Detecting System Emulators

Thomas Raffetseder, Christopher Kruegel, and Engin Kirda[⋆]

Secure Systems Lab, Technical University of Vienna, Austria
{tr,chris,ek}@seclab.tuwien.ac.at

Abstract. Malware analysis is the process of determining the behavior and purpose of a given malware sample (such as a virus, worm, or Trojan horse). This process is a necessary step to be able to develop effective detection techniques and removal tools. Security companies typically analyze unknown malware samples using simulated system environments (such as virtual machines or emulators). The reason is that these environments ease the analysis process and provide more control over executing processes. Of course, the goal of malware authors is to make the analysis process as difficult as possible. To this end, they can equip their malware programs with checks that detect whether their code is executing in a virtual environment, and if so, adjust the program's behavior accordingly. In fact, many current malware programs already use routines to determine whether they are running in a virtualizer such as VMware.
The general belief is that system emulators (such as Qemu) are more difficult to detect than traditional virtual machines (such as VMware) because they handle all instructions in software. In this paper, we seek to answer the question whether this belief is justified. In particular, we analyze a number of possibilities to detect system emulators. Our results shows that emulation can be successfully detected, mainly because the task of perfectly emulating real hardware is complex. Furthermore, some of our tests also indicate that novel technologies that provide hardware support for virtualization (such as Intel Virtualization Technology) may not be as undetectable as previously thought.

1 Introduction

The Internet has become an integral part of our lives. Today, we interact with hundreds of services, do business online, and share information without leaving the comfort of our offices or homes. Unfortunately, the Internet has turned into a hostile environment. As the importance of online commerce and business has increased, miscreants have started shifting their focus to Internet-based scams and attacks. Such attacks are easy to perform and highly profitable. A popular technique is to develop malware (such as a Trojan horse or spyware) that is installed on victims' machines. Once deployed, the malicious software can then

[⋆] This project was supported by the Austrian Science Foundation (FWF) under grants P-18157 and P-18764, the FIT-IT project Pathfinder, and the Secure Business Austria competence center.

be used to capture the victims' sensitive information (such as passwords or credit card numbers) and perform illegal online financial transactions.

When an unknown malware sample is obtained by a security organization such as an anti-virus company, it has to be analyzed in depth. The goal is understand the actions the malware performs, both to devise defense and detection mechanisms as well as to estimate the damage it can inflict. To perform the analysis, running the executable in a virtual machine such as the one provided by VMware [1] is a popular choice. In this case, the malware can only affect the virtual PC and not the real one[1]. A virtual environment also has the benefit that it offers tight control over program execution, allowing the analyst to pause the system at any time and inspect the contents of the memory. In addition, the analyst can make use of snapshots that capture the state of the system at a certain point in time. This allows us to observe the effects of different actions (e.g., what happens if the malware process is killed?; what happens if a certain registry key does not exist?) without having to reinstall the system after each experiment. Instead, one can just revert back to a previously stored snapshot.

Obviously, an important question is whether a malware program can detect if it is executed in a virtual environment. If malicious code can easily detect that it is running in a simulator, it could try to thwart analysis by simply changing the way it behaves. Unfortunately, it is possible to detect the presence of virtual machines (VMs) such as VMware. In fact, a number of different mechanisms have been published [2,3] that explain how a program can detect if it is run inside a VM. These checks and similar techniques are already used by malware (e.g., [4] is using a simple detection technique). Thus, the analysis results obtained by executing malicious code inside a VM become questionable. Because of the availability of checks that can identify virtual machines, there is a general belief among security professionals that software emulation is better suited for analysis than virtualization. The reason is that an emulator does not execute machine instructions directly on the hardware, but handles them in software. Also, a number of malware analysis tools (e.g., Cobra [5] or TTAnalyze [6]) have been presented recently that claim to be stealthy (that is, undetectable by malicious code) because they are based on software emulation.

In this paper, we aim to answer the question whether software emulation is as stealthy as hoped for. Unfortunately, our results show that there are several possible methods that can be used to distinguish emulated environments from a real computer. Most of these techniques aim at identifying elements of the computer hardware that are difficult to faithfully emulate in software. In addition, we developed a number of specific checks to detect Qemu [7], a popular system emulator that forms the basis for malware analysis tool such as TTAnalyze [6]. These checks allow a program to identify observable differences in the behavior of the CPU cache, the implementation of the instruction set (such as bugs present on a particular CPU), MSRs (model-specific processor registers),

[1] Note that the software emulating the PC itself may have implementation flaws that could allow malicious code to break out of the virtual PC. However, such errors are not common.

and the I/O system. Furthermore, some of our experiments also indicate that novel technologies such as the Intel Virtualization Technology may not be as undetectable as previously thought. Because of the complexity of providing a virtualized environment that precisely mimics the real system, including its timing properties, we believe that miscreants have opportunities to identify small deviations that can be used for detection.
The contributions of this paper are as follows:

- We discuss a number of possible approaches that can be used to distinguish between an emulated environment and a real computer.
- We describe in detail specific checks that allow a program to detect the presence of Qemu, a popular system emulator that is used as the basis of malware analysis tools such as TTAnalyze.
- We examine the extent to which our emulator detection techniques apply to the novel virtualization technologies recently introduced by processor manufacturers (such as Intel VT).

2 Virtual Machine Monitors (VMMs) and Emulators

Virtual machine monitors (VMMs) and emulators are computer programs that provide mechanisms to simulate the hardware. Guest software (typically operating systems) can run within this simulated environment as if they are executed on real hardware.

