Compatibility is Not Transparency: VMM Detection Myths and Realities

Tal Garfinkel, Keith Adams, Andrew Warfield, Jason Franklin
Stanford University,VMware,UBC/XenSource,Carnegie Mellon University

Abstract

Recent work on applications ranging from realistic honeypots to stealthier rootkits has speculated about building *transparent* VMMs – VMMs that are indistinguishable from native hardware, even to a dedicated adversary. We survey anomalies between real and virtual hardware and consider methods for detecting such anomalies, as well as possible countermeasures. We conclude that building a transparent VMM is fundamentally infeasible, as well as impractical from a performance and engineering standpoint.

1 Introduction

Recently there has been significant interest in providing virtual machine *transparency* – making virtual and native hardware indistinguishable under close scrutiny by a dedicated adversary, thus preventing VMM detection.

Interest in transparency stems from a variety of applications. Those exploring VMM-based worm detectors [9], malware detectors [18, 21] and honeypots [19, 8, 5] wish to disguise their VMs as normal hardware to avoid introducing an easy heuristic for detection and evasion. Anti-virus vendors are eager to cloak their use of VMs in identifying newly released exploits for similar reasons [23, 11]. Others have proposed offensive uses of virtualization in the form of VM-based rootkits (VMBRs) [10, 17, 24], hoping to leverage the transparency of VMMs to cloak their presence and provide an ideal attack platform [1]. We believe the transparency these proposals desire is not achievable today and will remain so.

The belief that VMM transparency is possible is based on a mistaken intuition that *compatibility* and *performance* imply *transparency* i.e. that once VMMs are able to run software for native hardware at native speeds, they can be made to look like native hardware under close inspection. This is simply not the case.

While commodity VMMs conform to the PC architecture, virtual implementations of this architecture differ substantially from physical implementations. These differences are not incidental: performance demands and practical engineering limitations necessitate divergences (sometimes radical ones) from native hardware, both in semantics and performance. Consequently, we believe the potential for preventing VMM detection under close scrutiny is illusory – and fundamentally in conflict with the technical limitations of virtualized platforms.

Previous discussions of VMM transparency have generally failed to acknowledge the breadth and depth of virtualization induced anomalies, instead narrowly focusing on a few simple techniques for detection [11]. The conflict between transparency and practical demands such as performance and ease of implementation have also been largely overlooked. Optimistic hopes that emerging hardware support will mitigate these issues is misplaced.

Our thesis is that preventing VMM detection in the face of a dedicated adversary is generally impractical. In the next section, we survey the many types of disparities between native and virtual hardware, and means of detection. We then discuss limitations associated with trying to ameliorate these differences. We end on a positive note, observing that while infeasible, VMM transparency is likely of marginal benefit to defenders in the foreseeable future as virtualization becomes ubiquitous.

2 Virtualization Anomalies

We begin with a taxonomy of virtualization induced anomalies and detection methods. Our discussion does not aim to be exhaustive, instead we hope to convey the scope of the VMM transparency problem, and the obstacles to its solution.

In our threat model the VMM *passively* avoids detection. To succeed, virtual hardware must be sufficiently similar to physical hardware to be indistinguishable to an adversary in the guest OS. We explicitly ignore *active* attempts at thwarting detection, through modifying guest

[1] Concern about virtualization-based rootkits has led to an advisory from Microsoft suggesting that CPU-based virtualization extensions be disabled in firmware on some systems [12].

code to disable detectors, on the grounds that such active approaches rely on *a priori* knowledge of the detector, making them inapplicable to the detection of novel—a.k.a. "zero-day"—detectors.

Our discussion makes no distinction between anomalies visible at kernel vs. user level, as we assume the reader can infer these by context. Further, this is often a false dichtomoy, as timing and logical semantics of virtual hardware are often as evident at user-level as they are in the kernel.

2.1 Logical Discrepancies

CPU Discrepancies. Logical discrepancies are semantic differences in the interfaces of real and virtual hardware. Most current VMM detection methods exploit differences in the virtual CPU interface of VMMs such as VMware Player or Microsoft VirtualPC that violate *x*86 architecture. Inaccuracies in the execution of some non-virtualizable instructions, such as *SIDT*, *SGDT*, and *SSL*, allow user-level inspection of privileged state [15].

