Ether: Malware Analysis via Hardware Virtualization Extensions

Artem Dinaburg*†, Paul Royal†*, Monirul Sharif*†, Wenke Lee†*
*Georgia Institute of Technology, Atlanta, GA, USA
†Damballa, Inc., Atlanta, GA, USA
{artem, paul, msharif, wenke}@{gtisc.gatech.edu, damballa.com}

ABSTRACT

Malware has become the centerpiece of most security threats on the Internet. Malware analysis is an essential technology that extracts the runtime behavior of malware, and supplies signatures to detection systems and provides evidence for recovery and cleanup. The focal point in the malware analysis battle is how to detect versus how to hide a malware analyzer from malware during runtime. State-of-the-art analyzers reside in or emulate part of the guest operating system and its underlying hardware, making them easy to detect and evade. In this paper, we propose a transparent and external approach to malware analysis, which is motivated by the intuition that for a malware analyzer to be transparent, it must not induce any side-effects that are unconditionally detectable by malware. Our analyzer, *Ether*, is based on a novel application of hardware virtualization extensions such as Intel VT, and resides completely outside of the target OS environment. Thus, there are no in-guest software components vulnerable to detection, and there are no shortcomings that arise from incomplete or inaccurate system emulation. Our experiments are based on our study of obfuscation techniques used to create 25,000 recent malware samples. The results show that *Ether* remains transparent and defeats the obfuscation tools that evade existing approaches.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Security

Keywords

Malware Analysis, Dynamic Analysis, Virtualization, Emulation, Unpacking

CCS'08, October 27–31, 2008, Alexandria, Virginia, USA.

1. INTRODUCTION

Malware–the increasingly common vehicle by which criminal organizations facilitate online crime–has become an artifact whose use intersects multiple major security threats (e.g., botnets) faced by information security practitioners. Given the financially motivated nature of these threats, methods of recovery now mandate more than just remediation: knowing what occurred after an asset became compromised is as valuable as knowing it was compromised. Concisely, independent of simple detection, there exists a pronounced need to *understand* the intentions or runtime behavior of modern malware.

Recent advances in malware analysis [17, 24, 25, 33] show promise in understanding modern malware, but before these and other approaches can be used to determine what a malware instance does or might do, the runtime behavior of that instance and/or an unobstructed view of its code must be obtained. However, malware authors are incentivized to complicate attempts at understanding the internal workings of their creations. Therefore, modern malware contain a myriad of anti-debugging, anti-instrumentation, and anti-VM techniques to stymie attempts at runtime observation [16, 42]. Similarly, techniques that use a malware instance's static code model are challenged by runtime-generated code, which often requires execution to discover.

In the obfuscation/deobfuscation game played between attackers and defenders, numerous anti-evasion techniques have been applied in the creation of robust in-guest API call tracers and automated deobfuscation tools [34, 38, 43, 47]. More recent frameworks [1, 3, 44] and their discrete components [15, 19] attempt to offer or mimic a level of transparency analogous to that of a non-instrumented OS running on physical hardware. However, given that nearly all of these approaches reside in or emulate part of the guest OS or its underlying hardware, little effort is required by a knowledgeable adversary to detect their existence and evade [26, 39].

In this paper we present a *transparent*, *external* approach to malware analysis. Our approach is motivated by the intuition that for a malware analyzer to be transparent, it must not induce any side-effects that are *unconditionally detectable* by its observation target. In formalizing this intuition, we model the structural properties and execution semantics of modern programs to derive the requirements for transparent malware analysis. An analyzer that satisfies these transparency requirements can obtain an *execution trace* of a program identical to that if it were run in an environment with no analyzer present. Approaches unable to fulfill these requirements are vulnerable to one or more *de-*

tection attacks–categorical, formal abstractions of detection techniques employed by modern malware.

Creating a *transparent* malware analyzer required us to diverge from existing approaches that employ in-guest components, API virtualization or partial or full system emulation, because none of these implementations satisfy all the transparency requirements. Based on novel application of hardware virtualization extensions such as Intel VT [6], our analyzer–called *Ether*–resides completely outside of the target OS environment– there are no in-guest software components vulnerable to detection or attack. Additionally, in contrast to other external approaches, the hardware-assisted nature of our approach implicitly avoids many shortcomings that arise from incomplete or inaccurate system emulation.

To demonstrate the efficacy of our approach we tested Ether with other academic and commercial approaches. Our testing included the analysis of specific in-the-wild malware instances that attempt to detect instrumentation and/or a virtual environment. In addition, we also surveyed over 25,000 recent malware samples to identify the distribution of obfuscation tools used in their creation; this knowledge was then used to create a synthetic sample set that represents the majority of the original corpus. The results of testing (presented in Section 5) show that *Ether* is able to remain transparent and defeat a large percentage of the obfuscation tools that evade existing approaches.

Our work represents the following contributions:

- A formal framework for describing program execution and analyzing the requirements for transparent malware analysis.
- Implementation of Ether, an *external*, *transparent* malware analyzer that operates using hardware virtualization extensions to offer both fine- (single instruction) and coarse- (system call) granularity tracing. *To motivate the use of our approach by the information security community, the GPL'ed source code for Ether is available for download at http://ether.gtisc.gatech.edu.*
- Broad-scale evaluation of current approaches using a proxy set of samples representing the majority of a recent, large malware corpus. *Copies of discrete samples referenced in this paper and the 25,000 malware sample corpus used for our survey are available to any academic or industry professional at an accredited organization.*

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 presents our model for programs and their execution, formal requirements for transparency, and abstract representations of failures in transparency that lead to detection attacks. Section 4 describes Ether's design and implementation, including an in-depth explanation of *how* hardware virtualization extensions are leveraged. Section 5 details the experiment selection process and how experimentation was performed, and provides an analysis of the results. Finally, Section 6 briefly describes future work and provides some concluding remarks.

2. RELATED WORK

Traditionally, anti-virus scanners have used simple emulation and API virtualization in their scanning engines [42]. More recent malware analysis efforts make heavy use of virtualized and emulated environments for their operation. Examples include systems derived from the BitBlaze project [3] (e.g., Polyglot [23] and Panorama [48]), Siren [22] and others. Honeypot-based projects also employ virtual environments for trapping and investigating malware [31, 36, 46].

Virtualization- or emulation-based approaches can be used to construct malware processing systems that provide a level of isolation between the guest and the host operating systems. In these systems, modifications made to the guest by an instance of malware can be quickly discarded, ensuring that each instance runs in the same sterile environment. Approaches that employ these ideas to obtain scalability include malware analysis services such as Norman Sandbox [13], CWSandbox [47] and Anubis [1].

Previous frameworks for fine-grained tracing of programs include VAMPiRE [43], BitBlaze [3] and Cobra [44]. Among these, VAMPiRE is in-guest, BitBlaze uses whole-system emulation, while Cobra traces malware at the same privilege level as itself. None of these frameworks use hardware virtualization extensions for their analysis capabilities or information gathering. Automated unpackers–common applications for fine-grained analysis–include PolyUnpack [40] and Renovo [32]. Renovo takes an out-of-guest approach, utilizing whole-system emulation for its unpacking engine. PolyUnpack uses an in-guest approach and hence runs at the same privilege level as the malware it is analyzing.

Frameworks which could be used for system call or Windows API tracing include Detours [29] and DynInst [4]. System call tracing using out-of-guest environments has been previously implemented in VMScope [30] and TTAnalyze [19], both of which are based on QEMU [20]. There are many in-guest approaches used to trace Windows API functions, which include older tools such as FileMon [5], RegMon [9], and more recently sandboxing environments such as CWSandbox and Norman Sandbox. These approaches use a combination of API hooking and/or API virtualization, which are detectable by malware running at the same privilege level [27].

