# Remote Detection of Virtual Machine Monitors with Fuzzy Benchmarking


Jason Franklin
Carnegie Mellon University

Mark Luk
Carnegie Mellon University

Jonathan M. McCune
Carnegie Mellon University

Arvind Seshadri
Carnegie Mellon University

Adrian Perrig
Carnegie Mellon University

Leendert van Doorn
Advanced Micro Devices



## Abstract

*We study the remote detection of virtual machine monitors (VMMs) across the Internet, and devise fuzzy benchmarking as an approach that can successfully detect the presence or absence of a VMM on a remote system. Fuzzy benchmarking works by making timing measurements of the execution time of particular code sequences executing on the remote system. The fuzziness comes from heuristics which we employ to learn characteristics of the remote system's hardware and VMM configuration. Our techniques are successful despite uncertainty about the remote machine's hardware configuration.*


## 1. Introduction

The ability to remotely detect a virtual machine monitor (VMM) is important in many circumstances: detecting malware that has been implemented with VMMs (e.g., VM-based rootkits [11, 16, 22]); detecting virtualized environments used for dynamic analysis such as honeypots [10, 19]; preventing data lifetime or freshness attacks that use VMMs [3, 7]; and preventing time-limited trial software from being reused. These uses for VMM detection span both the white hat and black hat communities. Even in the case of malicious uses of VMM detection technology, it is important for researchers to explore this space to facilitate development of countermeasures.

We are interested in remotely detecting the presence of a VMM on a particular system. Given the exact hardware specifications and the specific VMM implementation that may be present, detection using timing attacks is straightforward, as we show in this paper. However, given all known VMM implementations, and all possible hardware configurations, detecting the presence of a VMM on a platform with unknown hardware is an open problem. This problem spans a spectrum of VMM-detection scenarios, running from specific (easier to detect) to general (harder to detect) along two axes: VMM implementations and hardware configurations (see Figure 1). We do not claim that one point in the space is more useful or more important than another; rather, it is our goal to explore the space and gain some understanding of the challenges that lie within.

Complete knowledge of the remote system is available in some scenarios: suppose VM-based rootkits (VMBRs) become a significant threat in the wild. Anti-virus software makers are motivated to detect such threats, and may require that users specify their exact hardware configuration upon installation of the latest malware signatures. The anti-virus software companies' servers could then periodically challenge the users' systems to execute certain instruction sequences, designed to elicit slow performance in the presence of a VMM on a user's system. If performance is degraded sufficiently, the anti-virus software company suspects the presence of a VM-based rootkit. Due to the nature of VMBRs, out-of-band mechanisms may be necessary to inform the user of this detection, but that is outside the scope of this paper. However, several problems arise in this model. The information provided by the user about their system might be incomplete or wrong.

We devise *fuzzy benchmarking* as an approach to detect the presence of a VMM on a remote system with uncertainty about the remote system's exact hardware configuration or specific VMM implementation. We are able to successfully detect whether the remote system has an Intel Pentium IV, and whether it is running vanilla Linux, Xen 3.0.2, or VMware workstation. Further, our *fuzzy benchmarks* continue to work against a machine with hardware support for virtualization running Xen 3.0.2. While our results do not prove that all VMMs are detectable on all hardware platforms, they suggest that a motivated entity with some knowledge of the remote system in question is likely to be able to construct a *fuzzy benchmark* that demonstrates performance degradation when running on a VMM. As such, our work represents a first step towards general VMM detection techniques.

**Contributions.** This work makes the following contributions:

- We introduce the problem space relating to VMM detection.
- We propose the *fuzzy benchmark* approach based on timing VMM overhead on machines of uncertain configuration.
- We design and implement the *fuzzy benchmark* approach for some commodity architectures and VMMs.
- We evaluate our *fuzzy benchmark* experiments, which successfully detect the presence of commodity VMMs executing on commodity x86 hardware, including hardware with virtualization support.

**Outline.** The remainder of our paper is organized as follows. Section 2 provides an overview of the VMM detection problem. Section 3 motivates the design of our detection algorithm based on both theoretical and practical constraints. The implementation and experimental evaluation of our detection algorithm is presented in Section 4. We treat related work in Section 5. Finally, Section 6 offers our conclusions.

## 2. Problem Space

We define the problem of *VMM detection*, in which a program, called a *fuzzy benchmark*, executes on a remote host in order to determine if that remote host is a virtual machine running on virtualized hardware or a real machine running directly on hardware. We term the benchmark a *fuzzy benchmark* because uncertainties with respect to the hardware and VMM configuration of the remote system preclude the benchmark being designed for a specific target system. Further, a set of all hardware or VMM configurations is intrinsically vaguely defined, since the space of all possible configurations is infinite.

We follow Popek and Goldberg [13] in defining a virtual machine as an efficient, isolated duplicate of the underlying hardware. This definition imposes the three properties that a control program must satisfy to be termed a VMM: efficiency, resource control, and equivalence.