Since these and other discrepancies are unimportant to the vast majority of software, VMMs make no effort to hide them. Intel's VT and AMD's SVM eliminate many obvious discrepancies, leading to speculation that hardware virtualization makes great strides towards providing transparency. This view is mistaken for several reasons.

As with software VMMs, transparency is secondary to providing an efficient and compatible execution environment. Thus, architecturally visible differences exist in current CPU virtualization support, allowing straightforward detection of hardware-assisted hypervisors [2]. Paravirtual VMMs intentionally extend the CPU in non-transparent ways. Xen and Denali provide modified software MMU architectures to ease implementation and improve performance [22, 6] and commercial VMMs have embraced a similar approach [4]. Even with compatible and performant hardware virtualization, this approach offers benefits, and thus will likely remain common.

Off-chip Discrepancies. SVM and VT only address CPU virtualization. Off-CPU differences between physical and virtual hardware abound, both for reasons of engineering ease and I/O performance. For example, modern chipsets are difficult to model [3]. For simplicity, the VMware virtual platform always emulates an *i*440bx chipset, leading to absurd hardware configurations: two AMD Opteron CPUs and 8 GB of RAM in an Intel motherboard from the Clinton administration, for instance. Operating systems run in such bizarre environments, because the firmware layer makes OS scrutiny of the chipset unnecessary.

I/O paravirtualization is another source of strange looking hardware. VMware, for example, provides various network, SCSI, and video cards that look nothing like any physical device, have PCI device and vendor ID's specific to VMware, and require their own device drivers. A detector can easily flag such non-hardware as proof of a VMM.

The VMM could try to emulate a richer set of devices, allowing more plausible hardware configurations. However, writing and maintaining software models of modern devices is difficult and costly. The VMM implementer further risks introducing new opportunities for detection in the form of bugs or omissions in emulated devices. Maintaining a comprehensive library of such emulated devices is a mammoth task that would require continuous engineering effort as new devices become available.

The VMM may also present guest hardware via *device pass through*, allowing guests to program physical devices directly. While this approach appears to improve transparency, it introduces two major problems: First, the VMM must audit device interactions to prevent DMA-based access to arbitrary memory; this effectively requires a complete emulated device model in order to parse device requests. Second, such pass through devices may no longer be shared with other VMs, defeating one of the basic purposes of virtualization.

Hardware support for virtualization is progressing to allow VMM protection from DMA through IOMMUs [3]. Current IOMMU proposals do not provide restartable semantics for DMA faults, however. Thus, even in the presence of an IOMMU, resource-efficient pass through will require virtualization-specific device interfaces [14], defeating any transparency benefits of device pass through.

2.2 Resource Discrepancies

VMMs share physical resources with their guests, including CPU cycles, physical memory, and cache footprint. To survive physical reboots, persistent storage is also required. Irregularities in the availability of these resources can betray the presence of a VMM.

Consider the TLB. VMM and guest virtual address mappings both compete for the same small pool of TLB entries. Software that is sensitive to TLB pressure can detect VMM use of this shared resource.

To demonstrate the feasibility of this approach, we constructed a simple TLB-sizing utility which changes page table entries (PTEs) *without* executing a TLB-

[2] For example, native *x*86 CPUs block non-maskable interrupts (NMIs) after delivery of an NMI until execution of the *IRET* instruction, but VT hardware does not provide a corresponding "block NMIs" bit [7]. Similarly, native *x*86 CPUs hold off debug exceptions for a one-instruction window following *M*OV %SS instructions. AMD's SVM provides no information about pending debug exceptions if an exit occurs in such a window [2]. We constructed a simple SVM detector based on this discrepancy in less than 100 lines of C and assembly.

[3] Modern chipsets are complex and rapidly evolving. Beyond changes in basic bus and memory controllers, richer power managment and I/O, built in sound, video, and security functionality all contribute to this.

synchronizing instruction, and then performs loads through the altered PTEs. A load via the old mapping indicates a hit, while a load from the new mapping is the result of a miss. Our utility runs this TLB sizing algorithm twice; during the second run, we interleave the memory accesses with exiting operations that invoke the VMM. In the presence of a VMM, the second run computes a smaller TLB size, due to the entries consumed by the VMM.