2.1 Virtual Machine Monitors (VMMs)

Popek and Goldberg [8] describe a virtual machine (VM) as “an efficient, isolated duplicate of the real machine.” Furthermore, they define the key characteristics of a VMM as follows:

1. The VMM provides an environment for programs, which is essentially identical to the original machine. Any program run under the VMM should exhibit an effect identical to that demonstrated if the program had been run on the original machine directly, with the possible exception of differences caused by the availability of system resources and differences caused by timing dependencies.
2. Programs that run in this environment show at worst only minor decreases in speed. A statistically significant amount of instructions need to be executed on the real hardware without interception of the VMM.
3. The VMM is in complete control of system resources. That is, it is not possible for a program running under the VMM to access resources not explicitly allocated to it. The VMM is able to regain complete control of (possibly already allocated) resources at any time.

The second point is important to be able to distinguish between VMMs and emulators. Emulators do not execute code directly on hardware without intercepting the execution (see Section 2.2). Thus, emulators typically cause a decrease in speed. VMMs on the other hand, allow execution times close to native speed. To support virtualization, a processor must meet three requirements. These are defined by Robin and Irvine [2] as follows:

> **Requirement 1** The method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode. [. . .]
> **Requirement 2** There must be a method such as a protection system or an address translation system to protect the real system and any other VMs from the active VM.
> **Requirement 3** There must be a way to automatically signal the VMM when a VM attempts to execute a sensitive instruction. It must also be possible for the VMM to simulate the effect of the instruction.

A sensitive instruction is an instruction that, when executed by a virtual machine, would interfere with the state of the underlying virtual machine monitor. Thus, a VMM cannot allow a VM to execute any of these instructions directly on the hardware [2]. Unfortunately, it is not always possible for a VMM to recognize that a virtual machine attempts to execute a sensitive operation, especially on the Intel IA-32 architecture.

The IA-32 architecture's protection mechanism recognizes four privilege levels, numbered from 0 to 3. A higher number means lesser privileges. Usually, critical software such as the kernel of an operating system is executed with privilege level 0. Other, less critical software is executed with less privileges. Instructions that can only be executed in the most privileged level 0 are called privileged instructions. If a privileged instruction is executed with a privilege level that is not 0, a general-protection exception (#GP) is thrown [9].

Typically, VMMs run in privileged mode (privilege level 0) and a VM runs in user mode (privilege level > 0). If a VM calls a privileged instruction, it causes a general-protection exception (because the privileged instruction is executed with a privilege level greater than 0). The VMM can then catch the general-protection exception and respond to it. If all sensitive instructions have to be executed in privileged mode, a processor is considered to be *virtualizable*. That is, the VMM can catch every exception generated by the execution of sensitive instructions and emulate their proper behavior. Problems occur if there are sensitive, unprivileged instructions, as these do not cause exceptions. Unfortunately, this is the case for the IA-32 architecture, whose instruction set includes seventeen sensitive, unprivileged instructions, making the IA-32 architecture unvirtualizable [2]. This property can be exploited to detect that code is running in a virtual machine. A well-known example is the RedPill code developed by Rutkowska [3], which makes use of the SIDT Store Interrupt Descriptor Table Register.

2.2 Emulators

An emulator is a piece of software that simulates the hardware, and a CPU emulator simulates the behavior of a CPU in software. An emulator does not execute code directly on the underlying hardware. Instead, instructions are intercepted by the emulator, translated to a corresponding set of instructions for the target platform, and finally executed on the hardware. Thus, whenever a sensitive instruction is executed (even when it is unprivileged), the system emulator is invoked and can take an appropriate action. This property makes system emulators invisible to detection routines such as RedPill, and as a result, an appealing environment for malware analysis. Unlike with virtualization, the simulated guest environment does not even have to have the same architecture as the host processor (e.g., it is possible to execute code compiled for the IA32 instruction set as a guest on a Power PC hardware or vice versa).

There are two popular techniques to implement a CPU emulator; interpretation and (variations of) dynamic translation. An interpreter reads every instruction, translates it, and finally executes it on the underlying hardware. The simplest form of a dynamic translators take one source instruction, translate it and cache it. If a instruction is cached and needed again, the translator can use the cached code. The dynamic translation technique can be extended by translating blocks of instructions instead of single instructions [10].

The big disadvantage of emulators is the incurred performance penalty. In general, every instruction (or sequence of instructions) has to be analyzed and finally executed on the underlying hardware. Even though techniques such as dynamic translation and caching can be used to speed up the execution, emulators do not reach the speed of real hardware or VMMs. However, the fact that instructions are read and interpreted makes emulators powerful tools. In particular, it enables emulators to fake instructions.

For our experiments, we used Qemu [7], an open source system emulator. The software offers a "full system emulation mode", where the processor and periphery is emulated, and a "user mode emulation", where Qemu can launch executables compiled for one CPU on another CPU [7]. In our work, we focus on the full system emulation mode. For more information about Qemu see [7] and [11]. Note that we do not consider the Qemu acceleration mode [12], which executes (most) code directly on the hardware. The reason is that in this mode, Qemu behaves similar to a classic virtual machine and no longer as a system emulator (and thus, can be detected similar to VMware).

3 General Detection Vectors

In this section, we provide an overview of possible approaches to detect emulated environments. To this end, we have to identify software whose execution differs in some observable fashion from the direct execution on hardware. In the following Section 4, we discuss concrete techniques that allow a program to determine that it is running in an emulated environment on top of Qemu.