3. FORMAL FOUNDATIONS

In this section, we analyze the requirements for building a dynamic *transparent* malware analysis system (i.e., one that cannot be detected and evaded by the malware being analyzed). These requirements serve as guiding principles to the design and implementation of *Ether* (Section 4). We start with a simple, abstract model for program execution and present the basic definition and theorem of transparent malware analysis. We then extend our model to consider virtual memory, privilege levels, system calls, exception handling, and execution timing that are part of realistic program execution environments.

3.1 Abstract Model of Program Execution

We model a program's execution at the machine instruction level. Since a low level instruction can access memory and CPU directly, we consider a system state as the collection of the contents of memory and CPU registers. Let M be the set of all possible memory states and C be the set of all possible CPU states. We use I to denote all possible instructions. Each instruction can be considered a machine recognizable combination of opcode and operands stored at at a particular address in memory. The low-level semantics of an instruction defines how it updates the memory and CPU state.

We model any program P as a tuple (I_P, D_P) of code and

data. Here, $I_P \subseteq I$ is the set of all instructions belonging to the program, including any dynamically generated instructions. D_P is the set of static data used by the code where each element can be considered a value stored in a particular address. Execution of a program's instruction may require access to hardware resources, especially for performing I/O. Such hardware resources are usually managed by the underlying operating system, which executes predefined service routines on behalf of specific application level instructions. We define the surrounding runtime environment E for the program P that contains any software or hardware components that provide any service necessary for P's execution. In other words, the environment E contains the operating system, underlying hardware, virtual machine monitors and external inputs.

We define a transition function $\delta_E : I \times M \times C \rightarrow I \times M \times C$ to formally represent the semantics of executing an instruction in E, which is a combination of the machine-level execution semantics and the semantics defined by components in the environment. This function defines how an instruction execution in the environment updates CPU and memory state and determines the next instruction to be executed. Since we only need instructions belonging to I_P in order to obtain an execution trace of P in E, we use another transition function $\delta_{E,I_P} : I_P \times M \times C \rightarrow I_P \times M \times C$, which takes from δ_E a projection of instructions in I_P. The trace of the program P in E is $T(P, E)$, which is an ordered set defined as $T(P, E) = (i_0, i_1, i_2, ..., i_l)$, where $\delta_{E,I_P}(i_k, m_k, c_k) = (i_{k+1}, m_{k+1}, c_{k+1})$ for $0 \leq k < l$.

3.2 Transparent Malware Analysis

Suppose P is a malware program. Consider a dynamic malware analyzer P_A whose goal is to learn about P's activities (e.g., its execution traces). In order to analyze P, P_A or at least some of its components need to reside in the underlying environment E. The malware program P, on the other hand, will try to detect the presence of P_A and change or hide its actvities to thwart analysis. Thus, we want P_A to be *transparent* or undetectable by P.

Since P_A and P share at least some resource in the runtime environment (e.g., the CPU), a *covert channel* can exist that leaks information about the presence of P_A to the malware P. To prevent such leakage, the principle of *noninterference* [21] dictates that the execution of P_A *shall not interfere* with the execution of P. Intuitively, if noninterference is achieved, P has the same execution (for the same given input) regardless of P_A.

We model an attempt by P to detect the presence of P_A in E as a boolean function $d_P(E)$. P must perform its own d_P instructions (included as part of its code I_P, i.e., $d_P \subset I_p$) to get/infer information about P_A. For example, d_p can include an instruction to query the debugging flag in E. The results of this check can be used in the malware to alter its behavior when the analyzer P_A is detected. Before P_A is enabled in E, we have $d_p(E) = 0$. Denote the environment with P_A running as A. The transparency goal is to achieve $d_P(A) = d_P(E) = 0$ for any detection method d_P. Thus, we have the following definition:

DEFINITION 1. *Assume E is a runtime environment, and A is E with malware analyzer P_A running. P_A is transparent if for any malware P containing any detection logic d_P, $d_P(A) = d_P(E) = 0$.*

The above definition provides a starting point for analyzing the requirements of a transparent malware analyzer. Since enumerating all possible instruction sequences for the detection attempt d_P is an undecidable problem, we state the transparency goals in a different and more tractable way. Without loss of generality, we assume that P will alter its execution path when it detects P_A (i.e., if $d_P(E) = 1$) because the malware author would try to hide the behavior of the malware from an analyzer. That is, at the point of detection k, $\delta_{E,I_P}(i_k, m_k, c_k) = (i'_{k+1}, m'_{k+1}, c'_{k+1})$ if $d_P(E) = 1$, i.e., P executes i'_{k+1} instead of the instruction i_{k+1} that would have executed without the presence of an analyer. Ultimately, the goal of a *transparent* malware analyzer is to extract the same execution traces from malware as if the analyzer is not present. Thus, we have the following malware analysis theorem, which inherently guarantees the above definition of transparency:

THEOREM 1. *If E is a runtime environment and A is the same environment with the addition of a malware analyzer P_A, then P_A is transparent if and only if $T(P, E) = T(P, A)$ for any malware program $P = (I_P, D_P)$.*

PROOF. In both environments E and A the same external inputs are provided to the program P since they are modeled as part of the environments. The traces of the program P in these two environments are defined as $T(P, E) = (i_{E,0}, i_{E,1}, ..., i_{E,l_E})$, and $T(P, A) = (i_{A,0}, i_{A,1}, ..., i_{A,l_A})$.

For the "only if" part of the proof, assume that P_A is transparent; we will prove by induction that $T(P, E) = T(P, A)$. The base case is trivial because the program starts execution at the same instruction i_0, so $i_{E,0} = i_{A,0} = i_0$. For the induction hypothesis, assuming $i_{E,t} = i_{A,t}$ and we need to prove that $i_{E,t+1} = i_{A,t+1}$. Since P_A is transparent, according to Definition 1, we have $d_p(A) = d_p(E) = 0$, and P will not change its execution path. Further, $d_P(A) = 0$ implies that the data/values in $m_{E,t}$, $c_{E,t}$, $m_{A,t}$, and $c_{A,t}$ that are visible to P, and hence relevant to the execution of P, must be the same (otherwise, $d_P(A) = 1$). That is, from P's point of view, the execution semantics in A and E are equivalent. Therefore, the next instruction in I_P executed in A has to be $i_{E,t+1}$ as in E (i.e., $i_{A,t+1} = i_{E,t+1}$). Using induction, we have $T(P, E) = T(P, A)$. For the "if" part of the proof, we assume that $T(P, E) = T(P, A)$ for any malware P, and we need to prove that the P_A is transparent. We prove by contradiction. Assume that P_A is not transparent. According to Definition 1, we have $d_p(A) = 1$. Therefore, without loss of generality, P will alter its execution in A, which leads to the contradiction that $T(P, E) \neq T(P, A)$. Therefore, P_A is transparent. □

3.3 Requirements for Transparency

We use Definition 1 and Theorem 1 as guidelines to formulate the requirements for the design of a transparent malware analyzer. Our discussion here uses a generalization of several common hardware and operating system features such as privilege levels, virtual memory, and exception handling, which also covers other protection features provided by hardware virtualization. We first describe these features as parts of an extension to the basic program execution semantics introduced in Section 3.1, and then discuss the requirements for transparent malware analysis.

Similar to information flow models in multi-level security systems [21], the notion of *privilege* is essential for reasoning

about how to hide these changes from P. Suppose that there are n rings of privilege where 0 is the most privileged (or "highest") level and n is the least. Let the highest privilege level gained by the program P during execution in E be denoted by $\Pi_E(P)$.

