Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware

Xu Chen* Jon Andersen* Z. Morley Mao* Michael Bailey* Jose Nazario+
*University of Michigan – Ann Arbor +Arbor Networks

Abstract

Many threats that plague today's networks (e.g., phishing, botnets, denial of service attacks) are enabled by a complex ecosystem of attack programs commonly called malware. To combat these threats, defenders of these networks have turned to the collection, analysis, and reverse engineering of malware as mechanisms to understand these programs, generate signatures, and facilitate cleanup of infected hosts. Recently however, new malware instances have emerged with the capability to check and often thwart these defensive activities — essentially leaving defenders blind to their activities. To combat this emerging threat, we have undertaken a robust analysis of current malware and developed a detailed taxonomy of malware defender fingerprinting methods. We demonstrate the utility of this taxonomy by using it to characterize the prevalence of these avoidance methods, to generate a novel fingerprinting method that can assist malware propagation, and to create an effective new technique to protect production systems.

1 Introduction

Protecting end hosts from infections and break-ins remains a challenging problem today given the inherent software vulnerabilities in existing applications and commodity operating systems. Exploits and vulnerabilities are the primary means by which attackers gain unauthorized control over computer resources. Vulnerabilities are generally specific to particular software versions and configurations. Before attempting to compromise a system, attackers often perform reconnaissance by fingerprinting the attack target to discover specific artifacts or behavior of the target, possibly revealing certain properties that are useful for subsequent targeted infection attempts. The advantage of collecting such information about the target before further malicious behavior is to improve attack efficiency as well as to avoid detection.

A related essential goal of reconnaissance is to identify and avoid potential monitoring systems that attempt to analyze malware behavior. Such undesirable targets include honeypot-based monitoring systems [3] that collect information about attackers' behavior, including the very methods by which they attempt to identify these systems. In this perpetual arms race, this information is then used to better identify attackers in the future and to hide the evidence they use to identify monitors. Therefore, to avoid disclosing malware behavior to defenders, stealthy attackers intentionally avoid monitoring systems. From a defender's perspective, it would be best to hide or eliminate identifying information for both production and monitoring systems to increase attack difficulty. Unfortunately, such fingerprints are difficult to eliminate completely.

Based on the above discussion, we can generalize deployed computer systems into two broad categories: *production systems* actively used for real computing purposes and *monitoring systems* mainly used to attract and analyze attacker activities for detection purpose. The focus of our work is analyzing and exploiting the difference of malware execution under these two different environments. The contributions of our work are:

1. We introduce a detailed taxonomy that captures essential techniques for distinguishing between production systems and monitoring systems, which typically operate in virtualized and debugger environments.
2. We characterize the prevalence of malware evasion methods by executing 6,900 recently-captured malware samples under different environment — more than 40% of the total malware samples reduce their malicious behavior under virtual machines or with a debugger attached, and they account for potentially 90% of the Internet attacks during certain periods.
3. We provide a remote network-based reconnaissance to differentiate between virtual machines and plain machines, and even possibly between variants of virtual machines. To the best of our knowledge, this is the first remote network-based fingerprinting method for detecting VMs.
4. Given the prevalence of evasive malware, we introduce *a new paradigm of protecting production systems by*

1-4244-2398-9/08/$20.00 ©2008 IEEE

making them appear to be monitoring systems. We implemented several fingerprint imitation techniques and evaluated our method using real malware samples.

The paper is organized as follows: after briefly surveying related work in §2, we provide a taxonomy of malware evasion techniques, focusing on anti-virtualization and anti-debugging behavior in §3. We then describe three applications based on this taxonomy in §4, namely evasion-informed malware characterization, remote network fingerprinting, and deterring malware by imitating fingerprints. We discuss the implications and limitations of our work in §5 and conclude in §6.

2 Related Work

Understanding the execution behavior of malicious programs is of critical importance. Compared to static malware analysis [22], dynamic analysis of malware programs gives rich information about runtime behaviors. Recent work performs malware identification based on behavioral patterns [11, 12]. For example, behavior-based clustering of malware programs was proposed [8]. Another interesting approach [24] is to take advantage of virtualization — manipulating memory objects during run-time to exploit multiple execution paths of malware and building a full view of the malware behavior. Though it is known that evasive malware does exist, to the best of our knowledge, no existing work has extensively studied the difference in malware behavior across various execution environments.

Virtual machines can be an effective environment in which to study activities and techniques of attackers [4, 17, 20]. For example, VMs have been used to infiltrate a botnet to discover its internal structure [26]. Security researchers also use debuggers, such as WinDbg, SoftICE, etc., to extensively characterize malware behavior. The two approaches are similar in the sense that they both execute the real code, as opposed to static analysis. In reaction to this trend, attackers have sought to fingerprint VMs and debuggers during runtime [9, 14, 10] by checking either OS objects or benchmarking CPU instruction execution time to actively avoid monitoring systems. Some malwares, such as the IRC bot Agobot [7], actively seeks to detect the presence of VMs and then changes its behavior accordingly.