1. The **efficiency property** dictates that a statistically dominant subset of the virtual processor's instructions be executed directly by the real processor, without requiring intervention by the VMM.

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

3. The **equivalence property** dictates that the VMM provide an environment for programs which is essentially identical to that of the original machine. 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.

The resource availability exception to the equivalence property 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 the VMM shares the underlying hardware and hence consumes some 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 requirement that the VMM maintain control over system resources by interposing on control-modifying instruction.

In this paper, we exploit the timing dependencies to construct *fuzzy benchmarks* which demonstrate measurable overhead, even in the presence of uncertainty as to the remote system's hardware and possible VMM configuration. Uncertainties with respect to the hardware configuration include CPU microarchitecture, cache architecture, memory size, 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 [9] or AMD's SVM [5] technologies, further complicates VMM detection.

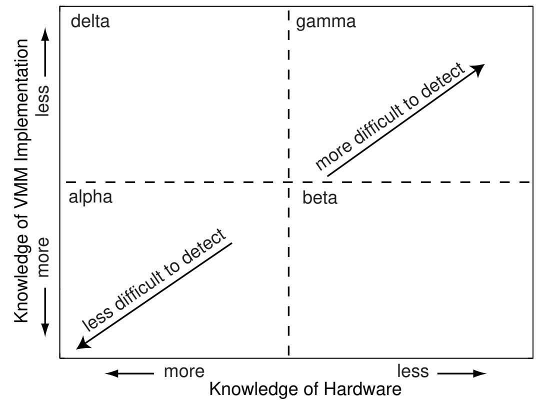



**Figure 1. VMM detection problem space.**

We envision the VMM detection problem space along two axes (see Figure 1): level of knowledge about the remote system's hardware configuration, and level of knowledge about the remote system's possible VMM implementation. For example, VMware has enjoyed a dominant position in the VMM market for years. Thus, it may be reasonable to assume that a remote system is using VMware. However, Xen has recently been growing in popularity. Still, guessing that the remote system is executing a specific version of Xen or VMware is unlikely to succeed if the remote system wants to hide its VMM.

One potential hurdle for timing-based detection techniques is the arrival of new hardware with support for virtualization. However, Adams and Agesen show that even with today's latest hardware support for virtualization, VMMs experience considerable performance overhead [1]. Our experimental results confirm these observations. We cannot predict what capabilities future hardware may have to improve virtualization performance; thus, our results reflect only the today's state of the art.

## 3. Design of Fuzzy Benchmarks

In this section, we describe our design of a VMM detection technology called *fuzzy benchmarking*, with applications in each of the four quadrants identified in Figure 1. Though we discuss some local measurements to aid understanding, our design facilitates remote detection of a VMM using a detection protocol across the Internet for each quadrant (results in Section 4). We first present our detection scenario and assumptions, then describe the remote VMM detection protocol, followed by the intricacies of *fuzzy benchmarks* for each quadrant.

### 3.1 Scenario and Assumptions

We have two parties in our VMM detection scenario, an external verifier and a target host, connected via the Internet. The external verifier would like to determine whether the target host is a virtual machine. To this end, the external verifier installs and executes some benchmarking code on the target host. The VMM, if present, is assumed to be a full system VMM such as VMware, Xen, or Virtual PC running on unmodified commodity x86 hardware (including but not limited to the Intel Pentium family, Intel VT [9], and AMD SVM [5]). The VMM may make use of hardware extensions to support virtualization, if present. We assume that the external verifier has root access to at least one VM running inside the VMM, which it uses to execute benchmarking code at the highest privilege level with interrupts turned off. If the target host is a VM, the VMM may execute at a higher privilege level, interrupts may only be disabled for this VM, and the VMM may return invalid information to direct inquiries from VMs about elapsed time or performance. We do not assume any knowledge concerning which quadrant of Figure 1 the target host is operating in. This is the source of the fuzziness in our benchmarking; the external verifier may know only a vaguely defined set of possible hardware configurations or VMM implementations that describe the target host.

## 3.2 Remote Detection Protocol

The remote verifier attempts VMM detection on the target host from a remote Internet site. The remote verifier is assumed to have an accurate clock which is used to measure the runtime of the *fuzzy benchmark* on the target host. Clock synchronization is not required between the machines because all timing measurements are made on the remote verifier.

The remote verifier installs and prepares for execution a *fuzzy benchmark* $B$ on the target host. $B$ executes when directed to do so by the remote verifier, and sends the remote verifier a notification of completion after execution finishes. Upon receiving the notification of completion, the remote verifier records the time elapsed since requesting the invocation of $B$. To determine if the detection algorithm $B$ was executed on top of a VMM, the remote verifier checks whether the execution time of $B$ is greater than a predetermined threshold value $\tau_R$ (described below) for the target host's set of suspected hardware platforms. If the execution time exceeds the threshold, the target host is considered to be a VM and hence a VMM was in control of the execution of $B$.

