Transparent System Introspection in Support of Analyzing Stealthy Malware

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In partial fulfillment

of the requirements for the degree

Doctor of Philosophy Computer Engineering

by

Kevin Joseph Leach

December 2016

APPROVAL SHEET

The dissertation
is submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Kevin Joseph Leach, author

The dissertation has been read and approved by the examining committee:

Dr. Westley Weimer, Advisor

Dr. Joanne Dugan, Committee Chair

Dr. Marty Humphrey

Dr. Ronald D. Williams

Dr. Laura E. Barnes

Dr. Angelos Stavrou

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, Dean, School of Engineering and Applied Science

December 2016

Abstract

The proliferation of malware has increased dramatically and seriously degraded the privacy of users and the integrity of hosts. Millions of unique malware samples appear every year, which has driven the development of a vast array of analysis tools. Malware analysis is often performed with the assistance of virtualization or emulation for rapid deployment. Malware samples are run in an instrumented virtual machine or analysis tool, and existing introspection techniques help an analyst determine its behavior. Unfortunately, a growing body of malware samples has begun employing anti-debugging, anti-virtualization, and anti-emulation techniques to escape or otherwise subvert these analysis mechanisms.

These anti-analysis techniques often require measuring differences between the analysis environment and the native environment (e.g., executing more slowly in a debugger). We call these measurable differences artifacts. Malware samples that use artifacts to exhibit stealthy behavior have increased the effort required to analyze and understand each stealthy sample. Additionally, traditional automated techniques fail against such samples because they produce measurable artifacts. We desire a transparent approach that produces no artifacts, thereby admitting the analysis of stealthy malware. We refer to this challenge as the debugging transparency problem. Solving this problem is thus concerned with reducing artifacts or permitting reliable analysis in the presence of artifacts.

We present a system consisting of two approaches to address the debugging transparency problem and then demonstrate how these components can apply to currently available computer systems. We present two techniques capable of transparently acquiring snapshots of memory and disk activity that can be used to analyze stealthy malware. First, we discuss a novel use of a custom Field-Programmable Gate Array that provides snapshots of memory and disk activity with no measurable timing artifacts. Second, we present a novel use of System Management Mode on x86 platforms that produces no functional artifacts at the expense of producing timing artifacts. Finally, we present an approach to evaluating the tradeoff space that exists between analysis transparency and the fidelity of introspection data provided by such an analysis system. Together, these approaches form a cohesive solution to the debugging transparency problem that admits analyzing stealthy malware.

Dedication

The graduate school experience has been a long endurance test. There are several people to whom I owe a tremendous amount of gratitude.

First, to Wes Weimer, thank you for serving as my doctoral advisor. Thank you for taking me as a student in spite of my coming from an entirely different field of research. Thank you for your tireless support throughout the doctoral program, and thank you for your eternal patience with me as a time-consuming student. Thank you for advising me even though so many of my publications did not include you as a co-author. Thank you for your professional and life advice, and thank you for improving my research acumen and writing skills. Most of all, thank you for helping me figure out what I want to do with my life. Without your help, I would not have been able to complete my doctoral degree.

To Angelos Stavrou, thank you for being my advisor while I completed my master's degree. Attending GMU was the best professional decision of my life, and you have been like a coach for becoming a better security researcher and professional. Thank you so much for the opportunities you have provided to me, and thank you for your timeless advice. I wish you the best of success in your career, your company, and your family.

To Fengwei Zhang, thank you for the years of productive collaboration. We certainly seem to be able to come up with research-worthy ideas. I am fortunate to be able to call you a colleague and friend. I hope there will be years of collaboration to come, and I hope you find all the success you deserve.

To Chad Spensky, thank you for setting me up with the internship at Lincoln Laboratory. Thank you for securing two years of funding for my doctoral studies—you helped me gain a lot of research freedom during my time at UVa. Thank you for being a friend and collaborator, and thank you for cooking such good barbecue. I hope we can continue to collaborate in the future.

To Marty Humphrey, thank you for enduring my sardonic comments in your operating systems class and in your office. I appreciate all of the general advice you have given me and the conversations we have had during my time at UVa.

To Joanne Dugan, thank you for your unwavering determination in seeing that the computer engineering program succeeds and that the computer engineering students are represented. The computer engineering program has been great to me, both as an undergrad and a doctoral student.

To Laura Barnes, thank you for tolerating me as a student when I first came to UVa as a graduate student. Your research group is one of the reasons I was able to stick it through in spite of a difficult first year.

To Ronald Williams, thank you for serving on my committee. Your question about caching during my proposal turned out to be a significant concern and led to a refined system. It certainly made reviewers happy on a subsequent journal submission.

To Natalie Edwards, thank you for your eternal patience and administrative help throughout the doctoral process. Thank you especially for making sure I got paid! Travel and reimbursement were made much easier due to your efforts.

To Nick Napoli, thank you for being a friend throughout this Iditarod through Hell they call graduate school. I hope we can remain friends and work together in the future.

To Lin Gong, thank you for being my friend since my first year at UVa. Thank you for introducing me to so many people in the department, and thank you for your confidence and conversations over the years.

To Jon Dorn, Kevin Angstadt, Jamie Floyd, Nora Evans, Erin Griffiths, and the rest of WRG, thank you for your help and support. Thank you for the laughs in the office, and thank you for tolerating my loud mechanical keyboard. To Dorn and Kevin A., thank you both for your help on the quadcopter demo while I was interning at GrammaTech. I hope you all have amazing success in whatever your future endeavors may be.

To Kate Highnam, my former undergraduate workhorse, thank you for your diligence and hard work. I wish you the best of luck.

I am extremely fortunate to have loving and supportive family. To my parents, Richard and Linda, thank you for your love and all your advice through the years. Thank you for your financial support, and thank you for putting up with me. Most of all, thank you for encouraging me to complete this doctoral degree. I was hesitant to complete the degree early on, but I am so glad I have seen it through to the end.

To my brother, Eric, thank you for helping me print large volumes of text. Thank you for your patience with me, and thank you for including me in the Fantasy Football league.

To Yu Huang, my wonderful fiancée and the love of my life, thank you for your love and support. Thank you for tolerating me, and thank you for being so patient and good-natured. I am lucky to have you in my life. Most importantly, thank you for helping me find somewhat-related Chinese quotes to place in my dissertation chapters. I can't wait for what the future holds for us. 我爱你。

Most of all, thank you, the reader, for taking the time to read my dissertation. Thank you for contributing to scientific research to bolster human knowledge. I hope you find this work informative.

Contents

List of Figures

List of Tables

Glossary

Artifact A measurable piece of data (i.e., a "tell") exposed by an analysis tool or instrumentation framework that code under test can use to subvert or disable such analysis tools. xiii, xviii, 6–8, 10, 11, 17, 21, 22, 25–27, 30, 31, 33, 35, 38, 40, 52, 58, 59, 61–63, 70, 74, 79, 83, 85–91, 97, 98, 103, 105, 109–111, 118–123, 125–127

Bare metal A native, non-virtualized system. 9, 26, 27, 61–63, 74, 86, 89

Basic Input/Output System A small piece of code that executes when a system is first powered on. It controls very low-level functions concerning the configuration of hardware (e.g., it configures interrupt delivery for connected components). xv, 20

BIOS Basic Input/Output System. xv, 18, 20, 23, 33, 67, 74–76, 80–83, 88, *Glossary:* Basic Input/Output System

Cat-and-mouse game An informal description of the constant retaliation between the attacker and defender communities in computer security. Once a new attack is revealed, a new defense against it is quick to follow, after which a new attack technique to circumvent it is revealed. 4, 89, 90

Commercial off-the-shelf An informal term describing products that are widely available through commercial vendors. As of 2016, examples of COTS software include Microsoft Word, MATLAB, and Adobe Photoshop. xv, 26

COTS Commercial off-the-shelf. xv, 26–28, 94, 95, *Glossary:* Commercial off-the-shelf

CPU Mode On Intel x86 CPUs, the mode of execution connotes how the CPU addresses system memory and performs operations. This approach was adopted so that Intel could maintain backwards compatibility as newer generation CPUs were introduced. Intel CPUs support 1) Real Mode, a legacy 16-bit addressing mode in which the CPU addresses real physical addresses, 2) Protected Mode, in which the CPU addresses virtual addresses for each process, and 3) System Management Mode. 14, 62, 66, 72, 82, 83, 86, 87

Direct Memory Access A architectural feature in computer systems that permits peripheral devices to directly read and write from system memory. For instance, a disk that reads a file from storage may be configured to directly write the file's contents into memory for subsequent use by a program. xvi, 8

DMA Direct Memory Access. xvi, 8, 25, 26, 29, 31, 33, 35, 40, 58, 93, 126, *Glossary:* Direct Memory Access

Field-programmable gate array Reprogrammable hardware. Often used for hardware prototype, FPGAs allow a developer to design a circuit and deploy it in a reusable device. xvi, 8

FPGA Field-programmable gate array. xvi, 8, 11, 25–30, 33, 39, 58, 61, 93, 123, 126, *Glossary:* Field-programmable gate array

Heisenbug An unexpected program behavior in which a program behaves differently while executing in a debugger from native execution. Similar to the observer effect in physics, where a system cannot be measured without affecting that system.. 106, 117, 123

Instruction set A set of instructions, or small operations, supported by a CPU. For instance, instruction sets may include simple mathematical operations such as addition and multiplication. 10, 14, 17, 21, 23, 127

Interrupt Asynchronous events produced by physical peripheral devices, such as keyboards, mice, and network interfaces. This event-driven approach enables computer systems to handle externally-produced events as they are produced instead of repeatedly polling for a peripheral's status. xvi, 14–17

Interrupt handler After an interrupt occurs, the CPU will begin executing a special piece of code that handles the request associated with the device that produced the interrupt. For example, a keyboard interrupt handler will check which key was pressed on the keyboard that caused an interrupt. 15–17

Introspection In the context of malware analysis, introspection is the examination of a sample's execution, including variable locations and values in memory, disk or network access, or other resource usage. This information helps an analyst understand the overall behavior and intent of a sample. xviii, 18, 94–96, 98–101, 103, 111, 113–116, 118, 120, 122–124

Malware A general name for any malicious computer code that takes control of a computer for nefarious purposes, including exhausting system resources or stealing sensitive information. xviii, 3, 4, 6, 7, 9, 10, 15, 18, 20–22, 27, 28, 30, 33, 36–38, 43, 48, 53, 54, 58, 62, 63, 70–72, 74, 75, 78, 81, 83–85, 87–89, 94, 95, 98, 105, 106, 125–127

Malware analysis The art of understanding the behavior of a malicious sample of code. Often, analysis is performed to document the sort of malicious activity a sample performs, with the end goal being the rapid detection of future instances of that malicious sample. xvi–xviii, 4, 6–11, 18, 21, 22, 25, 27, 33, 58, 62, 63, 74, 75, 80, 81, 85, 86, 89–91, 93, 95, 105, 124, 126

Master File Table In New Technology File System (NTFS), a special record that maintains metadata about all files stored on the disk. xvii, 42

MFT Master File Table. xvii, 42, 43, *Glossary:* Master File Table

New Technology File System Microsoft's proprietary filesystem used in Windows releases based on the NT kernel. xvi, xvii

NTFS New Technology File System. xvi, xvii, *Glossary:* New Technology File System

Performance counter A set of special-purpose registers that help developers assess their code's performance with respect to various metrics. For instance, a CPU may count the number of cache misses that occur during execution, which may inform a developer that they should rethink their data structures to take advantage of locality. 26, 33, 61, 71–73, 78, 87, 88, 126

Peripheral Component Interconnect Express An industry standard protocol for communications between a computer system and peripheral components attached to it. 9, 28, 29, 31, 33, 35, 40, 58, 93, 99, 113, 122, 123, 126

Ransomware A malicious computer program that extorts a victim. Often, ransomware will destroy personal data unless the victim pays a certain amount of money. 2

Remote System A computer system engaged by a human analyst or automated triage system that communicates with the System Under Test (SUT) during analysis. viii, xvii

Rootkit A sophisticated malicious program that takes control of the operating system, not just a single program. Rootkits often attempt to hide their presence by modifying kernel structures vital to the operating system. 27, 43–46, 58, 62, 63, 74, 82

RS Remote System. viii, xvii, xviii, 11, 12, 29, 62–67, 70, 71, 75–77, 80, 87, 100, *Glossary:* Remote System

SATA Serial Advanced Technology Attachment. xvii, 25, 28, 29, 33, 35, 39, 41, 58, 76, *Glossary:* Serial Advanced Technology Attachment

Secure Guard Extensions An extended set of instructions included with Intel CPUs starting in the mid 2010's. These instructions permit the creation of and interaction with secure regions of memory called *enclaves*. xvii

Semantic gap The challenge of gleaning meaningful semantic information from raw introspection data, such as variables and data structures from chunks of raw system memory. In the context of malware analysis, tools that reconstruct such semantic data are said to *bridge the semantic gap*. 22, 24, 26, 40, 66, 69, 80, 96, 101

Serial Advanced Technology Attachment As of 2016, the standard communications port and protocol used by storage devices in computer systems. xvii, 25

SGX Secure Guard Extensions. xvii, *Glossary:* Secure Guard Extensions

SMI System Managment Interrupt Handler. xviii, 20, 64–73, 75–79, 82, 83, 85–88, 90, 126, *Glossary:* System Managment Interrupt Handler

SMM System Management Mode. xv, xviii, 8, 9, 11, 14, 18, 20, 22, 59, 62–64, 66, 68–71, 73, 75–78, 80–83, 85–88, 90, 93, 99, 100, 113, 123, *Glossary:* System Management Mode

SMRAM System Management RAM. xviii, 20, 66, 72, 76, 78, 82, 83, 86–88, *Glossary:* System Management RAM

Social engineering The use of psychological manipulation by a malicious agent to coerce a victim to reveal sensitive information. 3

Stealthy malware A subset of malware that uses artifacts to determine whether it is being analyzed. Such samples take significant human effort to understand. xviii, 6, 7, 9–11, 20, 21, 27, 28, 36, 43, 46, 58, 62, 64, 77, 90, 91, 93–95, 105, 106, 122, 125–127

SUT System Under Test. xvii, xviii, 11, 12, 26, 29–32, 35, 36, 39, 41, 42, 46–50, 52, 58, 59, 62–68, 70, 71, 73, 75–77, 80, 81, 90, 93, 95, 100, 126, *Glossary:* System Under Test

System Management Mode A special mode of execution on Intel x86 CPUs much like Real or Protected modes. While in System Management Mode, the CPU logically atomically executes the System Managment Interrupt Handler (SMI). xviii

System Management RAM On x86 platforms, a special location in system memory that contains storage for the SMI. xviii, 20

System Managment Interrupt Handler A piece of code stored in the system's BIOS that is only executed when the CPU enters System Management Mode (SMM). xviii

System Under Test A computer system that executes a sample of stealthy malware as part of its analysis. The SUT communicates with the RS during analysis. xvii, xviii

Transparency The property by which a system does not change a passing signal. In this dissertation, malware analysis is said to be transparency if the malware sample cannot predictably measure the precense of the malware analysis tool—that is, the tool does not produce any artifacts. 7–11, 27, 31, 36, 58, 62–64, 72, 74, 76, 77, 80, 81, 83–86, 89–91, 93–100, 103, 105, 106, 111, 116–118, 122, 124, 126, 127

Virtual Machine Introspection A technique in which programs execute within a virtual machine and are analyzed externally. An external tool introspects inside the VM to examine the execution of the program. Often, malware analysis employs VMI tools to understand malicious behavior in an isolated environment. xix, 18

Virtualization, Virtual Machine, VM, Hypervisor A technique that allows an emulated computer system to run as a virtual machine (VM) process within a hypervisor system or virtual machine monitor. Virtualization permits the creation and management of isolated machines with controlled access to system resources. xviii, 6, 14, 17–19, 22, 24, 26, 27, 30, 36, 38, 40, 46, 51, 62, 63, 71, 73–75, 83–86, 90, 94, 95, 98, 105, 111

Virus A malicious computer program that exhausts system resources by automatically duplicating itself to other machines. 2

VMI Virtual Machine Introspection. xix, 18, 22, 30, 66, 69, 70, 99, *Glossary:* Virtual Machine Introspection

兵者，诡道也。故能而示之不能，用而示之不用，近而示之远，远而示之近。

War is an art of deception. We must show our enemy that we are incapable of winning while in fact concealing our true prowess; that we are inactive while in fact mounting the attack; that we are tracking some remote target while in fact stalking one nearby; that we are stalking some nearby target while in fact tracking one far away.