3.1 Differences in Behavior

The first property of VMMs [8] states that a virtualized environment should behave "essentially identical to the original machine" and that "any program run under the VMM should exhibit an effect identical with that demonstrated if the program had been run on the original machine directly" (see Section 2.1). Hence, this property should also hold for emulators. Therefore, executing the same piece of code in emulated and real environments should lead to the same results in order to have a complete and perfect emulation. If any difference can be found, one can detect emulated environments. Of particular interest are differences between the ways a real CPU and an emulated CPU behave.

CPU Bugs Modern CPUs are complex devices. They implement the specification of a given instruction set. However, due to failures in the design and implementation, some instructions do not behave as specified. Every processor family has its typical bugs, and the presence or absence of specific bugs might allow us to distinguish between a real and an emulated CPU (assuming that the emulated CPU does not reproduce specific bugs per default). Moreover, based on the information about the processor-model-specific bugs, it might even be possible to precisely determine a specific processor model. That is, we envision the possibility to leverage CPU bugs to create processor fingerprints. To obtain information about CPU bugs, we can draw from public documentation provided by the chip manufacturers.

Model-Specific Registers Another possible way to distinguish between the behavior of emulated and real CPUs are model-specific registers (MSRs). Model-specific registers typically handle system-related tasks and control items such as the debug extensions, the performance-monitoring counters, the machine-check architecture, and the memory type ranges (MTRRs) [9]. Note that these registers are machine-dependent. That is, certain registers are only available on specific processor models. Furthermore, the attempt to access a reserved or unimplemented MSR is specified to cause a general protection exception [13,14,15]. Thus, one can imagine a technique in which the model-specific registers of processors are enumerated to generate a specific fingerprint for different CPU models. The idea is that because of the highly system-dependent character of these registers, it might be possible that emulators do not support MSRs and, therefore, could be successfully detected. This analysis includes checking the availability of MSRs as well as interpreting their contents. A list of the MSRs of the Intel processors can be found in [15].

3.2 Differences in Timing

The performance of an application running in a virtual environment cannot be as high as on real hardware (see Section 2.1). This is because additional work such as translation or interception has to be done. Clearly, the *absolute performance* in an emulated environment is lower than that on the real hardware. However, using absolute performance values to detect virtual environments is difficult, as most computers differ with regards to hardware configurations. Performance results may even vary on the same hardware when using different operating systems and/or working loads.

Relative Performance To address the problem that absolute performance values are unreliable when used to detect emulated environments, another approach is to use what we call *relative performance*. With relative performance, a system is characterized by the performance ratio of two (or more) operations executed on a system. For example, one can compare the time that it takes to execute two different instructions to obtain a relative performance measure for a system. Of course, the choice of these instructions is important. Lang [16] suggested to compare the performance of accessing the control registers CR0 and CR3 with the performance of a simple NOP (no operation) instruction. Relative performance measurements can be created for emulated systems as well as for real environments. If the indicators between execution in real environments and virtual environments differ significantly, it shows that the execution times of the instructions did not change homogeneously. It is also possible to use the indicators generated on real environments as benchmarks that can later be used for comparison with indicators generated in virtual environments.

Another interesting aspect of relative performance is caching. The basic idea is to observe the effects caching has on real and emulated environments. To this end, a function is executed a number of times, and its execution time is measured. We expect that the first run will be much slower than all subsequent ones, because data and instructions will be cached after the first iteration. Then, we turn off caching and repeat the same test. This timing analysis can be used to examine the presence and effectiveness of caches. Because a processor cache is difficult to simulate, emulators may not support the CPU cache or may not support CPU cache control.

3.3 Hardware Specific Values

In simulated environments, all peripheral devices have to be emulated. These virtual devices include controllers, IDE/SCSI devices, and graphic cards. Certain emulators implement the virtual devices in a characteristic manner. Thus, various pieces of the emulated hardware are characteristic for certain emulators. Also, it might be possible to extract characteristic strings and hardware specific properties from devices. These strings include default vendor names, product information, and MAC addresses [16]. Most of today's hardware also includes monitoring software. By accessing these values in emulated environments, it is possible to determine differences to real hardware.

4 Results and Implementation Details

Based on the general detection vectors outlined above, this section provides a detailed discussion of our techniques to identify the system emulator Qemu [11]. While the results in this section are specific to Qemu, many ideas can be trivially applied to other emulators as well. Also, in some cases, we repeated tests under VMware. The goal was to determine whether a specific test to identify emulators would also apply to virtualizers.

We performed our experiments on an Intel Pentium 4 processor as well as on an Intel Core 2 Duo processor. The guest and host operating system used was

Linux Debian Testing with kernel version 2.6.16. Qemu was used with version 0.8.2, and VMware was used with version 5.5.2. We did not use the Qemu acceleration module [12] so that we could test the feature of the system emulator rather than that of the VMM.

4.1 Using CPU Bugs to Identify a Processor

This section describes our approach to fingerprint the CPU using CPU bugs. Here, we focus only on the Intel Pentium 4 processor. The document "Intel Pentium 4 Processor - Specification Update" [17] contains 104 errata, 45 of them are without a plan to fix (as of December 2006). We searched the document for errata that were not scheduled for fixing and seemed to be reproducible. Out of the 45 without a plan to fix, we selected the following two to implement:

Errata N 5: Invalid opcode 0FFFh requires a ModRM byte: 0FFFh is an invalid opcode that causes the processor to raise an invalid opcode exception (undefined opcode UD). This opcode does not require a ModR/M byte (the ModR/M byte is part of an instruction that immediately follows the opcode, typically specifying the instruction's addressing format). On the Pentium 4 processor, however, this opcode raises a page or limit fault (instead of the expected invalid opcode exception) when the "corresponding" ModR/M byte cannot be accessed [17].