In order to represent virtual memory, suppose V denotes all possible virtual memory states viewed by instructions executed at a specific privilege level. The entire memory state M can then be defined as $M = V^n$, where each member $m \in M$ is an n-tuple of memory states, and $m[k] \in V$ is the state of the virtual memory at ring k. Virtual memory mapping can be expressed by functions $\mu^-_{E,r\to k}$, which maps memory of ring r to ring k and $\mu^+_{E,k\to r}$, which maps memory of ring k to ring r for $r > k$. By assuming that $\mu^-_{E,r\to k}$ and $\mu^+_{E,k\to r}$ are in $m[k]$, we can express how a higher privilege ring k code can control how a lower privilege ring r views its memory.

We use exceptions and exception handling to represent a broad range of system features such as all privileged instructions (e.g. system calls), I/O, memory content or CPU register protection and access violations, and program and system faults. An exception occurs when an instruction execution requires services or data at a higher privilege level k than the current level r. A function $\phi_{E,r\to k}$ specifies the first instruction of the exception handler at ring k that handles the particular exception occurring at ring r.

The instruction execution semantics δ_E introduced in Section 3.1 can be extended to include two parts. The first is δ, the low level or *basic* instruction execution semantics that do not involve exceptions, and only deals with access to virtual memory and CPU registers (note that we consider I/O as exceptions). The second is $\delta_{E,\phi}$, the semantics that deal with exceptions (e.g., control transfers to and from exception handlers residing in privileged levels).

In order to achieve transparency (i.e., $d_P(A) = 0$ and hence $T(P, E) = T(P, A)$), the memory and CPU states visible to P need to be identical in both E and A. However, the presence of P_A and its analysis activities introduce changes to these entities. Using the extended model described above, we now formulate the requirements for hiding these changes and achieving transparency.

1. Higher Privilege: We require that the analyzer P_A have higher privilege than the maximum privilege a malware instance P can gain. If the maximum privilege gained by P is $\pi = \Pi_A(P)$, then P_A should reside in privilege levels $k < \pi$. For any memory state $m \in M$, besides the code and data of P_A, the memory mapping functions $\mu^-_{A,\pi\to k}$ and $\mu^+_{A,k\to\pi}$ as well as the exception handler function $\phi_{A,\pi\to k}$ should also reside in $m[k]$. Proper isolation and protection can be achieved by ensuring $\mu^+_{A,k\to\pi}$ does not map any of these components to virtual memory state of $m[\pi]$.

2. No non-privileged side effects: This requirement states that if P_A induces side-effects, access to them should be privileged and through exception handlers at a higher privilege level(s) than P's. This ensures that any access to the changes in the memory, CPU registers, etc., can be intercepted using an exception handler that can hide these side-effects from P. Similarly, since P_A can have timing side-effects, instructions that can access any notion of time should be privileged as well.

3. Identical Basic Instruction Execution Semantics: Recall that the basic execution semantics do not involve any exception. From the second requirement above, the basic semantics do not involve any side-effects introduced by P_A (which is privileged and requires exception handling). Thus, the identical basic semantics, plus transparent exception handling (see requirement 4 below), guarantee that the same instruction has the same execution (and will lead to the same next instruction) in both A and E.

4. Transparent Exception Handling: Suppose that when the t^{th} instruction is executed (in ring π), an exception occurs and the control is transferred to $\phi_{A,\pi\to k}$ in ring $k < \pi$. First, consider the case where there was no equivalent exception handler in environment E for the same instruction $i_{E,t}$ (i.e., $i_{E,t}$ was a basic instruction in E). In this case, the handler code must first guarantee the third requirement above by executing $i_{A,t}$ with the same semantics as for $i_{E,t}$. Then, it has to guarantee that execution is returned to $i_{A,t+1}$ (the same as $i_{E,t+1}$) at the end of exception handling. In addition, the changes in $m_{A,t+1}$ and $c_{A,t+1}$ by the exception handler should only be privileged side-effects to fulfill the second requirement above. Second, if this exception handler replaces an original exception handler $\phi_{E,\pi\to k}$ in E, it needs to have identical changes made to $m_{A,t+1}$ and $c_{A,t+1}$ as $\phi_{E,\pi\to k}$ would make to $m_{E,t+1}$ and $c_{E,t+1}$ (e.g. results of system calls remain the same). The cases when the handlers involve timing measurements are addressed separately, as in fifth requirement below.

5. Identical Measurement of Time: This requirement states that the timing information received by the t^{th} instruction $i_{A,t}$ in $T(P, A)$ is the same as it were in $T(P, E)$. It is necessary because P can use timing measurement to detect the presence of P_A. For P to have a continuous false view of time it is required that A maintain a *privileged* logical clock that is adjusted when exceptions (which include any access to the clock) are handled. Although the requirement of having identical measurement of time is very difficult to fulfill in practice, it can be decomposed into smaller requirements that may be easier to satisfy in many cases. Suppose that the time spent in E to move from tth instruction to $(t + 1)$th instruction is $\Delta_{E,t}$. We can define $\Delta_{E,t} = \Delta_{E,\delta,t} + \Delta_{E,\phi,t}$ where $\Delta_{E,\delta,t}$ is the time for basic instruction execution and $\Delta_{E,\phi,t}$ is the exception handling time. Similarly, $\Delta_{A,t} = \Delta_{A,\delta,t} + \Delta_{A,\phi,t}$. A needs to ensure $\Delta_{A,t} = \Delta_{E,t}$ by making some adjustments $\Delta'_{A,t}$, where $\Delta_{E,t} = \Delta_{A,t} - \Delta'_{A,t}$. Therefore, we have $\Delta'_{A,t} = \Delta_{A,\delta,t} + \Delta_{A,\phi,t} - \Delta_{E,\delta,t} - \Delta_{E,\phi,t}$. If the basic instruction execution requires the same amount of time in both A and E, we have $\Delta'_{A,t} = \Delta_{A,\phi,t} - \Delta_{E,\phi,t}$. Thus, when no exceptions occur for both E and A no adjustment in time is required. When the an exception occurs for A but not for E, which is usually due to having privileged side-effects, $\Delta_{A,\phi,t}$, the time spent by the exception handler ϕ_A, has to be determined and negated. When both environments have exceptions, if ϕ_A is essentially ϕ_E plus some extra activities, then the extra time (i.e., $\Delta'_{A,t}$) can be measured and negated because the activities belonging to ϕ_E are executed and can be timed. However, if ϕ_A replaces ϕ_E (i.e., implements different activities), then it is very difficult to measure $\Delta'_{A,t}$ because ϕ_E is not executed.

3.4 Fulfilling the Requirements

We will now use the requirements presented in Section 3.3 to analyze the transparency achievable by various malware analysis approaches. In particular, we describe which trans-

parency requirements the reduced privilege guest and full system emulation based approaches cannot satisfy, and discuss how hardware virtualization extensions in the x86 architecture can overcome these limitations.

Previous malware analysis approaches employ user level or kernel level counterparts residing in the host in which malware is analyzed; these include VAMPiRE and CWSandbox. Since malware that use rootkit components can gain kernel level privileges, these approaches cannot satisfy the first requirement of transparency.

Reduced privilege guest-based virtualization approaches (e.g., VMware [12] and VirtualPC [11] for x86) can fulfill the first requirement by emulating a few sensitive instructions in order to gain higher privilege over the OS kernel in the virtual machine. Therefore, the second requirement is partially satisfied by these approaches as they can remove certain memory and CPU side-effects by providing a virtual view of memory. However, these approaches are not designed with transparency in mind, and the communication medium between the guest and host operating systems introduces unprivileged side-effects. Moreover, instructions that can access time are not privileged, making these side-effects visible in such systems through time measurement. In contrast, full system emulators (e.g. QEMU) emulate the entire low level instruction execution semantics δ to gain privilege over the guest OS. They have privilege over all instructions executed, thereby fulfilling the second requirement.