The Art of War

1

Introduction

From 2009 to 2015, Volkswagen produced cars capable of cheating on emissions tests [124]. Volkswagen successfully exploited the fact that the emissions test administered by the Environmental Protection Agency (EPA) consists of a specific sequence of precise inputs. The car's computer was designed to detect this predictable sequence and alter the engine's behavior accordingly. When the computer detected the emissions test, it would enter a high efficiency mode that made it appear to perform well. In practice, however, the Volkswagen cars in question released as much as 10x the emissions required by the EPA [106] during normal operation. It would take years before the discrepancy between EPA test performance and actual road performance would be detected. In brief, because the EPA test offered measurable differences from typical driving patterns, Volkswagen was capable of telling precisely when the vehicle was under test, and thus *performed differently while the test was administered.* In this dissertation, we investigate a similar line of logic relevant to computer security: how can we test samples of code without those samples determining whether they are being tested?

Society's dependence on computers has grown drastically in the past few decades. From professional activities [167, 177] to social media [167, 179] to personal banking [95], computing is now a linchpin of modern day-to-day life in the developed world. Unfortunately, this increasing dependence has been met with rising interest in exploitation for nefarious purposes [144, 155, 246]. With so much confidential and personal information being stored in computers today, malicious agents can benefit tremendously from acquiring personal or confidential information stored on computer systems [110, 153, 171, 174]. As a result of this growth in high-profile malicious activity, securing computer systems has become a significant and important focus of society [166, 172, 173, 226].

1.1 History of Attacks Against Computer Systems

In decades past, clever developers would create programs that produced unexpected behavior for the sake of showing it was possible [56, 225]. For example, in 1971, `Creeper` [56] was the first widespread self-replicating software (i.e., the first computer *virus*), which simply printed a message. Later programs would maliciously cause damage—for example, in 1988, the `Morris Worm` [198] copied itself many times to multiple machines. Eventually, malicious developers would begin using such code for financial gain. For example, in 1989, the `AIDS` [224] program rendered all files inaccessible by encrypting them and demanded money to decrypt the files (indeed, this program was the first example of *ransomware*, software that holds information hostage until a ransom is met). These early attacks demonstrated some glaring problems with early computer system design.

While developers began addressing security concerns in computer systems, malicious agents would respond with increasingly complex malicious software and code, called *malware*. In 1991, the `Michelangelo` [63] virus devastated Australian computer systems by preventing computers from starting up. Later, during 2001, the `Code Red` [65] and `Nimda` [64] viruses exploited vulnerabilities in Microsoft webserver software, using new means to propagate. With the amount of personal and sensitive information stored on computers today, there is now a large financial incentive to compromise computer systems [30, 158, 159].

The financial incentive can come from two sources: selling personal information, or selling techniques for compromising systems. For example, in 2009, Virginia's Prescription Monitoring database was compromised [144], exposing social security numbers, driver's license numbers, and prescription records of many Virginians. In such cases, the data themselves proved valuable, and the malicious agents involved were using malware developed separately. However, malicious agents can additionally sell techniques used by malware for breaking in to computer systems. In such cases, the agent does not care about who or what the victim may eventually be; malicious agents may purchase such techniques to deploy their own attacks against computer systems. For example, new iOS malware can be sold for as much as $500,000 [178]. In response to the expanding market for attacks, companies now frequently pay substantial sums of money to private parties who discover vulnerabilities in their own code using *bug bounty* or *white hat* programs [87, 109, 113, 178] to compete with underground markets seeking to compromise their software.

This dissertation focuses on the distribution of malware, which takes advantage of weaknesses in the computer's or software's design to gain partial or complete control of a computer system.[1] As malware emerges, the computer security community responds in two

[1] Malicious agents can gain access to computer systems in a variety of ways, such as *social engineering* [114], the psychological manipulation of a person causing them to disclose sensitive information. For example, a

ways: 1) developers deploy a fix or design change that renders the malware inoperable, and 2) analysts deploy a technique for quickly detecting the malware to prevent it from subsequently executing on new systems. The malware community then responds by deploying new malware that circumvents previously deployed fixes or detection mechanism. This *cat-and-mouse-game* constantly challenges the malicious agents and security professionals to adapt [201, 214]. As a result, the security community benefits from techniques that facilitate rapid *analysis* of malware to understand its behavior so that fixes and detection procedures can be deployed.

1.2 Problems with Malware Analysis

The proliferation of malware has increased dramatically in the past few years, seriously eroding user and corporate privacy and trust in computer systems [30, 133, 134, 158, 159]. Kaspersky Lab products detected over six billion threats against users and hosts in 2014, consisting of almost two million specific, unique malware samples [133]. In 2015, this figure doubled to four million unique samples of malware [134]. McAfee reported that malware abundance has consistently increased by more than 40 million unique malware samples per quarter during 2015 [158, 159] alone. While automated techniques exist for detecting pre-identified malware [216], manual analysis is still necessary to understand new and unknown threats [89, 245]. Malware analysis is thus critical to understanding new infection techniques and to maintaining a strong defense [31].

Understanding a malware sample's behavior often involves executing the sample. As a result, analysts employ laboratory setups [73, 230] in which to run a sample of malware for two main reasons. First, executing a sample runs the risk of destroying data on the platform or performing other dangerous tasks that may compromise tasks subsequently performed using

malicious agent may pretend to be a bank employee to coerce a victim to give away their bank account information. However, as the focus of this dissertation is malware, these alternative malicious techniques are out of scope.

Table 1.1: Summary of Anti-debugging, Anti-VM, and Anti-emulation Techniques

Anti-debugging [42, 88]	
API Call	Kernel32!IsDebuggerPresent returns 1 if target process is being debugged
	ntdll!NtQueryInformationProcess: ProcessInformation field set to -1 if the process is being debugged
	kernel32!CheckRemoteDebuggerPresent returns 1 in debugger process
	NtSetInformationThread with ThreadInformationClass set to 0x11 will detach some debuggers
	kernel32!DebugActiveProcess to prevent other debuggers from attaching to a process
PEB Field	PEB!IsDebugged is set by the system when a process is debugged
	PEB!NtGlobalFlags is set if the process was created by a debugger
Detection	ForceFlag field in heap header (+0x10) can be used to detect some debuggers
	UnhandledExceptionFilter calls a user-defined filter function, but terminates in a debugging process
	TEB of a debugged process contains a NULL pointer if no debugger is attached; valid pointer if some debuggers are attached
	Ctrl-C raises an exception in a debugged process, but the signal handler is called without debugging
	Inserting a Rogue INT3 opcode can masquerade as breakpoints
	Trap flag register manipulation to thwart tracers
	If entryPoint RVA set to 0, the magic MZ value in PE files is erased
	ZwClose system call with invalid parameters can raise an exception in an attached debugger
	Direct context modification to confuse a debugger
	0x2D interrupt causes debugged program to stop raising exceptions
	Some In-circuit Emulators (ICEs) can be detected by observing the behavior of the undocumented 0xF1 instruction
	Searching for 0xCC instructions in program memory to detect software breakpoints
	TLS-callback to perform checks
Anti-virtualization	
VMWare	Virtualized device identifiers contain well-known strings [57]
	checkvm software [55] can search for VMWare hooks in memory
	Well-known locations/strings associated with VMWare tools
Xen	Checking the VMX bit by executing CPUID with EAX as 1 [10]
	CPU errata: AH4 erratum [10]
Other	LDTR register [186]
	IDTR register (Red Pill [193])
	Magic I/O port (0x5658, 'VX') [139]
	Invalid instruction behavior [24]
	Using memory deduplication to detect various hypervisors including VMware ESX server, Xen, and Linux KVM [240]
Anti-emulation	
Bochs	Visible debug port [57]
QEMU	`cpuid` returns less specific information [241]
	Accessing reserved MSR registers raises a General Protection (GP) exception in real hardware; QEMU does not [188]
	Attempting to execute an instruction longer than 15 bytes raises a GP exception in real hardware; QEMU does not [188]
	Undocumented `icebp` instruction hangs in QEMU [241], while real hardware raises an exception
	Unaligned memory references raise exceptions in real hardware; unsupported by QEMU [188]
	Bit 3 of FPU Control World register is always 1 in real hardware, while QEMU contains a 0 [241]
Other	Using CPU bugs or errata to create CPU fingerprints via public chipset documentation [188]

that system. Second, malware analysis often requires measuring a before and after snapshot of the system; having a known good starting image and controlled environment helps to determine the specific activities performed by a single malicious sample. *Virtualization* technology allows rapidly deploying a controlled computer configuration called a *virtual machine* (VM). A virtual machine provides some degree of controlled isolation. The advent of system virtualization technologies such as VMWare [222], Xen [78], and VirtualBox [168] has paved the way for a variety of computer security and analysis applications [11,20,26,28, 35,47,52,53,60,76,83,97,99,104,107,116,119–121,125,130,141,142,176,180,190,197,208, 212,238].

Malware analysis often employs such virtualization [70,71,90] and emulation [15,187,241] techniques that enable dynamic analysis of malware behavior. A given malware sample is run in a virtual machine (VM) or emulator. An program (or a VM plugin) *introspects* the malicious process to help an analyst determine its behavior [36]. Introspection is a technique used to unveil semantic information about a program without access to source code. Initially, conventional wisdom held that the malware sample under test would be incapable of altering or subverting the debugger or VM environment [101,191]. Unfortunately, malware developers can easily escape or subvert these analysis mechanisms using several anti-debugging, anti-virtualization, and anti-emulation techniques [24,42,57,88,186,188], which we refer to as *artifacts*. We summarize known artifacts in Table 1.1. The wide diversity of types of artifacts used by malware results in a time-intensive process to manually account for each of them when analyzing a sample. Chen *et al.* [57] reported that 40% of 6,900 given samples were found to hide or reduce malicious behavior when run in a VM or with a debugger attached. Thus, the analysis and detection of malware, including such *stealthy* malware, remains an open research problem which we call the *debugging transparency problem.*

There is a diverse array of anti-analysis artifacts as shown in Table 1.1. We divide artifacts into two broad types: software-based and timing-based. Software artifacts are functional properties that can reveal the presence of analysis, such as the value returned by `isDebuggerPresent` or improperly emulated features. Timing artifacts are introduced by the cost of executing the analysis framework. Debuggers that allow single-stepping through instructions incur a significant amount of overhead, which can be measured by the software under test. Malware samples that are aware of these artifacts can thus heuristically detect when they are being analyzed. Such stealthy malware requires additional effort to understand. Solving the debugging transparency problem is thus concerned with reducing artifacts or permitting reliable analysis in the presence of artifacts.

1.3 Transparent System Introspection as a Solution

We choose to focus on three aspects to help analyze stealthy malware. Many analysts employ existing analysis tools such as IDA Pro [126] or OllyDbg [244], but such tools introduce slowdowns or other detectable artifacts. Thus, we first desire a solution that provides *low overhead and low artifact* debugging capabilities. Second, these debugging capabilities ultimately rely on the ability to gather semantic information from a program. To provide information that the analysis requires, we observe that a solution should therefore successfully read and report 1) variable values and 2) dynamic stack traces [85] as well as disk activity. Third, we want to analyze tradeoffs between maintaining low artifacts and providing high-fidelity semantic information that is useful to an analyst.

To achieve these desired properties of a successful solution, we combine several insights to form the basis of a novel system for transparent malware analysis. First, specialized hardware exists that can be used to read a host's memory and disk activity with extremely low overhead and without injecting artifacts in the platform's software. Second, we can use existing program analysis techniques to reconstruct valuable semantic information from raw memory dumps, including variable values and dynamic stack traces. Finally, we can use these program analysis techniques to further examine the cost of maintaining transparency to providing high-fidelity semantic information.

We present a system consisting of three components that use the above insights to solve the debugging transparency problem. First, we discuss low-artifact memory acquisition in two ways. Our first component incorporates a novel use of custom Field-programmable gate array (FPGA) circuit, while our second component employs SMM on x86 platforms. The FPGA approach can access memory (and disk activity) with low overhead (and therefore fewer timing artifacts), and with only one visible functional artifacts (the Direct Memory Access (DMA) access performance counter). SMM on x86 platforms can transparently acquire physical memory associated with a single process. Compared to the FPGA approach, using SMM introduces no functional artifacts, but at the cost of increased timing artifacts. Both approaches are capable of providing a similar interface: the user provides an address, and the system returns the value of physical memory at that address.

Finally, our third component explores the tradeoff space that exists between maintaining transparency and the fidelity of the analysis. This component uses the snapshots produced by the FPGA and SMM approaches described above to reconstruct semantic information about a given program using well-known program analysis techniques. We can determine values of variables based upon known memory locations. Similarly, since activation records of function calls are stored on the stack, we can reconstruct a sequence of function calls if we know where the program's stack is stored. We use program analysis techniques to find this information given a binary and a sequence of raw memory dumps produced by the FPGA or SMM components, and then explore the how much value semantic information we can reconstruct based upon the level of transparency we wish to maintain.

1.4 Organization of This Dissertation

In brief, we present three components:

- In Chapter 3, we present an approach for transparently acquiring memory snapshots and disk activity of a bare metal system using PCI Express.
- In Chapter 4, we present an alternative approach for transparently acquiring memory snapshots of a bare metal system using Intel SMM.
- In Chapter 5, we present a discussion of the tradeoffs between maintaining transparency and the amount of variable and dynamic stack trace information that can be reconstructed from memory snapshots provided by components discussed in Chapters 3 and 4.

1.5 Malware Analysis Challenges

Malware analysis is an increasingly important area. As malware becomes more complex, the strain on analysis resources escalates. Engineers from MIT Lincoln Laboratory frequently take as long as one month to manually analyze a new single sample of stealthy malware [205]. With millions of new samples appearing every year, this time investment is not feasible, and is certainly no longer cost effective. Further, with the increased prevalence and reliance upon computing in our everyday lives, large-scale malware such as `Stuxnet` [246] and `Careto` [136] have caused significant financial damage. As for individual users, Kaspersky Labs reported 22.9 million attacks involving financial malware that targeted 2.7 million individual users [135] in year 2014 alone and 3.5 million individual users in 2015 [134].