To remotely detect VMM overhead, we must develop a program $B$ with sufficient VMM overhead to overcome variance in network latency, inaccuracies in timing, and any other sources of noise which may exist. To achieve adaptability in the face of future changes in the noise level, we develop $B$ with a configurable amount of VMM overhead, at the expense of the overall runtime of the *fuzzy benchmark*. The remainder of this section discusses how *fuzzy benchmarks* operate, including obtaining an appropriate value of $\tau_R$ for target hosts in each quadrant of Figure 1.

## 3.3 Fuzzy Benchmarking Architecture

We design a *fuzzy benchmark* $B$ which includes control-modifying CPU instructions that we empirically determine to cause VMM overhead across several VMM implementations. We choose the particular control-modifying CPU instructions (thereby requiring intervention by the VMM to maintain the resource control property) and then tune their number such that the VMM overhead is noticeable across the Internet. The execution time of $B$ on real machine $R$, $\tau_R$, is our threshold for distinguishing between the execution of $B$ on a real machine and execution on a virtual machine. The accuracy of our *fuzzy benchmark* is dependent on a correct value for $\tau_R$. One complexity that arises is how one determines the value of $\tau_R$ for machines of unknown hardware configuration (quadrants beta and gamma in Figure 1).

Since the execution time of $B$ is dependent on the underlying hardware, we require some knowledge of the hardware configuration. We have developed a heuristic that relies on the existence of hardware artifacts that "shine through" the VMM. An artifact shines through a VMM if it is possible to identify the existence of the artifact by observing the effects of commands executed in a VM. The hardware artifacts we discover are unique to a particular CPU microarchitecture and allow us to infer a portion of the configuration of the target host. In the context of Figure 1, this moves the level of knowledge in the external verifier further to the left. For example, the artifact we explore in Section 4 is the existence of a trace cache present in all Intel Pentium IV CPUs. This configuration information then allows for an estimation of the expected runtime of $B$ on $R$, $\tau_R$, since the external verifier can assume the target host is a member of the Intel Pentium IV family.

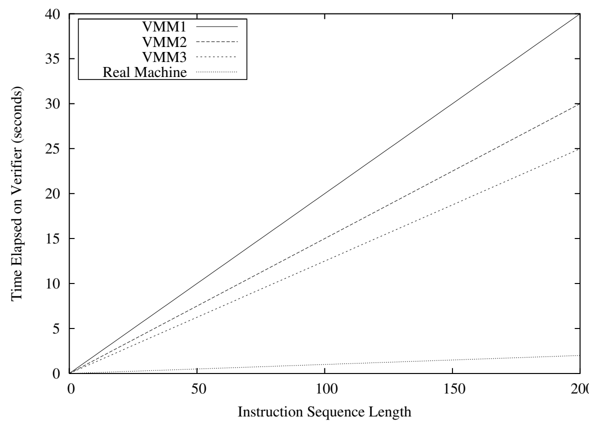



**Figure 2. Example VMM overheads for program $B$. Without a VMM executing, the instructions in $B$ complete rapidly. With a VMM, there is noticeable overhead.**

To select control-modifying instructions to induce VMM overhead, we measured the overhead of different control-modifying instructions on several different VMMs. After selecting particular instructions to include in the program $B$, we need to further tune the VMM overhead induced by $B$ by selecting the number of instructions. There are at least two factors in addition to network-induced noise that affect the VMM overhead of $B$. First, the configuration of the machine, for instance, Intel Pentium IV 2.0 GHz, has a direct effect on the execution time. Second, different virtual machine monitor implementation strategies result in different overheads (e.g., paravirtualization, binary rewriting, or the presence of a host OS under the VMM). The following analysis explains how we incorporate these two factors into our experiments in order to select the number of instructions in $B$.

First, we look at the optimistic case where we assume full knowledge of the configuration of the target machine, i.e., the bottom left of quadrant alpha in Figure 1. We then reduce the amount of configuration information that is known and develop a technique for estimating $\tau_R$ in the other quadrants.

### 3.3.1 Detailed Configuration Information – Quadrant Alpha

Given local access to a machine about which we know detailed configuration information, we can determine the number of instructions to include in a *fuzzy benchmark* by estimating the noise on the Internet and running a number of experiments to time the *fuzzy benchmark* directly on the real hardware and on different VMMs. The results of these experiments can generate a graph similar to Figure 2.

Figure 2 is a distilled version of Figure 9 from our experimental results, and serves to illustrate our point here. The upper three lines


**Figure 3. The required order of execution times of *fuzzy benchmark B* for successful VMM detection given a set of partial configuration information. FR (fastest real), SR (slowest real), FV (fastest virtual machine), and SV slowest virtual machine) are machines which conform to this partial configuration information.**

represent the runtimes of $B$ with a fixed set of control-modifying instructions under several different VMM implementations. The bottom line is the execution time of $B$ on the real hardware. To determine the required number of instructions denoted by $x$, we first generate equations for all the lines in the graph. We then derive the minimum number of instructions required to overcome our noise estimate as follows.