In response to these fingerprinting techniques, defenders may attempt to hide VM fingerprints [23] or develop more stealthy debuggers [29]. Despite these efforts, several works have shown limitations on how well the presence of a VM can be hidden. For example, legacy CPU instruction sets may contain instructions that simply cannot be virtualized [28], and the discrepancies revealed by physical resources are inherently difficult to eliminate [13, 27, 28, 19] due to the side effect of virtualization. To successfully fake such a genuinity test, a VM would need orders of magnitude more computing power than the hardware that it is emulating. Instead of focusing on eliminating fingerprints associated with monitoring systems, our work simplifies the problem by deterring attackers through production systems that appear as monitoring systems, thereby contributing to defenders' ability to compete in the arms race by misleading attackers.

3 Taxonomy of Anti-Virtualization and Anti-debugging Techniques

In an effort to avoid potential monitoring by adversaries, attackers often attempt to distinguish systems running on virtual machines and in debuggers from those running on plain machines. While a great deal of attention has been given to specific techniques used to detect these systems, (e.g., Redpill [5]) with few exceptions (e.g., Honeypot Detection [14]) the questions of how these techniques relate to each other, to the systems they are monitoring, and to the goals of the adversaries have remained largely unexplored. To better understand how to answer these questions, we have developed a taxonomy of anti-virtualization and anti-debugging techniques.

As is the case with the general class of adversarial fingerprinting, anti-virtualization and anti-debugging techniques are based on the assumption that systems of interest carry characteristics that differentiate them from "normal" systems. In our taxonomy, we group these techniques by the system abstraction at which they operate as well as the class of virtualization or debugging characteristics they exploit. In addition, for each method we develop a set of metrics to evaluate and differentiate the various classes. These include: 1) What is the least level of access required to uncover the characteristic? 2) How accurate is the method, assuming no evasion is done by the target? 3) What is the complexity involved in building a practical tool to detect this characteristic? 4) How difficult is it to try and mask this characteristic? 5) How can we imitate the characteristic on OSes running on plain machines to fool existing fingerprinting software? This analysis can be found in Table 1, with detailed explanations of the various categories occurring in subsequent sections.

3.1 Hardware

Both virtual machines and debuggers can make hardware-detectable changes to the system when they are present. For example, debuggers can set hardware breakpoints and a virtual machine, by definition, emulates hardware. Such hardware differences between debuggers or VMs and a native, non-instrumented environment are detectable.

Abstraction	Artifact	Accuracy	Access level	Complexity	Evasion	Imitation	Examples
Hardware	device	high	local network	medium	easy	easy	Reptile
	driver	high	local user	low	medium	easy	Roxio
Environment	memory	high	local root	medium	hard	medium	Reptile, Agobot, Peacomm.C
	system	high	local root	high	hard	hard	Reptile
Application	installation	high	local user	low	easy	easy	Reptile, Rbot, Phatbot
	execution	high	local user	low	medium	easy	Rbot, Phatbot
Behavioral	timing	medium	remote network	medium	medium	medium	Reptile, Nugache

Table 1. A taxonomy of common malware anti-virtualization and anti-debugging techniques.

3.1.1 Device

Virtual machines often create specific hardware devices with identifiable attributes, either overt or subtle. An overtly virtual device would include an Ethernet device from VMWare with a specific, well-known manufacturer prefix. A more subtle attribute would be a failure of the CPU emulator or translator to handle illegal opcodes. The VMWare VGA adapter, for example, has a well-known device string that is specific to that environment. Other VMWare devices have identifiable strings such as "BusLogic BT-958" and "pcnet32" on Linux, and in Windows on VMWare, registry keys associated with the SCSI disk drivers also have specific strings associated with them. Finally, the Bochs emulator has a debug port visible from the system specific to that emulated platform. In these cases, malware can use these features to determine that it is in a non-native environment. The `doo` tool [6] on Linux and Windows uses shell scripts that look for such fingerprints in the form of vendor strings and special hardware types in the Linux system messages and Windows registry table.

3.1.2 Driver

Most virtual machines and debuggers create telltale drivers that are specific to the tool being used and generally do not appear otherwise. For virtual systems, these drivers are needed to adapt the guest OS to the host OS. For debuggers, these drivers are needed to communicate with the rest of the system. The SoftICE kernel debugger, for example, uses virtual drivers to communicate with the kernel. These are identifiable as files with names such as `SICE`, `NTICE`, and `SIWVID`. Software can look for these drivers to determine that it is in a suspicious environment.

3.2 Execution Environment

The execution environment of a process is altered when it is in a virtual machine or running under a debugger when compared to a native machine under normal conditions. Kernel space memory values, for example, are usually slightly different between native and virtual systems. Furthermore, debuggers have to perturb process and sometimes kernel memory to instrument the process for inspection. In both cases, these differences can be used to mark an environment as "instrumented" to a malware process.