An analyzer based on hardware virtualization extensions can likewise satisfy the first and second requirements. The first requirement is satisfied because the analyzer can reside in a domain more privileged than the guest. This privilege is enforced in hardware by the analyzer residing in *ring -1*, which has higher privilege than rings 0 to 3. In addition, the contents of the analyzer's domain are completely isolated through the use of shadow page tables. Hardware virtualization extensions not only enable basic memory protections, but also offer privileged access to sensitive CPU registers and instructions including instructions that access time, such as `RDTSC`. A malware analyzer based on these extensions can therefore intercept and hide these side-effects from malware.

Neither reduced privilege guest-based approaches nor full system emulators can guarantee the third requirement. To elaborate, emulation-based approaches use low-level instruction execution semantics function δ' to simulate the entire low level execution semantics of δ. For reduced privilege guest-based approaches, δ' partially simulates δ. The low level instruction execution semantics of δ can be easily shown to be Turing complete. Likewise, δ' is also Turing complete. In addition, determining whether δ' is equivalent to δ requires determining whether all programs exhibit the same behavior under δ' and δ.

In automata theory, the above problem would be formally represented as the problem of determining whether the language of two Turing machines (L and L') are equal; it is otherwise known as the undecidable problem EQ_{TM} [41]. In practice, there have been attacks that detect full system emulator and reduced privilege guest-based approaches by exploiting incomplete emulation [27]. There is no way to guarantee the absence of such attacks. In contrast, hardware virtualization extensions rely on the same hardware execution semantics δ, thereby guaranteeing that the third requirement is satisfied.

The fourth requirement is an analyzer design issue and can be satisfied by all approaches, with careful design.

Finally, although emulators can have privileged access over instructions that can access the notion of time, it is difficult to provide a notion of time that is equivalent to the environment E. To elaborate, in emulators, almost all instructions have exception handlers managing their execution, and the identification of $\Delta_{E,t}$ is hard without having a real system execute these instructions in parallel. Moreover, the determination of $\Delta_{A,t}$ requires a cycle-count accurate execution simulator, which keeps track of the number of cycles required to execute an instruction in a real system.

In contrast, for hardware virtualization extensions-based approaches, the side effect on time is privileged because the instructions that access time (e.g., `RDTSC`) are privileged. As we describe next in Section 4, hardware virtualization extensions maintain a separate execution cycle count in the hypervisor, which allows an analyzer to adjust a cycle value before it is given to the guest. As such, while there still exist complex situations that are hard to satisfy, hardware virtualization extension-based approaches go a long way in satisfying the fifth requirement.

4. IMPLEMENTATION

In this section we describe Ether's architecture, the low-level details of how it performs instruction and system call tracing, and the efforts necessary to ensure transparency in accordance with the requirements for transparency. In addition, we describe implementation challenges and current architectural limitations that prevent full transparency.

4.1 Environment

To create Ether, we needed an analysis mechanism that was readily available to researchers and would allow for transparency. We deemed hardware virtualization extensions as the most appropriate, as they do not interfere with the original instruction stream I_P, the CPU registers C, the memory state M, the original exception handlers, or the original CPU transition function δ. In addition to this transparency, processors with such extensions are inexpensive and widely available.

Among available software that can utilize hardware virtualization extensions, we chose the Xen hypervisor [18] version 3.1.0 as the base for implementing Ether. Xen was chosen because it is a mature product, it is open source, and it has existing communication mechanisms that could be leveraged in Ether's implementation. Finally, our work could also be incorporated into the numerous research projects currently supporting Xen.

Among hardware virtualization platforms we selected Intel VT due to the available documentation, our familiarity with Intel processors, and the availability of Intel-based hardware. Finally, as the selection of the target operating system must well represent the actual install base, we chose Windows XP (Service Pack 2). Windows XP is the most common PC operating system in use today and therefore a preferred target of modern malware.

4.1.1 A brief overview of Xen

The Xen hypervisor is software that runs at the lowest and most privileged layer of the system. This layer presides over multiple operating systems (OSes), known as domains, with one domain having special privileges. The privileged

domain is referred to as domain 0; it serves as the administrative center of a Xen system. Domain 0 is the first domain started and the only domain with direct access to real hardware. The guest OSes, also referred to as domUs, rely on the hypervisor for privileged operations. The domUs may be commodity, unmodified operating systems. Ether runs as a component in the hypervisor layer, and as a userspace component in domain 0. The analysis target, Windows XP with Service Pack 2, runs in a domU.

4.1.2 A brief overview of Intel VT hardware Virtualization Extensions

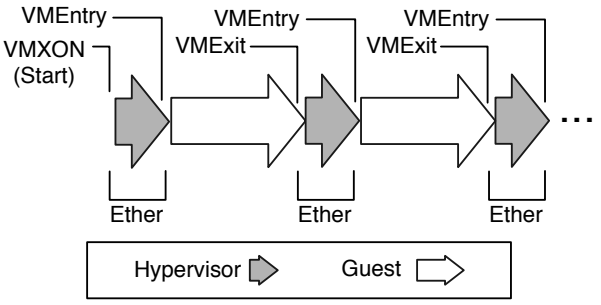

Figure 1: Processor operation for Intel VT. Ether operates between VMExits and VMEntries.

Intel VT hardware virtualization extensions are a set of instructions added to Intel processors to facilitate the virtualization of the x86 instruction set. These instructions enable two new processor modes, called VMX root mode and VMX non-root mode. The Xen hypervisor, and hence Ether, runs in VMX root mode. The domUs or guests, which we refer to as the *analysis targets*, run in VMX non-root mode.

Events called VM transitions change operation between the two modes. There are two different transitions: VMEntry and VMExit. A VMExit will transition from VMX non-root mode to VMX root mode, and a VMEntry will transition from VMX root mode to VMX non-root mode. Certain events in VMX non-root mode automatically cause a VMExit; these include certain exceptions, changes to the page directory entry pointer, and page faults.

Ether obtains control from the analysis target on VMExits and performs a VMEntry when it chooses to resume execution of the analysis target (i.e., the guest). After a VMExit, but before the next VMEntry, the guest is in a completely dormant state. An overview of this process is shown in Figure 1.

4.2 Analyzer Architecture

As shown in Figure 2, the architecture of Ether consists of a hypervisor component and a userspace component running in domain 0. Ether's hypervisor component is responsible for detecting events in the analysis target. Currently, such events include system call execution, instruction execution, memory writes, and context switches.

Ether's userspace component acts as a controller regulating which processes and events in the guest should be monitored. This component also contains logic to derive semantic information from analyzed events, such as translating a system call number into system call name or displaying system call argument content based on argument data type.

The analysis target consists of a Xen domU running Windows XP Service Pack 2. The only change we made to the default installation of Windows XP was disabling PAE and large memory pages. These modifications exist solely to make memory write detection easier in the initial Ether implementation and are not a limitation of our approach.

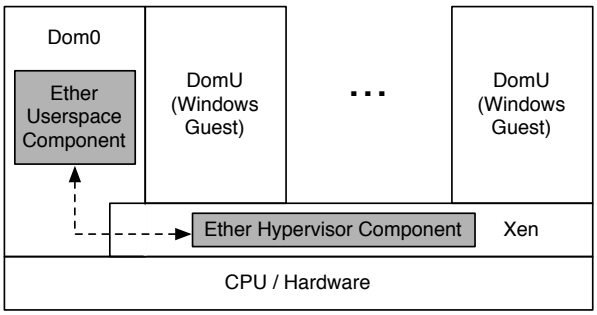

Figure 2: Ether's system architecture.

