Towards Sound Detection of Virtual Machines

Jason Franklin[1], Mark Luk[1], Jonathan M. McCune[1], Arvind Seshadri[1], Adrian Perrig[1], and Leendert van Doorn[2]

[1] Carnegie Mellon University, Pittsburgh, PA.
jfrankli@cs.cmu.edu, {mluk, jonmccune}@cmu.edu,
arvinds@cs.cmu.edu, perrig@cmu.edu
[2] AMD, Austin, TX. Leendert.vanDoorn@amd.com

Summary. We design, implement, and evaluate a practical timing-based approach to detect virtual machine monitors (VMMs) without relying on VMM implementation details. The algorithms developed in this paper are based on fundamental properties of virtual machine monitors rather than easily modified software artifacts. We evaluate our approach against two common VMM implementations on machines with and without hardware support for virtualization in a number of remote and local experiments. We successfully distinguish between virtual and real machines in all cases even with incomplete information regarding the VMM implementation and hardware configuration of the targeted machine.

1 Introduction

In their seminal work, Popek and Goldberg formally defined the essential properties that a program must satisfy to be termed a virtual machine monitor: efficiency, resource control, and equivalence [12]. In this article, we exploit the *timing dependency exception* to the equivalence property of a VMM to detect the presence of a virtual machine monitor (VMM) without relying on implementation details or software artifacts.

Virtual machine monitor detection has two direct implications for botnet remediation: first, it provides defenders with the ability to detect bots which utilize VMMs for improved stealth (e.g., VM-based rootkits [10, 18, 27]) and second, exploring VMM detection allows defenders to assess the extent to which intelligent bots can identify and potentially bias virtualized analysis environments such as high-interaction honeypots [9, 22, 26].

Due to the sophisticated nature of modern VMMs and significant variations between implementations, implementation-independent VMM detection is a difficult open problem. This difficulty is highlighted by the fact that most related work emphasizes implementation-dependent (software-artifact-based) techniques. These techniques have an inherent weakness: implementation-dependent detection techniques are easy to counter by modifying VMM implementations to mask or otherwise hide identifiable software artifacts.

In contrast to previous work, the detection algorithms developed in this paper are VMM implementation-independent and hardware-dependent. While the practicality of modifying VMM implementations to counter the multitude of current implementation-dependent detection techniques can be disputed, modifying the implementation of a VMM is inherently easier than modifying the underlying hardware, especially since in most cases the required software modifications are trivial. Our implementation-independent algorithms do not rely on software artifacts, making them difficult to counter without hardware modifications, a task which is difficult for organizations who rely on commodity hardware.

The main contribution of this article is the development of a class of implementation-independent VMM detection algorithms whose execution is noticeably different when executed inside a virtual machine versus when executed directly on the underlying hardware. We describe the design and implementation of our algorithms, their success detecting a number of VMMs including VMware [23,25], the Xen VMM [2] on standard hardware, and the Xen VMM on a machine with hardware assistance for virtualization.

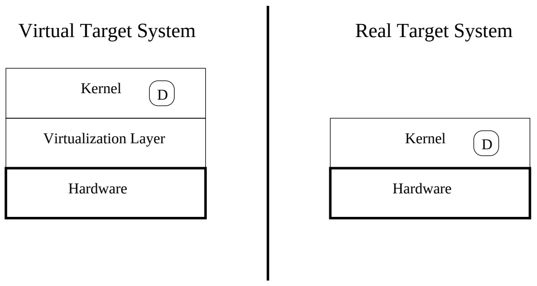


Fig. 1. VMM detection algorithm D on virtual and real target systems

We develop a class of VMM detectors that, when executing on a target system of unknown status (virtual or real) with access to a trusted external timer, can distinguish between a virtual and real target system (see Figure 1). Given the exact hardware specification and the specific VMM implementation that may be present, detection using timing is straightforward. However, given all known and possibly unknown VMM implementations, and all possible commodity hardware configurations, detecting the presence of a VMM on a platform with uncertain configuration is challenging. Hence, VMM detection spans a spectrum of scenarios, running from specific (easier to detect) to general (harder to detect) along two axes: VMM implementation ranging from known to unknown and hardware configuration ranging from known to unknown (see Figure 2). We explore this space of detection scenarios and address the challenges that lie within.

Complete knowledge of a system's hardware configuration is available in some scenarios, such as administratively controlled machines in corporations. As an example, consider the scenario where VM-based rootkits (VMBRs) become a signifi-

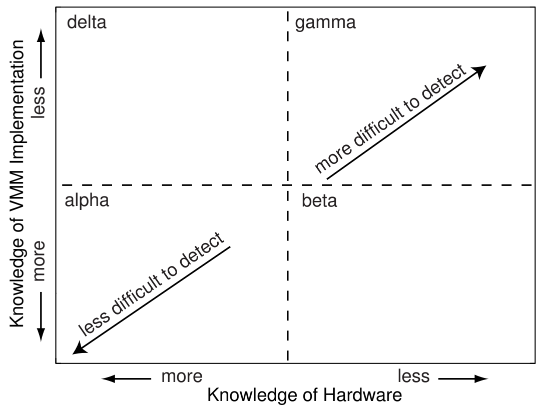


Fig. 2. Problem space

cant threat in the wild. Anti-virus software makers, motivated to protect their users against such threats, could ask users to specify their hardware (e.g., Pentium 4 2.0 GHz) upon installation of a VMM detector such as the one developed in this paper. Servers run by the anti-virus company could then periodically challenge the users' systems to execute our hardware-specific VMM detector, designed to elicit a detectable performance degradation when running in a VMM. If performance is degraded sufficiently, the anti-virus software company could begin a recovery on the users' VM-based rootkit infected machines. A challenge in this naive model is that the information provided by the user about their system might be incorrect, incomplete, or unavailable.

The techniques described in this paper successfully detect the presence of a VMM on a target system even with uncertainty about the system's exact hardware configuration and specific VMM implementation. Our approach exploits VMM timing dependencies to elicit measurable VMM overhead, even in the face of limited hardware and software configuration information. Uncertainties with respect to hardware configuration include CPU microarchitecture, cache architecture, and clock speed. Uncertainties with respect to the VMM implementation include optimizations such as the use of binary rewriting or paravirtualization. Hardware support for virtualization, such as Intel's VT [8] or AMD's SVM [5] technologies, further complicate detection.

In our evaluation, we are able to identify sufficient hardware configuration information for target systems and to ultimately distinguish between virtual and real machines. Further, our approach continues to work against VMMs that utilize hardware support for virtualization. Our experiments demonstrate the viability of our approach over a range of uncertainty. As such, the algorithms developed in this paper represent a promising step towards general VMM detection techniques.

1.1 Context

The best way to understand VMM detection and to understand the relationship between this paper and past work is to describe the arms race which is VMM detection. VMM detection is an arms race between detectors (which attempt to detect a VMM) and VMMs (which attempt to evade detection). Below we describe the stages of the arms race with each step labeled either current, emerging, or future to describe the chronology of the race.

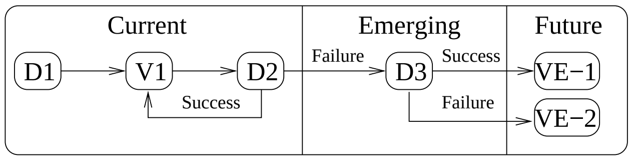


Fig. 3. VMM detection arms race

D1: Currently, detectors use software implementation-dependent artifacts such as communication back doors, process names, and perturbed locations of system components [19].

V1: VMMs evade detection by eliminating the specific artifacts used for detection. For example, VMMs mask names and values (i.e., location of the IDT, special processes, communication back doors etc...) or interpose on specific instructions which are used in detection [10].