3.2.1 Memory Artifacts

VMWare creates a "ComChannel" channel between the host and guest OS, allowing for inspection and control between the two systems. VMware and Virtual PC hooks work similarly [18, 2], and programs like `checkvm` [6] can look for these VMware hooks. Another VMWare feature that qualifies as a memory artifact is the interrupt descriptor table (IDT), whose presence is detected by tools such as "Red Pill" [5]. This table reliably resides at a well-known memory address that is different from a native Windows value. For debuggers, the Windows API sets a flag that is detectable using the IsDebuggerPresent() and CheckRemoteDebugger() API calls, helping to prevent debugger attachment loops, as well as software breakpoints. In both cases these are intentional markers and alterations to provide features, and often can be masked by the system with the right tools. Because these are accessible to any process with the right minimum permissions, the process can look for these markers and assume a modified environment.

3.2.2 Execution Artifacts

OS-level changes that are the results of bugs in the implementation are inadvertent artifacts and just as useful to the attacker as specific markers. These can include CPU instruction bugs in the virtual machine and how they are handled by the guest OS. Debuggers may also leave unintentional traces in the execution path and modify the call-stack, such as altering the UnhandledExceptionFilter or single byte instructions in the case of OllyDbg. These can be significantly harder to thwart for the honeypot or sandbox operator because they are often poorly defined and systemic flaws in the implementations.

3.3 Application

Some of the easiest methods to detect the presence of an instrumented environment is through the tools that are installed and executed on the system. Both virtual machines and debuggers usually have external applications that are visible to the process checking the environment.

3.3.1 Installation

Even if the processes associated with the sandbox environment are not executing, if the tools are installed with well-known names and in a well-known location, they can be used to mark a system as "suspicious" to a malicious executable. Both registry keys (in the case of Windows) and files on disks can be enumerated by the application to look for VMWare Tools, for example, or debuggers. These detection techniques are usually reliable to qualify a host as an instrumented environment, and are also easy to mask through simple, non-default installations.

3.3.2 Execution

The running processes, services, and windows associated with debuggers and virtual machine management can also be used to identify their presence to a suspicious process. Unless their names have been altered, the malware can enumerate processes, services and window titles to look for names associated with well-known virtual machine management software or debuggers. For instance, malware can assume that only a virtual machine would have a service named "VMtools," or that a window with the title "OLLYDBG" is associated with a debugger. This detection can be trivially defeated by renaming processes or through API call hooking.

3.4 Behavior

An artifact more difficult to conceal is the timing differences between two environments. When single-stepping a process through a debugger, for instance, the wall clock differences between any two points will grow dramatically when compared to a native execution. Malware can perform two time checks and infer that it is running under such conditions if unusual difference is seen. Similarly, some instructions take much longer to finish in virtual machine than in a normal machine due to virtualization overhead. A program continuously executing these instruction can soon tell VMs apart from plain machines.

3.5 Limitations of This Taxonomy

The taxonomy of monitoring environments described above is by no means a complete listing of all of the methods by which malcode could detect that it is running in an analysis environment. Techniques that we omit include those suggested by Zhou and Cunningham, for example, describing complex infection propagation mechanisms that are hard to defeat without the honeypot becoming a risk source [30]. Also, this taxonomy omits any detection of AV processes that may influence malcode behavior. Instead, this taxonomy focuses on commonly-found evasion mechanisms that we routinely observe in actual malware samples.

4 Application of Malware Evasion Taxonomy

In this section, we introduce three interesting applications of the malware evasion taxonomy. In §4.1, we attempt to develop an understanding of how recent malware behaves under different execution environments by characterizing the actual differences in terms of malicious behavior. So far, existing techniques of malware evasion all require access to the target host. We have developed a novel technique to *remotely* determine suspicious execution environment without any local host access. We outline this network-based fingerprinting technique in §4.2. Given the observed gap between the malware behavior under plain-machine execution and suspicious execution environments, we developed a novel technique to *imitate* suspicious environment fingerprints on production systems with low overhead to help deter malware. We describe this technique, its implementation and evaluation in §4.3.

4.1 Malware Characterization

To better understand the prevalence of the techniques used by malware to evade monitoring systems, in this section we characterize the difference of malware runtime behavior by running malware samples under three different environments (for Windows) — in plain machines, in virtual machines, and with a debugger attached. We execute 6,900 distinct malware samples[1] collected from September 3rd, 2006 to September 9th, 2007, using a variety of sources including Web page crawling, spam traps, and honeypot systems. When executed in Norman Sandbox and under a variety of anti-virus software, 99% of them are reported to be known malware [25].

4.1.1 Execution Environments and Results

We set up three standard Windows environments to execute malware samples. For *plain-machine execution*, we install Windows directly on a plain machine and automatically execute each malware sample after system booting finishes. For *virtual-machine execution*, we install Windows inside VMware Server running on Linux. For *debugger execution*, we install Windows on a plain machine, but execute samples with WinDbg attached, using the command line: `cdb.exe -o -g -G malware.exe`. We studied Windows XP SP2 without applying any additional

[1] For interested readers, the scanning results from various anti-virus software can be downloaded from `http://www.eecs.umich.edu/robustnet/malware/avdetails.tar.gz`.

patches in our analysis, given the time frame in which the malware samples are obtained.