1.6 Threat Model

In the security literature, we often enumerate the capabilities of a hypothetical attacker to help solidify the circumstances in which a new approach may apply. This enumeration is referred to as a *Threat Model*.

The system presented in this dissertation entails malware analysis. We must therefore define the scope of the malware that we analyze. We assume a malware sample can compromise the operating system after executing its very first instruction. This assumption is strong because it means we do not have a chance to observe the sample before it compromises the system—rather, it begins executing malicious activity as soon as possible. We further assume the malware can use unlimited computational resources. This strong assumption precludes solutions that do not address fundamental behavior inherent to the malicious sample—for instance, we cannot declare the problem solved by claiming the malware takes too long to complete any malicious activity. In contrast, we assume that the physical hardware is trusted; hardware trojans [217] are thus out of scope. As hardware trojans are a fertile area of research in the hardware security community [34, 68, 94, 115, 189, 229, 231, 252], this assumption is reasonable when considering samples of software instead of circuit design or layout.

1.7 Thesis Statement

> It is possible to develop a transparent malware analysis system capable of analyzing stealthy malware samples by independently considering timing and functional artifacts.

This dissertation consists of three research components that, together, form a cohesive system supporting the automated, transparent analysis of stealthy malware. First, we discuss a hardware-assisted introspection approach using a custom FPGA circuit to maintain low timing artifacts. Second, we discuss an alternative hardware-assisted approach using SMM to achieve low function artifacts. Third, we discuss tradeoffs between maintaining transparency and the quality of semantic data recoverable from data produced by the first two components. Together, these techniques help solve the debugging transparency problem in a way that can be broadly applied.

1.8 System Overview

We present a system consisting of two hosts, 1) a *System Under Test (SUT)*, and 2) a *Remote System (RS)*. The SUT is the platform that contains the code we want to observe (e.g., a stealthy malware sample), while the RS is used by an analyst to guide introspection and debugging on the SUT. This architecture is favorable because it isolates duties: the SUT executes its code under test transparently. The RS, in turn, is responsible for debugging and analysis. Figure 1.1 gives a high-level architectural diagram of our system.

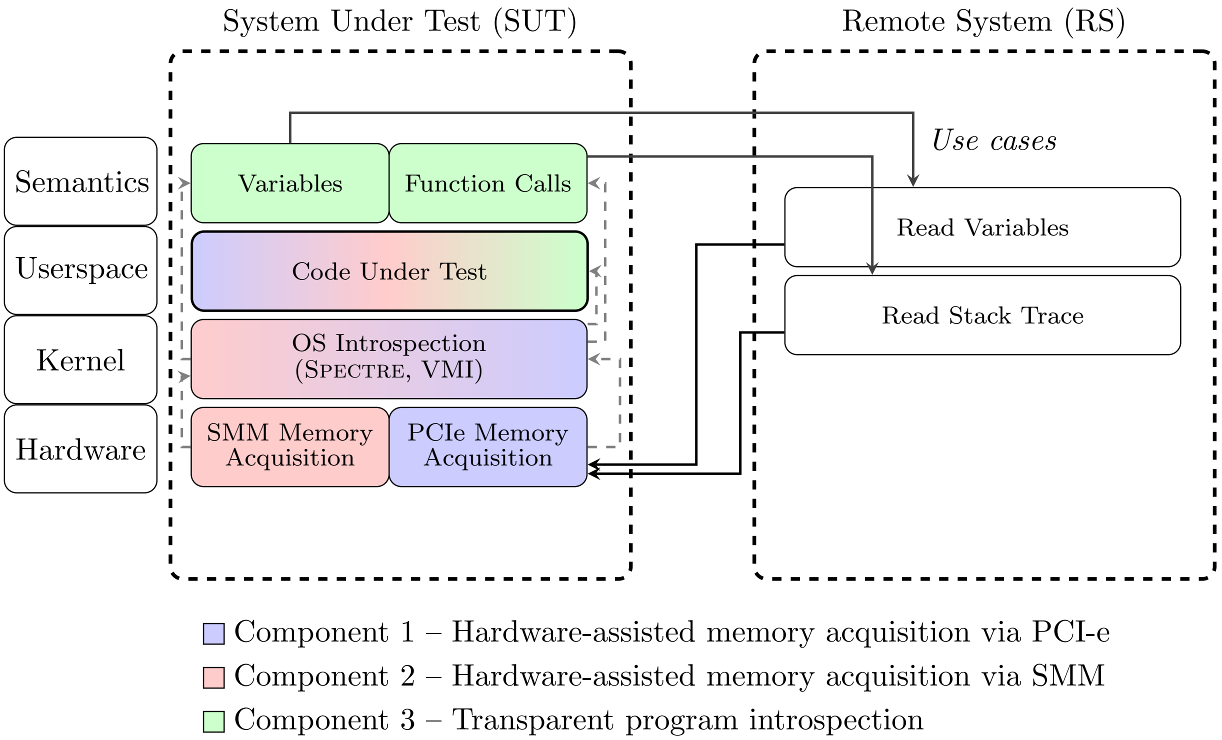


Figure 1.1: System structure. The system on the left (SUT) executes the sample of code to be instrumented. Various components shown in the shaded boxes correspond to chapters in this dissertation. The system on the right (RS) contains the various use cases an analyst may want to perform. The arrows show how the components presented in this dissertation relate to a particular use case.

工欲善其事，必先利其器。

The diligent worker must sharpen his tools.

Analects of Confucius

2

Background and Related Work

In this chapter, we introduce terms, vocabulary, and background material used in, or related to, this dissertation.

2.1 Computer Architecture and Operating Systems

In this section, we briefly introduce three concepts from computer architecture and operating systems as they relate to this dissertation:

1. Intel CPU basics,

2. interrupts,

3. virtualization and virtual machines, and

For further information regarding computer architecture and operating systems, we refer the reader to Hennessy and Patterson [118] and Silberschatz *et al.* [199].

2.1.1 Intel CPU Basics

Intel CPUs date back to the early 1970s [128]. Since the 8086 in 1976, all Intel desktop CPUs have supported the same basic set of *instructions*—small operations the CPU performs in sequence to realize the final intent of a program. The set of instructions used by Intel CPUs is referred to as the *x86 instruction set.* In a running system, program code is stored in the system's memory, and the CPU continuously requests instructions from memory to execute. As of 2016, x86 CPUs are configured to run in *Protected Mode* so that multiple programs can run on the CPU without interfering with each other's memory[1](i.e., the programs are protected from one another). In contrast, older software such as MS-DOS [218] would run in *Real Mode*, in which the CPU would access real physical addresses of memory. In this dissertation, we explore a novel use of a third mode of execution called *System Management Mode (SMM)*, which is discussed in Section 2.2.

In addition, Intel CPUs employ a layered security mechanism when executing instructions. Code executes in one of four *rings* (i.e., ring 0 through ring 3), and each ring is associated with a particular privilege level. For instance, high-privilege operating system code executes in ring 0, where the CPU allows executing highly-privileged instructions that influence system stability. In contrast, typical program code runs in ring 3, where the CPU prevents the program from executing highly privileged instructions. Programs can request that the OS perform privileged tasks on its behalf through the use of *system calls.* Thus, the CPU

[1] Protected Mode x86 instructions use virtual addresses managed by the operating system and Memory Management Unit. The details are elided here for brevity. See Silberschatz [199] for more details.

depends on the OS to ensure privileged instructions are executed safely. If, however, malware compromises the OS (i.e., it compromises ring 0), it can potentially execute such privileged instructions, altering system stability and compromising the integrity or privacy of data on the system.

2.1.2 Interrupts

Computer systems interact with the outside world through peripheral devices such as keyboards and mice. Additionally, computers use internal clocks to keep tasks in lock step while executing code. For example, a computer running multiple tasks may configure a clock to raise an alarm every 10ms to indicate when the computer should switch between tasks. In a similar vein, the keyboard may be configured to raise an alarm every time a key is pressed. Support for such event-based response elicitation is achieved through the use of *interrupts*. In essence, the CPU of the computer system has a physical interrupt pin whose state is changed when a particular event occurs. In practice, there are many types and priorities associated with interrupts from particular peripheral devices (e.g., a timer interrupt may be a higher priority than a keyboard interrupt). Once the CPU circuitry detects that an interrupt has occurred, it saves its progress and begins executing special code called the *interrupt handler* to service the device that caused the interrupt. The interrupt handler is usually specific to the type of device causing the interrupt. For example, a keyboard interrupt handler will determine which key was pressed by the user, while a network interrupt handler will read or write information received by or sent to the computer over the network.

Figure 2.1 illustrates the high level operation of a typical timer interrupt. The CPU must be shared between multiple tasks executing on the system. Thus, a timer interrupt is raised by a device at regular intervals (every 10ms as of 2008 [199]) to force the CPU to switch between tasks. Once an interrupt is raised, the CPU begins executing special code called the *interrupt handler*, which services the particular device raising the interrupt. In the case of a

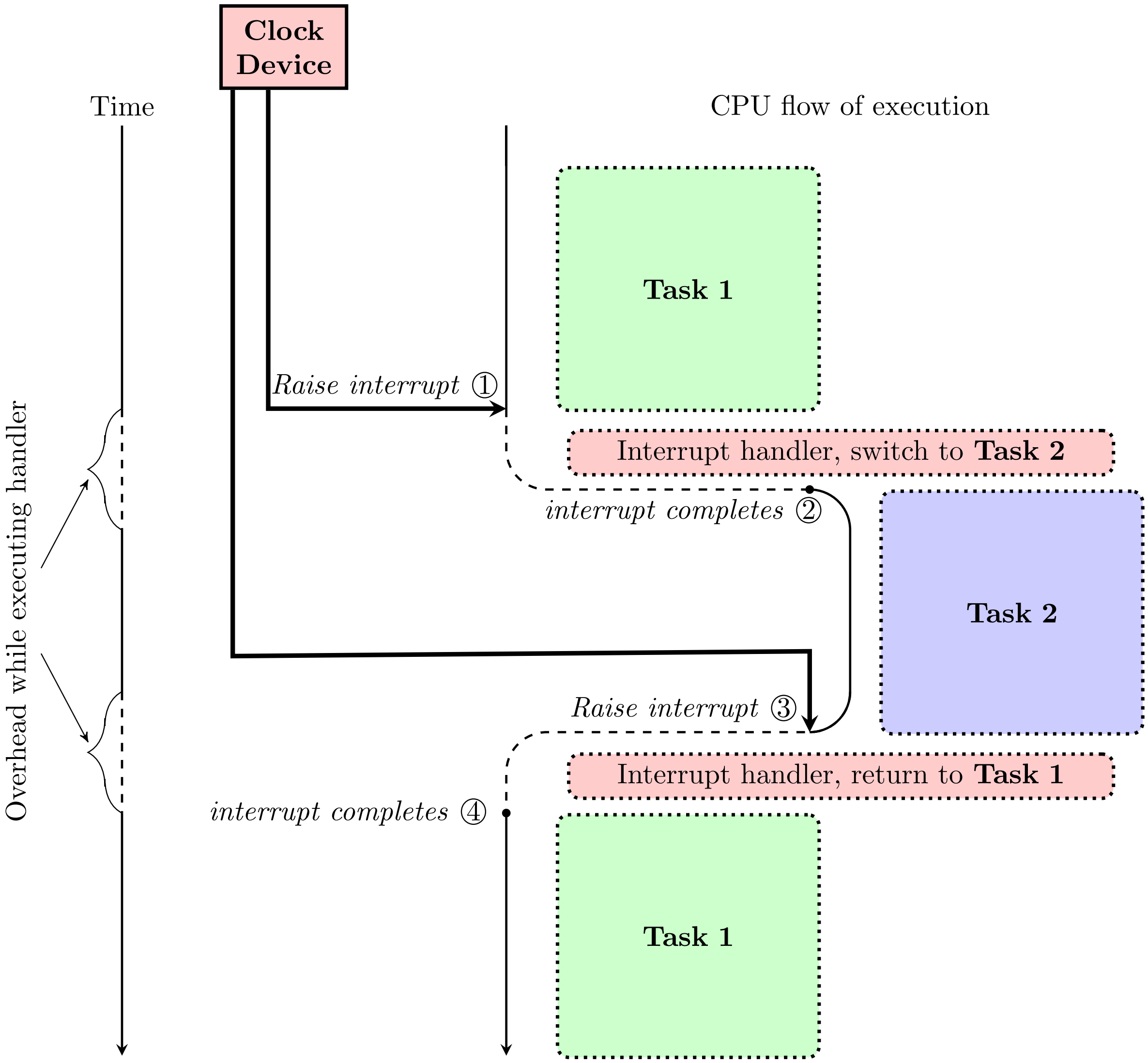


Figure 2.1: High level workflow for interrupts. A device such as a keyboard or internal timer is configured to deliver interruptss when it requires service. This figure considers a typical timer device that the system uses to force the CPU to share execution time between two tasks. In Step ①, the timer device raises an interrupt, forcing the CPU to begin executing the *interrupt handler* to service the timer and switch tasks. This time is considered overhead because the CPU is not executing useful work for processes. In Step ②, the interrupt completes, at which point the CPU begins executing Task 2. The process repeats in Steps ③ and ④, except that the CPU resumes execution in Task 1.

timer interrupt, the CPU executes code to save its progress on the current task and decide which task to execute next. Once the interrupt handler completes, the CPU can resume execution where it left off before the interrupt. In the case of the timer interrupt, the CPU will begin executing the newly selected task.

The interrupt handler is often considered wasted time because servicing the interrupt does not contribute useful work to the progression of the tasks executing on the CPU. In this dissertation, we use regularly-scheduled interrupts to instrument every single instruction that the CPU executes. Thus, the amount of overhead produced during the interrupt handler is crucial to understanding the extent to which our system produces timing artifacts.

2.1.3 Virtualization

One job of the operating system is to ensure that a single computer system can be shared among multiple programs that need executing. As early as the 1960s, large mainframe computers would be shared among multiple users and applications. In a similar vein, a single computer system can be used to manage multiple instances of smaller *virtual* computer systems. These *virtual machines (VMs)* are essentially computer programs that emulate portions of a computer system, allowing one large system called a *hypervisor* to manage many virtual machines concurrently. Virtual machines allow a user access to a logically isolated machine that exists solely as software within a larger machine. This architecture enables rapid reconfiguration of a computer system (e.g., to change system memory available to the virtual machine) and higher utilization (if there are 10 virtual machines with 5% utilization, they can be combined on a single hardware platform with 50% utilization overall). Virtualization technologies such as VMWare [222], VirtualBox [168], Xen [61], and QEMU [32] are widely known and studied [11, 44, 46, 58, 99, 118, 120, 125, 182, 190, 191, 199, 238].