D2: Detectors search for previously unknown software artifacts. If found, return to step V1 otherwise continue.

D3: In the absence of previously unknown software artifacts, detectors search for implementation-independent perturbations such as timing (this article). If found, continue, else jump to VE-2.

VE-1: Unable to evade implementation-independent detection, VMMs either remain detectable or violate an assumption of the arms race. One possible violation is that VMMs continue to operate on commodity hardware. It's possible that hardware support for virtualization will eliminate VMM overhead. We suspect this to be unlikely for multiple reasons: hardware-support is meant to fill holes in current architecture's virtualization support and to ease the implementation of VMMs, it is not designed to optimize or otherwise hide the presence of a VMM. We evaluate the results of violating this assumption in Section 6.

VE-2: With both implementation-dependent and implementation-independent detectors eliminated, VMMs successfully evade detection.

Organization. Section 2 discusses necessary background including the formal properties of a VMM. Section 3 sketches a sound approach to VMM detection. Section 4 discusses the algorithm and protocol design for a class of VMM detection algorithms. Section 5 describes the implementation of a detector. Section 6 presents our experimental results. We present a security analysis in Section 7 and discuss limitations and possible extensions in Section 8. We cover related work in Section 9 and conclude in Section 10.

2 Background

We follow Popek and Goldberg in defining a virtual machine as an efficient, isolated duplicate of the underlying hardware [12]. This definition imposes the three properties that a control program must satisfy to be termed a virtual machine monitor: efficiency, resource control, and equivalence. To explain these three properties, we must first introduce some terminology.

2.1 Instruction Types

We classify the underlying instructions of a machine based on their behavior. An instruction is **privileged** if it can only be executed in the highest processor privilege level, and executing it at any other privilege level results in a trap to a higher privilege level. Privileged instructions are characteristics of the underlying hardware and are invariant over a particular instruction set architecture. An instruction is **sensitive** if it can interfere with the state of a memory-resident VMM. An instruction is innocuous if it is not sensitive.

2.2 Virtual Machine Properties

Informally, the **efficiency property** dictates that programs run in a virtualized environment show no more than minor decreases in speed. Since minor decrease in speed is difficult to quantify, a parallel requirement of the efficiency property is that a statistically dominant subset of the virtual processor's instructions be executed directly by the real processor.

The **resource control property** dictates that a VMM maintain complete control of system resources. This requires that it be impossible for an arbitrary program running in a VM on top of a VMM to affect system resources, e.g., memory and peripherals, allocated to a different VM or the VMM itself.

The **equivalence property** dictates that a VMM provide an environment for programs which is essentially identical to that of the original machine. Formally, any program P executing with a VMM resident in memory, with two possible exceptions, must perform in a manner *indistinguishable* from the case when the VMM did not exist and P had the freedom of access to privileged instructions that the programmer had intended. The two possible exceptions to the equivalence property result from resource availability and timing dependencies.

2.3 Exceptions

The **resource availability exception** states that a particular request for a resource may not always be satisfied. As a result, a program may be unable to function in the same manner as it would if the resource were made available. This exception exists because a VMM shares the underlying hardware and hence consumes resources.

The **timing dependency exception** states that certain instruction sequences in a program may take longer to execute. Hence, assumptions about the length of time required for the execution of an instruction might lead to incorrect results. This exception results from the possibility of a VMM occasionally intervening in certain instruction sequences.

These exceptions allow for the theoretical possibility of detecting a virtual machine monitor. If these exceptions did not exist, a VMM that perfectly satisfied the equivalence property would be impossible to detect. In this paper, we study how VMM detectors can be written which exploit these exceptions to unmask virtualized machines.

3 Approach

We sketch the design of a sound detection algorithm that exploits the timing dependency exception of a VM to distinguish between real and virtual machines.

3.1 Definitions

A *VMM detection algorithm* is a decision procedure which when given as input a target machine M outputs $accept$ if M is a virtual machine and $reject$ if M is a real machine. Let V be a virtual machine. A detection algorithm D is *sound* if and only if when $D(M)$ outputs $accept$, M is a virtual machine. A detection algorithm D is *complete* if and only if on input V, D halts and outputs $accept$. In order to eliminate any dependence on a particular VMM implementation, the approach described below is based on an idealization of a control program which satisfies the required properties of a VMM with the two previously mentioned exceptions. We term such a program an *idealized VMM*.

3.2 Intuition behind Detection Algorithms

Failure to control the execution of a sensitive instruction executed in a virtual machine (VM) can result in a loss of control over system resources. Since this is a violation of the resource control property, a VMM must strictly control the execution of sensitive instructions. The need to completely control system resources imposes stringent requirements on the execution of any instruction which has the potential to affect system resources.

Classes of instructions that can potentially affect system resources include sensitive-privileged instructions, sensitive-unprivileged instructions, and innocuous-privileged instructions. Innocuous-unprivileged instructions can be directly executed on the underlying hardware as they pose no risk of state corruption or control modification. It is the potentially control-modifying instructions that necessitate the existence of timing dependencies when a program executes in a VM.

When a VMM interposes on the execution of instructions that can affect system resources, VMM overhead is encountered. The VMM overhead of an instruction is the additional number of cycles required to execute the instruction in a VMM versus executing the instruction directly on real hardware. We exploit this overhead to distinguishing between real and virtual machines.

We give an intuition as to why positive VMM overhead is independent of VMM implementation. Assume positive VMM overhead does not exist. Then, either the VMM overhead is zero or it is negative. If the VMM overhead is negative, then the addition of a VMM actually increases the speed of the real machine, clearly a contradiction. If the VMM overhead is zero and instructions execute in a positive amount of time, then the VMM cannot interpose on instructions to maintain resource control. A program which does not maintain resource control is not a VMM, hence we arrive at a contradiction.

In our previous argument, we implicitly assumed that VMMs execute without hardware assistance for virtualization. The recent commoditization of hardware support for virtualization could reduce or in the extreme case eliminate VMM overhead. Previous work has show that even with current generation hardware support for virtualization, VMMs experience considerable performance overhead [1]. In addition, our experimental results confirm these observations. Since we cannot predict how future hardware might improve virtualization performance; the results of this paper only apply to current architectures.

3.3 VMM Detection Algorithm

We are interested in the class of detection algorithms that exploit the timing dependency exception to distinguish between real and virtual machines. We describe this class of algorithms as follows.

Let R_C be a real machine with configuration C and let M_C be a virtual or real machine with identical configuration C. Let `Benchmark` be a program with k control-modifying instructions each with VMM overhead o. Execute `Benchmark` on R. Store the time required for `Benchmark` to complete in $R_C(Benchmark)$. Execute `Benchmark` on M. Store the time required for `Benchmark` to complete in $M_C(Benchmark)$. Compare $M_C(Benchmark)$ and $R_C(Benchmark)$. If $M_C(Benchmark)$ is greater than $R_C(Benchmark)$ by at least $k * o$, output accept, else output reject.

4 Algorithm and Protocol Design

We present the design of our detection algorithm and protocol.

4.1 Algorithm Design

A number of complexities surface while implementing the detection algorithm developed in the previous section. First, a `Benchmark` with control modifying instructions must be constructed. Second, the execution time of a `Benchmark` on the real machine must be measured. Third, the execution time of a `Benchmark` on the target machine must be measured. Each of these entails additional complexities, explanations of which follow.

Designing for Overhead. As we previously argued, because of the inherent properties of a VMM, the VM should not be able to execute a program with control-modifying instructions as fast as the real machine. We design a `Benchmark` to include control modifying instructions empirically determined to have an overhead across implementations and validate our selection against a VMM of unknown implementation. We choose the particular control-modifying instructions and then tune their number such that the VMM overhead is remotely (e.g., across the Internet) noticeable.