Only one malware sample is executed during each iteration to prevent potential interference, after which the whole system is rolled back to a clean state to prevent disruption across executions. No network traffic, except for DNS query, is permitted, to prevent spreading infection to other hosts. For each malware sample, we execute it for two minutes to capture most, if not all, of its behavior. We believe two minutes is sufficiently long for most malware samples to exhibit malicious behavior. We execute malware samples that behave differently across environments for a slightly longer duration to ensure that we capture their key behavior.

We use the Backtracker system [20] to capture the system-level behavior of malware samples and compare execution traces across environments. This includes program execution events related to every malware execution, including disk read/write, registry table read/write, memory mapped read/write, process fork, inode update, file execution of the malware process and its child processes. We treat the events that cause persistent state changes and system state changes to be malicious, including file/registry key/named pipe modification and process fork/execution.

Some malware samples cause the system to crash or to reboot so that we cannot collect complete execution traces. To ensure fair comparisons, we ignore 217 (3.1%) samples with missing data in at least one of the execution setups. Further investigation shows that some of these malware samples directly cause Windows to reboot, even under a freshly-installed system. Some samples only reboot Windows under VM execution, which we will discuss later. It is also possible that some malware programs crashed because of our instrumented Backtracker system.

We ignore another 461 (6.7%) samples that do not exhibit any malicious behavior under plain-execution, mostly due to an incompatible Windows version or lack of user interaction.

4.1.2 Impact of Virtual Machines

Among the remaining 6,222 samples, 5,929 (95.3%) of them exhibit the same behavior under VM execution as under plain execution. 167 (2.7%) samples exhibit fewer malicious behaviors. In those cases, the VM environment prevented 225 file modifications, 188 process creation attempts, 203 named pipe read/writes and 5,102 registry modifications. 75 (1.2%) samples crashed and caused Doctor Watson to start. Overall, at least 4% ($\frac{167+75}{6222}$) of the malware samples exhibit less malicious behavior under VM executions. Further analysis shows that 60 malware samples directly reboot Windows under VM, which results in missing execution traces for them. We suspect these cases are caused by malware intentionally rebooting the system to evade VM-based analysis.

4.1.3 Impact of Debuggers

For the same 6,222 malware samples, 3,662 (58.5%) of them exhibit the same behavior with debuggers attached as plain execution. Quite surprisingly, 2,481 (39.9%) of them have fewer malicious behaviors, reducing the number of file modifications or creations by 8,406, that for registry key modifications by 57,510, named pipe read/writes by 3,201, and file execution by 2,150. Another 28 samples seem to have additional process execution behavior in the debugging environment, but further investigation shows that a file, `removeMeXXXX.bat`, is created on the disk and then executed to remove the malware executable and the bat file itself. We suspect that these malware samples try to eliminate their traces upon detecting debugging environments. Overall, around 40% of malware samples exhibit less malicious behavior in debugging environments.

4.1.4 Popularity of Environment-aware Malware

Our malware dataset only records the first day when a particular sample was observed, without any indication of popularity. Fortunately, mwcollect.org [1] keeps a detailed log of when and which malware sample is observed by their honeypot machines. We obtain the hit log from mwcollect.org and correlate it with our malware samples.

Only 101 samples appear in both data sets during the overlapping time period. These samples account for a total of 68912 hits in the mwcollect database. Among these samples, 20 of them have less malicious behavior under VM and account for 11928 (**17.3%**) of the total hits. 69 of them have less malicious behavior with debugger attached and account for 64719 (**93.9%**) of the total hits. This result might be biased because of the small number of overlapping samples. However, it clearly demonstrates the tremendous popularity of environment checks present in modern malware samples.

4.1.5 A Malware Case Study: Storm Worm

We take a particular malware sample as an interesting example to illustrate the behavioral difference across environments. We compare the execution traces for a malware program with MD5 hash `6d0e98688ec3ce31479e02dad96882e0`, a known variant of the currently prevalent Storm Worm.

Under plain execution, the malware extracts two files, `dNVeHk3.exe` and `wincom32.sys`, and then executes `dNVeHk3.exe`. During the execution, a registry key is modified to disable the Windows Firewall/Internet Connection Sharing (ICS) service. Under VM execution, no files are extracted or executed, but the registry key is modified. Interestingly, under debugger execution, it terminates

quickly without modifying any system state. This again confirms our observation that current malwares are increasingly more intelligent at avoiding debugger and VM-like environments.

4.2 Fingerprint Remote Virtualized Hosts

To the best of our knowledge, all the current evasion techniques happen after the executable is uploaded and executed on the target host, which could very well be a monitoring system. From the attacker's point of view, they may not want their programs to even land on those monitoring systems, in order to delay detection and signature generation on them and thus prolong their prevalence.

In this section, we briefly introduce two methods that can be used by malware to detect remote virtualized hosts. We found that it is possible to accurately detect a remote VM host by sending a few hundreds SYN packets, even when network delays are as high as 300ms. Our method can even differentiate among different VMM types, namely VMware and Xen, which could be potentially used to exploit different vulnerabilities on each VMM system.