Given a model:

$$\left\{ \begin{array}{ccc} VMM_1(x) & = & a_1 x \\ VMM_2(x) & = & a_2 x \\ VMM_3(x) & = & a_3 x \\ RealMachine(x) & = & bx \end{array} \right\}$$

We set $a = min(a_1, a_2, a_3)$ and denote $FastestVMM(x) = ax$. With a noise estimate of $n$, we select $x$ such that

$FastestVMM(x) - RealMachine(x) \gg n$

or equivalently, $x \gg \frac{n}{a-b}$. Since $n$ is less than a second in practice and our VMM overhead is configurable into the tens of seconds, selecting $x$ based on local experiments presents few difficulties. We describe the configurability of *fuzzy benchmarks* in Section 4 when we present some *fuzzy benchmarks* in detail.

### 3.3.2 Fuzzy Configuration Information – Quadrants Beta, Delta, and Gamma

We now examine the case where we have fuzzy configuration information for the target host. For example, we have configuration information derived from the use of our hardware discovery heuristic. In this case, we determine the correct number of instructions to execute based on a number of estimates and experiments. We assume that we have local access to a machine with partial configuration information which matches that of the target host, though even the partial configuration information may be hard to obtain in, e.g., quadrant gamma.

**Gaining knowledge of the target host's hardware configuration.** As an example, imagine that the partial configuration information we have identifies just the processor microarchitecture (e.g., Intel Pentium IV). Since the target host 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 of $B$ for different configurations and use these bounds for detection. In addition, since our runtime estimates for $B$ will not be as accurate as in the full configuration information case, we must design $B$ such that the execution time of $B$ is ordered as in Figure 3. Essentially, executing $B$ on the fastest VMM on the fastest real machine that matches the partial configuration information should take longer than executing $B$ directly on the slowest real machine matching the partial configuration information.

Our approach is to determine the range of processor speeds available given our partial configuration information and to use these values to approximate $B's$ execution time under different configurations. 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 $B$ (detailed in Section 4) 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 Intel Pentium IV.[1]

As in Section 3.3.1, we experimentally determine $FastestVMM(x) = ax$ and $RealMachine(x) = bx$ by running tests with different VMMs (including no VMM) 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 of $B$ on the real hardware of $S$. This gives us a runtime estimate on $S$ of $SlowestReal(x) = p * RealMachine(x)$. 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 $FastestVirtual(x) = u * FastestVMM(x)$. To determine the minimum number of instructions required to overcome our noise estimate, we require $FastestVirtual(x) > SlowestReal(x) + n$ or equivalently, $x > \frac{n}{au-bp}$.

**Gaining knowledge of the target host's VMM implementation.** Thus far, we have described techniques for increasing the external verifier's knowledge of the remote hardware (moving to the left in Figure 1). To obtain high confidence in a VMM detection, knowledge of the target host's hardware configuration is insufficient; we must also gain additional knowledge about the VMM implementation on the target host.

Suppose preliminary experiments with a target host suggest the presence of a VMM, e.g., induced TLB flushes are unusually expensive (due to the existence of shadow page tables). Consider a VMM that uses binary rewriting to maintain control. The VMM rewrites control-modifying instructions which might require a trap into the VMM instead replacing them with innocuous instructions that modify virtualized system state. This use of binary rewriting may make some otherwise slow control-modifying instructions more efficient. If the presence of a binary-rewriting VMM is suspected, additional timing-based experiments can be performed to exploit the overhead induced by binary rewriting. We successfully perform exactly this experiment in Section 4, initially detecting that a VMM is likely to be present because of TLB flush overheads, and then refining our knowledge of that VMM by detecting the use of binary rewriting (VMware Workstation uses binary rewriting; Xen 3.0.2 does not).

## 4. Implementation and Evaluation

We present implementation details and an evaluation of experiments which allow a remote verifier to learn about the hardware configuration and potential VMM implementation on a target host. We first describe our experimental setup, then discuss the actions necessary to ensure timing integrity for our experiments. Mech-

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

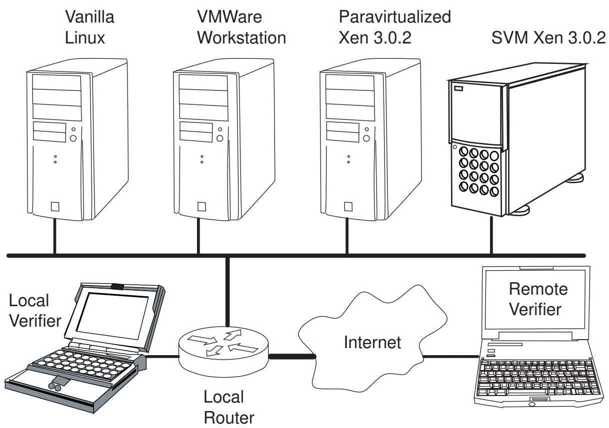



**Figure 4. Experimental machine and network setup.**

anisms that can detect the hardware architecture of an unknown remote system are presented next. Finally, we provide a detailed discussion of the development of code that induces overhead in virtual machine monitors, including the ability to distinguish between different VMM implementations.