Establishing Reference Times. The execution time of a `Benchmark` on R_C, denoted $Baseline(R_C)$ is our reference for distinguishing between virtual and real machines with hardware configuration C. The performance of our algorithm is directly related to the accuracy with which we can measure $Baseline(R_C)$. A central complexity in establishing an accurate reference time is how to establish this value for machines of unknown configuration.

Since the execution time of a `Benchmark` is dependent on the underlying hardware, clearly we require some knowledge of the hardware configuration to establish $Baseline(R_C)$. The greater the amount of information we have about the hardware configuration, the easier it is to distinguish between real and virtual machines, however, as we require more configuration information, the number of scenarios where our detector may work is reduced.

While our approach is independent of the mechanism used to determine the configuration of the machine in question, in order to develop an end-to-end VMM detection algorithm, we proceed as follows. To start, we assume we have no configuration information about the machine in question and that we cannot trust the machine's direct responses to configuration inquiries. Assuming we know the configuration of the machine in question greatly limits the scenarios in which our detection algorithm is applicable. Further, trusting a virtual machine's direct response to configuration questions can result in our acceptance of incorrect timing measurements.

We develop a heuristic approach to identify unknown hardware which works well in practice. Our heuristic, which we call hardware discovery, uses the existence of hardware artifacts that "shine through" a VMM. The hardware artifacts we discover are unique to a particular architecture and allow us to infer a portion of the configuration of the machine. This configuration information then allows for an estimation of $Baseline(R_C)$. We explain our techniques for hardware discovery and runtime estimation in the coming sections.

Measuring Execution Times in a VM. Timing the execution of a `Benchmark` on M necessitates the existence of a reliable timing source. If M is a virtual machine, the VMM may return timing measurements which do not accurately characterize the execution time [10]. To overcome this complexity, we allow the detector to contact an external timing source.

To remotely detect VMM overhead, we must develop a `Benchmark` with sufficient VMM overhead to overcome possible measurement noise. Potential sources of noise include variance in network latency, inaccuracies in timing, and variance in execution times resulting from caching. To overcome this noise, we develop techniques to configure the amount of VMM overhead to a nearly arbitrary extent.

4.2 `Benchmark` Design

Constructing a `Benchmark` requires that we determine which control-modifying instructions and the correct number of these instructions to execute. Below we discuss how a `Benchmark` can designed to have a variable amount of VMM overhead based on the specific instructions used and their number.

Selecting Instructions

To select the correct control-modifying instructions to induce VMM overhead, we measured the overhead of different sensitive-privileged instructions on several different VMMs. We use sensitive-privileged instructions, as opposed to sensitive-unprivileged instructions, because sensitive-unprivileged instructions violate the resource control property [14]. The results of these measurements are presented in Section 6.

Number of Instructions

After selecting particular instructions, we need to further tune the VMM overhead induced a `Benchmark` by selecting the number of instructions. There are two primary factors that affect the VMM overhead of a `Benchmark`. First, the processor configuration of a machine, for instance, Intel Pentium IV 2.0 GHz, has a direct effect on the execution time. Second, different VMM implementation techniques have different levels of overhead. The following analysis explains how we incorporate these two factors into our experiments in order to select the number of instructions in a `Benchmark`.

4.3 Measuring and Approximating Execution Times

First, we assume full knowledge of the configuration of the target machine. We then limit the amount of configuration information that is known and develop an approximation technique for estimating the runtime of a `Benchmark` over a class of machines.

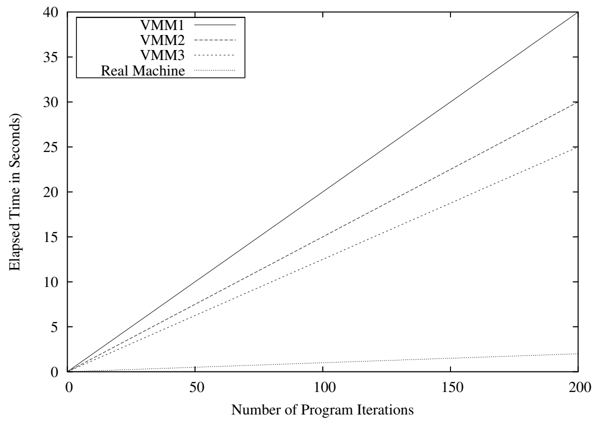


Fig. 4. Example VMM overhead of a `Benchmark`. Without a VMM executing, the instructions complete rapidly. With a VMM, there is noticeable overhead.

Timing With Complete Configuration Information

For purposes of demonstration, we imagine a scenario where we know the exact hardware configuration of the machine which we wish to distinguish as real or virtual, and we have access to a local machine of identical configuration. In this case, we can execute our detection code on the identically configured local machine and measure its execution time for use as a baseline for remote detection.

Given access to the local machine, we can determine the correct number of instructions to execute by estimating the noise in our experiments and running a number of experiments. We execute a `Benchmark` on the real hardware of the local machine and under different VMMs, while varying the number of instructions. The results look similar to Figure 4.

This graph is a hypothetical example based on our experimental results. The upper lines represent the runtimes of a `Benchmark` with a fixed set of control-modifying instructions under several different VMM implementations. The bottom line is the execution time on the real hardware. To determine the required number of instructions, we first fit equations to all the data points in the graph. We then use these equations to determine the minimum number of instructions required to overcome our noise estimate.

$$Let \quad Model(R_C) = \left\{ \begin{array}{cc} VMM_1(x) & = a_1 x \\ VMM_2(x) & = a_2 x \\ VMM_3(x) & = a_3 x \\ RealMachine(x) & = \ bx \end{array} \right\}$$

with $a = min(a_1, a_2, a_3)$ and $FastestVMM(x) = ax$. Given a noise estimate of n, the minimum required number of iterations x such that $FastestVMM(x) - RealMachine(x) > n$ is $x > \frac{n}{a-b}$. Since n is small in practice and our VMM overhead is configurable to an almost arbitrary extent, selecting x based on local experiments presents few difficulties.

In the above example, which is based on our experimental results, we have $a = 0.125$ and $b = 0.01$. If we assume our experimental noise $n = 20ms$ (based on a network latency variation of 10 ms), a `Benchmark` must run at least 175 iterations.

Approximate Timing With Incomplete Configuration Information

We now examine the case where we have incomplete configuration information for the target machine. In this case, we determine the correct number of instructions to execute based on a number of estimates and experiments. We assume we have access to a machine with partial configuration information which matches that of the target machine.

As an example, imagine that the partial configuration information we have identifies just the processor type (e.g., Pentium IV). Since the remote machine we are attempting to distinguish as virtual or real may run at a different clock speed than the machine we are using for our experiments, we need to bound the runtime a `Benchmark` for different configurations and use these bounds for detection. In addition, since our baseline execution time will not be as accurate as in the full configuration information case, we must design the `Benchmark` such that its execution time is ordered as in Figure 5. Essentially, executing a `Benchmark` on the fastest VMM on the fastest real machine that matches the partial configuration information should take longer than executing the `Benchmark` directly on the slowest machine matching the partial configuration information.

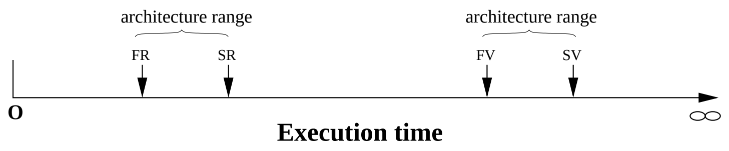


Fig. 5. The required order of execution times for a `Benchmark` for different configurations. Given some configuration information, FR is the fastest real machine, SR is the slowest real machine, FV is the fastest real machine running the fastest VMM, and SV is the slowest real machine running a VMM.