Figure 2.2 illustrates a typical hypervisor and virtual machine configuration. A single physical system shares its physical hardware resources among multiple logically isolated virtual machines (VMs). The hypervisor software allocates virtual resources that are used by the virtual machines residing in the system. Each VM thus gains the illusion of total control over its virtual resources. The hypervisor is ultimately responsible for ensuring that each VM receives its fair share of resources according to its configuration (e.g., a VM may be configured to receive 2 CPU cores out of 10 cores available on the underlying physical system). The hypervisor uses historical resource consumption information to determine how much of a particular resource should be made to each VM at a given time. For instance, if a VM has been active on the CPU for the past 10 seconds, the hypervisor may force the VM off of the CPU to allow another VM to run instead.

The isolation and rapid configurability of virtual machines has been used in the analysis of malware. A sacrificial virtual machine can be deployed with a malware sample executing inside the VM. This allows the malware to completely take over the virtual machine without affecting the underlying system. In practice, special hypervisor software is used to *introspect* inside of the virtual machine to understand how the malicious sample behaves [11,20,26,28, 35,47,52,53,60,76,83,97,99,104,107,116,119–121,125,130,141,142,176,180,190,197,208, 212,238]. Such Virtual Machine Introspection (VMI) is the prevailing technique for malware analysis [74, 103, 116, 129, 130, 151, 202, 207]. In this dissertation, we explore well-known weaknesses in virtual machine introspection and present a cogent solution for the analysis of stealthy malware.

2.2 System Management Mode

Our second component (Chapter 4) makes extensive use of *System Management Mode (SMM)* [127]. SMM is a mode of execution similar to Real and Protected modes available on x86 platforms. It provides a transparent mechanism for implementing platform-specific system control functions such as power management. It is initialized by the Basic Input/Out-

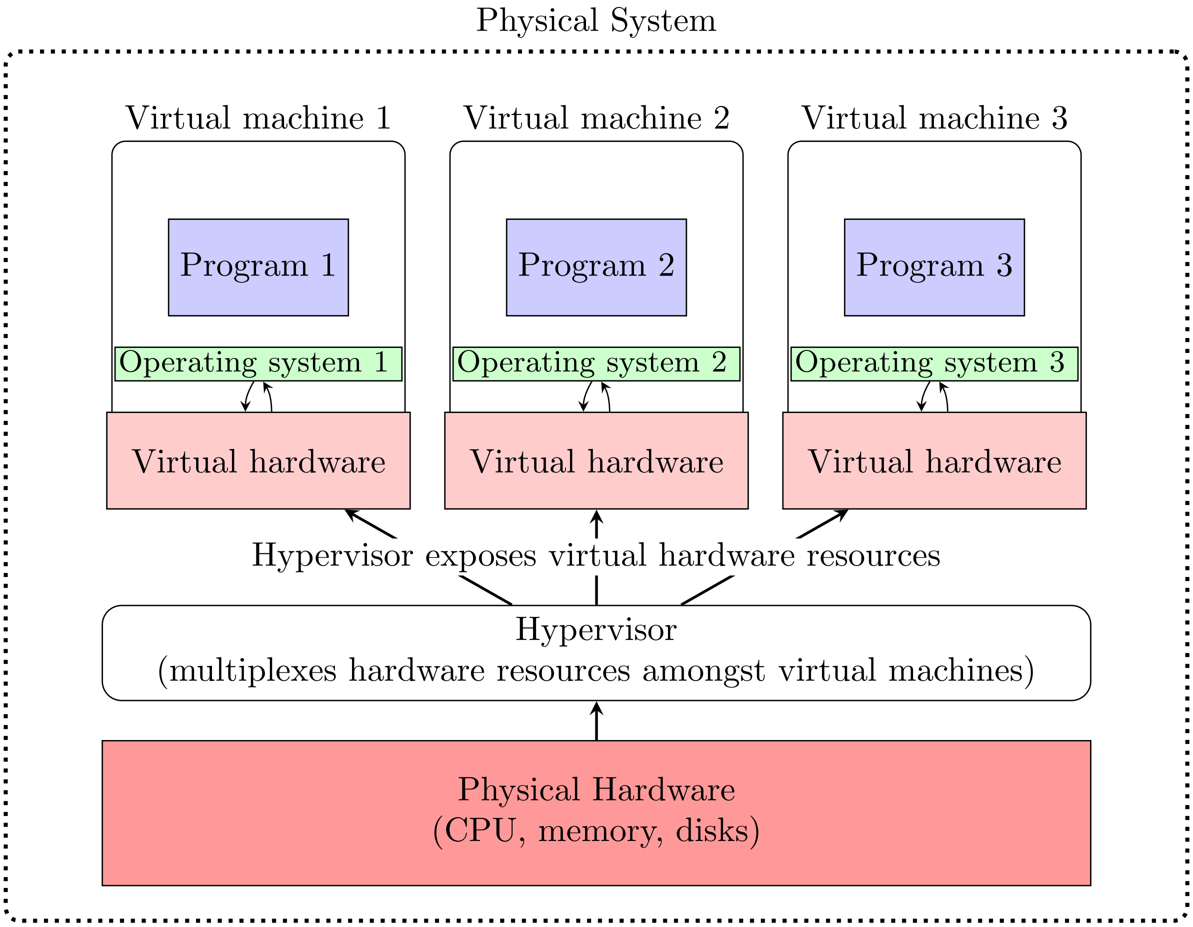


Figure 2.2: Overview of virtualization. A single physical system shares its physical hardware resources among multiple virtual machine instances that are logically isolated from each other. Special control software called a *hypervisor* manages the physical resources and allocates virtual resources that are used in turn by the virtual machines (VMs) residing in the system. Each virtual machine contains its own operating system that gains the illusion of total control over its virtual resources. Programs running within each virtual machine engage their respective operating systems for resources. Programs and operating systems running inside each virtual machine are ideally isolated from, and unaware of, programs and operating systems on other virtual machines. The hypervisor layer is responsible for ensuring a fair allocation of system resources among the virtual machines by measuring and accounting for those resources requested and consumed by each virtual machine.

put System (BIOS). SMM was originally designed to enable the hardware to save power by monitoring which peripherals were being used and turning off peripherals when idle. SMM is transparent to the operating system, so its use in power management did not require extensive driver support. We make use of this transparency aspect to execute analysis code unbeknownst to code executing in both user and kernel space.

SMM is triggered by asserting the *System Management Interrupt (SMI)* pin on the CPU. This pin can be asserted in a variety of ways, which include writing to a hardware port or generating Message Signaled Interrupts with a PCI device. Next, the CPU saves its state to a special region of memory called *System Management RAM (SMRAM)*. Then, it atomically executes the SMI handler stored in System Management RAM (SMRAM). SMRAM cannot be addressed by the other modes of execution. This caveat therefore allows SMRAM to be used as secure storage. The SMI handler is loaded into SMRAM by the BIOS at boot time. The SMI handler has unrestricted access to the physical address space and can run privileged instructions. SMM is thus a convenient means of storing and executing OS-transparent analysis code.

2.3 Stealthy Malware

As of the 2010s, malware detection and analysis tools rely on virtualization, emulation, and debuggers [24, 42, 55, 88] (also, see Table 1.1). Unfortunately, these techniques are becoming less applicable with the growing interest in, and prevalence of, *stealthy malware* [57]. Malware is stealthy if it makes an effort to hide its true behavior. This stealth can emerge in several ways (see Table 1.1 for a more complete list).

First, malware can simply remain inactive in the presence of an analysis tool. Such malware will use a series of platform-specific tests to determine if certain tools are in use. If no tools are found, then the malware executes its malicious payload.

Second, malware may abort its host's execution. For example, a sample may attempt to execute an esoteric instruction that is not properly emulated by the tool being used. In this case, attempting to emulate the instruction may lead to raising an unhandled exception, crashing the analysis program.

Third, malware may simply disable defenses or tools altogether. For instance, OllyDbg 1.10 [244] would crash when attempting to emulate `printf` calls with a large number of '%s' tokens [242]. This type of malware may also infect kernel space and then disable defenses by abusing its elevated privilege level.

Stealthy malware analysis entails significant manual engineering effort because each stealth technique applied by a sample must be accounted for during analysis. As this class of malware grows in volume, the manual analysis and reverse engineering effort required is too burdensome. A firm understanding of stealthy malware is and important part in reducing the overall effort spend analyzing such stealthy malware samples.

2.4 Artifacts

Stealthy malware evades detection by concluding whether an analysis tool is being used to monitor its execution and then changing its behavior. This means there must be some piece of evidence available to the malware that it uses to make this determination.[2] This may be anything from execution time, (e.g., debuggers make programs run more slowly), to I/O device names (e.g., if a device has a name with 'VMWare' in it), to emulation idiosyncrasies (e.g., QEMU fails to set certain flags when executing obscure corner-case instructions). We coin a novel term for these bits of evidence: *artifacts*. Ultimately, we seek more transparent instrumentation and measurement of malware by reducing or eliminating the presence of these artifacts.

[2] An analog in other scientific fields is the *observer effect*, which refers to observable changes that result from the act of observation.

2.5 Malware Analysis

Stealth techniques employed by malware have necessitated the development of increasingly sophisticated techniques to analyze them. For benign or non-stealthy binaries, numerous debuggers exist (e.g., OllyDbg [244], IDA Pro [126], GNU Debugger [108]). However, these debuggers can be trivially detected in most cases (e.g., by using the `isDebuggerPresent()` function). Such anti-analysis techniques led researchers to develop more transparent, security-focused analysis frameworks using virtual machines [15, 70, 71, 90, 203, 241] which typically work by hooking system calls to provide an execution trace which can then be analyzed. System call interposition has its own inherent problems [100] (e.g., performance overhead) which led many researchers to decouple their analysis code even further from the execution path. Virtual-machine introspection (VMI) inspects the system's state without any direct interaction with the control flow of the program under test, thus mitigating much of the performance overhead. VMI has prevailed as the dominant low-artifact technique and has been used by numerous malware analysis systems [74, 103, 116, 130, 151, 202, 207]. Jain *et al.* [129] provide an excellent overview of this area of research. However, introspection techniques have very limited access to the semantic meaning of the data that they are acquiring. This limitation is known as the *semantic gap problem.* There is significant work in the semantic gap problem as it relates to memory [20, 75, 98, 130, 152] and disk [50, 145, 156] accesses. All VM-based techniques thus far have nevertheless been shown to reveal some detectable artifacts [57, 186, 188, 193] that could be used to subvert analysis [91, 185].

The semantic gap problem requires reconstructing useful information from raw data, but this reconstruction process is completely separate from the method of acquisition. Numerous techniques have been discussed which further decouple the analysis code from the software under test by moving the analysis portion into System Management Mode [21, 227, 228, 249] or onto a separate processor altogether [27, 161, 163, 181, 251].

2.6 Debugging

Most popular debuggers, such as the GNU Debugger (GDB), Pin [154], OllyDbg [244], or DynamoRio [43], work by modifying the instructions of the process being debugging by adding jump or interrupt instructions that will periodically call back to the debugging framework to report the requested data. These techniques have obvious performance impacts as they execute multiple instructions in-line within the debugged process, and are known to be trivial to detect. More security-focused dynamic analysis engines either emulate the CPU in software [25, 111, 232] or execute the instructions on the hardware and monitor memory accesses or system calls to trigger their instrumentation [72,202,204,249]. However, these techniques have also been shown to expose numerous artifacts and be susceptible to subversion [91, 185].

Hardware debuggers, such as JTag, Asset InterTech's PCT [18], or ARM's DStream [17], expose no software artifacts, but still have a significant performance impact as they typically function by iteratively sending a sequence of instructions to the CPU and analyzing system state upon completion of those instructions. These debugging interfaces are typically orders of magnitude slower than the CPU that they are debugging, and thus introduce inherent performance restrictions. The aforementioned timing constraints make single-stepping techniques cumbersome for large programs, and are also easily detectable by malicious programs. Further, non-malicious processes dependent on time such as network servers may be unstable in such environments. These techniques have also been widely unexplored in a security context and are typically reserved for debugging the BIOS and boot loaders. This approach explores low-artifact hardware-based debugging techniques for general software programs.

2.7 Memory Introspection

The idea of memory introspection is by no means a new field of study. Numerous tools have been created to facilitate memory introspection on both physical and virtual machines over the years [51, 77, 157, 175, 228], as well tools that bridge the semantic gap. Bhushan et al. [129] summarized the research in the field with virtual machines, however the area of live hardware-based memory introspection is mostly unexplored. In general, these techniques require instrumentation to acquire memory, which is augmented to bridge the semantic gap to extract high-level information from the system being analyzed. Many analysis techniques use expert knowledge to reconstruct features of popular operating systems and processes, as in Volatility [20]. Others use automated techniques to reproduce native process functionality, such as enumerating running processes, via an introspection framework [75, 98, 130, 152]. A third class of techniques achieve isolation by moving security critical introspection functions to a higher-level hypervisor or external process [105, 207]. While a few hardware-based techniques currently exist [181, 228, 251], they either have a very specific focus or a significant performance overhead. We are unaware of any hardware-based systems capable of analyzing arbitrary programs and providing debugging-like information that includes variable values and stack traces while maintaining transparency from the software under test.

笑里藏刀。

Hide a knife behind a smile.

The Thirty-Six Strategems

3

Hardware Assisted System Introspection via Custom FPGA

To support and carry out transparent malware analysis, we require transparent access to a host's memory and disk. In this Chapter and Chapter 4, we discuss two techniques toward this end. First, we present a novel use of a custom FPGA circuit to rapidly acquire memory contents via Direct Memory Access (DMA) and disk activity via Serial Advanced Technology Attachment (SATA) interposition. This technique has the benefit of producing very little measurable timing artifacts and minimal functional artifacts—specifically, software can measure the DMA performance counter. In Chapter 4, we present an alternative approach that does not expose any functional artifacts while instead exposing timing artifacts. In both cases, these hardware-assisted introspection techniques provide a similar interface for memory introspection: a desired address is given as input, and a value of the process's memory at that address is returned as output. For disk introspection, our FPGA technique permits 1) rapidly logging disk activity data and 2) rapidly restoring the disk to a known good state after completion. Together, this transparently-acquired high-fidelity memory and disk data contribute to a cohesive solution to the debugging transparency problem.

We present LO-PHI (Low-Observable Physical Host Instrumentation), a novel component capable of analyzing software executing on Commercial off-the-shelf (COTS) bare metal machines without the need for any additional software on those machines. LO-PHI permits accurate monitoring and analysis of live-running physical hosts in real-time with a minimal addition of plug-and-play components to an otherwise-unmodified SUT. We have taken a two-pronged approach that is capable of 1) instrumenting machines with actual hardware to minimize artifacts, or 2) using hardware virtualization to maximize scale. This permits a tradeoff between transparency, scale, and cost as appropriate. Our architecture uses physical hardware- and software-based sensors to monitor a SUT's memory and disk activity as well as simulate its keyboard, mouse, and power. The raw data collected from our sensors is then processed with modified open source tools (i.e., `Volatility` [20] and `SleuthKit` [50]), to bridge the semantic gap by converting raw sensor data into human-readable, semantically-rich output. Because LO-PHI is designed to collect raw low-level data, it is both operating system and file system agnostic. Our framework can easily be extended to support new or legacy operating systems and filesystems as long as they are physically capable of interacting with our FPGA.