To test for this bug, we first allocate two pages in memory. The instruction with opcode 0FFFh is put at the very end of the first page. In the next step, we use the Linux `mprotect` system call to remove all access permissions from the second page. Finally, the code at the end of the first page is invoked. On Pentium 4 processors, we receive a segmentation fault (the page fault is processed by the kernel and a segmentation fault is sent to the user mode program), because the processor attempts to load the byte following the opcode as the ModR/M byte. This fails, because the page is marked as not accessible. With Qemu and non-Intel Pentium 4 processors, the invalid opcode on the first page is sufficient to raise an exception.

Errata N 86: Incorrect Debug Exception (#DB) May Occur When a Data Breakpoint is set on an FP Instruction: The IA-32 architecture contains extensions for debugging. These include various debug registers and debug exceptions. A complete description of the debug facilities can be found in [15]. An incorrect debug exception (#DB) may occur if a data breakpoint is placed on a floating point instruction. More precisely, the exception occurs if the floating point instruction also causes a floating point event [17].

To test for this bug, we set a data hardware breakpoint on a floating point instruction. In the next step, we have to make this instruction raise an exception. To this end, we used the FDIV (floating point division) inststruction to cause a division by zero exception. Our tests show that on Pentium 4 machines, the machine halts because the processor raises an incorrect debug exception (i.e., a data breakpoint set on an instruction should not raise a debug exception). This did not occur with Qemu or non-Pentium 4 processors.

4.2 Using Model-Specific Registers (MSRs) to Identify a Processor

According to Intel's software developer's manual [13,14,15], trying to access a reserved or unimplemented model-specific register (MSR) causes a general protection (#GP) exception. However, Qemu did not raise any exceptions when trying to access a reserved or unimplemented MSR. Hence, Qemu can be easily detected using this approach.

In our particular test, we attempt to write to the reserved MSR `IA32_MCG_RESERVED1` (number 18BH) (for list of the MSRs of the Intel processors see [15]) with the appropriate WRMSR instruction, using the following GNU C code:

```
asm volatile("WRMSR;" : : "c" (0x18b));//input: IA32_MCG_RESERVED1
```

This code, run in a kernel module (MSRs can only be accessed in privileged mode), should result in a general protection fault. Executed on real hardware (Intel Pentium 4 and Intel Core 2 Duo), it behaves as expected. The same code executed in a Qemu environment does not raise any exception.

Extending the idea above, we can generate a fingerprint for a specific processor by checking for the existence of certain MSRs. To this end, if a given MSR exists on the CPU. To test the existence of a register, we use the `rdmsr_safe` macro defined in the Linux kernel. This macro defines the assembler code for accessing a specified MSR and includes exception handling code. A user mode program accesses this kernel module using the character device. This user mode program can now extract a list of all existing MSRs. Then, it can compare this list with a database that stores the available MSRs for known processors to determine the exact model.

4.3 Instruction Length

In [9], it is stated that "the Intel processor sets a limit of 15 bytes on instruction length. The only way to violate this limit is by putting redundant prefixes before an instruction. A general-protection exception is generated if the limit on instruction length is violated." We discovered that the CPU emulated by Qemu does not set this limit.

One of the available prefixes is the Repeat String (REP) operation prefix (for a comprehensive list of prefixes see [18]). This prefix can be added to a string instruction and repeats the string instruction a number of times. If this prefix is repeated more than 14 times consecutively before a string move operation, the CPU raises an illegal instruction exception. The reason is that the instruction length limit is violated (e.g., $15 * REP(1byte) + 1 * MOVSB(1byte) > 15bytes$). However, the same instruction executed in a Qemu environment does not raise this exception. Note that the idea about the REP prefix was first posted in [19].

4.4 Alignment Checking

The IA32 architecture supports alignment checking. When code is run with the least privileges, alignment of memory addresses can be checked by the CPU.

To this end, the AM (alignment mask) flag in the CR0 (control register 0) and the AC (alignment check) flag in the EFLAGS register must be set. When alignment checking is activated, unaligned memory references generate alignment exceptions (#AC) [9]. Qemu does not support this processor feature, and thus, can be distinguished from a real processor.

To test for the existence of the alignment checking feature, we developed a kernel module that sets the alignment flag in the control register 0 (usually, this flag is set already by the operating system). Then, a user mode program sets the alignment check flag in the EFLAGS register and attempts to access an address that is both unaligned and invalid. If alignment checking is supported, an alignment exceptions (#AC) is thrown. If alignment checking is not supported, a segmentation fault occurs, because the CPU tries to access an invalid address. As expected, this code executed under Qemu results in a segmentation fault, indicating that alignment checking is not supported by Qemu.

4.5 Relative Performance - Comparison of Instructions

The goal of relative performance tests is to distinguish between real and emulated environments using timing analysis. The idea is to measure the absolute time that it takes to execute a privileged instruction and compare it to the time is takes to execute an unprivileged instruction. More precisely, the execution of an unprivileged instruction serves as the baseline. The execution time of the privileged instruction is then divided by this baseline time, yielding a relative performance number. This relative performance number is used to identify a simulated environment. The rationale is that a real processor has to perform different tasks than a simulated CPU when handling privileged instructions, and these different tasks manifest themselves by yielding different relative performance results.