The approach we develop is to determine the range of processor speeds available given our partial configuration information and to use these values to approximate the execution time under different configurations. Since our detection code is CPU bound, it is possible to estimate the runtime of a `Benchmark` given only a few experiments on a single machine and a number of easily determined public values.

Given the partial configuration information we know, we determine the processor speed of the fastest machine available and denote this as F. While this value increases over time, the configurable nature of the overhead elicited by a `Benchmark` makes it possible to compensate for this increase. We denote the speed of the slowest machine satisfying our partial configuration information as S. The processor speed of the machine we are using for local experiments is denoted M. At the time of writing this paper, $F = 3.8GHZ$ and $S = 1.3GHZ$ for the Pentium IV[3].

As described above, we experimentally determine $FastestVMM(x) = ax$ and $RealMachine(x) = bx$ by running a small number of tests on the local machine M. We then use the ratio of the speed of the local machine to the speed of the slowest possible machine, $p = \frac{M}{S}$, to estimate the runtime a `Benchmark` on the real hardware of S. This gives us a runtime estimate on S of $SR = p{*}RealMachine(x)$. Similarly, we use the ratio of the speed of the local machine to the fastest machine, $u = \frac{M}{F}$, to estimate the runtime on the fastest virtual machine. This gives us $FV = d{*}FastestVMM(x)$. To determine the minimum number of instructions required to overcome our noise estimate, we require $FV > SR + n$ or equivalently, $x > \frac{n}{au-bp}$.

Returning to the above example and the Pentium IV, we have $a = 0.125$, $b = 0.01$, $M = 2.0;\ GHz$, $p = \frac{2.0}{1.3}$, and $u = \frac{2.0}{3.8}$. If we assume that our experimental noise $n = 20ms$ a `Benchmark` must run at least 471 iterations, more than twice as many as in the complete configuration information case.

4.4 Protocol Design

In our scheme, a trusted agent external to the target system denoted by V interacts with an instance of a detection algorithm D on a target machine M. V measures the start and end times of D by either invoking D remotely or receiving a communication immediately before D executes. After execution completes, D sends V a notification of completion.

D contains a specially crafted sequence of instructions called the `Benchmark`. The `Benchmark` is designed to elicit externally noticeable differences in execution time between virtualized and non-virtualized execution environments. D executes on the target host at the highest privilege level with interrupts turned off.

Upon receiving the notification of completion, V records the time elapsed since invocation of D. To determine if the detection algorithm D was executed in a VMM, V performs a lookup into a precomputed table of baseline execution times for the target host's hardware platform. If the execution time exceeds the threshold set for the slowest real machine of the specified configuration, the target machine M is considered to be a virtual machine.

5 Implementation

We detect the presence of a VMM based on performance measurements of instruction sequences, which we execute in a loop called the benchmarking loop. We use a sequence of instructions inside of a loop rather than as a straight line program to ease experimentation. We iterate the loop containing control-modifying instructions until we generate enough overhead for detection. Unless stated otherwise, our loop iterates 2^{17} times. We experimentally selected this value.

[3] `http://www.intel.com/products/processor/pentium4`

We implemented our `Benchmarks` as Linux kernel modules. Their instructions always execute at the same privilege level as the kernel itself, which depends on the hardware architecture and the presence or lack of a VMM. To measure execution time locally, we use the `rdtsc` (read time-stamp counter) instruction before and after the benchmarking loop. To obtain measurements using an external or remote verifier, a user-level program `measured` runs on the target system and listens for a TCP connection from the verifier. When a connection is established, `measured` immediately tries to open a file that our kernel module adds to the `/proc` filesystem. This results in a call to a function in our module, which immediately suspends the calling process, disables interrupts, and begins execution of the benchmarking loop. When the benchmarking loop finishes, interrupts are re-enabled, the calling process gets woken up, and its file-open succeeds. Without even reading any data from the file, `measured` sends a packet back across its TCP connection, indicating to the verifier that execution of the benchmarking loop is complete.

6 Evaluation

We first describe the VMMs evaluated in our experiments and our experimental setup, then the actions necessary to ensure timing integrity for our experiments. Mechanisms that can detect the hardware architecture of an unknown remote system are presented next. Finally, we provide the results of both local and remote experiments, culminating in successful detection.

6.1 VMM Implementations

We evaluate our approach against two common virtual machine monitor implementation techniques [15]: full virtualization and paravirtualization. Both of these techniques are used to virtualize operating system instances rather than processes on one operating system; however, they differ in their approach to achieving this goal.

In full virtualization, the virtual replica of the underlying hardware exposed is functionally identical to the underlying machine. This allows operating systems and applications to run unmodified. Full virtualization is typically implemented in one of two ways: (1) with full support from the underlying hardware, affording maximum efficiency; and (2) without full support from the underlying hardware, requiring sensitive instructions to be emulated in software.

A popular full system virtualization VMM is VMware Workstation [23,25], hereafter referred to as simply VMware. VMware runs inside of a host operating system – as opposed to running on the raw hardware – and exposes an accurate representation of the x86 architecture to guest operating systems. This causes VMware to suffer a performance overhead during the execution of certain privileged instructions, since they must be emulated in software.

In paravirtualization, the virtual replica of the underlying hardware exposed is similar to the underlying machine, but it is not identical. This is done when the underlying machine architecture consists of sensitive instructions which are not privileged.

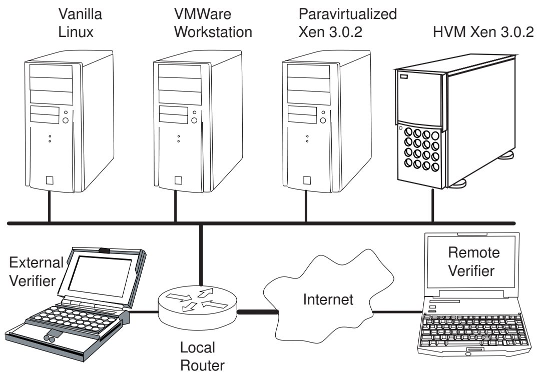


Fig. 6. Experimental machine and network setup

Paravirtualized VMMs have the drawback that operating systems must be modified to run on them; however, they enable efficient virtualization to be performed even when hardware support for full virtualization is unavailable.

Xen is an open-source x86 virtual machine monitor that uses paravirtualization to achieve high performance [2]. Xen presents a software interface to the guest OS that is not identical to the actual hardware. Therefore, the guest operating system needs to be modified before it can run on Xen. Paravirtualization is trivially detectable from within a guest OS, as certain features of the underlying hardware will be broken or missing. Full virtualization on Xen can be accomplished with hardware support, e.g., Intel Vanderpool Technology (VT) [8] or AMD SVM [5].

6.2 Experimental Setup

We use six machines in our VMM detection experiments. Figure 6 shows these machines and their network connectivity. Three of the machines are identical 2.0 GHz Intel Pentium IV systems. These systems run vanilla Linux, VMware Workstation, and paravirtualized Xen 3.0.2, respectively. The fourth machine has hardware extensions to support virtualization (e.g., Intel VT [8] or AMD SVM [5]) and runs Xen 3.0.2. The last two machines are used as verifiers in experiments where timing measurements are made remotely. One of these is on a separate subnet from our machines running VMMs, separated by one hop through a router, which we call the *external* verifier. The other is located remotely at another university, which we call the *remote* verifier. Average ping times to the external and remote verifiers are 0.4 ms and 16 ms, respectively. All CPU-scaling and power-saving features are disabled on the external and remote verifiers during experiments to prevent the clock frequency of the CPU in the verifier from changing.