## 4.1 Experimental Setup

We use six machines in our VMM detection experiments. Figure 4 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 xenolinux on Xen 3.0.2, respectively. The fourth machine has AMD SVM [5] hardware extensions to support virtualization 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 *local* verifier. The other is located remotely at another university, which we call the *remote* verifier. Average ping times to the local and remote verifiers are 0.4 ms and 16 ms, respectively. All CPU-scaling and power-saving features are disabled on the local and remote verifiers during experiments to prevent the clock frequency of the CPU in the verifier from changing.

We detect the presence of a VMM based on performance measurements of specially crafted instruction sequences, which we execute in a loop called the benchmarking loop. We iterate the loop containing the sequence to generate enough overhead for detection. Unless stated otherwise, our loop iterates $2^{17}$ times. We selected this value experimentally.

Our benchmarking loops execute within a Linux kernel module. 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 a local 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.

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 SVM support, are run against an unprivileged user domain which is the only other domain running besides the privileged domain 0.

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 perform VMM detection in the worst-case of an idle system.

## 4.2 Timing Integrity

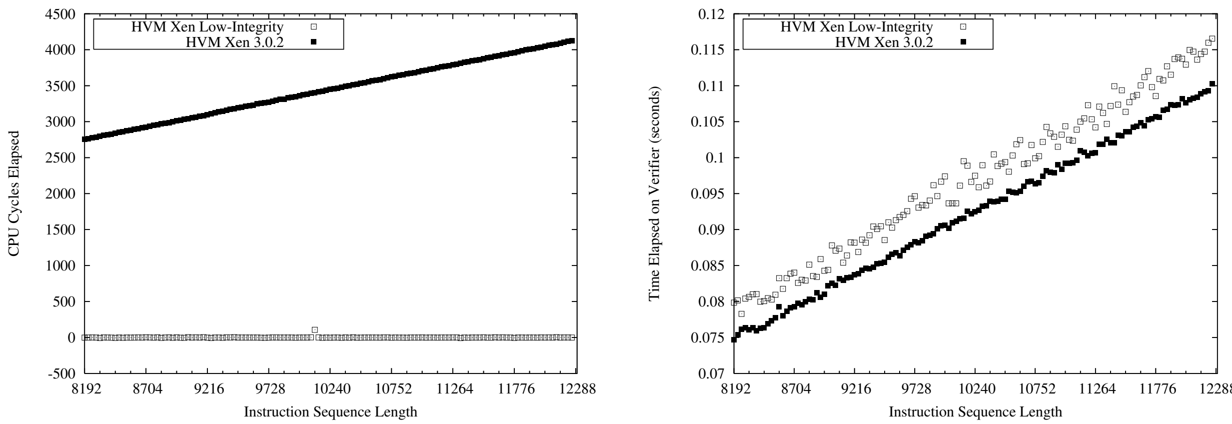
A VMM has total control over its guest OSes. Thus, we cannot trust a VMM to return valid answers to `rdtsc` "in the wild" [11]. Figure 5 compares internal (local) versus external timing measurements for the exact same experiment run on two variants of SVM Xen. One variant is the standard 3.0.2 release. The variant labeled as "Low-Integrity" in figures is actually an unstable development release of Xen with a bug in the code that 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 5(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 5(b) shows a rerun of the same experiment, except that timing is performed by a local 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, though we omit experimental results due to lack of space. In fact, VMware provides a configuration option for VMs called `monitor_control.virtual_rdtsc` [20]. When set to `true`, a virtual counter in the VMM is used to provide values for guest OS calls to `rdtsc`. When set to `false`, VMware allows guest OS calls to `rdtsc` to access the CPU's true timestamp counter.

## 4.3 Identifying Target Host Hardware Architectures

Inducing significant overhead in a VMM can result in long runtimes, which we detect by measuring runtime from a separate sys-




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

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

**Figure 5. Timing integrity using internal versus local verifiers.**

tem. 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 that is useful for identifying an unknown remote system as having an Intel Pentium IV CPU, thereby providing more knowledge in the context of Figure 1. If a system is known to be equipped with a Pentium IV, we can bound its expected performance and establish 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 [2]; 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.