4.2.1 Remote Timing Test

The TCP timestamp option [16] is intended to improve high-performance, high-bandwidth connections by giving the sender a more accurate way to measure RTT (round trip time). A TCP timestamp clock increases monotonically with fixed frequency ($FREQ$) between 1Hz and 1000Hz. TCP timestamp was first exploited by Kohno *et al.* [21] as a convert channel to reveal a target host's physical clock skew, which uniquely identifies a physical machine.

Our observation is that virtualized hosts have a more perturbed clock skew behavior (visualized in Figure 1). The difference in *randomness* is because OSes running on plain machines keep accurate timing by receiving regular hardware interrupts generated by hardware oscillators, while guest OSes rely on VMM-generated software interrupts, which can be lost or delayed. We use this discrepancy to determine whether the target is running on a VM. We follow two steps to remotely measure randomness:

Determining TCP Clock Frequency: There are only a few common TCP clock frequencies; Windows hosts exhibit a clock frequency of 10Hz, while most new distributions of Linux exhibit clock frequencies of 100Hz, 250Hz, etc. To determining a target host's TCP clock frequency, for each TCP packet from the target host, we record our system time t when the packet is received and the TCP timestamp T stored in the option field. By taking two packets separated by a few seconds, we can estimate how quickly the TCP timestamp is incremented by calculating $F = (T_1 - T_2)/(t_1 - t_2)$ and truncating it to the nearest possible TCP clock frequency. We find this method works for all tested operating systems running on both plain machines and VMs.

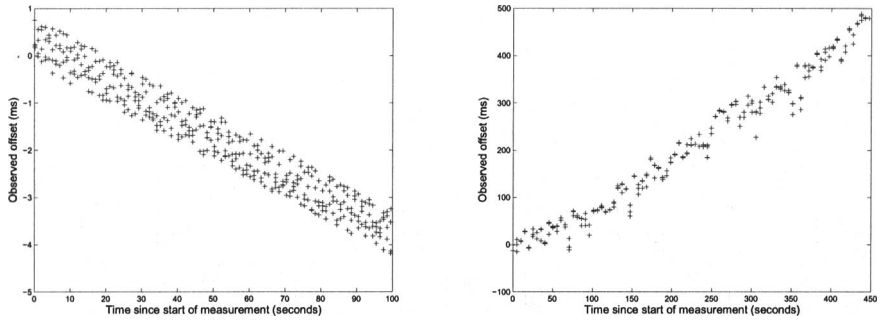
Characterizing Randomness: For any packet from the target host, we consider its TCP timestamp T_i and the time we receive it t_i. Given that we have the TCP clock frequency $FREQ$, we can transform the TCP timestamps into clock readings. Given a timestamp T_i, the time elapsed since the first packet should be roughly $(T_i - T_0)/FREQ$ seconds. Then, we have two clocks to look at: 1) $x_i = (t_i - t_0)$ which is the time elapsed locally [2]; 2) $w_i = (T_i - T_0)/FREQ$, which is the time elapsed on the target host. We can plot (x_i, y_i), where $y_i = x_i - w_i$ is the clock skew. Since the two clocks come from different physical devices, the clock skew (y_i) becomes greater as time passes, as shown in Figure 1.

We use *linear least squares fitting* to get a line $f(x) = Sx + q$. We then quantify the deviation of each point from this line using the squared error (SE): $SE_i = [f(x_i) - y_i]^2$. Correspondingly, the sample mean of SE (MSE) is calculated as $\overline{SE} = \frac{\sum_i [f(x_i) - y_i]^2}{N}$, which we use as the randomness indicator.

To obtain TCP packets, we actively initiate TCP connections to a few known ports — port 22 for Linux standard `ssh` service and 3389 for Windows remote desktop service. Each new TCP connection gives us one TCP timestamp sample. We stop probing when the samples give a converged linear least squares fit $f(x)$. All our experiments finish within a few minutes, even when probing at a very slow rate of one connection per second. Passive sniffing is a better choice, if we have access to a machine within the same subnet of the target host.

After obtaining the randomness indicator MSE from a remote host, we compare it with the *baseline* values from OS installed on plain machines. For Linux machines, we have a theoretical estimate of MSE. We examined various versions of the Linux kernel source and found TCP timestamps are a simple truncation of the hardware clock using $FREQ$, so the TCP timestamp is updated exactly every $h = 1/FREQ$ seconds. The deviation of the TCP clock to real clock should ideally be uniformly distributed across $[-h/2, h/2]$. Thus the mean of the theoretical error should be $\mu_{SE} = \frac{\int_{-h/2}^{h/2} x^2 dx}{h} = \frac{h^2}{12}$. For Linux hosts with $FREQ = 250Hz$ and $FREQ = 1000Hz$, the theoretical mean for SE should be $1.33ms^2$ and $0.083ms^2$ respectively. Various experiments have verified that this estimate is very accurate. Also, we found that the randomness in network delay within a short period is usually not large enough to cause a significant increase in MSE.

[2] We assume the local system clock is accurate because it can achieve a $1\mu s$ accuracy — much finer-grained than TCP clocks.



(a) Clock skew of a Linux host (1000Hz) (b) Clock skew of a Linux on VMware (250Hz)

Figure 1. Clock skew of Linux hosts on a plain machine and a VM.