4.3 Using Intel VT for Malware Analysis

To present Ether as a full-featured malware analyzer we required that it be able to monitor the instructions executed by a guest process, any memory writes a guest process performs, and any system calls a guest process makes. We chose these low- and high-level operations due to their usefulness in malware analysis and their ability to demonstrate the efficacy of Ether performing both coarse- and fine-grained tracing. The challenges to successful implementation included using a mechanism that was not intended for malware analysis (i.e., Intel VT) while maintaining the original level of transparency provided by hardware virtualization extensions. Given that Intel VT extensions do not provide explicit support for any of these monitoring activities we performed an in-depth investigation of Intel VT to create novel ways that fulfill our monitoring requirements. Below we first describe how Ether uses VT extensions to analyze malware and then how it can maintain transparency.

4.3.1 Monitoring Instruction Execution

Instruction execution monitoring relies on Ether's privilege over the analysis target in handling debug exceptions, and in guaranteeing a debug exception occurs after the execution of every instruction. Ether guarantees the occurrence of a debug exception after every instruction by setting a flag called the trap flag in the analysis target. Upon handling a debug exception caused by the forced trap flag, Ether will once again set the trap flag for the next instruction, thereby inducing a debug exception after every instruction. Ultimate control of which exceptions reach the analysis target rests in Ether, so all induced debug exceptions are hidden from the analysis target. This control allows Ether to execute the target process one instruction at a time while preventing it from detecting Ether's presence. This form of instruction stepping via the trap flag was first implemented in VAMPiRE, even though its approach is in-guest and hence vulnerable to in-guest detection attacks.

4.3.2 Monitoring Memory Writes

Ether monitors memory writes by using shadow page tables and privilege over the guest in handling page faults.

Ether induces a page fault at every attempted analysis target memory write, traps the fault, and prevents it from reaching the guest. Faults occurring due to normal guest operation are forwarded to the guest. In this manner all memory write attempts are transparently intercepted.

Shadow page tables refer to the actual page tables the hardware uses for address translation, as the guest is never permitted to do its own translation. The hypervisor is responsible for synchronizing shadow page tables with the guest's page table contents, as well as ensuring the guest can only map memory explicitly allocated for it.

Ether causes page faults on write attempts by removing writable permissions from shadow page table entries. When Ether detects a fault caused by itself, the Ether userspace component is notified of an attempted memory write, and the fault is then hidden from the analysis target. Faults resulting from normal guest operation are passed along. After notification, the faulting page is set to writable in the shadow page table and the faulting instruction is re-executed. Upon completion of the instruction, all pages are once again marked read-only in preparation for the next instruction.

4.3.3 Monitoring System Call Execution

Ether's novel system call execution monitoring exploits features of the x86 fast system call entry mechanism to inform Ether of system calls executed by the analysis target. The system call interception mechanism uses a special register present on all modern x86 processors to cause a page fault at a chosen address during system call invocation. A fault at this address will then signal to Ether that a system call was executed in the analysis target. In addition to tracing system calls executed via the fast call entry mechanism, Ether can trace system calls made via deprecated methods as well. Semantic information such as system call arguments and return value may also be obtained by Ether.

To properly describe Ether's system call tracing technique, some background on the fast system call entry mechanism of x86 processors is required. This method is used for system calls on all Windows versions starting with XP, and on Linux kernels starting with 2.6. The fast system call entry mechanism uses the `SYSENTER` instruction to raise privilege and jump to a pre-defined address in the kernel. This address is stored in a special register called `SYSENTER_EIP_MSR`, access to which is only permitted from kernel mode. Whenever a userspace application requires system services, it specifies the service number and parameters in an implementation dependent manner, and then executes `SYSENTER`. `SYSENTER` changes the privilege mode to kernel mode, the stack pointer to the kernel mode stack, and the instruction pointer to the value in `SYSENTER_EIP_MSR`.

Ether sets the value of the analysis target's `SYSENTER_EIP_MSR` to a chosen value on a page guaranteed to be not present, and stores the original value in Ether's memory. Whenever the analysis target attempts system call execution, a fetch page fault will occur at the chosen address. When Ether encounters such a fault in the analysis target, it indicates a system call was attempted. The userspace component of Ether is notified of system call execution, and the instruction pointer of the analysis target is reset to the expected value of `SYSENTER_EIP_MSR`. Execution of the analysis target resumes as if `SYSENTER` jumped directly to the expected address, instead of our chosen address.

The arguments, return address, and return value of a system call may be gathered from the guest by a series of guest memory reads. The exact location of the arguments, return value, and return address is guest operating system dependent.

System calls executed via deprecated methods can also be monitored by Ether. While all operating system provided libraries in Windows XP will use the `SYSENTER` instruction to execute system calls, support for system call execution via software interrupt 0x2E is still present in the kernel. To trace system calls executed by this older method, Ether can change the interrupt descriptor table entry of interrupt 0x2E to point to a non-present page. A VMExit caused by a page fault at that address would indicate system call execution via `INT 2E`. Alternatively, AMD's hardware virtualization solution provides a direct means to cause VMExits on software interrupts [14].

4.3.4 Limiting Scope to a Chosen Process

Ether can limit analysis scope to a chosen process in the analysis target. Processes are identified for monitoring during a context switch in the analyzed environment. When a context switch occurs, Ether gains control and uses virtual machine introspection [28] to obtain the upcoming process name from the analysis target. If the process matches a target specified to Ether, analysis is enabled. In the current implementation, the target process may either be specified by name or by the process page directory entry. Enabling process monitoring exactly when a process is about to run guarantees Ether does not miss any events executed by the process.

Ether gains control on every context switch by leveraging a feature of Intel VT that causes a VMExit every time the page directory entry pointer is accessed in the guest OS. As the page directory pointer must be updated every context switch to change address spaces, it is guaranteed that Ether will detect all guest context switches.

4.4 Maintaining Analyzer Transparency

Despite making several modifications to the guest, Ether maintains transparency of the analyzer by ensuring such changes are undetectable. Other changes made by Ether, such as those to shadow page tables, are changes outside of the analysis target and therefore transparent. Transparency of the analyzer does not imply hiding the presence of a virtualized environment, but hiding the presence of Ether in a virtualized environment.

4.4.1 Hiding the Trap Flag

Ether is able to conceal that it has the set trap flag in the guest by intercepting the few instructions that can read this flag, and altering their behavior to hide the flag's presence. The only instruction which can directly read the presence of the trap flag is `PUSHF`. Ether intercepts this instruction and changes its result to provide the environment-expected state of the flag. Besides `PUSHF`, the `INT` instruction reads the value of the flag indirectly. We monitor this instruction as well, and fix the flags value pushed on the stack to match the value expected by the analysis target.

At times the analysis target has set the trap flag and expects to receive debug exceptions. Ether detects the setting of the trap flag by the `POPF` instruction, and when the analysis target expects the trap flag to be set, Ether forwards debug exceptions along as appropriate. Any debug excep-

tions not caused by the trap flag are automatically forwarded to the analysis target as they are not caused by Ether.

4.4.2 Page Table Modifications

The page table modifications made by Ether are to shadow page tables, and not the page tables stored in the guest. Therefore, the guest is not aware that the shadow page tables exist, and hence cannot detect their modification or presence. Ether can determine which faults were caused by normal guest operation, and which faults it purposely created. Any faults caused by normal operation are forwarded to the guest. Conversely, any faults caused by Ether are handled by Ether and never passed along.

4.4.3 SYSENTER_EIP_MSR

Ether mediates all access to the `SYSENTER_EIP_MSR` register and can therefore conceal any modifications of the register from the analysis target. Complete mediation is achieved because any access of the register automatically causes a VMExit. Ether saves any value the analysis target attempts to write into `SYSENTER_EIP_MSR`, and uses this value any time the guest-expected value of this register is required. Although technically modified, as observed from the analysis target the CPU state C is unchanged.

4.5 Potential Attacks

While theoretically resilient against in-guest detection attacks, current architectural restrictions make some of these attacks possible, and Ether is also vulnerable to a class of timing attacks using external timing sources.