In the remainder of the paper, we sometimes refer to a target host as “VMware” or “Xen”, when in fact we mean the guest OS running on VMware or Xen. All experiments run against Xen, with or without HVM support, are run against an unprivileged user domain which is the only other domain running besides the privileged domain 0.

In our experiments, we execute the benchmarking loop in the same privilege level as the OS kernel. Once the benchmarking loop executes on the target host, it turns off interrupts and executes a sequence of instructions that will experience detectable performance differences depending on the presence or absence of a VMM. Interrupts were disabled to improve the accuracy of timing measurements. Once the sequence of instructions executes, the VMM detection code re-enables interrupts and sends a notice of completion to the verifier.

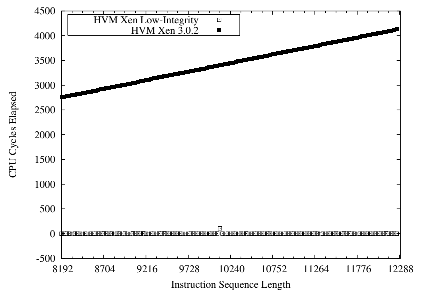
We must address one more issue before delving into our benchmarking loops: the issue of a heavily loaded target host. We compare the case where the target host is not running a VMM with the case where it is. If there is no VMM, then disabling interrupts in the benchmarking loop truly disables them. The benchmarking loop executes to completion without interruption, rendering the load on the target host irrelevant. If the target system is a guest running on a VMM, interrupts are *at least* disabled in that guest VM. Thus, only code executing in other guest VMs on the same VMM can affect performance. If another heavily loaded guest exists alongside the target guest, the performance of the target guest may be degraded. This performance degradation only applies on systems running VMMs, and will thus *improve* our chances of successfully detecting the VMM. All of our experiments are run without any extra load on the VMMs, hence we evaluate our VMM detection approach in the worst-case of an unloaded system.

6.3 Timing Integrity

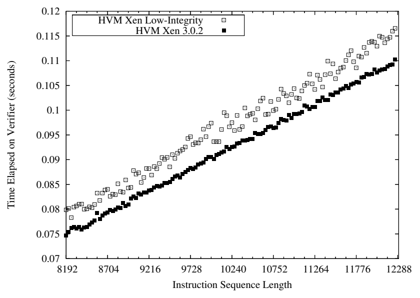
A VMM has total control over instructions executed by the guest OSes. Thus, we cannot trust a VMM to return valid answers to `rdtsc` “in the wild” [10]. Figure 7 compares internal (local) versus external timing measurements for the exact same experiment run on two variants of HVM Xen. One variant is the standard 3.0.2 release. The variant labeled as “Low-Integrity” in the figure is actually an unstable development release of Xen with a bug in the code which handles `rdtsc`. It is illustrative here because a party who wishes to thwart local VMM detection may intentionally modify their VMM to return such invalid timing measurements.

Figure 7(a) shows the internal timing measurements for a loop of a sequence of arithmetic instructions which clears interrupts at the beginning of each loop iteration. Xen 3.0.2 behaves as expected, with longer instruction sequences requiring longer to execute. In contrast, “Low-Integrity” Xen does not show any overhead whatsoever. In fact, some of the elapsed times are negative. Figure 7(b) shows a rerun of the same experiment, except that timing is performed by an external verifier. Local `rdtsc` calls are now unnecessary, and the runtime of the two experiments is nearly identical.

VMware Workstation can be made to demonstrate similar behavior. In fact, VMware provides a configuration option for VMs called `monitor_control.virtual_rdtsc` [24]. When set to `true`, a virtual counter in the VMM is used to provide values for guest OS calls to `rdtsc`. When set to



(a) Low timing integrity. Elapsed cycles measured internally using `rdtsc`. The same experiment yields dramatically different timing results on two variants of HVM Xen on the same physical machine.



(b) High timing integrity. Elapsed time measured via an external verifier. The same experiment yields similar results, even though one VMM was returning incorrect responses to `rdtsc` instructions.

Fig. 7. Timing integrity using internal versus external verifiers

`false`, VMware allows guest OS calls to `rdtsc` to access the CPU's true timestamp counter.

6.4 Identifying Remote Architectures

Inducing significant overhead in a VMM can result in long runtimes, which we detect by measuring runtime from a separate system. However, without some idea of the hardware architecture of the remote system in question, it is difficult to interpret timing results correctly. In this section, we describe a technique which is useful for identifying an unknown remote system as having an Intel Pentium IV CPU. If a system is known to be equipped with a Pentium IV, we can bound its expected performance (as demonstrated in Section 4). This bound is what allows for the establishment of a runtime threshold, above which it is likely that the target system is running a VMM. The Netburst Microarchitecture of the Intel Pentium IV family includes a trace cache with consistent specifications across all currently-produced Pentium IV CPUs [3]; our hardware discovery heuristics detect the presence of the trace cache. Other relevant characteristics of the Pentium IV microarchitecture include an out-of-order core and a rapid execution engine.

The trace cache stores instructions in the form of decoded μops rather than in the form of raw bytes which are stored in more conventional instruction caches [17]. These *traces* of the dynamic instruction stream permit instructions that are noncontiguous in a traditional cache to appear contiguous. A trace is a sequence of at most n instructions and at most m basic blocks (a sequence of instructions without any jumps) starting at any point in the dynamic instruction stream. An entry in the trace cache is specified by a starting address and a sequence of up to $m - 1$ branch outcomes, which describe the path followed. This facilitates removal of the instruction decode logic from the main execution loop, enabling the out-of-order core to schedule multiple μops to the rapid execution engine in a single clock cycle. In the case

```
rdtsc                ;; get start time
mov $131072, %edi    ;; n = 131072
loop:
xorl %eax, %eax      ;; begin special
addl %ebx, %ebx      ;; instr. seq.
movl %ecx, %ecx
orl %edx, %edx
...                  ;; 1K – 16K instr.
sub $1, %edi         ;; n = n − 1
jnz loop             ;; until n = 0
rdtsc                ;; get end time
```

Fig. 8. Example assembly code used to fill trace cache with register-to-register arithmetic instruction sequences without data hazards. These arithmetic instructions each decode to a single μop on Intel Pentium IV CPUs.

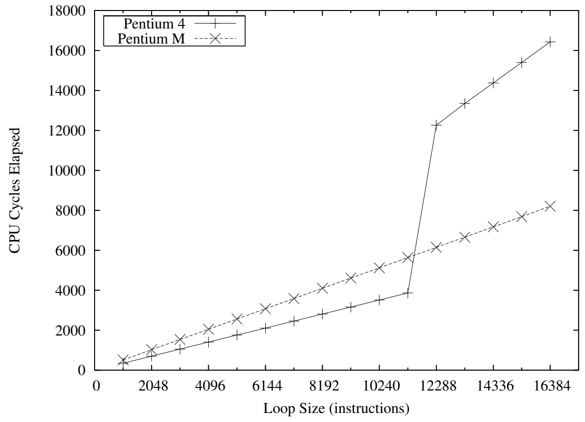


Fig. 9. When sequences of register-to-register arithmetic instructions without data hazards populate the trace cache of an Intel Pentium IV, a CPI of $\frac{1}{3}$ is attainable. Once an instruction sequence exceeds the trace cache's maximum size of 12KB, the CPI becomes 1. No such effect is visible on a Pentium M (an architecture without a trace cache). Cycles measured locally with `rdtsc`.

of register-to-register arithmetic instructions without data hazards, it is possible to retire three μops every clock cycle. Register-to-register x86 arithmetic instructions (e.g., `add`, `sub`, `and`, `or`, `xor`, `mov`) decode into a single μop. Thus, it is possible to attain a Cycles-Per-Instruction (CPI) rate of $\frac{1}{3}$ for certain sequences of instructions.