For Windows, we have no access to the source code, so we do not know how the TCP timestamps are generated. Therefore, we measured the SE of clock skew from Windows hosts and came up with an empirical value. It turns out that the MSE is roughly $1667ms^2$ for Windows running on plain machines — exactly twice as large as the theoretical value of $1/(10^2 * 12) = 833ms^2$, like because Windows generates TCP timestamps using a proprietary algorithm. In our experiments we use $1667ms^2$ as the baseline for Windows installed on plain machines.

Fingerprinting a remote host is simply done by comparing the randomness indicator MSE with baseline randomness values. Experimental results are shown in Table 2. For Linux hosts running on plain machines, the MSE is extremely close to theoretical values, even for target hosts whose RTT is 300ms away from our test machine. On the contrary, Linux hosts installed on virtual machines within the same subnet exhibit orders of magnitudes larger MSE than theoretical values. Interestingly, Xen introduces much less randomness than VMware does, probably because they have different algorithms for firing software interrupts. On Windows machines, the randomness is already very large due to the relatively small $FREQ$ value. However, the randomness introduced by VMM is still very obvious — VMware adds around $270ms^2$ to MSE, while Xen adds around $130ms^2$. We have applied the Z-test to obtain the statistical confidence of the likelihood that the target host is running under different virtualized platforms. The details of this method is omitted for simplicity.

To imitate such fingerprint on plain machines, we can obfuscate the TCP timestamps by introducing extra randomness. On the other hand, we can erase such fingerprint on VMs by normalizing the TCP timestamp at the VMM layer. The details are omitted due to space limit.

4.2.2 MAC Address

Besides the TCP clock, we can also exploit properties related to virtualized network devices. The Media Access Control (MAC) address is a unique identifier for network hardware devices. VM software by default set MAC addresses of virtual network interfaces within particular ranges. By extracting the MAC address of a raw packet, we can infer whether the sender is using a real NIC or a virtual one. Fingerprinting a subnet of machines can be done within a few seconds by sending ARP queries and gathering MAC addresses from the reply messages. However, this fingerprinting method assumes that the attacker has compromised at least one host within the same subnet of the target host and has acquired root access of that compromised machine to sniff packets.

4.3 Deterring Malware by Imitating Debuggers and VMs

Given the goal of most anti-debugging and anti-virtualization code is to avoid detection by defenders, a common reaction by a malware upon detecting these environments is to suppress any additional behavior that might disclose it presence. Given this tendency, one possible approach to reducing the amount of malicious behavior exhibited on a defender's system would be to make it appear as a monitored environment. In this section, we discuss our work in building a set of tools that are capable of mimicking those monitoring systems on production systems for the purpose of deterring malware. We discuss our general emulation approach as well as an evaluation of these techniques using our malware sample set.

4.3.1 Approach

In order to deter attackers, we use several techniques described in our taxonomy in §3 to disguise production systems as virtualized and debuggers. These include:

- **Drivers:** For Windows hosts, we created a program that changes the driver information strings in the Windows registry table to appear to be VMware. To

	Linux			Windows		
Setup	Plain 1kHz	VMware 1kHz	Xen 100Hz	Plain 10Hz	VMware 10Hz	Xen 10Hz
$MSE(ms^2)$	**0.0854**	**245.8**	**23.1**	**1671**	**2042**	**1804**
Base Line (ms^2)	0.0833	0.0833	8.33	1667	1667	1667

Table 2. Remote timing fingerprinting results for various setups.

fake the presence of SoftICE, we modify the API `Openfile()`, which is used to open device drivers. If a program requests `\\.\NTICE`, we return a non-NULL value to imitate the existence of SoftICE driver.

- **System:** In order to appear virtualized, we also intercepted system calls that would have triggered exceptions in non-virtualized hosts. Since we do not have direct access to the source code of Windows, we used Detours [15] to intercept Windows API calls, and in particular `KiUserExceptionDispatcher()` in `ntdll.dll` is intercepted in order to imitate VM hook fingerprints. If the register values match the VM checks, we return artificial values without causing any exception. In addition, we modify `KiUserExceptionDispatcher()` to deal with malware searching for `int 1`.

- **Memory:** With the Windows Detours package, we are able to further modify the behavior of `IsDebuggerPresent()` and `CheckRemoteDebuggerPresent()`. On a production machine where no debugger is present, we can return `true` for both API calls for any executables. This can successfully fake the presence of WinDbg under these checks.

- **Execution:** To fool run-time anti-debugging checks we also created an empty Windows application with window name `OllyDbg` so that a direct window search will falsely conclude the existence of OllyDbg.

These tools are certainly quite simple and superficial, only serving as a proof-of-concept. It is not our purpose to imitate all the possible existing fingerprints.

4.3.2 Evaluation

To evaluate our system, we ran our 6,900 malware samples in two environments: one in a vanilla Windows XP install, and one with our imitated VM and debugger fingerprints installed. Of these samples, 264 have missing execution traces, which we ignore. Also excluded are those 461 samples that do not exhibit any malicious behavior, even under real environments. We compare the execution results for the remaining 6205 samples.