All of the source code for LO-PHI is available under the Berkeley Software Distribution (BSD) license at `http://github.com/mit-ll/LO-PHI`.

We focus our discussion of LO-PHI on our ability to perform automated malware analysis on physical machines, and demonstrate its usefulness by showcasing its ability to analyze classes of malware that trivially evade existing dynamic analysis techniques.

In summary, LO-PHI is a novel approach to instrumenting 1) native, bare metal computer systems, and 2) virtual machines. For hardware instrumentation, LO-PHI uses a custom-designed FPGA built from COTS components that allow the collection of raw physical memory snapshots and disk activity from a live-running system. For virtual machines, LO-PHI uses custom-developed hypervisor plugins that acquire memory snapshots and disk activity from a virtual machine guest. Together, LO-PHI contributes to an end-to-end stealthy malware analysis system. We demonstrate that LO-PHI introduces minimally-measurable timing artifacts. We show that LO-PHI is capable of analyzing sizable corpora of malware and stealthy malware.

This chapter is organized in the following manner. First, we introduce a high level architecture of this hardware-assisted introspection component. Secondly, we present preliminary experiments showing potential artifacts exposed by this component. Third, we apply this component to the analysis of real malware via five experiments:

1. We develop and analyze a stealthy rootkit by using well-known stealth techniques, showing that LO-PHI works on controlled samples where other state-of-the-art tools fail (Section 3.6.1).

2. We analyze a corpus of well-labeled malware, showing that LO-PHI can expose malicious behavior consistent with the ground truth labels (Section 3.6.2).

3. We analyze a corpus of unlabeled malware, showing that LO-PHI can rapidly provide useful introspection data when applied to malware (Section 3.6.3).

4. We analyze an open-source stealthy malware prototype, `pafish`, using LO-PHI, `Anubis`, and `Cuckoo`, and show that LO-PHI remains transparent to stealthy malware during

analysis.

5. We analyze a corpus of coarsely-labeled stealthy malware, showing that LO-PHI is transparent enough to convince stealthy samples to execute under its control (Section 3.6.5).

3.1 Introspection Using PCI Express and SATA

We use a FPGA board with a PCI Express connector to create a *sensor* capable of acquiring memory values transparently. PCI Express supports very high speed transfers. Customized hardware in the form of an FPGA allows us to take full advantage of this high bandwidth. Thus, we can acquire many samples of memory values with low overhead, and therefore few timing artifacts. This approach is also attractive because we can use COTS components to develop the sensor. As a result, we can potentially drastically lower the effort involved in acquiring memory values.

In addition, our FPGA board also includes SATA connectors. SATA is the most common disk connector as of 2016. By attaching a disk to one SATA connector, and the motherboard to the other SATA connector, we can program the FPGA board to record communications between the motherboard and disk. From this SATA communications data, we can 1) reconstruct useful high-level semantic data relating to file operations, and 2) store write activity separately thereby admitting rapid disk restoration. Thus, this approach is attractive because it can save time between back-to-back analyses of different samples.

3.2 Implementation Details

We use a Xilinx ML507 development board to build a prototype of LO-PHI. This FPGA has PCI Express and Gigabit Ethernet connectors. We designed an FPGA that supports reading

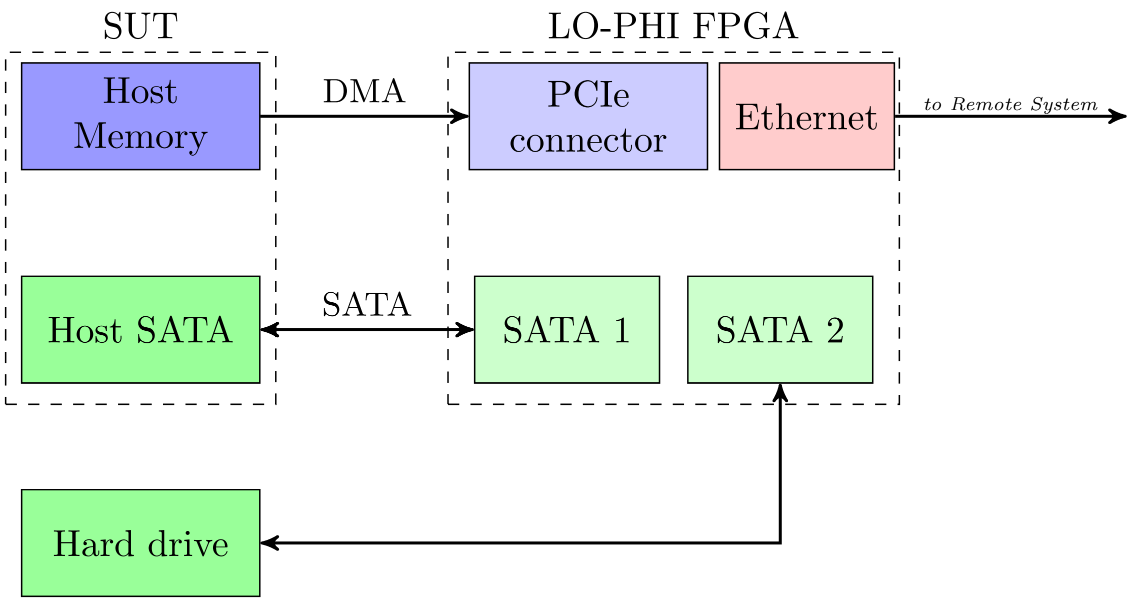


Figure 3.1: Architecture of LO-PHI. The FPGA circuit connects to the SUT's PCI Express port. We interpose the SATA connector by connecting the hard drive to one SATA port and the SUT's motherboard to the other SATA port. We spoof DMA packets over the PCI Express bus to gain high-bandwidth access to memory. By interposing the SATA connector, we gain access to all disk activity. The FPGA sends memory and disk data over its Gigabit Ethernet connector from the SUT to the RS.

memory over PCI Express via DMA, then passes the values over Ethernet to the RS. In our setup, the board communicates with the system under test via PCI Express. As we have assumed hardware is trusted in our Threat Model (Section 1.6), there are no enforcement mechanisms to stop a peripheral from reading arbitrary memory locations.[1] This method has been widely studied [51, 195, 228] and exploited [19, 77, 82, 117, 146, 195, 210]. However, this use of this caveat in transparent memory acquisition for the legitimate study of stealthy malware is novel. The RS connects via Gigabit Ethernet. An analyst then uses the RS to orchestrate an introspection session on the SUT.

[1] Intel's IO Memory Management Unit (IOMMU) optionally configures the specific ranges of memory accessible by DMA from each peripheral device. IOMMU is out of scope as it is not yet in common use as of 2016.

3.3 Addressing Artifacts Exposed via FPGA-Based Introspection

While hardware-level introspection provides numerous desirable security guarantees that are not available for software-based solutions (e.g., hardware segregation of analysis code), it is still critically important to introduce minimal artifacts and, in the ideal case, none at all. We emphasize that the artifacts produced by LO-PHI are likely unusable by extant malware as of 2016 for subversion because the malware would lack a baseline for comparison. For example, to assess the presence of timing artifacts on a SUT instrumented with LO-PHI, a hypothetical malware sample would first require access to an equivalent system without LO-PHI instrumentation to establish ground truth timing.

Nevertheless, we enumerate the artifacts introduced by our FPGA instrumentation. We discuss potential shortcomings in LO-PHI's physical host instrumentation by comparing it against similar instrumentation in a virtual machine meant to represent prevailing VMI techniques [11, 20, 26, 28, 35, 47, 52, 53, 60, 76, 83, 97, 99, 104, 107, 116, 119–121, 125, 130, 141, 142, 176, 180, 190, 197, 208, 212, 238]. In particular, we compare memory bandwidth and disk throughput by comparing performance under the following four configurations:

1. Physical SUT, *without* instrumentation
2. Physical SUT, *with* LO-PHI instrumentation
3. Virtual SUT, *without* instrumentation
4. Virtual SUT, *with* LO-PHI-like instrumentation

In these performance experiments, the physical SUT consisted of a Dell T7500 equipped with a 6-core Xeon X5670 CPU, 2GB of RAM, and a WD3200AAKX disk drive. Our virtual machine was instantiated with one core of a Xeon E5-2665 CPU on a Dell T7600, 1GB of RAM, and a 10GB hard disk.

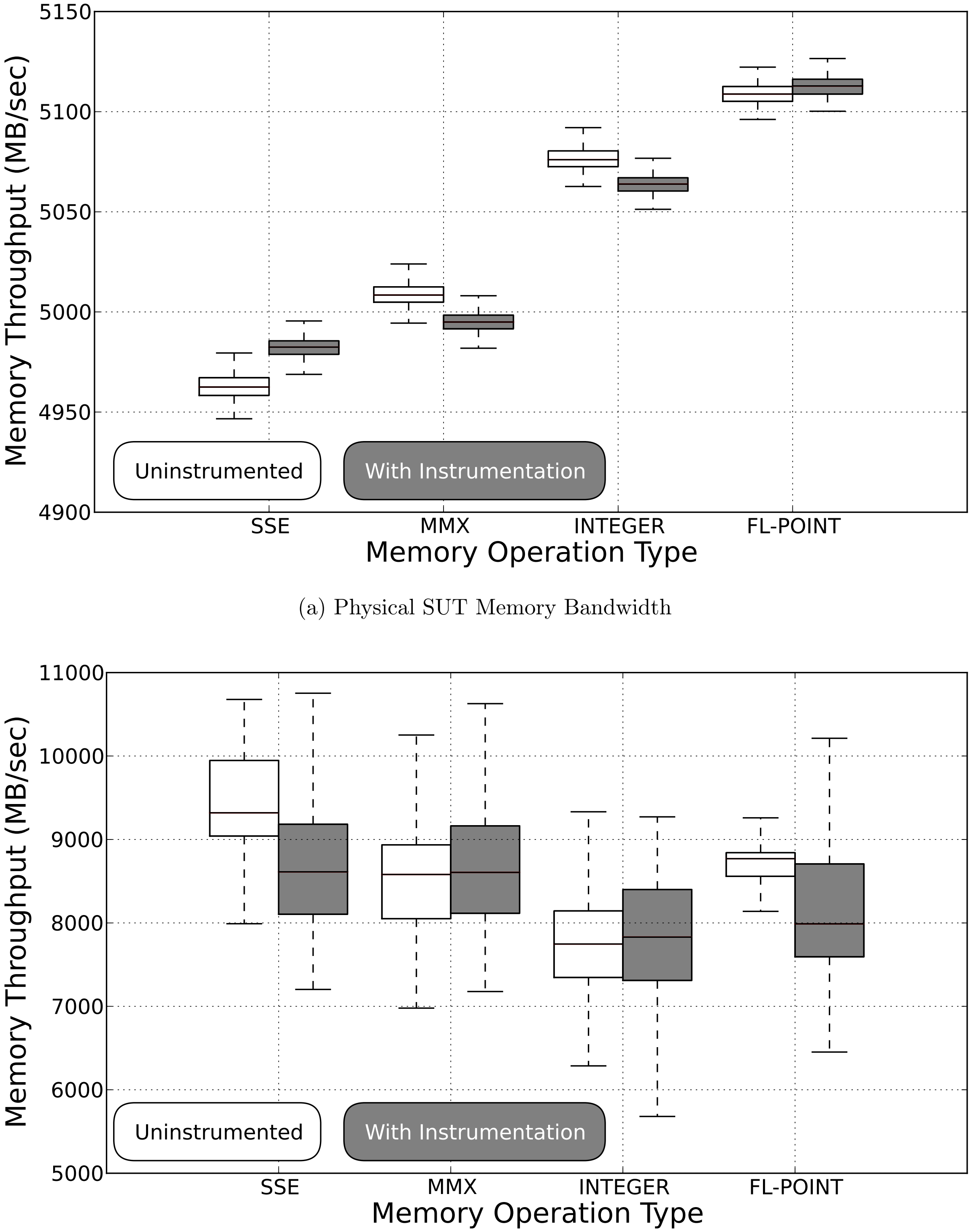
3.3.1 Addressing Memory Bandwidth Artifacts

As main memory is a shared resource, there is likely to be a performance impact (i.e., a timing artifact) whether instrumenting a physical or virtual SUT. We quantify this performance impact using `RAMSpeed`, a popular RAM benchmarking application [123]. We ran `RAMSpeed` on the same system with and without our instrumentation on a fresh install of Windows XP with no other active processes. In each case, we conducted four experiments designed to stress each of the `INT`, `MMX`, `SSE`, and `FL-POINT` instruction sets. Each of these experiments consists of 500 trials, which evaluate and average the performance of copy, scale, sum, and triad operations. To ensure that the memory reads for our instrumentation were not being cached, we configured our sensors to continuously read the entire memory space to produce the worst-case performance impact. The memory polling rates were dictated by the hardware and our particular implementation.[2]

Figure 3.2 appears to indicate that our instrumentation has a discernible effect on the system. However, the deviation from the uninstrumented median is 0.4% in the worst case (SSE in Figure 3.2a). We were unable to to definitively attribute any deviation to our polling of memory. While our memory instrumentation may have some impact on the system, the rates at which we are polling memory do not appear to be frequent enough to predictably degrade performance. This result indicates that systems like ours could poll at significantly higher rates while still remaining undetectable. For example, PCI Express implementations can achieve DMA read speeds of 3 GB/sec [14], which could permit a new class of introspection capabilities.[3] Nevertheless, to detect any deviation in performance, the software being analyzed would need to have the ability to baseline our SUT, which is not practical for adversaries constrained by our Threat Model (Section 1.6).

[2] In this work, the maximum polling rate achievable by PCI Express is roughly one snapshot per 3200 bus cycles. We explore the tradeoff space between polling rate and transparency in Chapter 5.

[3] To this end, we have successfully achieved rates as fast as 60 MB/sec using SLOTSCREAMER [92]; however, the implementation is not stable enough to incorporate into our framework and is left for future work



(a) Physical SUT Memory Bandwidth

(b) Virtual SUT Memory Bandwidth

Figure 3.2: Memory bandwidth reported by `RAMSpeed` on a physical SUT. The white box-plots correspond to native memory bandwidth under no instrumentation, while the filled box-plots correspond to memory bandwidth under LO-PHI instrumentation.