We can imagine VMM counter-measures for this detection method. The VMM might model a hardware TLB of native size, using a partial shadow page table as the current "virtual TLB" contents. However, similar methods can detect VMM pressure on data and instruction caches, and could include replacement policy along with size as a detection criterion. In the limit, the VMM must run every guest memory access through a software cache simulator to avoid detection, incurring absurd performance overheads, and leaving the VMM even more vulnerable to detection by our next set of techniques.

2.3 Timing Discrepancies

Virtual and physical environments differ in their timing characteristics. These differences are not simply "virtualization overhead" that can be addressed by making hardware or software faster – they can manifest as variance in latency, relative differences in the latencies of *any* two operations, and the behavior of these latencies over time. Consider some examples.

Device virtualization is a rich source of timing anomalies. For example, a PCI device register that takes a hundred cycles to read on physical hardware might require only a single cycle when the virtual hardware register is in the processor cache – cache behavior may in turn cause the virtual access time to exhibit higher variance than a hardware access. In this case, it is not the operation's latency *per se* that betrays the presence of a VMM, but the run-to-run variance in latency.

Memory virtualization also has guest-visible performance consequences [20]. VMMs use page protection for local and global resource management, memory-mapped I/O (MMIO) emulation, and protection of the VMM itself. Guest accesses to VMM-protected pages often induce *hidden page faults*, eliciting a performance discrepancy that can span three decimal orders of magnitude. While some of these memory overheads will be eliminated by future hardware developments such as AMD's NPT and Intel's EPT technologies [13], resource management and MMIO overheads will remain regardless of the hardware mechanism used to effect them.

Virtualization of privileged instructions is a well-understood source of timing discrepancies, due to the hardware and software overheads of a round-trip to the VMM. While hardware virtualization support shrinks some of these overheads, it bloats others [1]. On the balance, hardware virtualization makes timing-based detection no less feasible; while it changes the timing fingerprint of the virtual environment relative to a software-only approach, the fingerprint still differs from that of native execution.

Detectors can use a variety of time sources:

Local Time Sources. Timers and periodic interrupt sources provide the simplest mechanisms for measuring latencies. Examples include the hardware timestamp counter (*rdtsc*), PIT, ACPI timer and local APIC timer.

Guests can also use a host of relative time sources. Most hardware betrays timing information implicitly, in that hardware operations have a predictable average latency. Guest software can use these latencies as a time source by racing events against one another.

For example, we might construct a race between the innocuous *NOP* instruction and the virtualization-sensitive *CPUID* instruction. Thread A repeatedly spends its entire quantum executing *NOP*s and incrementing a count of completed operations, while thread B, bound to a separate logical processor, does the same with *CPUID*s. Since thread A takes no exits, the VMM has no opportunity to throttle the rate of *NOP*s. Over time, an observing thread will see the ratio of *NOP*s to *CPUID*s converging on a higher value in a virtual environment than in a physical one.

In this example we have assumed the presence of multiple CPUs, but this race construction technique can use any source of concurrency. DMA transfers, interrupt latencies, the memory hierarchy, and OS scheduler activity, for example, all provide possible sources of timing data for detection.

Remote Time Sources. A VM can use nearly any communication from the outside world as a clock. Possible remote time sources range from overt use of the NTP protocol to covert timing channels, such as subtle variations in inter-packet arrival times. Any communication with an outside entity will provide access to a covert channel, guaranteeing access to covert clocks – a VM can then measure performance discrepancies with a high integrity remote time source that is difficult to detect or modify even in the presence of an active VMM warden.

A VMM can interpose on and modify local or remote time sources, in an attempt to undermine timing based detection. This alone does nothing to prevent timing attacks. Any alterations a VMM makes must still "look real" relative to other time sources, thus necessitating the use of extreme measures like simulation. Approaches like adding randomness to either local or remote timing channels [5] merely introduces another source of anomalies for measurement.

3 Trade-offs and Implications

We've considered a number of VMM detection strategies, and have found the options for foiling these strategies limited. While our list is incomplete, the ease of imagining new detection methods suggests that VMM transparency is difficult to the point of impracticality. We now take a step back to look at transparency from the VMM implementor's point of view, and find an inherent tension between VMM performance and transparency. We then consider the ramifications of a world without transparent VMs.