Intel has published the size of the trace cache in the Pentium IV CPU family – 12K μops. However, the parameters m and n, as well as the number of μops into which x86 instructions decode, have not been published. We performed an experiment where we executed loops of 1024 to 16384 arithmetic instructions devoid of data hazards on Pentium IV systems running vanilla Linux 2.6.16. Figure 8 shows the structure of our benchmarking loop. Figure 9 shows the results of this experi-

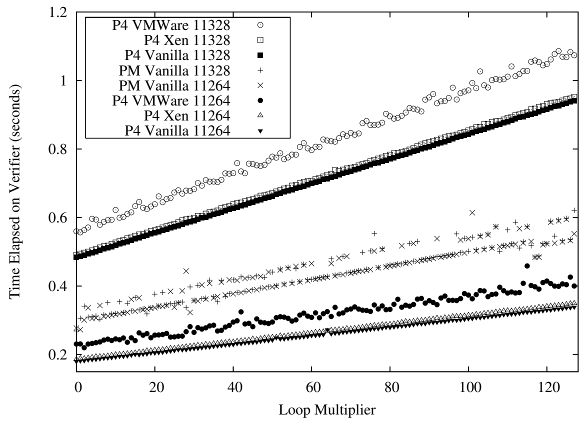


Fig. 10. Trace cache overhead timed remotely from another university. Sequences of either 11264 or 11328 arithmetic instructions with no data hazards are executed in a loop. The number of loop iterations is defined by $2^{17} + 2^{10}k$, where k is the Loop Multiplier on the X-axis. With and without a VMM, the Pentium IV architecture shows a considerable jump in overhead for a small number of additional instructions. In contrast, the Intel Pentium M (legend: PM) shows no such jump.

ment when run using the `rdtsc` – read time-stamp counter – instruction to measure the elapsed CPU cycles locally. On the Pentium IV, the CPI is $\frac{1}{3}$ until the number of instructions reaches Intel's published trace cache capacity of 12K μops. We also ran this experiment locally on a laptop equipped with a Pentium M CPU; no unusual caching effects are observed (note that a CPI of less than 1 is obtained for the entire loop).

At this point we know enough about the trace cache in Pentium IV CPUs to construct a loop that has sufficient trace cache overhead to be detectable over the Internet. As described above, the exact details of how the trace cache generates its traces are not published. We performed additional experiments like those of Figure 9 locally and determined that a benchmarking loop composed of a sequence of 11264 arithmetic register-to-register instructions fits inside the trace cache, but that a sequence of 11328 instructions does not fit. That these figures are less than 12K is expected, as there are additional instructions executed to maintain loop counters and jump back to the beginning of the loop. Thus, executing these sequences multiple times should cause the performance of the larger loop to suffer disproportionately with respect to its added length.

Since the benchmarking loops contain only innocuous instructions, VMMs allow them to execute directly. The exaggerated performance difference between the two loops is largely unaffected by the presence of a VMM. Figure 10 shows the results of an experiment designed to demonstrate this effect. The top three lines are the execution time for the smaller sequence (11264 instructions per loop iteration) on vanilla Linux, paravirtualized Xen, and VMware Workstation. The bottom three lines show the same with the larger sequence (11328 instructions per loop iteration). The middle two lines show the two sequences executed on a Pentium M running vanilla Linux; this serves to illustrate how minimal the runtime difference between the loops is when there is no trace cache involved. The gap between the execution time of loops of the smaller sequence and loops of the larger sequence is considerable making this overhead identifiable across the Internet.

6.5 Inducing Detectable VMM Overhead

Given the results of the previous section, we have partial configuration information about the remote architecture of the target host. For example, we know the CPU is a member of the Pentium IV family. As described in Section 4.3, we need sufficient overhead to distinguish between the slowest member of the CPU family running a native OS and the fastest member of the CPU family running a guest OS on a VMM.

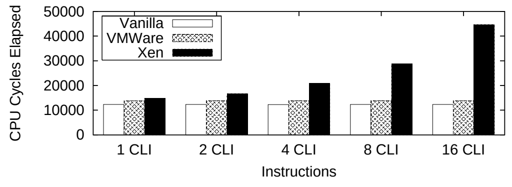
Recall that to detect a VMM, we must induce significant performance overhead. As described in Section 4, we use sensitive-privileged instructions which result in the execution of additional code inside the VMM. While we do not have space to exhaustively treat all sensitive instructions, we select a few and analyze their overhead on Xen 3.0.2 and VMware Workstation on an Intel Pentium IV. The instructions we consider are `cli` (clear interrupts), `mov %cr0, %eax` (read processor control register 0), `mov %cr2, %eax` (read processor control register 2), and `mov %cr3, %eax`; `mov %eax, %cr3` (read and write processor control register 3, which contains the physical address of the base of the page directory).

We next analyze these selected instructions locally on Xen 3.0.2, VMware Workstation, and vanilla Linux to understand their behavior (Section 6.5). Armed with this knowledge, we construct a remote attack that successfully detects the presence of a VMM across the Internet (Section 6.6).

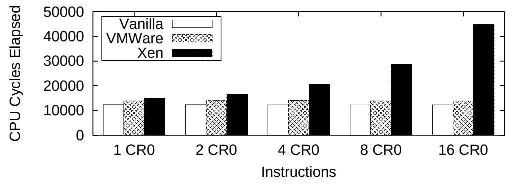
Per-Instruction Overhead

We configured VMware with the configuration setting `monitor_control.virtual_rdtsc = false` to provide guest OSes with direct access to the CPU's timestamp counter. Paravirtualized Xen 3.0.2 allows its guests to access the time stamp counter by default. Thus, we can run local experiments to analyze per-instruction overhead. Our analysis is based on experiments where a small number of one of the sensitive instructions in question are inserted in between sequences of register-to-register arithmetic instructions. For each sensitive instruction, we evenly space 1, 2, 4, 8, or 16 instances of that instruction among 12,256 arithmetic instructions. We selected 12,256 to ensure that trace cache effects would not add noise to our results. We cannot be sure how the trace cache would impact a smaller sequence of instructions because the exact μop structure of these sensitive instructions is not published.

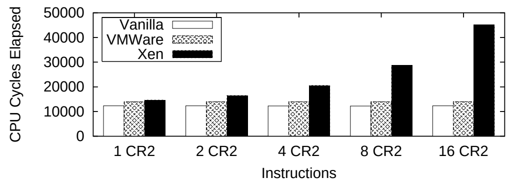
Figure 11 shows the results of local performance measurements. Figures 11(a), 11(b), and 11(c) yield very similar results. VMware Workstation shows a consistent minor overhead above vanilla Linux. In contrast, Xen's performance degrades significantly with each additional sensitive instruction. However, for CR3, we read its



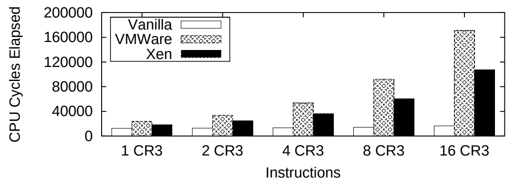
(a) `cli` (Clear Interrupts)



(b) `mov cr0, %eax` (Read Processor Control Register 0)



(c) `mov %cr2, %eax` (Read Processor Control Register 2)



(d) `mov %cr3, %eax; mov %eax, %cr3` (Read and then write Processor Control Register 3)

Fig. 11. Local execution times for selected sensitive instructions

current value and then rewrite that value. CR3 contains the physical address of the base of the page directory, thus the VMM must interpose on access to CR3 to uphold the resource control property. As Figure 11(d) shows, VMware Workstation incurs considerable overhead when it handles a write to CR3.