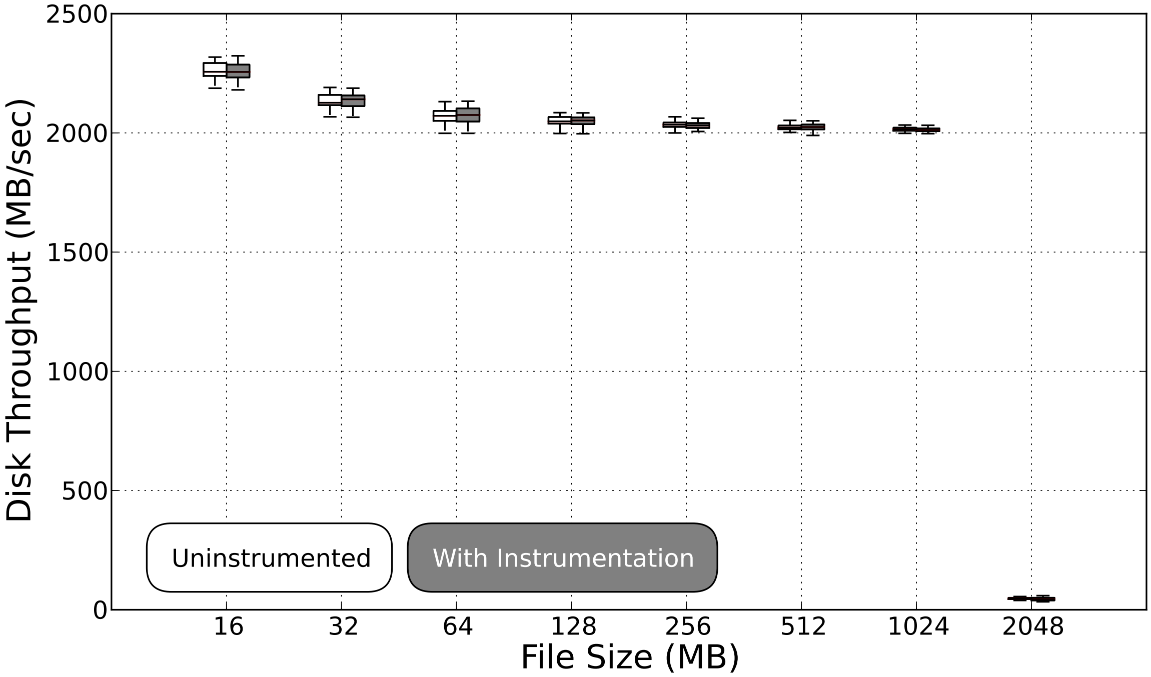
While performance concerns are a universal problem with instrumentation, adding hardware to a physical configuration introduces numerous additional artifacts that must also be addressed. When using PCI Express, the BIOS and operating system can see our hardware via enumerating the PCI Express bus. This inevitably reveals our presence on the machine; nevertheless, mitigations do exist (e.g., masquerading as a different device). To avoid detection, LO-PHI's FPGA card could trivially use a different hardware identifier for each analysis to avoid signature-based detection. However, even with a masked hardware identifier, Stewin *et al.* [209] demonstrated that all of these DMA-based approaches will reveal some artifacts that are exposed in the CPU performance counters. Similar techniques could be employed by malware authors in both physical and virtual environments to detect the presence of a polling-based memory acquisition system such as ours. These anti-analysis techniques may necessitate more sophisticated acquisition approaches left for future work.

3.3.2 Addressing Disk Throughput Artifacts

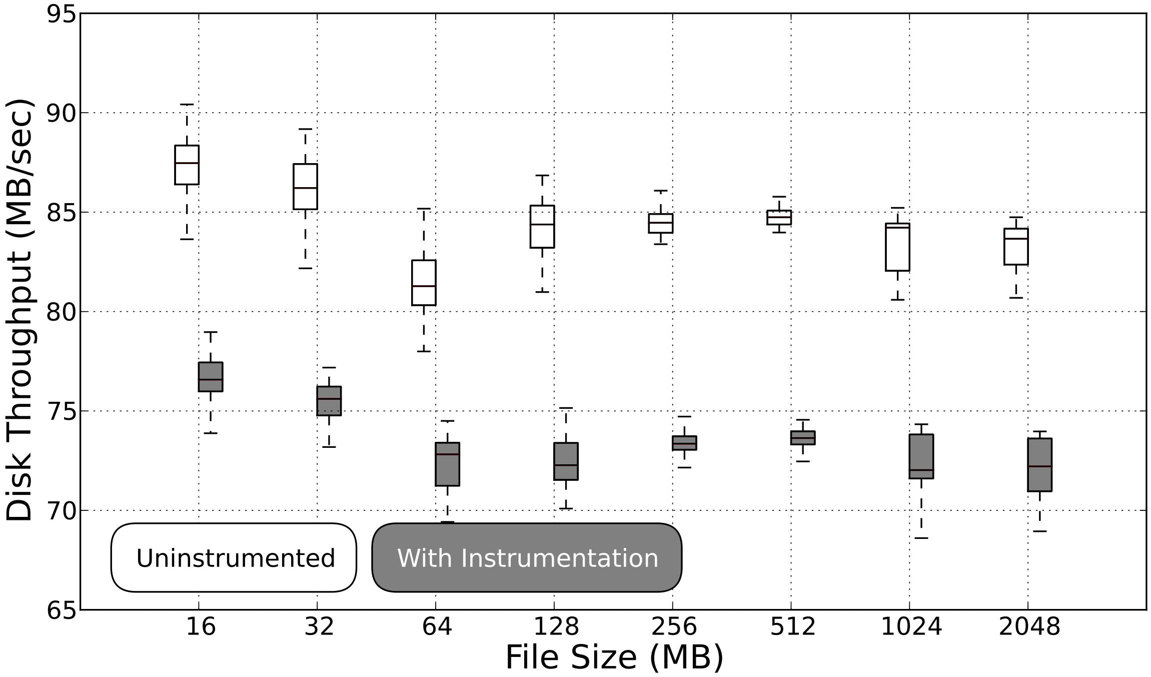
To quantify the performance impact of LO-PHI's disk instrumentation, we similarly employed a popular disk benchmarking utility, `IOZone` [2]. While `IOZone`'s primary purpose is to benchmark the higher-level filesystem, any performance impacts on disk throughput should nonetheless be visible to the tool. We used the same setup as the previous memory experiments (Section 3.3.1) and ran `IOZone` 50 times for each case, both with and without our instrumentation, monitoring the read and write throughputs with a record size of 16MB and file sizes ranging from 16MB to 2GB.

Our hardware should only be visible when we intentionally delay SATA frames to meet the constraints of our Gigabit Ethernet link.[4] In practice, we rarely observed the system cross this threshold; however, `IOZone` is explicitly made to artificially stress the limits of a filesystem. For smaller files, caching masks most of our impact as these cache hits limit the accesses that

[4] We designed the system with this delay to minimize our packet loss since UDP does not guarantee delivery of packets.



(a) File reads



(b) File writes

Figure 3.3: Filesystem throughput comparison as reported by IOZone on Windows XP, with and without instrumentation on a physical machine. We collected 50 samples for each box plot.

actually reach the disk. The caching effect is more prevalent when looking at the raw data rates (e.g., the median uninstrumented read rate was 2.2GB/sec for the 16MB experiment and 46.2MB/sec for 2GB case).

Figures 3.3a and 3.3b show the read and write throughputs for our `IOZone` experiments. The discrepancies between the read and write distributions are attributed to the underlying New Technology File System (NTFS) and optimizations in the hard drive firmware. Figure 3.3a shows that our instrumentation is essentially indistinguishable from the base case when reading, the worst case being a degradation of 3.7% for 2GB files. With writes however, where caching offers no benefit, the effects of our instrumentation are clearly visible, with a maximum performance degradation of 14.5%. Under typical operating conditions, throughputs that reveal our degradation are quite rare. In these experiments, the UDP data rates observed from our sensor averaged 2.4MB/sec with burst speeds reaching as high as 82.5MB/sec, which directly coincide with the rates observed in Figure 3.3b, confirming that we are only visible when throttling SATA to meet the constraints of the Ethernet connection.

In the case of virtual machines, we would expect to have no detectable artifacts on a properly-provisioned host aside from the presence of a Kernel Virtual Machine (KVM) [112]. This is because our instrumentation adds very little code into the execution path for disk accesses, and uses threading to exploit the numerous cores on our system. More precisely, our instrumentation only adds a memory copy operation of the data buffer, which is then passed to a thread to be exported. Our experimental results confirmed this hypothesis as we were unable to identify any consistent artifacts in our `IOZone` tests.

3.3.3 Summary: Addressing Disk and Memory Artifacts

On a physical SUT, LO-PHI produces no measurable differences in 1) memory bandwidth via spoofing DMA packets over the PCI Express, and 2) disk throughput via SATA traffic

Table 3.1: `IOZone` (Virtual SUT) — No LO-PHI

Test	**Avg (Kb/s)**	**Std Dev (Kb/s)**
Write Throughput	36264.01	896.47
Rewrite Throughput	38123.61	480.73
Read Throughput	38077.38	296.29
Reread Throughput	38129.14	426.98

Table 3.2: `IOZone` (Virtual SUT) — With LO-PHI

Test	**Avg (Kb/s)**	**Std Dev (Kb/s)**
Write Throughput	37689.81	1724.58
Rewrite Throughput	39933.51	611.40
Read Throughput	38734.89	1519.16
Reread Throughput	38873.33	1420.16

interposition. This suggests that LO-PHI is not currently detectable by extant stealthy malware as of 2016. Results were similar for a virtual SUT In brief, LO-PHI is capable of acquiring raw memory and disk introspection data from both a physical and virtual SUT while maintaining transparency.

3.4 Malware Experimental Framework

To facilitate experimentation, we built a scalable infrastructure capable of running arbitrary binaries on either a physical or virtual machine with a specified operating system. Our software infrastructure consists of a *master* which accepts job submissions and delegates them to an appropriate *controller*. A given controller is initialized with a set of SUTs, both physical and virtual, that serve as the controller's worker pool. Upon a job submission, the controlled first downloads a script, which describes the actions to perform on the SUT, and submits the job to a scheduler. This scheduler then waits for a SUT of the appropriate type (i.e., physical or virtual) to become available in the pool, allocates it to the analysis, and runs the requested routines. We stored all of our malware samples, analyses, and results in

```
# Reset our disk using PXE
machine.machine_reset()
machine.power_on()
# Wait for OS to appear on network
while not machine.network_get_status():
    time.sleep(1)
# Allow time for OS to continue loading
time.sleep(OS_BOOT_WAIT)
# Start disk capture
disk_tap.start()
# Send key presses to download binary
machine.keypress_send(ftp_script)
# Dump memory (clean)
machine.memory_dump(memory_file_clean)
# Start collection network traffic
network_tap.start()
# Get a list of current visible buttons
button_clicker.update_buttons()
# Start our binary and click any buttons
machine.keypress_send('SPECIAL:RETURN')
# Move our mouse to imitate a human
machine.mouse_wiggle(True)
# Allow binary to execute (Initial)
time.sleep(MALWARE_START_TIME)
# Dump memory (interim)
machine.memory_dump(memory_file_interim)
# Take a screenshot (Before clicking buttons)
machine.screenshot(screenshot_one)
# Click any new buttons that appeared
button_clicker.click_buttons(new_only=True)
# Allow binary to execute (3 min total)
time.sleep(MALWARE_EXECUTION_TIME-elapsed_time)
# Take a final screenshot
machine.screenshot(screenshot_two)
# Dump memory (Dirty)
machine.memory_dump(memory_file_dirty)
# Shutdown Machine
machine.power_shutdown()
```

Figure 3.4: Python script for running a malware sample and collecting the appropriate raw data for analysis.

a MongoDB [162] database. Samples were submitted using a custom File Transfer Protocol (FTP) server and a command line tool that interfaced with the master to instantiate a given analysis script, which are stored on the master and dynamically sent to the controller.

Because of the duality of our framework, we wrote one script (shown in Figure 3.4) that: 1) resets our machine to a clean state, 2) takes a memory snapshot before and after execution, 3) attempts to click any graphical buttons, 4) captures screenshots, and 5) captures all disk activity throughout the execution. To download and execute an arbitrary binary (Figure 3.4, line 12), our implementation uses hotkeys to open a command line interface, executes a recursive FTP download to retrieve the files to be analyzed, and then runs a batch file to execute the binary. From the resulting data, we reconstruct the changes in system memory, in addition to a complete capture of disk activity generated by the binary.

To identify any graphical buttons that the malware may present, we used the `Volatility windows` module to identify all visible windows with buttons, then spoofed USB packets to move the mouse to each button's location and click it.[5] Our analysis framework also attempts to remove any typical analysis-based artifacts by using a random file name and continuously moving the mouse during the execution of the binary. Similarly, when possible, we also properly shutdown the system at the end of the analysis to force any cached disk activity to be flushed.

In our analysis setup, both the physical and virtual environments had a 10GB partition for the operating system and 1GB of volatile memory. The operating system, Windows 7, was placed into a Hibernate state to minimize the variance between executions and also reduce the time required to boot the system. To minimize the space requirements of our system, we compress our memory snapshots before storing them in our databases to save space. Finally, the virtual machines' networks were logically divided to ensure that samples did not interfere with each other. The physical environment consisted of one machine, so no network isolation

[5] The `Volatility windows` module was configured to search for windows that had an atom class of 0xc061 or an atom superclass of 0xc017, which indicate a button.

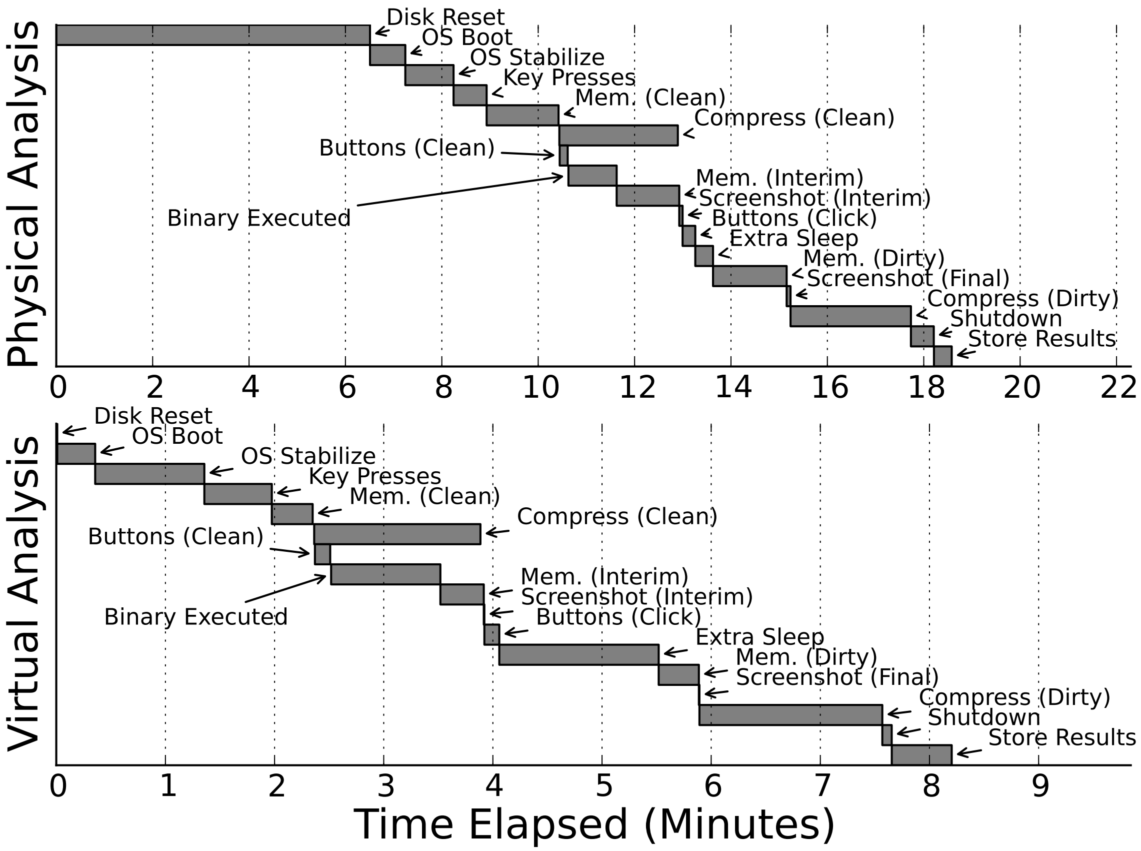
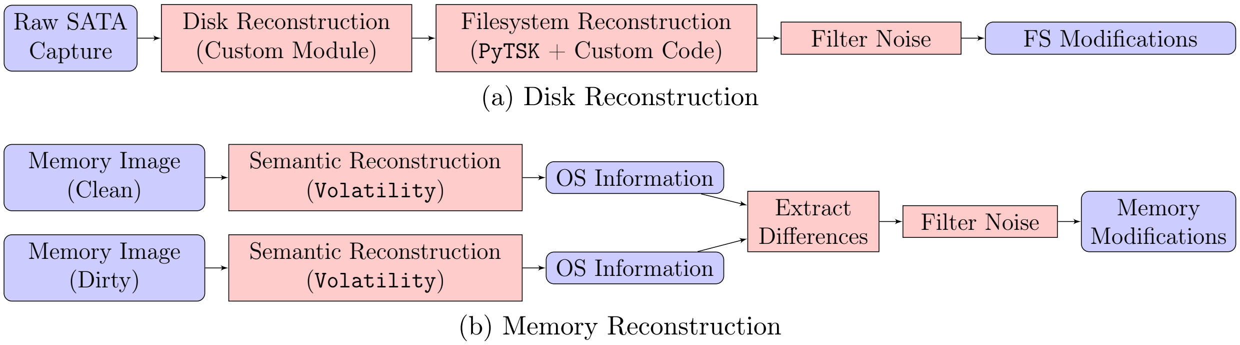


Figure 3.5: Time spent in each step of binary analysis. Both environments booted a 10GB Windows 7 (64-bit) hibernated image with 1GB of system memory.

was required.