Practical Emulation Overhead. VMMs may attempt to hide timing discrepancies by manipulating local time sources, a technique known as *time dilation*. Time dilation can allow virtual time to progress more slowly than real time, buying the VMM time to conceal its overheads.

To achieve transparency, time dilation must account for all latencies appropriately. In the limit, modeling all absolute and relative time sources reduces to cycle-accurate simulation of the entire system, entailing extreme engineering hardship and huge runtime overhead. Commercial emulators that provide a logical view of hardware without cycle-accuracy, such as AMD's SimNow or VirtuTech's Simics, incur a performance penalty on the order of 10x. Cycle accurate simulators such as SimOS incur overheads on the order of 25,000x [16] to provide full accuracy.

Perhaps some VMMs would accept such performance in the name of stealth, but this tactic would be self-limiting. If the practice became common, attackers could simply rule out attacking machines whose performance is below that of desktop machines of 10 to 20 years ago; we propose that the number of valuable targets running on such meager hardware is vanishingly small.

Even if dilation of local time sources were practical, the entire effort would be undermined as soon as a network is attached due to the intractable problem of remote timing channels.

Relative Accuracy vs. Absolute Cost. Time dilation, and other heroic efforts at preventing local timing detection, ironically make the hypervisor *more* vulnerable to remote detection; as the hypervisor spends more absolute time emulating local hardware, the gap between local and remote time sources widens.

In general, accurate simulation of any one component (e.g. memory, peripherals, processor attributes) provides greater leverage for detection attempts based on relative time differences. Carried to its logical conclusion, the cycle-accurate simulation approach is all-or-nothing: if any guest operation escapes from the cycle-accurate prison, all of the VMM's efforts at cycle-accuracy only make the VMM's presence more obvious. Since we've argued that trapping all remote time sources is intractable, the cycle-accurate simulation approach is a dead end for systems with network connectivity.

Real System Performance. Users are primarily concerned with the *utility* of their VMs: that VMs run correctly, are performant, and use minimal system resources. VMM implementors sweat blood over speed. To achieve high performance, commercial VMMs make heavy use of the systems design principle of optimizing for the common case.

VMs diverging from the common case may encounter performance cliffs. For detection software, such cliffs are dead giveaways that a VMM is present, but for VMM implementors, they are simply the consequence of a trade-off that allows common cases to run fast. Stealth and performance are thus, to some extent, in inevitable conflict.

For example, VMMs that use shadow paging manage a cache of *shadow page tables*. The VMM derives a shadow page table for each guest page table, allowing efficient virtual address mappings for the corresponding guest virtual address space. This cache has a finite capacity, however. By using an enormous number of page tables, the guest can conduct an exhaustion attack on the shadow page cache, leading to high rates of hidden page faults and an easily detected performance cliff.

Other cliffs result from the VMM's dynamic adaptations to guest behavior. VMware's VMM, for example, uses a cache of binary translations of guest kernel code, enforcing coherency via page protection of the source binary [1]. These translations evolve over time, e.g. by producing special translations for accesses to memory-mapped I/O devices.

In typical guests, this technique performs well. However, its performance rests on many assumptions, e.g., that self-modifying code is rare, and that the past behavior of an instruction predicts its future behavior. A guest can violate these assumptions to cause an easily observed performance degradation. Even with hardware virtualization extensions, this problem remains, as binary translation is a useful technique for avoiding hot spots causing frequent (expensive) `VMEXITs` [1].

These cliffs do not reflect a weakness in the construction of VMMs, but rather one in the assumption of transparency. By attacking the mechanisms that a VMM uses to provide high performance for normal guests, a diabolical guest will always be able to elicit worse performance from the system.

Don't worry – Be Happy. Those relying on VMM-based systems for monitoring have expressed concern over VMM detection while posing a variety of incomplete solutions. We believe these concerns are largely unwarranted. Virtualization has made massive inroads into enterprise data centers, a trend that is expected to continue. Soon, malware that limits itself to non-virtualized platforms will be passing up a large percentage of commercial, military and institutional targets. To the degree that malware disables itself in the presence of VMs, VMs become even more attractive for production systems. In the long run, malware authors are motivated to operate regardless of the presence of a VMM.