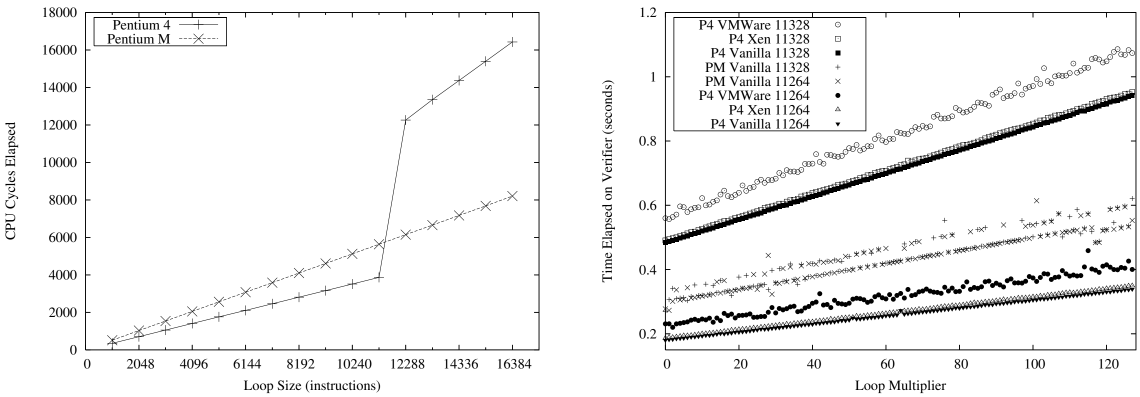
The trace cache stores instructions in the form of decoded $\mu$ops rather than in the form of raw bytes which are stored in more conventional instruction caches [15]. These *traces* of the dynamic instruction stream ensure that instructions that are noncontiguous in a traditional cache to appear contiguous. A trace is a sequence of at most $n$ instructions and at most $m$ basic blocks (a sequence of instructions without any branches) 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 $\mu$ops to the rapid execution engine in a single clock cycle. In the case of register-to-register arithmetic instructions without data hazards, it is possible to retire three $\mu$ops every clock cycle. Register-to-register x86 arithmetic instructions (e.g., `add`, `sub`, `and`, `or`, `xor`, `mov`) decode into a single $\mu$op. 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 $\mu$ops. However, the parameters $m$ and $n$, as well as the number of $\mu$ops into which x86 instructions decode, have not been published. We performed an experiment where we executed benchmarking loops of 1024 to 16384 arithmetic instructions devoid of data hazards on Pentium IV systems running vanilla Linux 2.6.16. Figure 6 shows the structure of our benchmarking loop. Figure 7(a) shows the results of this experiment when run using the `rdtsc` instruction to measure the elapsed CPU cycles locally. It is evident on the Pentium IV that the CPI is indeed $\frac{1}{3}$ until the number of instructions reaches Intel's published trace cache capacity of 12K $\mu$ops. 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).

```
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
```

**Figure 6. Example assembly code used to fill trace cache with register-to-register arithmetic instruction sequences (which decode to a single $\mu$op) without data hazards on Intel Pentium IV CPUs.**

At this point we know enough about the trace cache in Pentium IV CPUs to construct a benchmarking loop that has sufficient overhead to be detected remotely across the Internet, thus allowing the remote verifier to gain knowledge about the target host's hardware configuration. 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 7(a) locally and determined that a benchmarking loop composed of a sequence




(a) 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`.

(b) 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.

**Figure 7. Measurements of trace cache overhead.**

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 7(b) 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, and we are able to detect this even across the Internet.

## 4.4 Inducing Detectable VMM Overhead on the Target Host

At this point, we have some idea about the remote architecture on which we are trying to detect a VMM. For example, we know the CPU is a member of the Pentium IV family. As described in Section 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.

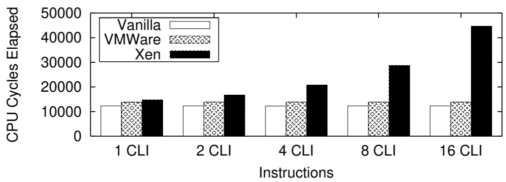
Thus, we must induce significant performance overhead in a VMM. As described in Section 3, we use control-modifying instructions which result in the execution of additional code inside the VMM. While we do not have space to exhaustively treat all sensitive instructions, we select a few and analyze their 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 induces a TLB flush).

We next analyze these selected instructions locally on Xen 3.0.2, VMware Workstation, and vanilla Linux to understand their behavior (Section 4.4.1). Armed with this knowledge, we construct a remote attack that successfully detects the presence of a VMM across the Internet (Section 4.4.2). We also present a technique which causes instructions which would normally execute efficiently on VMware to perform poorly (Section 4.4.3), thereby allowing us to distinguish between Xen and VMware.

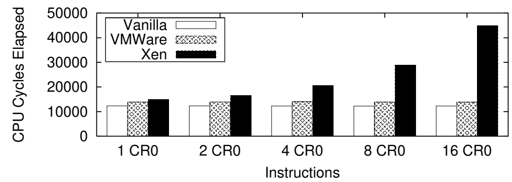
### 4.4.1 Per-Instruction Overhead

We configured VMware with
`monitor_control.virtual_rdtsc = false`
so that VMware allows guest OSes to have 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 $\mu$op structure of these sensitive instructions is not published.

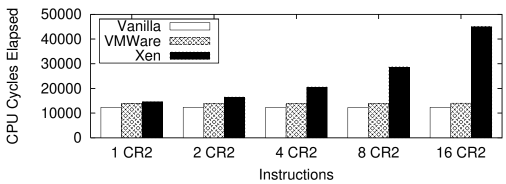
Figure 8 shows the results of local performance measurements.