Comparison of behaviors We found that 610 malware samples have reduced malicious behavior with our imitated fingerprints. Our tool set altogether prevented 1,850 file operations, 578 process creations, and 15,680 registry modifications. Besides these prevented operations, 27 samples also created the file `removeMeXXXX.bat` to erase themselves. 19 samples caused `dwwin.exe` (Microsoft Doctor Watson error reporting tool) to execute. These indeed show that our imitation tools are capable of misleading malwares into detecting the existence of virtual machines and debuggers. Overall, 2,625 (42.2%) samples reduced their malicious behavior under either VM execution or debug execution. Our simple tools are shown to be able to create similar effects on 656 (25%) of them.

The detailed comparison of the effectiveness of VM, debugger and our imitated fingerprints in terms of deterring malicious behavior is shown in Table 3. We only created very few imitated fingerprints, especially those related to debuggers, yet our tools still effectively deterred one fourth of what a real debugger can and already more than what VM can deter. Overall, these results demonstrate great promise of our approach.

With respect to the Storm Worm variant mentioned in §4.1, we found that our imitated fingerprints are capable of achieving the same effect as VM execution, meaning that file extractions and executions are successfully prevented. However, a registry key is still modified, indicating our imitation of debuggers is not complete. The reason might be that we only superficially changed the result of the API `IsDebuggerPresent()`, while a program can directly read the corresponding bit in its process PEB. Imitating such fingerprints to a deeper level can help deter such sophisticated malware.

Our experiments are mostly performed under Windows XP SP2, which prevented us from logging the network activities due to software incompatibility. To further understand the effectiveness of preventing network behavior, we set up a smaller experiment to run only those 656 malware samples under Windows XP, without any patches installed. We found that our imitated fingerprints can deter 4,029 (70.5%) of the 5,717 network connection attempts made during plain execution, compared to 5,674 (99.2%) that can be deterred by a debugger, disregarding the fact that we only implemented a few debugger fingerprints.

Performance Overhead For VMware driver and OllyDbg windows insertion, the introduced overhead to the system is negligible, given that they are not accessed by any

	Process	File	Registry	Named Pipe
VM	188	225	5102	203
Debugger	2150	8406	57510	3201
Our Tools	578	1033	15680	839

Table 3. Malicious behavior prevented by different setups.

legitimate applications and do not consume much computation resources.

VMware hook and debugger checking API interception are handled by the Detours package [15]. The overhead is around 15μs for intercepting a function call [15]. Moreover, we only intercept a few functions that are rarely executed in normal programs but used by malware. For example, exception handler is only executed when there is a significant software fault.

4.3.3 Improving Imitated Fingerprints

During our experiment for evaluation, we tracked all potential reconnaissance actions to get insight on improving our system, corresponding to many of the checks described in §3. We found that among the 656 samples that reduce malicious behavior with our imitated fingerprints, `IsDebuggerPresent()` was checked for 1,040 times, SoftICE driver checked for 458 times, VM hook checked for 29 times. In addition, we found 5,162 search attempts for the window name "creating rules for malware.exe," 654 attempts for "Hidden Process Requests Network Access," and numerous others for `FileMon`, `APIMonitor` and `Regmon`, indicating evasive actions against AV software, like AVP and Kaspersky, and monitoring programs. We can also apply malware multi-path execution techniques [24] to reveal the condition checks that clearly separate benign and malicious behavior — they are likely related to the discrepancies that malware cares about. We can enrich our imitated fingerprints based on these inputs.

5 Discussion

Intel-based VMs is a current topic with much research and industry activity. As hardware and operating system support for VMs improve, some existing VM fingerprints may disappear, e.g., instruction fingerprints. On the other hand, as VMs become more prevalent in production systems, the value of knowing that a particular OS is running inside a VM diminishes. For example, the most sophisticated worm so far, the Storm Worm, does not seem to care about virtual machine environment. However, we should notice that there always exists a significant gap between monitoring systems and production systems in general. Debuggers, as an example, are prevented by a large portion of malware. Designing monitoring systems to be identical to production systems is challenging; thus, there always exist discrepancies to be potentially exploited.

By imitating fingerprints that only exist in monitoring systems, we have shown that it is possible to deter many sophisticated malware. However, a question that remains is to what extent we should imitate such discrepancies. For example, we can return a fake result by hooking the `IsDebuggerPresent()` API, but malware can further check the process PEB directly. We argue that for malwares to detect monitoring systems, having false negatives is much more disastrous than having false positives. In this case, malware programs should exit even if only one of the tests returns true. In this case, we can always raise the bar a bit higher to help win this arms race.

Another question is how we deal with attackers' attempts to detect our imitated fingerprints. For example, we hook the `IsDebuggerPresent()` API call, but do not change the corresponding flag in PEB. This inconsistency is an obvious indication that certain system artifacts are intentionally arranged purely for being fingerprinted. It is possible that this may lead attackers to program higher levels of intelligence into more stealthy malware, e.g., testing the actual behavior of a driver, instead of simply detecting a device driver name. We would argue that as the escalation worsens, attackers would suffer from more overhead to detect sophisticated fingerprints. Also, the existence of such inconsistency already indicates the target host is probably monitored and guarded.

