant overhead.

To address this problem, the latest Intel processors, such as the Intel Xeon E52600 v2 product family, provide a new feature called APIC virtualization or APICv. With APIC virtualization, most interrupt related activities do not need to trap into the hypervisor. Thus, overhead is reduced and I/O throughput is increased.

Since Linux kernel 3.9, APIC virtualization is supported in KVM. In its implementation, accessing most APIC registers does not trap, but there are a few exceptions. For example, the local APIC ID register, the Task Priority register, and the Current Counter for Timer register will still trap. Therefore, accessing these APIC registers will incur much more time than accessing the other APIC registers. Based on this timing difference, we can detect the APIC virtualization feature. And a successful detection of this feature suggests that the underlying hypervisor kernel should be newer than or equal to 3.9.

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: extensibility. In fact, we have made it open source on github3, and we hope 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 higher precision our detection can achieve. And this will certainly accelerate our development process and our support for the other hypervisors.

In addition, developing multiple test cases for one kernel version would also reduce the possibility of false positives.

7 RELATED WORK

We survey related work in three categories: detection of a specific hypervisor, attacks against hypervisors, and operating system fingerprinting.

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.

1. https://github.com/jidongxiao/hyperprobe
for detecting the underlying hypervisors and are briefly described as follows. RedPill [46] and Scooby Doo [47] 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 [48]. By contrast, the detection technique proposed in [48] 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 [10], [49]. 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.

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 [8], [9]. Perez-Botero et al. [50] characterized various hypervisor vulnerabilities by analyzing vulnerability databases, including SecurityFocus [51] and NIST’s Vulnerability Database [52]. Their observation is that almost every part of a hypervisor could have vulnerabilities. Ormandy [53] 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.

Operating System Fingerprinting

Operating system fingerprinting is crucial for both attackers and defenders. Prior research in this area can be divided into three categories. The first category is network based fingerprinting, a popular technique mainly used by attackers. In particular, tools like Nmap [54], [55] and Xprobe [56] have been widely used and extensively studied. These tools work by examining the TCP/IP traffic patterns and matching them against a database of known results. The second category is virtualization based fingerprinting [57], [58]. The key idea of OS-Sommelier [57] is using a cloud environment where the guest OS memory is present, system administrators can compute a hash for the kernel code of each guest OS; as different guest OSes should produce a different hash value, system administrators can differentiate each guest OS, achieving the goal of guest OS fingerprinting. In [58], the authors observed that, in a virtualized environment where memory deduplication works at the hypervisor level, the memory deduplication mechanism usually causes accumulated access delay for the deduplicated memory pages. Therefore, one can load different OS images into its own memory; if there is another virtual machine running the same OS co-resident with the attacker’s virtual machine, the identical pages will be deduplicated, and by measuring the access delay, one can detect whether or not that specific operating system is running in co-resident virtual machines.

The third category is USB based [59], [60]. In a representative work of this type, Bates et al. [59] proposed to use USB devices to identify different host systems. The main premise of this work is that there is a timing variation between different operating systems when communicating with a specific USB device. Leveraging this time variation and some machine learning techniques,
system administrators can determine the identity of each host system. Compared with all these OS fingerprinting techniques, Hyperprobe differs in two aspects. First, it has a different threat model. Hyperprobe works inside a virtual machine, and attempts to retrieve the information of the underlying hypervisor, specifically its kernel version. Second, it employs a very different approach. In particular, our implementation mainly relies on the knowledge of the evolution of KVM. As far as we know, we are the first to systematically examine the KVM patches over the past seven years and study the evolution of KVM development.

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 six real clouds, as well as in our own testbed.

REFERENCES


Jidong Xiao received his PhD degree in Computer Science from the College of William and Mary, Williamsburg, in 2016. He is an assistant professor at Boise State University. His research focuses on cybersecurity, with a particular emphasis on operating system security and cloud security.

Lei Lu received his PhD degree in computer science from the College of William and Mary, Virginia, in 2014. He is currently a senior member of technical staff at VMWare Inc., Palo Alto.

Hai Huang received his PhD degree in Computer Science and Engineering from University of Michigan, Ann Arbor in 2006. He is a Research Staff at IBM T. J. Watson Research Center. His research interests include cloud computing, file and storage systems, systems management, software testing and anomaly detection, and energy and power management.

Haining Wang received his Ph.D. in Computer Science and Engineering from the University of Michigan at Ann Arbor in 2003. He is a Professor of Electrical and Computer Engineering at the University of Delaware, Newark, DE. His research interests lie in the areas of security, networking system, and cloud computing.