For our experiment, we used a kernel module developed by Lang [16]. This kernel module reads either from the control register CR0 or from CR3, and then writes back the value previously read. To measure the time that it takes to execute instructions, the processor's time stamp counter (TSC) is used. Reading and writing to the control register CR0 (see Figure 1) is relatively faster on both VMware and Qemu than on real hardware. Interestingly, the relative timings for VMware and Qemu are very similar. Accessing the control register CR3 (see Figure 2) shows a significant timing difference between VMware and the real hardware; the real hardware is up to 100 times faster than VMware. Qemu, while significantly faster than VMware, is still slower than the real hardware.

4.6 Relative Performance - Cached Instructions

Instruction and data caches have a major influence on the performance of program execution. Thus, we were asking the question whether different timing behavior can be observed between a simulated and a real processor when executing the same piece of code for a large number of times. The idea is that caching will speed up the execution of all iterations after the first one. However,

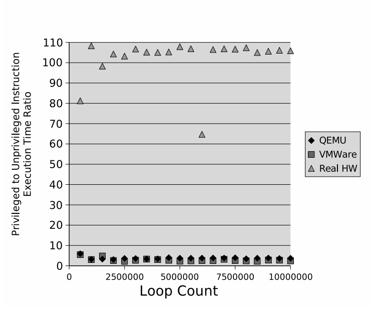

Fig. 1. Reading/Writing CR0 (TSC) on the Pentium 4 1.8 GHz

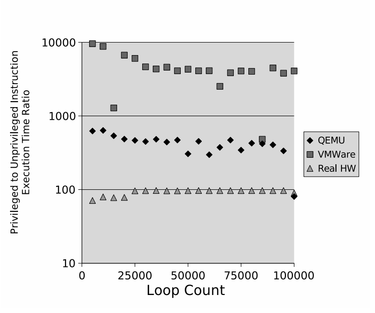

Fig. 2. Reading/Writing CR3 (TSC) on the Pentium 4 1.8 GHz

the relative speed-up could be different between depending on the environment. In addition, the timing tests were repeated when caching was disabled. Again, we wished to determine whether a different timing behavior could be seen.

For the cache timing tests, we wrote a function that contained a loop. This loop executed a sequence of simple, arithmetic instructions 100 times. For each test, this function was invoked six times (each invocation was called a run). When caching is enabled, we expect the first invocation (the first run) to last slightly longer. The reason is that the caches have to be filled. Starting from the second run, all others last shorter. To measure the execution time, the RDTSC instruction (Read Time-Stamp Counter) is used. Because the CPU supports out-of-order execution (that is, instructions are not necessarily executed in the order they appear in the assembler code), we need an instruction that forces all previous instructions to complete before we read the TSC. The CPUID instruction is one of these serializing instructions [20]. The timing code was executed in a kernel module. This allowed us to disable interrupts for each test run and therefore, to guarantee that only the timing of our test code rather than that of the interrupts handlers was measured and led to repeatable results.

After the first set of experiments with caching, we disabled caching and repeated the experiments. To disable caching, the processor has to enter the no-fill cache mode (i.e., set the CD flag in control register CR0 to 1 and the NW flag to 0) and all caches need to be flushed using the WBINVD instruction. Finally, the MTRRs need to be disabled and the default memory type has to be set to uncached or all MTRRs have to be set for the uncached memory type. [9] Turning on the cache again works the opposite way.

The charts showing the results of our experiments in this section are structured as follows: We executed six independent test runs. Each of the test runs consisted of six calls to one function. In the chart, the first bar of each test run shows the time (in cycles) of the first call, the second bar shows the time of the second call, and so on. The results are shown in Figures 3 to 8.

Two main conclusions can be distilled from the figures. One is that caching has a much more pronounced effect on the execution times when an emulated

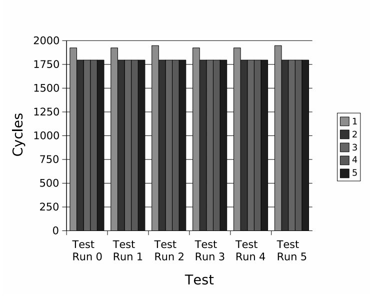

Fig. 3. Real Hardware (Cache On) - Pentium 4 1.8 GHz

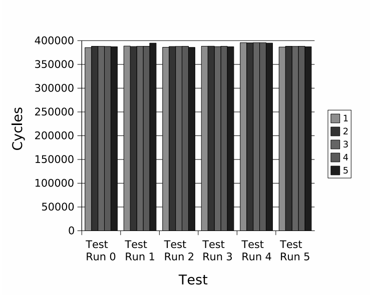

Fig. 4. Real Hardware (Cache Off) - Pentium 4 1.8 GHz

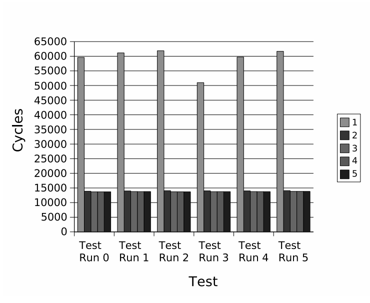

Fig. 5. Qemu (Cache On)

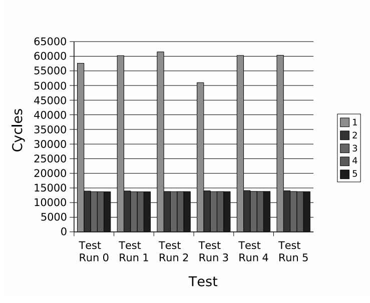

Fig. 6. Qemu (Cache Off)