While reading and writing CR3 does not induce the worst overhead on Xen, the overhead is still significant. In the next section, we show how we use reads and writes to CR3 to detect a VMM across the Internet.

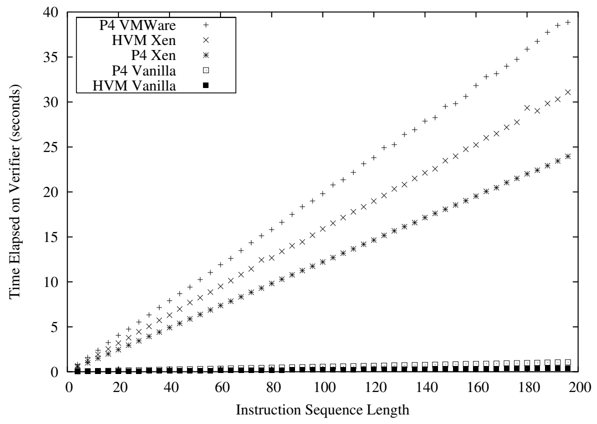


Fig. 12. Overhead resulting from reading and writing x86 Control Register 3 multiple times consecutively. Without a VMM executing, these instructions complete rapidly. With a VMM, there is sufficient overhead for remote detection via thresholding. Timed remotely from another university.

6.6 Successful Detection

We have established that an instruction sequence of reads and writes to CR3 results in VMM overhead when the target system is running either VMware or Xen. We used a loop containing a sequence of such instructions in our remote detection experiment. Although we did not include HVM Xen in our analysis of per-instruction overheads in the previous section, we include it in this experiment to validate our approach.

Figure 12 shows the results of our experiment, where the remote verifier is located at another university. We are able to induce extremely high overhead; code which executes in under 2 seconds on a native system takes more than 20 seconds to execute when running on either paravirtualized Xen, HVM Xen, or VMware Workstation. This is far above the amount of overhead necessary to overcome network latencies, allowing us to conclude that our approach to VMM detection is feasible.

7 Security Analysis

We have shown in the previous sections that it is possible to craft code which has pathological performance on a VMM, while still executing efficiently on bare hardware. This discrepancy provides an avenue through which motivated parties can detect VMMs. Recall that the execution of a detection algorithm has three logical stages:

Stage 1. For a target machine R_C, locate a hardware artifact to establish the configuration C of the machine.

Stage 2. Establish a reference time, $Baseline(R_C)$, for distinguishing between virtual and real machines with hardware configuration C.

Stage 3. Develop and execute a `Benchmark` which when running on top of a VMM on the fastest available machine for the architecture in question executes sufficiently slower than the `Benchmark` running in a native OS on the slowest available machine for the architecture in question.

We analyze the security of each stage individually, describing techniques which a VMM might deploy to evade or resist detection.

7.1 Stage 1 and 2 Evasion

A VMM can corrupt the results of stages 1 and 2 by masking all possible hardware artifacts that are observable through the VMM and simulating alternative artifacts from a slower machine. If a VMM were able to successfully simulate a slower machine, the baseline value established in stage 2 would be larger than necessary. This larger value might allow a VMM to execute a `Benchmark` without sufficient overhead to identify its presence.

Consider the case of a VMM running on an Intel Pentium IV. If this VMM is able to hide the existence of the trace cache, perhaps by masquerading as an Intel Pentium 3, then as a result of the speed difference between the Pentium IV and the Pentium 3, a detection attack may complete before the detection threshold for the Pentium 3, even with the overhead of the VMM.

For a VMM to successfully masquerade as a different architecture requires the following to be true: the configuration of the target machine is not known a priori and the VMM is able to simulate a slower device during stage 1 while still running at normal speed during stage 3. To successfully hide all hardware artifacts, the VMM would need to be a full system simulator. To execute at normal speed during stage 3, the VMM would have to be able to identify when the detection code is running since running a cycle-accurate simulator on its own incurs delays that are orders of magnitude larger than the overhead of any modern VMMs, making the simulator timings off the charts [16].

7.2 Stage 3 Evasion

To describe our assumptions with respect to a VMM's ability to evade detection, we specify two models of VMM behavior: experiential VMMs and propositional VMMs. Our models follow from partitioning the arms race of Section 1.1 based on a VMM's level of omniscience.

Experiential VMM. An experiential VMM has posteriori knowledge of experientially observed detectors but lacks identifiable information (i.e. process name, code signatures, etc.) for all detectors. It may deploy general countermeasures to evade detection such as virtualizing local timing sources (i.e., rdtsc, performance counters, etc.), but isn't able to analyze programs to infer their intent. Experiential VMMs may have a finite list of signatures to identify detectors, but are unable to prevent all detection attempts.

Propositional VMM. A propositional VMM has a priori knowledge of detectors and evades detection by disabling or tampering with detection attempts either before or during its execution.

A propositional VMMs is the case where the VMM can identify all detection algorithms and trivially thwart detection. Recent work on verifiable code execution on untrusted devices assumes a similar model of adversarial omniscience, however is not useful for VMM detection because it does not work across an uncontrolled network, such as the Internet [21].

Correctly identifying a detection attack makes it possible for the VMM to interpose and tamper with the execution of the attack. If the VMM realizes it is under a detection attack prior to the execution of the benchmarking loop, it may be able to prevent the detection attack from executing correctly, perhaps returning a valid response in the correct amount of time for a non-virtualized host.

Identifying that a particular code segment is a detection algorithm may be difficult. One potential approach is to rely on the unique structure of our detectors, for example, long sequences of the same operations, few or no I/O operations, and control-flow graphs with limited branching. These properties might provide sufficient invariants to generate signatures that match detection algorithms.

Even with the unique properties of our benchmarking loop, there are a number of difficulties inherent in evading detection. First, identification techniques could introduce false positives which would affect benign applications, secondly, a single false negative allows for the detection of the VM.

8 Discussion

We discuss limitations and potential extensions of our approach.

8.1 VMM Implementation Independence

While commodity VMMs aren't VMBRs specifically designed to thwart detection, they are implemented using the same techniques. As discussed in Section 3, these techniques necessitate the existence of VMM overhead. If hardware assisted VMMs become more common, then this overhead may be reduced, however our results show that current generation systems provide sufficient overhead for detection.

8.2 User-Level Detection

The detectors developed in this paper run at kernel-level rather than at user-level. In most scenarios, running a kernel-level detector is a reasonable assumption since the system's administrator is interested in detecting VMBRs. Administrators and users regularly run kernel-level integrity checkers and attackers continue to perform remote root exploits to gain administrator status. Statistical techniques may be necessary to overcome the resulting noise that user-level detection would incur.

8.3 Local VMM Detection

Rather than identify a target host as virtual or real by using an external source of time, local VMM detection aims to demonstrate to a user if their platform is virtual or real without a trusted time source. One potential approach is for a detector to observe the relative inter-leavings of short code sequences which are executed concurrently as a relative timing attack. If code sequences can be developed whose inter-leavings are virtualization sensitive, such an approach may be able to eliminate the requirement of a trusted time source.

8.4 Widespread Virtualization

As more and more machines run VMMs, the existence of a VMM becomes less of an anomaly. However, to dismiss VMM detection as useless in the face of widespread virtualization is too harsh. Legacy machines without VMMs will likely persist for many years to come. VMM detection algorithms like the ones developed in this paper can help protect these machines against VMBRs when upgrading is not an option. We believe that VMM detection will remain useful as long as non-virtualized platforms exist.