6 Conclusion

Our work is the first to develop a detailed taxonomy of evasion techniques that are actively used by modern malware to avoid monitoring systems based on virtualization and debugger characteristics. These techniques span different layers of the computer system with varying levels of difficulty to be performed, obfuscated, and imitated. As a direct application of the taxonomy, we performed large-scale experiments over 6,900 recent malware samples to understand the behavioral gap among plain-machine execution, virtual machine execution, and debugger attached execution. Our results show that a significant percentage of malware samples actively evade monitoring systems by exhibiting less malicious behavior. Despite various powerful evasion techniques, none can be used to detect a remote networked monitoring system. To fill this gap, we developed a novel technique that detects a remote networked virtual machine based on its clock skew behavior. Finally, we proposed a novel approach to mislead attackers and subse-

quently deter them from infecting target hosts, by making production systems operating on plain machines appear as monitoring systems. We demonstrate that only a few lightweight imitated fingerprints can already deter a fairly significant portion of malware samples and a large portion of malicious behavior. We believe our work makes important progress to assist defenders in combating emerging threats of evasive malware through a novel deterrence technique.

References

[1] Collaborative malware collection and sensing. http://alliance.mwcollect.org.

[2] Detect if your program is running inside a vm. http://www.codeproject.com/system/vmdetect.asp.

[3] The honeynet project. http://project.honeynet.org.

[4] Honeypotting with VMware - basics. http://www.seifried.org/security/ids/20020107-honeypotvmware-basics.html.

[5] Red pill. http://invisiblethings.org/papers/redpill.html.

[6] Scoopy doo. http://www.trapkit.de/research/vmm/scoopydoo/scoopy_doo.htm.

[7] C. Associates. Win32.agobot. http://www3.ca.com/securityadvisor/virusinfo/virus.aspx?id=37776, July 2004.

[8] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, , and J. Nazario. Automated Classification and Analysis of Internet Malware. In *10th International Symposium on Recent Advances in Intrusion Detection (RAID 2007)*, September 2007.

[9] J. Corey. Advanced honeypot identification. Phrack magazine, January 2004. http://www.phrack.org/fakes/p63/p63-0x09.txt.

[10] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, and L. van Doorn. Remote detection of virtual machine monitors with fuzzy benchmarking. *ACM SIGOPS Operating System Review*, April 2008.

[11] D. Gao, M. Reiter, and D. Song. Behavioral Distance for Intrusion Detection. In *8th International Symposium on Recent Advance in Intrusion Detection (RAID 2005)*, September 2005.

[12] D. Gao, M. Reiter, and D. Song. Behavioral Distance Measurement Using Hidden Markov Models. In *9th International Symposium on Recent Advance in Intrusion Detection (RAID 2005)*, September 2006.

[13] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. 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.

[14] T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. In *Systems, Man and Cybernetics (SMC) Information Assurance Workshop, 2005. Proceedings from the Sixth Annual IEEE*, June 2005.

[15] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium, pp. 135-143. Seattle, WA, July 1999*, July 1999.

[16] V. Jacobson, R. Braden, E. Lagache, and M. K. Claffy. Tcp extensions for high performance. RFC 1323, May 1992.

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

[18] K. Kato. Vmware backdoor i/o port. http://chichat.at.infoseek.co.jp/vmware/backdoor.html.

[19] R. Kennell and L. H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *12th USENIX Security Symposium*, August 2003.

[20] S. T. King and P. M. Chen. Backtracking Intrusions. In *Proceedings of the 2003 Symposium on Operating Systems Principles*, October 2003.

[21] T. Kohno, A. Broido, and K. C. Claffy. Remote physical device fingerprinting. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 211–225, Washington, DC, USA, 2005. IEEE Computer Society.

[22] C. Krügel, W. K. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX Security Symposium*, pages 255–270, 2004.

[23] T. Liston and E. Skoudis. On the cutting edge: Thwarting virtual machine detection. http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pd%f.

[24] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[25] J. Oberheide, E. Cooke, and F. Jahanian. Rethinking Antivirus: Executable Analysis in the Network Cloud. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Security (HOTSEC '07)*, Aug 2007.

[26] G. H. Project. Tracking Botnets, 2005. http://www.honeynet.org/papers/bots.

[27] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting System Emulators. In *Proceedings of 10th Information Security Conference (ISC), Lecture Notes in Computer Science, Springer Verlag.*, 2007.

[28] J. S. Robin and C. E. Irvine. Analysis of intel pentium's ability to support a secure virtual machine monitor. In *USENIX Security Symposium*, August 2000.

[29] A. Vasudevan and R. Yerraballi. Stealth breakpoints. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 381–392, Washington, DC, USA, 2005. IEEE Computer Society.

[30] C. C. Zou and R. Cunningham. Honeypot-aware advanced botnet construction and maintenance. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 199–208, Washington, DC, USA, 2006. IEEE Computer Society.