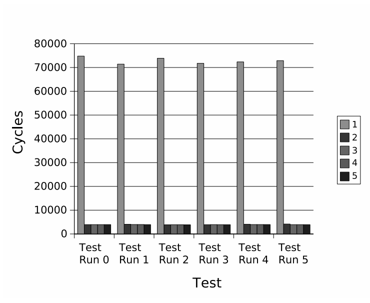

Fig. 7. VMware (Cache On)

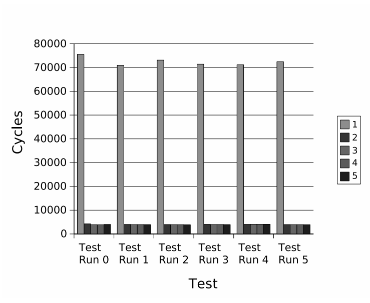

Fig. 8. VMware (Cache Off)

(or virtual) environment is used. The second conclusion is that for simulated environments, the timing results are the same, independent of whether caching is active or not. In other words, both Qemu and VMware discard requests of software to disable caching. Both observations can be leveraged to distinguish between emulated and real processors.

5 Detecting Hardware-Supported VMMs

This section discusses our preliminary research in detecting virtual machine systems that make use of hardware extensions recently proposed by the two largest x86 (IA-32 architecture) processor manufacturers, Intel Corp. and Advanced Micro Devices, Inc. (AMD). The specification of Intel's technique (called Intel VT: Intel Virtualization Technology) can be found in [21] and [15], and AMD's technique (called AMD-V: AMD Virtualization) can be found in [22] and [23]. Here, we only focus on the virtualization technique used by Intel processors.

5.1 The Intel Virtualization Technology (Intel VT)

In Section 2.1, we outlined the problems when attempting to virtualize processors that are based on the Intel IA-32 architecture. In particular, we discussed the problem of seventeen sensitive, but unprivileged instructions that make this architecture unvirtualizable. To address this problem, Intel introduced appropriate virtual-machine extensions. The key idea of these extensions is to provide a separate operation mode for VMMs. The operation mode for VMMs is called the root mode, while the mode for guest systems (virtual machines) is called the non-root mode. The code executed in root mode runs with higher privileges than all the code in the guest systems (including, of course, the guest operating system). If a guest system, even when running in processor level 0, attempts to execute instructions that would interfere with the state of other guest systems or the VMM, the processor recognizes this instruction and notifies the VMM. The VMM can then react to this instruction, and in particular, emulate the proper behavior to the guest, but limit (or prevent) its effects on the state of other virtual machines or the VMM itself. For example, moving a register to one of the control registers (e.g., CR0) would certainly interfere with the VMM and other running virtual machines. However, this instruction causes a VM exit and an immediate trap to the VMM. The VMM can then restrict the effects of the instruction and simulate its correct behavior to the guest. [15]

The Intel manual [15] states that "there is no software-visible bit whose setting indicates whether a logical processor is in VMX non-root operation. This fact may allow a VMM to prevent guest software from determining that it is running in a virtual machine." This sounds promising, as it could potentially allow one to create a analysis environment that would be invisible to malware. Interestingly, it also opens up completely new possibilities for stealth malware. That is, a rootkit could install itself as a virtual machine monitor and virtualize the user's operating system. In theory, such a rootkit could not be detected by conventional methods. Prototypes of this type of malware are the well-known BluePill [24], Vitriol [25], or SubVirt [26]. Unfortunately, despite our requests, we were not able to get access to either Blue Pill or Vitriol, and thus, could not test our detection techniques with these rootkits. Instead, we had to focus on the widely used VMware Workstation 5.5.2 [1], which has support for Intel VT, and the open source program KVM, a kernel-based virtual machine for Linux that also leverages the new Intel extension [27].

Timing Analysis In Section 4.6, we presented a timing analysis that used cache effects to allow us to distinguish between a system emulator and a real machine. Recall that system emulators and VMMs use the cache provided by the underlying hardware. Thus, their cache behavior is very similar to that of real hardware. As expected, code and data that has been accessed recently is delivered much faster when requested again. Interestingly, however, disabling the cache in the virtual environment does not lead to any changes in the observed behavior of the processor. That is, the virtual machine and the system emulator behaved *identically* independent of whether caching was enabled or disabled. Of course, timing results show that execution is significantly slower on a real machine when caching is disabled.

The discussion above shows that system emulators and traditional VMMs do not allow the guest system to disable the cache. This provides a straightforward mechanism for a program to determine whether it is running in a virtual environment. The question is whether the detection approach also succeeds when the VMM uses the Intel VT hardware extensions. The results of our analysis are shown in Figures 9 to 12 (The results of the test runs using the Core 2 Duo processor, without any virtualizers or IVT enabled, are comparable to the results on the Pentium 4 processor (see Figures 3, 4, 7 and 8) and therefore not listed here again.) It can be clearly seen that our cache timing analysis has the ability of distinguishing between a virtual environment and real hardware, even when Intel VT is used. Of course, a VMM that is aware of our timing tests might attempt to adjust its behavior such that it appears as real hardware. For example, we use time stamp counters (TSC) to measure the cycles that our timing program is executing. Thus, the VMM could modify the value of the TSC appropriately whenever we access it (using an instruction that traps into the VMM). Nevertheless, even though the TSC can be set to arbitrary values, the VMM would have to adjust the value depending on the number and the type of executed instructions. This is a non-trivial task, as any deviation from the expected result would serve as an indication that the test code is running in a virtual environment.