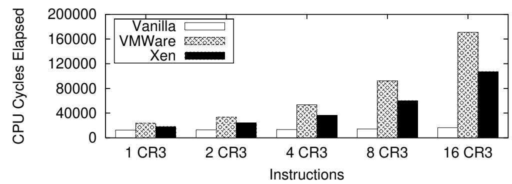

(a) `cli` (Clear Interrupts)




(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)

**Figure 8. Local execution times for selected sensitive instructions.**

Figures 8(a), 8(b), and 8(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 read/writes (Figure 8(d) – causing a TLB flush in the VM), VMware Workstation incurs considerable overhead above both Xen and vanilla Linux.

We now have two techniques which induce overhead differently on two different VMM implementations – CR3 read/writes and CR0 reads. While CR3 read/writes do not induce the worst overhead on Xen, the overhead is still significant. In the next section, we show how we use CR3 read/writes to detect a VMM across the Internet.

### 4.4.2 Successful Remote Detection

We have now determined that an instruction sequence of reads and writes to CR3 results in overhead when the target system is running a VMM, but reasonable performance when the target system is executing natively. We used a loop containing a sequence of such instructions in our remote detection experiment. Although we did not include SVM Xen in our analysis of per-instruction overheads in the previous section, we include it in this experiment.

Figure 9 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, SVM Xen, or VMware Workstation. This is far above the amount of overhead necessary to overcome network latencies, and we conclude that remote VMM detection is feasible.

### 4.4.3 Remotely Distinguishing between Xen and VMware

VMware Workstation uses binary rewriting to emulate many sensitive instructions efficiently. This is one reason its performance is able to closely track that of the native system in Figure 8. However, for correctness, VMware must still be capable of detecting dynamic changes to program executables. For example, self-modifying code may appear innocuous when first loaded, but may then morph itself into one or more sensitive instructions.

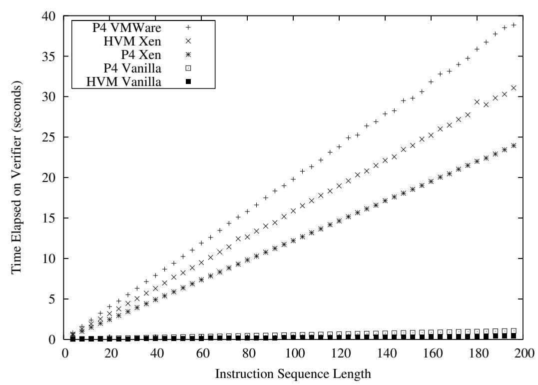
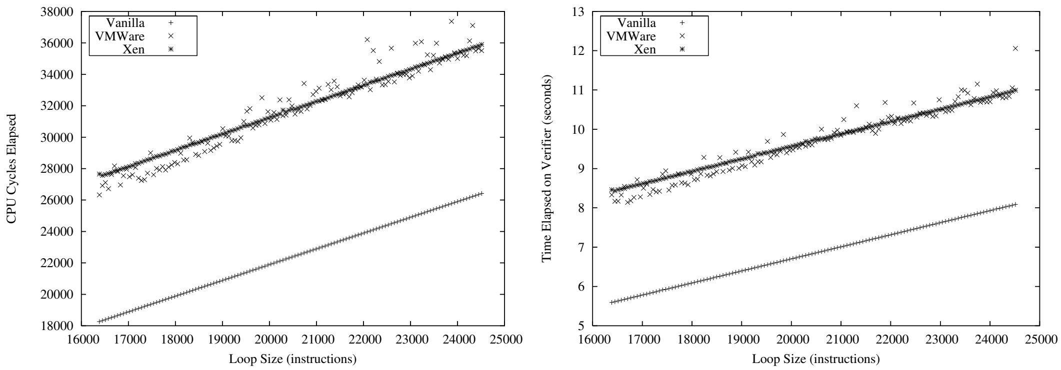



**Figure 9. CR3 read/write overhead timed remotely from another university.**

Figure 10 shows some assembly code involving `mov %cr0, %eax` instructions which are dynamically rewritten each loop iteration. The `sub` instruction immediately following the `nop` instruction gets overwritten with `mov %cr0, %esi` during each loop iteration. Although it is not shown in the figure, additional self-modifying code changes this instruction back to a `sub` again. This oscillation stymies VMware's ability to efficiently perform binary rewriting.

Referring back to Figure 8(b), one would expect a particular program which contains `mov %cr0, %eax` instructions to execute much more rapidly on VMware than on Xen. However, Figure 11 shows the results of a local experiment and one timed by a local verifier where VMware's performance closely tracks that of Xen. Note that the same number of sensitive instructions are executed for all data points; the number of arithmetic instructions included in the loop is what varies on the X-axis. In conclusion, using self-modifying code gives the remote verifier the ability to distinguish between some different VMM implementations.