4.5.1 Attack Classes

Classic in-guest detection attacks cannot detect Ether due to it being completely outside of the analysis target. Also, none of the few modifications Ether makes to the guest environment are unconditionally detectable. Ether is, however, vulnerable to a certain class of timing attacks, and in the current implementation, memory hierarchy attacks. Detection methods and our mitigation of them are outlined below.

In-Memory Presence: Traditional detection attacks which rely on detecting the presence of an analyzer in memory will always fail against Ether, as it has no in-guest memory presence.

CPU Registers: Ether hides the few changes it makes in CPU state from the analysis target so that it is unable to detect deviation from a native hardware environment.

Memory Protection: Ether modifies only the shadow page tables, which are inaccessible to the analysis target. That is, the analysis target is unable to detect changes to shadow memory permissions. However, in the current implementation, Ether does indirectly modify the memory hierarchy (the cache and the TLB). This is due to an architectural limitation; further details are provided in Section 4.6.

Privileged Instruction Handling: Ether uses built-in hardware mechanisms to intercept only certain privileged instructions and exceptions and as necessary, forwards these exceptions to the guest. From the viewpoint of the guest, no handler is ever modified, and privileged instructions have the same effects as in a native environment.

Instruction Emulation: Ether executes all instructions on the actual processor. Therefore, Ether does not suffer from emulation inaccuracies inherent in full system x86 emulators; the transition function δ remains unmodified.

Timing Attacks: As described in Section 3.3, there are two fundamental issues with avoiding timing attacks: controlling queries about time, and answering those queries with the expected time value. Ether controls the in-guest view of the `RDTSC` instruction, the APIC timer, the 8254 timer chip, as well as any periodic time-based interrupts and other guest time queries. Section 3.3 outlines the requirements for a correct reply to a guest time query. This correct reply, $\Delta_{E,t}$, is the wall-clock time when Ether is present, $\Delta_{A,t}$, reduced by $\Delta'_{A,t}$, the amount of overhead added by Ether's presence. $\Delta'_{A,t}$ is equal to the time spent in the analyzer minus the time spent in the guest exception handler. Since Ether always calls the native exception handlers, if any, for any exceptions the guest must process, $\Delta'_{A,\phi,t}$ consists of the time spent switching privilege to Ether, the time spent in Ether's handler, and the time to switch to the original guest handler. The time spent in the handler is easily measured using the wall-clock and times for privilege changes, and can be calculated empirically. Therefore, Ether would be able to adjust the time returned to the guest by the amount of overhead, and remain untraceable by timing attacks which rely on host-based time sources.

Additionally, the Intel VT architecture has a special field, called the `TSC_OFFSET`, which the processor will automatically add to any queries of the time stamp counter. This value must be set to the calculated value of $\Delta'_{A,t}$. Any other clock sources, all of which are under control of Ether, can be adjusted accordingly.

As Ether can only mitigate in-guest timing detection attacks, a class of detection attacks utilizing time from external inputs is still possible. However, preventing such attacks would require detecting covert channels over a shared medium, which is known to be undecidable [45].

Attacks which detect virtualization itself, and not the presence of Ether, will cease to be relevant in the future. As virtualization spreads to home computers and enterprise desktops, detecting virtualization will no longer be indicative of detecting an analysis environment. As a consequence, any malware which would not run in virtualized environments would preclude itself from infecting an increasing percentage of legitimate targets.

4.6 Architectural Limitations

Intel VT suffers from some architectural limitations which may allow Ether to be detected under certain circumstances. Different hardware virtualization extensions exist that do not suffer from such limitations.

Intel VT suffers from two main flaws which allow the current implementation to be detected by observing implicit changes to the memory hierarchy. The first flaw is that Intel VT flushes the TLB on every VMExit. A detection method such as the one proposed by [37] can detect this implicit change. A hypervisor could run in non-paged mode with caches off in an effort to avoid using the extended memory hierarchy at all. Intel VT's second flaw is that paging mode must be turned on before entering VMX Root Mode. AMD's competing virtualization solution, AMD-V, does not suffer from either of these limitations. A hypervisor which did not use paging and ran with caches off would be undetectable by these memory hierarchy detection methods.

5. EXPERIMENTS AND EVALUATION

In this section, we describe two tools based on Ether: EtherUnpack and EtherTrace. EtherUnpack traces memory writes and single instructions (i.e., fine-grained tracing), while EtherTrace traces system calls (i.e., coarse-grained tracing). We use these tools to evaluate Ether and compare it against current approaches.

5.1 Fine-Grained Tracing: Unpacking

To measure the effectiveness of fine-grained tracing on real malware, we used Ether to create a generic, automated unpacker called EtherUnpack. Before presenting the results of experiments using EtherUnpack, we describe why an unpacker is a ideal test for Ether's resiliency against current malware threats when performing fine-grained tracing.

5.1.1 Methodology

Packing is a term used to describe the obfuscation and encryption of program code to thwart static analysis. The result of packing is that signature-based approaches fail to identify packed malware as malicious. Opposite to packers, *unpackers* are programs which attempt to obtain the original code hidden by the packer.

To prevent unpacking, many current packers also attempt to thwart dynamic analysis by using anti-debugging and anti-VM techniques. Therefore, an Ether-based unpacker serves as an excellent testbed for the framework, since packers will attempt to expose any lack of transparency exhibited by Ether's approach or implementation. In addition, given the fine-grained nature of automated unpacking, packers are likely to have the greatest chance of exposing Ether's potential weaknesses.

5.1.2 Packer Use in Current Malware

To evaluate EtherUnpack against current packed malware, we obtained a recent malware corpus. Samples were collected between January and March 2008 from honeypots, mail filters, proxy monitors, web crawling, file sharing networks, and other sources. To classify the samples, we surveyed them using PEiD [7], a signature-based packer detector, and a PEiD signature database from SANS ISC [8]. The resulting set consisted of 25,118 malware instances, unique according to MD5 value. The distribution of packers used to obfuscate packed malware in the corpus appears in Figure 3.

Of note, the *Other* section in Figure 3 represents a multitude of packers, where each comprises less than one percent of the total collection.

5.1.3 Effectiveness

As used in Renovo, we define unpack-execution to occur when a process executes memory it previously wrote. Detection is performed by monitoring memory writes and executed instructions. Whenever a program's execution goes to an area of memory previously written, EtherUnpack flags that area as dynamically generated code and extracts it. Note that some samples contain multiple packing layers, which may require Ether to perform additional, subsequent unpacking even after it first detected dynamically generated code.

The samples used for unpacking in the above comparison were the exact same set evaluated in Renovo. EtherUnpack was able to recover hidden code from all tested samples, which also represent the vast majority of packers from our obfuscation tool survey. In contrast, PolyUnpack failed on many of the samples due to being detected by various in-guest detection techniques (e.g, detecting that an API had been hooked). Renovo could not unpack the Obsidium and Armadillo samples due to incorrect processor emulation, while its reason for failing to unpack Themida with VM is unclear.

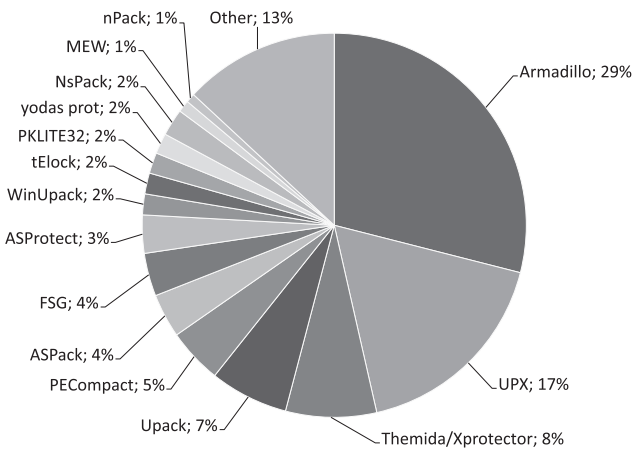

Figure 3: Distribution of obfuscation of tools used to transform malware.