The respective runtimes for each portion of our analysis is shown in Figure 3.5. We ensured that every binary executed for at least three minutes before retrieving a final memory snapshot and resetting the system. We used `Volatility`'s `screenshot` module to collect screenshots on physical machines and were extracted from the captured memory snapshots. Note that most of the time taken in the physical case is due to our resetting of the SUT's state using `Clonezilla` [41], waiting for the system to boot, and memory acquisition. The reset and boot process could be decreased significantly by writing a custom PXE loader,[6] or completely mitigated by implementing copy-on-write as part of our FPGA SATA interposi-

[6] The Portable eXecution Environment (PXE, pronounced *pixie* is a specification for describing a networked-enabled boot process in which a system boot configuration is retrieved from a network server.



(a) Disk Reconstruction

(b) Memory Reconstruction

Figure 3.6: Binary analysis workflow. Rounded nodes represent data and rectangles represent data manipulation.

tion. Similarly, the memory acquisition times could be more comparable to the virtual case, if not faster, by optimizing our PCI Express implementation. Finally, full system snapshots could reduce the time spent setting up the environment to mere seconds. While full system snapshots are trivial with virtual machines, capturing them is still an open problem for physical machines.

We note that LO-PHI may miss transient memory modifications made by the binary between our clean and dirty memory snapshots. To analyze the transient behavior of a binary, LO-PHI could be used to continuously poll the system memory during execution. However, while this has the potential to produce higher fidelity, we do not feel that our current polling rates are fast enough to warrant the tradeoff between the produced DMA artifacts and usefulness of the output. We explore this area of research in Chapter 5.

3.5 Bridging the Semantic Gap in LO-PHI

Before any analysis can be conducted, we must first bridge the semantic gap—that is, translate our memory snapshots and SATA captures, which contain low-level, raw, data into high-level, semantically-rich, information.

3.5.1 Memory

To extract operating-system-level modifications from our memory captures, we run a number of `Volatility` plugins on both clean and dirty memory snapshots to parse kernel structures and other objects. Some of the general purpose plugins include `psscan`, `ldrmodules`, `modscan`, and `sockets`, which extract the running processes, loaded dlls, kernel drivers, and open sockets resident in memory. Similarly, we also run more malware-focused plugins such as `idt`, `gdt`, `ssdt`, `svcscan`, and `callbacks` which examine kernel descriptor tables, registered services, and kernel callbacks.

3.5.2 Disk

The first step in our disk analysis is to convert the raw capture of the SATA activity into a 4-tuple containing the 1) disk operation (e.g., read or write), 2) starting sector, 3) total number of sectors, and 4) data. Our physical drives, as with most drives available in 2016, used an optimization in the SATA specification known as Native Command Queuing (NCQ) [69]. NCQ reorders SATA Frame Information Structure (FIS) requests to achieve better performance by reducing extraneous disk head movement and then asynchronously replies based on the optimal path. Thus, to reconstruct the disk activity, our SATA reconstruction module must faithfully model the SATA protocol to track and restore the semantic ordering of FIS packets before translating them to disk operations. Upon reconstructing the disk operations, these read/write transactions are then translated into disk events (e.g., filesystem operations, Master Boot Record modification, slack space modification) using our analysis code which is built upon `Sleuthkit` and `PyTSK` [5]. Since `Sleuthkit` only operates on static disk images, our module required numerous modifications to keep system state while processing a stream of disk operations. Intuitively, we build a model of our SUT's disk drive and step through each read and write transaction, updating the state at each iteration and reporting appropriately. This entire process is visualized in Figure 3.6a. Unlike previous work [156], which

Master File Table (MFT) modification (Sector: 6321319)	
Filename	/WINDOWS/.../drivetable.txt→/.../Desktop/New Text Document.txt
Allocated	0 → 1 **Unallocated** 1 → 0 **Size** 132 → 0
Modified	2014-11-07 20:07:06 (1406250) → 2015-02-19 15:47:17 (3281250)
Accessed	2014-11-07 20:07:06 (1406250) → 2015-02-19 15:47:17 (3281250)
Changed	2014-11-07 20:07:06 (1406250) → 2015-02-19 15:47:17 (3281250)
Created	2014-11-07 20:07:06 (1406250) → 2015-02-19 15:47:17 (3281250)

...

MFT modification (Sector: 6321319)	
Filename	/.../Desktop/New Text Document.txt →/.../Desktop/LO-PHI.txt
Changed	2015-02-19 15:47:17 (3281250) → 2015-02-19 15:47:25 (3437500)

Figure 3.7: Example log produced from creating `LO-PHI.txt` on desktop.

was designed for NTFS, our approach is generalizable to any filesystem supported by tools similar to `Sleuthkit`. An example output from creating the file `LO-PHI.txt` on the desktop is shown in Figure 3.7:

Note that we can infer from this output that the filesystem reused an old MFT entry for `drivetable.txt` and updated the filename, allocation flags, size, and timestamps upon file creation. A subsequent filename and timestamp update were then observed once the new filename, `LO-PHI.txt`, was entered.

3.5.3 Filtering Background Noise

While the ability to provide a complete log of modifications to the entire system is useful in its own right, it is likely more relevant to extract only those events that are attributed to the binary in question. To filter out the activity not attributed to a sample's execution, we first build a controlled baseline for both our physical and virtual SUTs by creating a dataset from 10 runs using a benign binary: `rundll32.exe` with no arguments.[7] We then use our analysis

[7] `rundll32.exe` is a program that loads other components and services in Windows. In this work, we consider it a reasonable baseline because it is essentially a minimal working executable program in Windows.

framework to extract all of the system events for those trials and created a filter (akin to `diff`) based on the events that frequently occurred in this benign dataset. We noted that two of our memory analysis modules (i.e., `filescan` and `timers`) produced particularly high false positives and proved less useful for our automated analysis. To reduce false positives in our disk analysis, we decouple the filenames from their respective Master File Table (MFT) record number.

3.6 Experimental Evaluation of LO-PHI

In the next five sections, we demonstrate the practicality of LO-PHI with five targeted experiments. The experiments are intended to evaluate LO-PHI's ability to:

- detect the behavior elicited by a controlled rootkit developed with typical anti-analysis techniques (Section 3.6.1)
- detect the behavior elicited by real malware, confirmed with ground truth (Section 3.6.2)
- scale and extract meaningful results from unknown malware samples (Section 3.6.3)
- remain transparent to a controlled stealthy malware sample where other techniques fail to do so (Section 3.6.4)
- analyze malware samples that employ anti-analysis and evasion techniques (Section 3.6.5)

3.6.1 LO-PHI Experiment 1: Custom Rootkit

To verify that LO-PHI is, in fact, capable of extracting behaviors of malware, we first evaluated our system with known malware samples for which we have ground truth. We first evaluate a rootkit that we developed using techniques from The Rootkit Arsenal [40] (Section 3.6.1)

Offset	**Name**	**PID**	**PPID**
0x86292438	AcroRd32.exe	1340	1048
0x86458818	AcroRd32.exe	1048	1008
0x86282be0	AdobeARM.exe	1480	1048
0x864562a0	*$$_rk_sketchy_server.exe*	1044	1008

(a) New Processes (pslist)

PID	**Port**	**Protocol**	**Address**
1048	1038	UDP	127.0.0.1
1044	*21*	*TCP*	*0.0.0.0*

(b) New Sockets (sockets)

Selector	**Base**	**Limit**	**Type**	**DPL**	**Gr**	**Pr**
0x320	0x8003b6da	0x00000000	CallGate32	3	-	P

(c) GDT Hooks (gdt)

Name	**Base**	**Size**	**File**
hookssdt.sys	0xf7c5b000	0x1000	C: \...\lophi\hookssdt.sys

(d) Loaded Kernel Models (modscan)

Table	**Entry Index**	**Address**	**Name**	**Module**
0	0x0000f7	0xf7c5b406	NtSetValueKey	hookssdt.sys
0	0x0000ad	0xf7c5b44c	NtQuerySystemInformation	hookssdt.sys
0	0x000091	0xf7c5b554	NtQueryDirectoryFile	hookssdt.sys

(e) SSDT Hooks (ssdt)

Created Filename
/.../lophi/$$_rk_sketchy_server.exe
/.../lophi/hookssdt.sys
/.../lophi/sample_0742475e94904c41de1397af5c53dff8e.exe

(f) Disk Event Log (81 Entries Truncated)

Figure 3.8: Post-filtered semantic output from rootkit experiment (Section 3.6.1).

Our rootkit achieves stealth by adding hooks to the Windows Global Descriptor Table (GDT)[8] and System Service Dispatch Table (SSDT)[9] that hide any directory or running executable with the prefix `$$_rk` and then opens a malicious FTP server. The rootkit module is embedded inside a malicious PDF file that drops and loads a malicious driver file (`hookssdt.sys`) and the FTP server executable (`$$_rk_sketchy_server.exe`). Figure 3.8 shows the complete post-filtered results obtained when running this rootkit through our framework. Note that we received identical results for both virtual and physical machines, which exactly matches what we would expect given our ground truth. We clearly see our rootkit drop the files to disk (Figure 3.8f), load the kernel module (Figure 3.8d), hook the kernel (Figure 3.8e and Figure 3.8c), and then execute our FTP server (Figure 3.8a and Figure 3.8b). We have omitted the creation of numerous temporary files by Adobe Acrobat Reader and Windows as well as accesses to existing files (81 total events) in Figure 3.8f to save space, however all disk activity was successfully reconstructed. Note that we can trivially detect the presence of the new process as we are examining physical memory and are not foiled by execution-level hooks.

We also ran our rootkit on the `Anubis` and `Cuckoo` analysis frameworks. `Anubis` failed to execute the binary, likely due to the lack of Acrobat Reader or some other dependencies. Cuckoo produced an analysis with very similar file-system-level output to ours, reporting 156 file events, compared to our 81 post filtered. However, Cuckoo did not find our listening socket or our GDT and SSDT hooks in their output. While our FTP server was definitely executed, and thus created a listening socket on port 21, it is possible that our kernel module may not have executed properly on their analysis framework. Nevertheless, we feel that our ability to introspect memory to find these obvious tells of malware is a notable distinction. Subsequently, the failure to execute such a simple rootkit also emphasizes the importance of

[8] The GDT is a data structure in memory used by Intel CPUs that describes how to treat other locations of memory. For example, segments of memory can be denoted read-only, writable, or executable.

[9] The SSDT is used by Windows to quickly locate code to perform general tasks such as creating new processes.

having a realistic software environment as well as a hardware environment. We address this issue for our analysis in Section 3.6.5.

3.6.2 LO-PHI Experiment 2: Labeled Malware

While LO-PHI's ability to analyze a stealthy rootkit is quite useful, we also wanted to assess whether LO-PHI can provide useful introspection information when analyzing real stealthy malware samples. We obtained a set of 213 malware samples that were constructed in a clean-room environment and were accompanied by their source code with detailed annotations. All the binaries in this experiment were executed on both physical and virtual machines that were running Windows XP (32bit, Service Pack 3) as their operating system.

For the analysis of these 213 well-annotated malware samples, we first performed a blind analysis, and then later verified our findings with the labels. Note that there were samples that exhibited more behaviors than those listed here, only the most prevalent behavior is discussed here.

VM-detection

We found 66 of these samples employed either anti-VM or anti-debugging capabilities. However, none of the 66 anti-VM samples performed QEMU-KVM detection; instead they focused on VMWare, VirtualPC, and other virtualization suites. As expected, all of the samples executed their full payload in both our virtual and physical analysis environments (recall we use QEMU-KVM for our virtualized SUT). The ground truth indicated 66 such samples, and our analysis with LO-PHI was able to uncover all of them.

New Processes

We found that 79 of the samples created new long-running processes detected by our memory analysis. The most commonly created process was named `svchost.exe`, which occurred in 15 samples. In addition, 2 other samples had variations of `svchost.exe`, i.e., `dddsvchost.exe` and `cbasvchost.exe`. These 17 samples wrote their own `svchost.exe` binary to disk, which was detected by our filesystem analysis, and executed the binary, which opened up a TCP listening socket on port 1053. Port 1053 is associated with the "Remote Assistance" service by the Internet Assigned Numbers Authority (IANA). The second most common process was named `bot.exe` and occurred in 12 samples, and four of these 12 samples also had the third most common process, which was named `dwwin.exe`. The `dwwin.exe` binary claimed to be Dr. Watson [160], a debugger included in Windows, but also appeared to be injected with malicious code. The four samples each created two UDP listening sockets on ports 1045 and 1046, one owned by `bot.exe` and the other owned by `dwwin.exe`, respectively. We inferred from this behavior that these two groups of samples were derived from the same two malware families and contained Remote Administration Tools (RATs), which we confirmed with the ground truth labels.

We also found three samples that executed the SUT's legitimate `firefox.exe` browser, but loaded with a suspicious library `needful.dll` that they wrote to disk. The `firefox.exe` process opened TCP listening sockets on ports 1044 and 1045 in two of the three samples, suggesting that these samples were also RATs attempting to masquerade as the Firefox browser. This supposition was also confirmed by the ground truth data.