9 Related Work

Most related work either detects VMMs based on implementation details, use techniques which make assumptions that limit their applicability, or relies on the integrity of values returned from the VMM. In contrast, our detection algorithm has a higher degree of independence with respect to the implementation of the VMM on the target host, uses a hardware discovery heuristic to identify the configuration of remote devices, and incorporates a remote timing and decision maker to eliminate the need to trust the VMM.

Delalleau proposed a scheme to detect the existence of a VMM by using timing analysis [4]. The proposed scheme requires a program to first time its own execution on a VMM-free machine in a learning phase. Then, when the program infects a suspect host of known configuration, its execution time is compared against the results from the learning phase. Because the result of the learning phase is dependent on the exact machine configuration and the scheme is not designed to produce a configurable overhead, it is unclear how practical it is to deploy such a detection algorithm in practice.

Execution path analysis (EPA) [20] was first proposed in Phrack 59 by Jan Rutkowski as an attempt to determine the presence of kernel rootkits by analyzing the number of certain system calls. Although the main idea can also apply to detect VMMs, EPA has several severe drawbacks. The main drawback is that it requires significant modification to the system (debug registers, debug exception handler) that could be easily detected and consequently forged by the underlying VMM.

Pioneer [21] is a primitive which enables verifiable code execution on remote machines. As part of the inherent challenge of verifiable code execution, Pioneer needs to determine whether or not it is running inside a VMM. The solution in Pioneer is to time the runtime of a certain function that also reads in the interrupt enable bit in the EFLAGS register. This function is pushed into the kernel and is expected to run with interrupts turned off. However, if it was running inside a VMM, the output of the EFLAGS register would be different than expected. Although promising, Pioneer assumes that the external verifier knows the exact hardware configuration of the target host. We eliminate this assumption and rely on hardware artifacts to discover the target host's hardware configuration. In addition, the minimal timing overhead of the Pioneer checksum function makes remote usage of Pioneer difficult.

There are a number of previously developed techniques from the blackhat community. Redpill[4] is an example detection algorithm used to detect the VMware virtual machine monitor. Redpill operates by reading the address of the Interrupt Descriptor Table (IDT) with the `SIDT` instruction and checking if it has been moved to certain locations known to be used by VMware. This algorithm can be easily fooled since it relies on the VMM to return the correct address of the IDT [10]. Similar to Redpill, VMware's Back[5] is a software-dependent detection attack which uses the existence of a special I/O port, called the VMware backdoor. This I/O port is specific to the VMware virtual machine and hence can be used to detect VMware.

Holz and Raynal describe some heuristics for detecting honeypots and other suspicious environments from within code executing in said environment [7]. Dornseif et al. study mechanisms designed specifically to detect the Sebek high-interaction honeypot [6]. Unlike these approaches, the detection algorithm we have constructed are not based upon specific software artifacts.

Vrable et al. touch briefly on non-trivial mechanisms for detecting execution within a VMM [26]. They allude to the fact that although a honeynet may be able to perfectly virtualize all hardware, an attacker may be able to infer that it is executing inside a VMM through side channel measurements.

Robin and Irvine analyzed the Intel Pentium's architecture and ISA [14] and pointed out problems in implementing a secure VMM on the Intel Pentium architecture. For instance, certain instructions break hardware virtualization requirements because they read sensitive registers and/or memory locations (e.g., the clock register and interrupt registers), but are not privileged instructions. Execution of such instructions does not raise an exception, and thus allows the attacker to read sensitive system data. However, the VMM can perform binary translation when it loads the process into memory, and change all such instructions into system calls. Alternatively, the VMM can expose a paravirtualized version of the underlying hardware, which Xen does on the Intel x86 architecture [2].

Remote physical device fingerprinting can be used to detect VMMs if the external verifier can directly interact with two different virtual machines running on the same host [11]. Our approach only requires the existence of a single VM and hence is

[4] `http://invisiblethings.org/redpill.html`
[5] `http://chitchat.at.infoseek.co.jp/vmware/`

useful in the case of virtual machine based rootkits [10]. Also, defending against remote physical device fingerprinting is as simple as disabling or masking the TCP option timestamps. HoneyD is an example virtual honeypot which defends against remote physical device fingerprinting [13].

10 Conclusions

The main contribution of this article is the development of a detection algorithm whose execution differs from the perspective of an external verifier when a target host is virtual (versus when it is executed directly on the underlying hardware). Our detection algorithm is based on the timing dependency exception property of a virtual machine monitor. We presented results where a single benchmarking program generates sufficient overhead on several different virtual machine monitors to be remotely detectable across the Internet. Included in our analysis is a machine with hardware virtualization support. The success of our detection algorithm against this platform demonstrates that hardware support for virtualization is not sufficient to prevent VMM detection.

11 Acknowledgments

We thank Garth Gibson and Adam Pennington for their instruction and guidance in the early stages of this project. We thank Michael Kozuch for his insightful comments and useful discussions. Finally, we thank Ahren Studer for his assistance preparing a preliminary version of this paper.

References

1. K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.
2. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2003.
3. D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, Singhal R, B. Toll, and K. S. Venkatraman. The microarchitecture of the Intel Pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(1), February 2004.
4. G. Delalleau. Mesure locale des temps d'execution: application au controle d'integrite et au fingerprinting. In *Proceedings of SSTIC*, 2004.
5. Advanced Micro Devices. AMD64 virtualization: Secure virtual machine architecture reference manual. AMD Publication no. 33047 rev. 3.01, May 2005.
6. M. Dornseif, T. Holz, and C. Klein. Nosebreak - attacking honeynets. In *Proceedings of the 2004 IEEE Information Assurance Workshop*, June 2004.

7. T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. In *Proceedings of the IEEE Workshop on Information Assurance and Security*, June 2005.
8. Intel Corporation. Intel virtualization technology. Available at: `http://www.intel.com/technology/computing/vptech/`, October 2005.
9. X. Jiang, D. Xu, H. J. Wang, and E. H. Spafford. Virtual playgrounds for worm behavior investigation. In *8th International Symposium on Recent Advances in Intrusion Detection (RAID '05)*, 2005.
10. S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2006.
11. T. Kohno, A. Broido, and K. Claffy. Remote physical device fingerprinting. In *IEEE Symposium on Security and Privacy*, May 2005.
12. G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17, July 1974.
13. N. Provos. Honeyd: A virtual honeypot daemon. In *Proceedings of the 10th DFN-CERT Workshop*, 2003.
14. J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the USENIX Security Symposium*, 2000.
15. R. Rose. Survey of system virtualization techniques. Available at: `http://www.robertwrose.com/vita/rose-virtualization.pdf`, March 2004.
16. M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, Winter 1995.
17. E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, November 1996.
18. J. Rutkowska. Subverting Vista kernel for fun and profit. Presented at Black Hat USA, 2006.
19. J. Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. `http://invisiblethings.org/papers/redpill.html`, 2004.
20. J. Rutkowski. Execution path analysis: finding kernel rootkits. *Phrack*, 11(59), July 2002.
21. A. Seshadri, M. Luk, E. Shi, A. Perrig, L. VanDoorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of the Symposium on Operating Systems Principals (SOSP)*, 2005.
22. S. Staniford, V. Paxson, and N. Weaver. How to 0wn the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium (Security '02)*, 2002.
23. G. Venkitachalam and B. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Technical Conference*, 2001.
24. VMWare. Timekeeping in VMWare virtual machines. Technical Report NP-ENG-Q305-127, VMWare, Inc., July 2005.
25. VMWare. VMWare Workstation. Available at: `http://www.vmware.com/`, October 2005.
26. M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity and containment in the potemkin virtual honeyfarm. In *Proceedings of the Symposium on Operating Systems Principals (SOSP)*, 2005.
27. D. D. Zovi. Hardware virtualization-based rootkits. Presented at Black Hat USA, August 2006.

Index