To determine whether EtherUnpack successfully processed packed samples, we compared the unpacked layers to original program code. Specifically, we checked the unpacked layers for an identifying string. The string consists of 32 bytes starting at a fixed offset in the original program code. This offset was not chosen at random. We selected code that was guaranteed to always be executed, but that did not contain any Windows API calls. Certain packers will replace calls to Windows API functions with new code having identical functionality – the original code will never be executed by the obfuscated program. Before testing, we verified that the search string does not appear in the packed test binaries.

To compare EtherUnpack against current approaches, we tested it alongside two other automated unpacking tools. These were PolyUnpack, a generic in-guest unpacking tool and Renovo, an external unpacker based on the BitBlaze Binary Analysis Platform. The results of our testing are shown in Table 1.

5.1.4 Performance

Ether's fine-grained tracing, on which EtherUnpack is based, is not meant to be used for real-time analysis. While tracing an application one instruction at a time makes Ether extremely accurate, such tracing induces a significant performance penalty. Ether's fine-grained tracing is most useful as a forensic tool.

5.1.5 Emulator Resistant Malware

Malware authors, aware that emulated environments are used to analyze their creations, have begun incorporating anti-VM techniques into malware to evade emulated environments. Examples include Armadillo [2] and Themida [10], which are highly emulation and virtualization resistant. Subsequent to experimentation with EtherTrace, we performed a cursory examination of our malware corpus to identify ad-

Packing Tool	EtherUnpack	Renovo	PolyUnpack
Armadillo	yes	no	no
Aspack	yes	yes	no
Asprotect	yes	yes	yes
FSG	yes	yes	yes
MEW	yes	yes	yes
Molebox	yes	yes	no
Morphine	yes	yes	yes
Obsidium	yes	no	no
PECompact	yes	yes	no
Themida	yes	yes	no
Themida VM	yes	no	no
UPX	yes	yes	yes
UPX S	yes	yes	yes
WinUPack	yes	yes	no
yoda's Prot.	yes	yes	no

Table 1: Effectiveness of generic unpackers.

ditional samples which would not run in QEMU. We selected QEMU because many novel, external malware analysis tools (e.g., Anubis, Renovo, and Panorama) are based on QEMU or one of its derivatives.

All of our testing was done on QEMU version 0.9.1, the latest stable release as of this writing. In testing, we found least three different categories of samples that failed to execute properly in QEMU. In addition, an analysis of emulation detection techniques present in these emulator-resistant samples reveals all exploit incorrect CPU emulation.

The first category contained any sample packed with the tElock tool; we identified 442 such samples in our corpus. These samples comprise well known malware, such as RBot, SDBot and the Spy.Banker trojan. Binary analysis showed that tElock uses an undocumented opcode, `F1`, which causes interrupt vector 1 to be issued by the processor on normal hardware. However, in QEMU, this instruction had the effect of freezing the entire guest OS, rendering the guest unusable even when tElock is executed from guest userspace.

The second category was represented by a new, in-the-wild sample provided to us by CERT-LEXSI. As outlined in [35], the CERT-LEXSI sample exploits the difference between real hardware and emulated hardware values of reserved FPU Control Word bits. Lastly, we found a version of PCPrivacyTool (a fake anti-spyware program) that failed to operate correctly. The PCPrivacyTool sample encountered an invalid memory access while executing the `LDDQU` instruction in QEMU; the same instruction executes without issue on a real CPU.

Other currently non-utilized methods exist to reliably detect QEMU. To demonstrate, we created a synthetic QEMU detection method that relies on improper emulation. Our detector uses the multi `REP` prefix detection method outlined in [26]. The detection relies on placing 15 `REP` prefixes before a single-byte instruction. This configuration makes the total instruction length 16 bytes – illegal on x86 where the maximum instruction length is 15 bytes. On real hardware, an illegal instruction exception is generated by the CPU. QEMU does not generate such an exception. Even though public release of this detection occurred in late 2006, the issue has remained unresolved and the method still reliably detects QEMU. The source code for our QEMU detector appears in Appendix A.

In conclusion, EtherUnpack was able to reveal hidden code

Packing Tool	EtherTrace	Anubis	Norman
None	yes	yes	yes
Armadillo	yes	no	no
UPX	yes	yes	yes
Upack	yes	yes	yes
Themida	yes	yes	yes
PECompact	yes	yes	yes
ASPack	yes	yes	yes
FSG	yes	yes	yes
ASProtect	yes	no	yes
WinUpack	yes	yes	yes
tElock	yes	no	yes
PKLITE32	yes	yes	yes
yoda's Prot.	yes	yes	no
NsPack	yes	yes	yes
MEW	yes	yes	yes
nPack	yes	yes	yes
RLPack	yes	yes	yes
RCryptor	yes	yes	yes

Table 2: Effectiveness of sandboxing environments.

from tElock, the CERT-LEXSI sample, Armadillo, Obsidium, and Themida because it does not rely on correct CPU emulation, and instead utilizes native hardware for instruction execution. Ether was also able to trace the PCPrivacyTool and our synthetic QEMU detection sample without issue. Since it is inherently impossible to ensure the equivalence of an emulated processor to a real processor, more emulation inconsistencies are likely present in QEMU. Therefore, as an alternative to QEMU, we propose the use of hardware virtualization-based approaches such as Ether.

5.2 Coarse-Grained Tracing: System Calls

In contrast to automated unpacking, system call tracing represents a more coarse-grained type of tracing, wherein discrete system calls represent easily identifiable actions such as file and registry access, process and thread creation, and network activity. System call behavior is useful to malware analysis because it can be used to identify malware startup mechanisms, command and control channels, and access to or theft of sensitive information. To evaluate Ether's ability to perform coarse-grained tracing, we created EtherTrace, a tool for tracing Windows native API functions, which are the Windows equivalent of Unix system calls.

5.2.1 *Effectiveness*

To assess the completeness of EtherTrace's functionality we created a synthetic sample that performs a set of file and registry operations. These operations were chosen because of their direct mapping to the Windows native API; that is, they could be used to easily confirm EtherTrace's ability to perform successful tracing. To determine how EtherTrace would perform against modern malware, we packed our synthetic sample with the 15 most popular obfuscation tools identified from our malware survey. We compared EtherTrace against Anubis and Norman Sandbox; the results appear in Table 2.

To confirm that EtherTrace successfully traced a given sample, we ran the sample in the guest and inspected the trace logs that were generated. For testing Anubis and Norman Sandbox, we uploaded the samples to their web submission forms and examined the output. In examining the

Benchmark	Untraced	Traced	Change
HTMLRender	0.97 pg/s	0.62 pg/s	35.95%
FileDecrypt	64.87 MB/s	64.09 MB/s	1.19%
HDD	11.34 MB/s	8.14 MB/s	28.29%
TextEdit	68.83 pg/s	19.37 pg/s	71.86%
Image	14.86 MPix/s	14.69 MPix/s	1.12%
FileCompress	2.7 MB/s	2.66 MB/s	1.67%
FileEncrypt	16.07 MB/s	15.53 MB/s	3.39%
VirusScan	11.14 MB/s	11.09 MB/s	0.41%
MemLatency	7.75 MAcc/s	3.12 MAcc/s	59.8%
RARTime	40.87 s	45.29 s	10.81%

Table 3: Performance of EtherTrace on application benchmarks.

results, all samples performed either all or none of the expected file and registry operations.