Also, turning off the cache involves setting a certain bit in the control register 0 (CR0), invalidating the cache (using the WBINVD instruction), and writing to a certain MSR (using the WRMSRS instruction). All these instructions cause VMX exit transitions. and it would be possible for the VMM to simply disable caching on the underlying processor. However, this would slow down the VMM as well as all other running guest systems executed on the processor. This is certainly not practical for VMMs in production settings (i.e., a program that provides an *isolated* duplicate of the real machine). On the other hand, a rootkit could simply turn off caching, as its main goal is to remain hidden.

6 Related Work

Malicious code is a significant security problem on today's Internet. Thus, a large body of previous work deals with different solutions to detect and analyze mal-

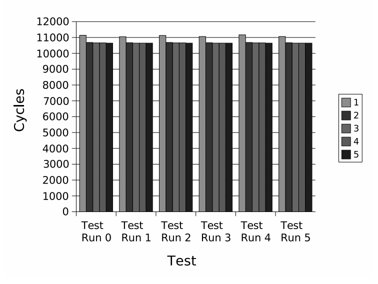

Fig. 9. KVM (IVT On - Cache On)

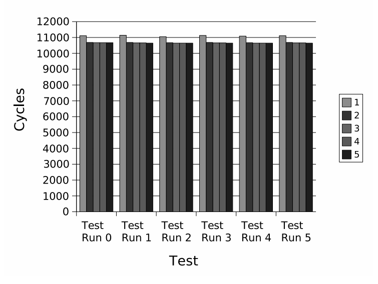

Fig. 10. KVM (IVT On - Cache Off)

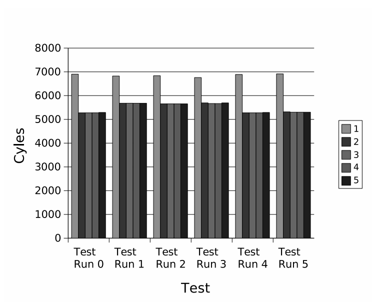

Fig. 11. VMware (IVT On - Cache On)

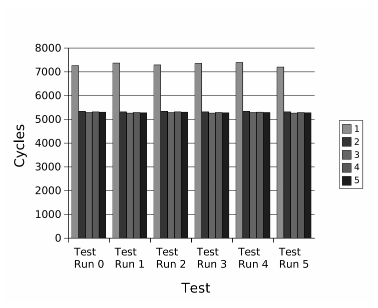

Fig. 12. VMware (IVT On - Cache Off)

ware. In general, these solutions can be divided into two groups: *static analysis* and *dynamic analysis* techniques.

Static analysis is the process of analyzing a program's code without actually executing it. This approach has the advantage that one can cover the entire code and thus, possibly capture the complete program behavior. A number of static binary analysis techniques [28,29,30] have been introduced to detect different types of malware. The main weakness of static analysis is the fact that the code analyzed may not necessarily be the code that is actually run. In particular, this is true when the malware makes use of code obfuscation.

Because of the many ways in which code can be obfuscated and the fundamental limits in what can be decided statically, dynamic analysis is an important and popular alternative. Dynamic techniques analyze the code during run-time. While these techniques are non-exhaustive, they have the significant advantage that only those instructions are analyzed that the code actually executes. Of course, running malware directly on the analyst's computer is not possible, as the malicious code could easily escape and infect other machines. Furthermore, the use of a dedicated stand-alone machine requires to reinstall the system after every test. Thus, the question arises in which environment a malware sample should be executed.

Virtual machines such as VMware [1] are a common and comfortable choice as an environment for malware analysis. Unfortunately, malicious code has adapted, and malware authors increasingly include routines into their creations that check for the presence of such virtual machines. These checks are actually quite straightforward, as the Intel IA-32 instruction set contains a number of instructions that are unvirtualizable [2]. Based on these instructions (and a number of alternative methods), a variety of detection techniques have been implemented. This includes RedPill [3], but there are also other VMware fingerprinting suites such as "Scooby Doo - VMware Fingerprint Suite" [31]. Recent publications also deal with general virtualization anomalies [32] and the usage of timing benchmarks to detect the presence of VMMs [33].

To address the problem of analysis environments that are easy to detect by malware, researchers have proposed systems based on emulation. For example, Cobra [5] and TTAnalyze [6] are two malware analysis tools that are based on full system emulation. While there was a general belief that these system are stealthier (i.e., less easy to detect), the extent to which this is true was unclear. In this paper, we analyzed possible approaches that can be used to identify system emulators. Moreover, we presented a number of concrete checks that allow us to detect Qemu (the emulator that TTAnalyze [6] is based on). The work closest to ours is a recent paper by Ferrie [34]. In his paper, Ferrie demonstrates mechanisms to detect a large number of virtual machine monitors, but he also addresses the issue of identifying emulators. The difference to this work is that we present a variety of approaches to detect system emulators, while Ferrie only focuses on a few specific implementation bugs of system emulators that incorrectly reproduce the behavior of specific instructions.

Finally, there has been recent work by processor vendors such as Intel and AMD to provide hardware extensions that help to make a virtual environment undetectable. While clearly a step into the right direction, there are properties (especially timing related ones) that are difficult to model precisely in a virtual environment and thus, provide malicious code with the possibility for detection (as our preliminary results indicate).