Whither VMBRs? The question, "Might VM-based rootkits (VMBRs) be useful if they were small, clever, hardware-accelerated, etc., enough?" no doubt remains in some readers' minds. We think not. No matter how minimal the hostile VMM is, it must consume physical resources, perturb timings, and take measures to protect itself from the guest, leaving it no less susceptible to detection than other VMMs. Further, one of King et al.'s primary motivations for introducing VMBRs was to provide a simpler environment to build malware than found in current kernel based rootkits [10]. Highly resource-constrained VMBRs would defeat this purpose.

Even if VMBR detection were difficult, VMBR prevention is trivial and highly effective. As King et al. note, a stub VMM that refuses to load unsigned VMMs provides complete protection from VMBRs.

Perhaps the most concise argument against the utility of VMBRs is: "Why bother?" VMBRs change the malware defender's problem from a very difficult one (discovering whether the trusted computing base of a system has been compromised), to the much easier problem of detecting a VMM.

4 Conclusion

The compatibility VMMs provide *seems* just a small step away from transparency; intuition suggests that the tiny gap between native and virtual platforms must only be a small matter of programming, a dash of additional hardware support, etc., away from vanishing. We have challenged this view by surveying the wide range of dissimilarities between real and virtualized platforms, both on principle and using examples from today's VMMs. While tomorrow's VMMs will change, performance will remain paramount. Consequently, virtual and native hardware are likely to remain highly dissimilar, and thus amenable to discrimination.

5 Acknowledgements

This work was supported in part by TRUST (The Team for Research in Ubiquitous Secure Technology), which is supported by National Science Foundation (award CCF-0424422). Jason Franklin performed this research while on appointment as a U.S. Department of Homeland Security (DHS) Fellow under the DHS Scholarship and Fellowship Program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Homeland Security. We are also very greatful for feedback and discussions with Ole Agesen, Mendel Rosenblum, Jim Chow, and Jim Mattson.

References

[1] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[2] AMD. *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual*, May 2005.

[3] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification*, Feb. 2006.

[4] Z. Amsden, D. Arai, D. Hecht, and P. Subrahmanyan. Paravirtualization API Version 2.5. `www.vmware.com/pdf/vmi_specs.pdf`.

[5] K. Asrigo, L. Litty, and D. Lie. Using VMM-based Sensors to Monitor Honeypots. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, June 2006.

[6] 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 19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

[7] Intel Corporation. *Intel® Virtualization Technology Specification for the IA-32 Intel® Architecture*, April 2005.

[8] X. Jiang and D. Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. In *Proceedings of 13th USENIX Security Symposium*, Aug. 2004.

[9] E. Jonsson, A. Valdes, and M. Almgren. HoneyStat: Local Worm Detection Using Honeypots. In *Proceedings of Seventh International Symposium on Recent Advances in Intrusion Detection*, Sept. 2004.

[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. Liston and E. Skoudis. On the Cutting Edge: Thwarting Virtual Machine Detection . `http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf`, July 2006.

[12] Microsoft. CPU Virtualization Extensions: Analysis of Rootkit Issues. `http://www.microsoft.com/whdc/system/platform/virtual/CPUVirtExt.mspx`. Windows Hardware Developer Central, October 20, 2006.

[13] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3), Aug. 2006.

[14] PCI SIG. *PCI I/O Virtualization Specifications*.

[15] J. Robin and C. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*, Aug. 2000.

[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] J. Rutkowska. Subverting Vista Kernel for Fun and Profit. Presented at Black Hat USA, Aug. 2006.

[18] S. Sidiroglou, J. Ioannidis, A. D. Keromytis, and S. J. Stolfo. An Email Worm Vaccine Architecture. In *Proceedings of the First Information Security Practice and Experience Conference*, 2005.

[19] 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 20th ACM Symposium on Operating Systems Principles*, Oct. 2005.

[20] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[21] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proceedings of Network and Distributed Systems Security Symposium*, Feb. 2006.

[22] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[23] L. Zeltser. Virtual Machine Detection in Malware via Commercial Tools. `http://isc.sans.org/diary.php?storyid=1871`. Handlers Diary, November 19, 2006.

[24] D. D. Zovi. Hardware Virtualization-Based Rootkits. Presented at Black Hat USA, Aug. 2006.