The results indicate once again that Armadillo, which is quite popular in current malware, provides strong anti-analysis protections and detected both Anubis and Norman Sandbox. Besides Armadillo, Anubis failed to trace tElock, which crashed after reporting failure of an internal CRC check. Reasons why Anubis failed to trace ASProtect or why Norman Sandbox failed to trace yoda's Protector are unclear. In contrast to both Anubis and Norman, EtherTrace successfully traced all samples.

5.2.2 Performance

As a final experiment, we measured the performance of EtherTrace's system call tracing where return values are not needed. This type of tracing can serve as input to a wide variety of applications, ranging from host based intrusion detection systems to file and registry access monitors. To perform our tests, we selected two tools: PCMark '05, an industry standard benchmarking application and WinRAR 3.71. Using PCMark '05, we performed a standard series of tests provided by the application. To test using WinRAR, we compressed every file in the Program Files directory of a default Windows XP installation; we selected this benchmark due to its mix of system calls, I/O, and CPU Utilization. The results of testing appear in Table 3.

Tracing, as expected, adds extra latency to system calls. Benchmarks which are sensitive to latency, such as web browsing, incur a higher performance penalty. However, the majority of this latency is due to notification of the Ether userspace component; a full in-hypervisor implementation would have much lower latencies. In addition, even in the current implementation, system calls which require I/O access are relatively unaffected by the extra latency.

6. CONCLUSION

In this paper we have presented *Ether*, a transparent and external malware analyzer that is based on hardware virtualization extensions such as Intel VT. *Ether* does not induce any unconditionally detectable side-effects by completely residing outside of the target OS environment. As a result, malware cannot detect the presence of *Ether*. In our experiments, we evaluated *Ether* and several other state-of-the-art analyzers on the obfuscation techniques used to obfuscate 25,000 recent malware samples. The results show that *Ether* remains transparent and defeats the obfuscation tools that evade the existing approaches.

In future work, we will focus on improving resistance to timing attacks and memory hierarchy detection attacks.

7. ACKNOWLEDGMENTS

The authors would like to thank Jonathon Giffin and Jon Larimer their insightful feedback. Additional thanks go to Robert Edmonds for his assistance in performing the malware survey and CERT-LEXSI for providing us with the in-the-wild malware sample that checks for the presence of emulated hardware.

This material is based upon work supported in part by the National Science Foundation under Grant No. 0716570 and the Department of Homeland Security under Contract No. FA8750-08-2-0141. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and the Department of Homeland Security.

8. REFERENCES

[1] Anubis: Analyzing Unknown Binaries. http://anubis.seclab.tuwien.ac.at.
[2] Armadillo. http://www.siliconrealms.com.
[3] BitBlaze Binary Analysis Platform. http://bitblaze.cs.berkeley.edu.
[4] DYNINST API. http://www.dyninst.org.
[5] FileMon for Windows. http://technet.microsoft.com/en-us/sysinternals/bb896642.aspx.
[6] Intel Virtualization Technology. http://www.intel.com/technology/virtualization.
[7] PEiD. http://www.peid.info.
[8] PEiDSO. http://handlers.sans.org/jclausing/userdb.txt.
[9] RegMon for Windows. http://technet.microsoft.com/en-us/sysinternals/bb896652.aspx.
[10] Themida. http://www.oreans.com/themida.php.
[11] VirtualPC. http://www.microsoft.com/windows/products/winfamily/virtualpc/.
[12] VMWare. http://www.vmware.com.
[13] Norman Sandbox Whitepaper. http://www.norman.com/documents/wp_sandbox.pdf, 2003.
[14] AMD64 Architecture Programmer's Manual, Volume 2: System Programming, 2007.
[15] TEMU: The BitBlaze Dynamic Analysis Component. http://bitblaze.cs.berkeley.edu/temu.html, 2007.
[16] P. Bacher, T. Holz, M. Kotter, and G. Wicherski. Know your enemy: Tracking botnets. http://www.honeynet.org/papers/bots, 2005.
[17] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated Classification and Analysis of Internet Malware. In *RAID*, 2007.
[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP*, pages 164–177, 2003.
[19] U. Bayer, C. Kruegel, and E. Kirda. TTanalyze: A Tool for Analyzing Malware. In *EICAR*, pages 180–192, 2006.
[20] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *ATEC*, pages 41–41, 2005.

[21] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, 2003.

[22] K. Borders, X. Zhao, and A. Prakash. Siren: Catching Evasive Malware (Short Paper). In *S&P (Oakland)*, pages 78–85, 2006.

[23] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis. In *CCS*, 2007.

[24] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-Aware Malware Detection. In *S&P (Oakland)*, pages 32–46, 2005.

[25] M. Christodorescu, C. Kruegel, and S. Jha. Mining Specifications of Malicious Behavior. In *ESEC/FSE*, pages 5–14, 2007.

[26] P. Ferrie. Attacks on Virtual Machine Emulators. Symantec Advanced Threat Research, 2006.

[27] P. Ferrie. Attacks on More Virtual Machines. http://pferrie.tripod.com/papers/attacks2.pdf, 2007.

[28] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *NDSS*, 2003.

[29] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *WINSYM*, pages 135–143, 1999.

[30] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. In *CCS*, pages 128–138, 2007.

[31] X. Jiang, D. Xu, H. J. Wang, and E. H. Spafford. Virtual Playgrounds for Worm Behavior Investigation. In *RAID*, pages 1–21, 2005.

[32] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A Hidden Code Extractor for Packed Executables. In *WORM*, 2007.

[33] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *ACSAC*, pages 91–100, 2004.

[34] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *ACSAC*, pages 431–441, 2007.

[35] F. Perigaud. New Pill? http://cert.lexsi.com/weblog/index.php/2008/03/21/223-new-pill, 2008.

[36] N. Provos and T. Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley Professional, Reading, 2007.

[37] T. Ptacek. Side-Channel Detection Attacks Against Unauthorized Hypervisors. http://www.matasano.com/log/930/side-channel-detection-attacks-against-unauthorized-hypervisors/, 2007.

[38] D. Quist and Valsmith. Covert Debugging: Circumventing Software Armoring. In *Black Hat USA*, 2007.

[39] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting System Emulators. In *ISC*, pages 1–18, 2007.

[40] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *ACSAC*, pages 289–300, 2006.

[41] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[42] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[43] A. Vasudevan and R. Yerraballi. Stealth Breakpoints. In *ACSAC*, pages 381–392, 2005.

[44] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained Malware Analysis using Stealth Localized-executions. In *S&P (Oakland)*, pages 264–279, 2006.

[45] C. Wang and S. Ju. The Dilemma of Covert Channels Searching. In *ICISC*, pages 169–174, 2005.

[46] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *NDSS*, 2006.

[47] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2), 2007.

[48] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS*, 2007.

APPENDIX

A. QEMU DETECTION CODE

```
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>

int seh_handler(
      struct _EXCEPTION_RECORD *exception_record,
      void *established_frame,
      struct _CONTEXT *context_recorcd,
      void *dispatcher_context)
{
      printf("Not QEMU!\n");
      exit(0);
}

int main(int argc, char *argv[])
{
      uint32_t handler = (uint32_t)seh_handler;

      printf("Attempting detection\n");

      __asm("movl %0, %%eax\n\t"
            "pushl %%eax\n\t"::
            "r" (handler): "%eax");

      __asm("pushl %fs:0\n\t"
            "movl %esp, %fs:0\n\t");

      __asm(".byte 0xf3,0xf3,0xf3,0xf3,0xf3,0xf3,"
                  "0xf3,0xf3,0xf3,0xf3,0xf3,0xf3,"
                  "0xf3,0xf3,0xf3,0x90");

      __asm("movl %esp, %eax");
      __asm("movl %eax, %fs:0");
      __asm("addl $8, %esp");

      printf("QEMU Detected!\n");
      return EXIT_SUCCESS;
}
```