7 Conclusion

In this paper, we presented a number of techniques that can be used to detect system emulators. These techniques make use of specific CPU bugs, model-specific registers (MSRs), instruction length limits, alignment checking, relative performance measurements, and specific hardware and I/O features that are difficult to emulate. The conclusion that we have to draw from our experiments is that emulators are not necessarily stealthier than virtual machines. It is theoretically possible to adapt a system emulator to address each of the specific detection methods that we have outlined above. However, it is still an arms race as the adversary can find new indicators that are not covered yet. The underlying problem is that virtual machines and system emulators are not written with malicious code analysis in mind. Thus, it is typically enough to provide a simulated environment that is sufficiently close to the real system. Unfortunately, as far as security analysis is concerned, the emulation has to be perfect. Otherwise, the malware under analysis can discover that it is not running in a real environment and hence, adapt its behavior. Our experiments also indicate that some of our tests might be applicable to detect new virtualization technologies recently introduced by Intel and AMD, making the task of creating a perfectly stealth analysis environment very difficult.

Given the discussion in this paper, it may seem as if security researchers are losing the fight against malware authors with respect to hiding their analysis infrastructure. However, note that virtualization and system emulation are increasingly gaining popularity among a broad spectrum of computer users. These environments are not only useful for malware analysis, but also make system maintenance and deployment much easier. As a result, malicious code cannot expect any longer that a virtual environment is an indication of an analyst examining the code. Instead, it could also be a production server that uses virtualization technology.

References

1. : VMware Inc. `http://www.vmware.com/` (2006)
2. Robin, J.S., Irvine, C.E.: Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In: Proceedings of the 9th USENIX Security Symposium, Denver, Colorado, USA (August 14–17, 2000)
3. Rutkowska, J.: Red Pill... or how to detect VMM using (almost) one CPU instruction. `http://invisiblethings.org/papers/redpill.html` (2004)
4. Classified by Symantec Corporation: W32.Toxbot.C. `http://www.symantec.com/security_response/writeup.jsp?docid=2005-063015%-3130-99&tabid=2` (2007)
5. Vasudevan, A., Yerraballi, R.: Cobra: Fine-grained Malware Analysis using Stealth Localized-Executions. In: IEEE Symposium on Security and Privacy. (2006)
6. Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: A Tool for Analyzing Malware. In: 15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR). (2006)
7. : Qemu - open source processor emulator. `http://fabrice.bellard.free.fr/qemu/` (2006)
8. Popek, G.J., Goldberg, R.P.: Formal requirements for virtualizable third generation architectures. Communications of the ACM **17**(7) (July 1974) 412 – 421
9. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1 (2006)
10. May, C.: Mimic: a fast system/370 simulator. In: Conference on Programming Language Design and Implementation - Papers of the Symposium on Interpreters and interpretive techniques. (1987)
11. Bellard, F.: Qemu, a Fast and Portable Dynamic Translator. In: USENIX 2005 Annual Technical Conference, FREENIX. (2005)
12. Bellard, F.: Qemu Accelerator Module. `http://fabrice.bellard.free.fr/qemu/qemu-accel.html` (2006)

13. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture (2006)
14. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z (2006)
15. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2 (2006)
16. Lang, J.: Personal Correspondence (2006)
17. Intel Corporation: Intel Pentium 4 Processor - Specification Update (2006)
18. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M (2006)
19. : VirtualPC 2004 (build 528) detection (?). `http://www.securityfocus.com/archive/1/445189` (2006)
20. Intel Corporation: Using the RDTSC Instruction for Performance Monitoring (1997)
21. Intel Corporation: Intel Virtualization Technology Specification for the IA-32 Intel Architecture (2005)
22. Advanced Micro Devices, Inc.: AMD64 Architecture Programmer's Manual Volume 2: System Programming (2006)
23. Advanced Micro Devices, Inc.: AMD I/O Virtualization Technology (IOMMU) Specification (2006)
24. Rutkowska, J.: Introducing Blue Pill. `http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.ht%ml` (2006)
25. Zovi, D.D.: Hardware Virtualization Rootkits. In: BlackHat Briefings USA. (2006)
26. King, S.T., Chen, P.M., Wang, Y.M., Verbowski, C., Wang, H.J., Lorch, J.R.: SubVirt: Implementing malware with virtual machines. In: IEEE Symposium on Security and Privacy. (2006)
27. : KVM: Kernel-based Virtual Machine. `http://kvm.sourceforge.net/` (2007)
28. Christodorescu, M., Jha, S.: Static Analysis of Executables to Detect Malicious Patterns. In: Usenix Security Symposium. (2003)
29. Christodorescu, M., Jha, S., Seshia, S., Song, D., Bryant, R.: Semantics-aware Malware Detection. In: IEEE Symposium on Security and Privacy. (2005)
30. Kruegel, C., Robertson, W., Vigna, G.: Detecting Kernel-Level Rootkits Through Binary Analysis. In: Annual Computer Security Application Conference (ACSAC). (2004)
31. Klein, T.: Scooby Doo - VMware Fingerprint Suite. `http://www.trapkit.de/research/vmm/scoopydoo/index.html` (2006)
32. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is Not Transparency: VMM Detection Myths and Realities. In: Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI). (May 2007)
33. Franklin, J., Luk, M., McCune, J.M., Seshadri, A., Perrig, A., van Doorn, L.: Remote Detection of Virtual Machine Monitors with Fuzzy Benchmarking. Carnegie Mellon CyLab (2007)
34. Ferrie, P.: Attacks on Virtual Machine Emulators. In: AVAR Conference, Auckland, Symantec Advanced Threat Research (2006)