(a) Timed locally using `rdtsc`. (b) Timed by a local verifier.

**Figure 11. Result of benchmark loop implemented with self-modifying code.**

```
rdtsc                     ;; get start time
mov $131072, %edi         ;; n = 131072
call 1f                   ;; pushes ebp
1: nop
pop %ebp                  ;; read esp
add $5, %ebp              ;; offset to sub
1: sub $1, %ecx           ;; gets overwritten
mov $0x83, 0x0(%ebp)      ;; overwrite
mov $0xe9, 0x0(%ebp)      ;; three
mov $0x01, 0x0(%ebp)      ;; bytes
xorl %eax, %eax           ;; begin arith.
...                       ;; 1K – 16K instr.
mov %cr0, %esi            ;; priv. op.
...                       ;; 1K – 16K instr.
sub $1, %edi              ;; n = n − 1
jnz 1b                    ;; until n = 0
rdtsc                     ;; get end time
```

**Figure 10. Loop including self-modifying code.**

## 5. Related Work

Most related work falls into two categories: techniques which detect VMMs based on implementation details and techniques which make assumptions that limit their applicability.

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) [17] 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 a severe drawback: it requires significant modification to the system (debug registers, debug exception handler) that could be easily detected and consequently forged by the underlying VMM.

Robin and Irvine studied the instruction set of the Intel Pentium processor to determine its suitability for implementing a VMM [14]. They found that the instruction set of the Pentium processor is not fully virtualizable since there are some instructions effecting the correctness of the VM's execution that fail silently without trapping to the VMM. Such instructions have to be simulated by a VMM for correctness of the VM's execution thereby leading to a time overhead when compared to execution on native hardware.

There are a number of previously deployed detection techniques. Redpill[2] 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 [11]. Similar to Redpill, VMware's Back[3] 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 [8]. Dornseif et al. study mechanisms designed specifically to detect the Sebek high-interaction honeypot [6]. Unlike these approaches, the *fuzzy benchmarks* we have constructed are not based upon specific software implementations details.

Vrable et al. touch briefly on non-trivial mechanisms for detecting execution within a VMM [21]. They allude to the fact that although a honeynet maybe be able to perfectly virtualize all hardware, an attacker may be able to infer that it is executing inside a VMM by certain side channels.

Pioneer [18] 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 config-

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

uration 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.

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 [12]. Our approach only requires the existence of a single VM and hence is useful in the case of virtual machine based rootkits [11]. Rutkowska [16] and Zovi [22] proposed the use of hardware virtualization support (AMD Pacifica [5] and Intel VT [9]) to create virtual machine-based rootkits rootkits.

## 6. Conclusions

The main contribution of this paper is the development of a fuzzy benchmarking program whose execution differs from the perspective of an external verifier when a target host is a virtual machine (versus when it is executed directly on the underlying hardware). Our benchmarking program is based on the timing dependency exception property of a VMM. We presented results where a single benchmarking program generates sufficient overhead on three different VMMs 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.

Most related work either detects VMMs based on implementation details, use techniques which make assumptions that limit their applicability, or rely 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 the target host, and incorporates a remote timing and decision maker to eliminate the need to trust the VMM.

## 7. 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 article. Jason Franklin performed this research while on appointment as a U.S. Department of Homeland Security (DHS) Fellow. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of CMU, DHS, DOE, ORISE or the U.S. Government or any of its agencies.

## 8. References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of ASPLOS*, October 2006.

[2] 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.

[3] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*, August 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. Garfinkel, B. Pfaff, J. Chow, and M. Rosenblum. Data lifetime is a systems problem. In *Proceedings of the ACM SIGOPS European Workshop*, September 2004.

[8] T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. In *Proceedings of the IEEE Workshop on Information Assurance and Security*, June 2005.

[9] Intel Corporation. Intel virtualization technology. Available at: `http://www.intel.com/technology/computing/vptech/`, October 2005.

[10] X. Jiang, D. Xu, Helen J. Wang, and E. H. Spafford. Virtual playgrounds for worm behavior investigation. In *Proceedings of RAID*, 2005.

[11] 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.

[12] T. Kohno, A. Broido, and K. Claffy. Remote physical device fingerprinting. In *IEEE Symposium on Security and Privacy*, May 2005.

[13] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17, July 1974.

[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] 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.

[16] J. Rutkowska. Subverting Vista kernel for fun and profit. Presented at Black Hat USA, 2006.

[17] J. Rutkowski. Execution path analysis: finding kernel rootkits. *Phrack*, 11(59), July 2002.

[18] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. Van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of SOSP*, 2005.

[19] 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.

[20] VMWare. Timekeeping in VMWare virtual machines. Technical Report NP-ENG-Q305-127, VMWare, Inc., July 2005.

[21] 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 SOSP*, 2005.

[22] D. D. Zovi. Hardware virtualization-based rootkits. Presented at Black Hat USA, August 2006.