Data Exfiltration

We successfully detected 46 samples that attempted to collect and exfiltrate data through a combination of our disk and memory analysis. We initially flagged two particular samples because they appeared to be exfiltrating data via external IPs over port 25, which is reserved for the Simple Mail Transfer Protocol (SMTP). Our disk analysis of these samples showed a number of suspicious file reads, including reads of Firefox's `cert8.db` and `key3.db` for all user profiles stored on the SUT. These files store user installed certificates and saved passwords, respectively, and there were no Firefox processes running during the execution of those samples. Searching for similar suspicious disk behavior in the rest of the labeled set yielded 44 additional samples that appeared to be exfiltrating data. Again, our detections correctly matched the ground truth data.

Worms and Network Scanning

We detected approximately 30 (of 30 labeled samples) that propagated worms and scanned the network. These samples contacted a significant number of IP addresses and created a large number of network sockets in a five minute window. For example, eight of the samples contacted over 140 IP addresses, and 13 samples opened more than 2000 sockets. The same 13 samples appeared to target external IP addresses over port 135, which is associated with Microsoft Remote Procedure Call (RPC), a service that has had remote exploitable vulnerabilities targeted by worms in the past [215].

Command and Control

We detected 14 samples that attempted to contact external servers over TCP port 6667, which is associated with the Internet Relay Chat (IRC) protocol. IRC is also commonly used as a Command and Control (C2) mechanism for remotely controlling malware [184], which was the case for these samples as confirmed by the ground truth data.

Domain Name Service Queries

These malware samples would call out to a variety of different hosts—we could check Domain Name Service (DNS) queries to verify which hosts were implicated by a sample's malicious behavior. The most common DNS queries were for the hostnames `579.info` (55 samples), `windowsupdate.net` (16 samples), `time.windows.com` (11 samples), `wpad` (11 samples), and `google.com` (10 samples). The ground truth data indicated that some of these queries were intended as red herrings, while other queries were for actual contact with more suspicious hostnames such as `irc.site406.com` and `asdf.it`.

Kernel Modules

We detected three samples that unloaded the `ipnat.sys` driver and appeared to gain persistence by replacing it with a malicious version.

3.6.3 LO-PHI Experiment 3: Unlabeled Malware

In this experiment, we demonstrate our framework's ability to scale and extract useful results from completely unknown malicious binaries, which were obtained from the same source as the labeled data and also said to target Windows XP. The physical SUT was the same as described previously (Dell T7500 with 1GB of RAM), but the virtual machines were instantiated on a server with six quad-core Xeon X5670s (24 logical cores) and 68GB of RAM. This enabled us to instantiate a pool of 20 virtual machines with instrumentation. Due to the vast difference in runtimes and resources, we were able to run far fewer samples in our physical environment. We ran 1091 samples in both environments. We present the general types of behaviors detected by LO-PHI in this section. There is no ground truth data associated with these samples to compare results against. However, we feel that the findings clearly demonstrate the usefulness of our system. Basic statistics for our analysis of

Table 3.3: Overall statistics for unlabeled malware (Section 3.6.3).

Observed Behavior	**Number of Samples**
Created new process(es)	765
Opened socket(s)	210
Started service(s)	300
Loaded kernel modules	20
Modified GDT	58
Modified IDT	10

these unlabeled samples are shown in Table 3.3.

New Processes

We found that 70% of the wild samples created new processes that persisted until the end of our analysis. The most common names are shown in Table 3.4. Unsurprisingly, most of the malware appeared to either start legitimate processes or masquerade as innocuously-named processes. We discovered 4 samples that started a process with the same name as the currently logged in user. We found 11 samples created at least 10 new processes on the SUT, one of which created an unusual 84 new processes.

Table 3.4: Top processes created by wild malware (Section 3.6.3).

New Process	**Number of Samples**
IEXPLORE.exe	31
dwwin.exe	30
svchost.exe	30
explorer.exe	14
urdvxc.exe	13
dfrgntfs.exe	13
wordpad.exe	12
defrag.exe	12

Sockets

About 19% of the wild samples opened at least one network socket. The most commonly-opened sockets are shown in Table 3.5. Three samples stood out as potential worms or network scanners as they created over 1900 sockets; the next highest sample created a mere 44 sockets. Unlike our labeled set, none of the wild malware seemed to use obvious C2 channel ports such as 6667 (IRC). For example, only one sample sent traffic over port 80.

Table 3.5: Top 6 sockets (by port and protocol) created by wild malware (Section 3.6.3).

Port	**Protocol**	**Number of Samples**
1038	UDP	58
1039	TCP	42
1042	TCP	37
1038	TCP	36
1040	TCP	36
1041	TCP	32

Services

About 27.5% of the wild samples started and installed at least one new system service. Most of these services suspiciously claimed to be hardware drivers such as USB or audio drivers. For example, over 250 samples loaded a driver claiming to be `hidusb.sys` (for Human Interface Devices over USB), possibly as an attempt to perform key logging.

3.6.4 LO-PHI Experiment 4: Paranoid Fish

First, we highlight our ability to analyze evasive binaries with a ground truth sample. We chose Paranoid Fish (`pafish` v054) [170], a proof-of-concept open-source tool that demonstrates various VM detection and anti-debugging techniques used by actual malware. When `pafish` is executed, it writes a file to disk for each artifact that it observes. `pafish` is currently able to detect most popular analysis frameworks. For example, `pafish` detected 7

Table 3.6: Artifacts detected by `pafish` in `Anubis`.

hi_qemu
hi_sandbox_NumberOfProcessors_less_2_GetSystemInfo
hi_sandbox_NumberOfProcessors_less_2_raw
hi_sandbox_drive_size
hi_sandbox_drive_size2
hi_sandbox_mouse_act
hi_sandbox_physicalmemory_less_1Gb

Table 3.7: Artifacts detected by `pafish` in `Cuckoo`.

hi_CPU_VM_rdtsc
hi_CPU_VM_rdtsc_force_vm_exit
hi_sandbox_mouse_act
hi_sandbox_drive_size
hi_sandbox_drive_size2
hi_hooks_deletefile_m1
hi_virtualbox

artifacts when run against `Anubis` [15] (shown in Table 3.6 and 7 different artifacts when run against `Cuckoo` (shown in Table 3.7).

On the contrary, when executed in our physical analysis environment, `pafish` only detected two artifacts: small disk size and small system memory (shown in Table 3.8).

In this instance, we were using a 750GB hard drive with a 10GB partition on it and 1GB of physical memory. These artifacts are very easily removed by simply adding more hardware to the SUT. However, this has a direct effect on the time per sample in an automated environment, as a larger disk image must be restored, and larger memory snapshots requires more time and space. Nevertheless, LO-PHI is able to analyze `pafish` without detection.

Table 3.8: Artifacts detected by `pafish` in LO-PHI.

hi_sandbox_physicalmemory_less_1Gb
hi_sandbox_drive_size2

Table 3.9: Description of `Volatility` modules used for evaluating evasive malware.

psscan	Enumerates processes using pool tag scanning. (Capable of finding processes that have previously terminated (inactive) and processes that have been hidden or unlinked)
envars	Extracts environment variables from processes in memory.
ssdt	Lists the functions in the Native and GUI SSDTs.
netscan	Enumerates network sockets using pool tag scanning.
ldrmodules	Enumerates modules in the Virtual Address Descriptor (VAD) and cross-references them with three unique PEB lists: InLoad, InInit, and InMem.
driverirp	Enumerates all DRIVER_OBJECT structures in memory
psxview	Helps detect hidden processes by enumerating PsActiveProcessHead using the following methods: PsActiveProcessHead linked list, EPROCESS pool scanning, ETHREAD pool scanning, PspCidTable, Csrss.exe handle table, and Csrss.exe internal linked list.

3.6.5 LO-PHI Experiment 5: Coarsely-Labeled Malware

In this experiment, we exhibit LO-PHI's ability to analyze evasive malware, which thwart existing analysis frameworks. Because we aim to analyze more modern malware samples compared to previous experiments discussed above, we ran these analyses on the same hardware, but with Windows 7 (64-bit) as our operating system. Subsequently, we also installed numerous potentially vulnerable and frequently targeted applications [66]. Specifically, Acrobat 9.4.0,[10] Flash 10.1.85.3, Java 7u0 (64-bit), Firefox 38.0.1, Chrome 43.0.2357.64 (64-bit), .NET 4.5.2, and Python 2.7 (64-bit). The analysis was performed exactly as described above. However, the `Volatility` modules used were limited to those that supported Windows 7, from which we selected the following to use in our analysis: `psscan`, `envars`, `ssdt`, `netscan`, `ldrmodules`, `driverirp`, and `psxview` (see Table 3.9). It is worth noting that the `ssdt` and `driverip` modules did not return any findings in our dataset.

We obtained a set of 429 coarsely-labeled evasive malware samples from Kirat *et al.* [140]

[10] This was the last release before strict sandboxing.

in their previous work. Because these samples were specifically labeled as evasive, we only present the findings from executing them in our physical environment. While we knew these samples employed evasion techniques capable of evading most popular analysis frameworks, we were not given the intended effect or target operating system of the samples as we were with the samples in Section 3.6.2. Thus, we do not make claims as to specific intent of the malware discussed here. We present our aggregated findings below, which indicate that our framework successfully avoided their evasive behaviors. The dataset consisted of malware labeled as using the evasion techniques outlined in Table 3.10. A summary of our findings is presented in Table 3.11.

Table 3.10: Evasive malware dataset.

Technique Employed	**# Samples**
Wait for keyboard	3
Bios-based	6
Hardware id-based	28 of 82
Processor feature-based	62 of 134
Exception-based	79 of 197
Timing-based	251 of 690 total

Table 3.11: Summary of anomalies detected in `Volatility` modules and GUI buttons found in our evasive dataset when executed in our physical environment on Windows 7 (64-bit).

		Volatility Module				
		envars	**netscan**	**ldrmodules**	**psxview**	**buttons**
Malware Label	Keyboard	0	3	1	0	1
	Bios	3	6	6	6	0
	Hardware	28	27	28	26	11
	Processor	53	54	59	51	7
	Exception	76	79	77	76	7
	Timing	229	247	231	239	4

Wait for keyboard

Due to the small number of samples employing this type of technique, we were not able to draw any interesting conclusions from these samples. However, all of these samples appeared to execute successfully. One presented an error dialog window that our framework was able to locate and click, which appeared to kill the sample. This particular sample also made a DNS query to `goldcentre.ru`. The other two had no notable effects on our system.

BIOS-based

All of the examples in this category appeared to trigger their payload. That is, they were unsuccessful in detecting our analysis framework and exhibited some interesting behaviors. Every sample attempted to create an output network connection to `smtp.mail.ru`. Two of them attempted to determine their IP addresses using "whatismyip" services. The samples also spawned new processes that persisted throughout our analysis, most masquerading as existing Windows services. The `psxscan` module indicated that the processes `122.exe` and `123.exe` were spawned in two cases; `explorer.exe` was also spawned by two of the samples. Most interestingly, one of the samples created a hidden `svchost.exe` which was invisible to every process enumeration method except `psscan`.

Hardware-id-based

These samples also exhibited interesting behaviors. Most notably, 23 of them started `TrustedInstaller.exe`, while 25 of the original processes continued running for the duration of our analysis, and the others appeared to spawn new processes. All of the samples also attempted to reach out to network resources: 24 of them attempted to connect to `219.235.1.127:80`, one attempted to connect to `62.75.235.238:443`, and two attempted TCP connections to either `8.8.8.8` or `8.8.4.4`, both Google-owned DNS servers, on port

53, which is the DNS port for UDP communications. All of the samples imported at least 32 modules, with the most active sample importing 156 unique modules. Finally, 11 of them appeared to present buttons that were detected and clicked by LO-PHI, and two of them set the particularly interesting environment variables `9Yy9Y9YYy9YYy` and `YYY9YYY9YYY99`, which both had the value of `E4EC4E2160D8E128C919C56915BFED6C`.

Processor feature-based

These samples produced the least compelling findings. While most of them persisted, or installed new processes, 11 had no new processes in memory. Those that did spawn new processes had filenames similar to before, with four of them once again loading `TrustedInstaller.exe`, three starting a more stealthy `netsh.exe`, 1 spawning a malicious `taskhost.exe`, and, perhaps the least stealthy sample, launching `trojan.exe`. Most of them also exhibited network activity, primarily DNS traffic, with eight of the samples querying a variation of `boxonline`, and seven of the samples attempting reach port 8 on various IP addresses. More interestingly, one of the samples attempted to contact 219.235.1.127, and then opened a local listening socket. A single sample in this set also set the `SEE_MASK_NOZONECHECKS` environment variable to “1”, which is a variable that will hide security warnings in Windows XP. This leads us to believe that at least some of the malware in this set was targeting an older version of windows, and likely explains why some of the samples appeared to have no effect. Two of samples also presented dialog boxes and the button “OK” was clicked by LO-PHI.

Exception-based

The exception-based malware samples also exhibited similar behavior, with all but three of the samples spawning new processes or continuing to execute for the duration of our analysis. Unsurprisingly, many of these samples also attempted to engage the network.

There appeared to be two distinct clusters that reached out to various domains with the strings `boxonline` (31 samples) and `backupdate` (26 samples), with the others calling out to unique domains. The `boxonline` samples indicate that these may be the same class of malware that was previously observed in the processor-feature-based samples. Again, a few of the samples appeared to present a graphical interface with the text "OK," which was successfully clicked by LO-PHI.

Timing-based

This was our largest dataset, and thus yielded the most diverse findings. Again, a majority of the samples (193 out of 251) spawned new processes or persisted throughout our analysis. Some interesting names included: `skype.exe`, which was launched by one process and also hidden from normal windows process enumeration; `taskhost.exe`, which was spawned in a hidden state by 22 processes and a less-stealthy manner by 10 other samples; `conhost.exe`, which was also spawned in a stealthed state; and one sample spawned `facebook.exe`. Once more, we saw four samples set the `SEE_MASK_NOZONECHECKS` environment variable, indicating that Windows XP was likely their intended target. This dataset also had a significant number of samples (156) making `boxonline` DNS queries, and five of the samples querying `backupdate`. None of these samples produced network traffic aside from DNS.

While our analysis did not indicate malicious behavior in all of the samples in this dataset, we were able to detect typical malware behavior from a large majority. Some of our findings indicate that at least some of the samples were targeting Windows XP, which could explain the lack of anomalies for the few that appeared benign. Nevertheless, we feel that our findings are more than sufficient to showcase LO-PHI's ability to analyze evasive malware without being subverted, and subsequently produce high-fidelity results for further analysis. In fact, behaviors like unlinked `EPROCESS` entries and listening sockets can be exceptionally difficult to detect with software-based methodologies. Because LO-PHI has a complete view of the entire memory space and disk activity, the ability for the malware to hide its presence is greatly hindered.

3.6.6 Evaluation Conclusions

In Sections 3.6.1, 3.6.2, 3.6.3, 3.6.4, and 3.6.5, we presented results of rigorous experimentation that demonstrate that LO-PHI is a powerful transparent malware analysis tool. First, we wrote a custom rootkit using well-known stealth techniques, and showed how LO-PHI was capable of providing useful analysis information. Second, we blindly analyzed 213 well-labeled stealthy malware samples using LO-PHI, matching all of the ground truth data correctly. Third, we analyzed a corpus of 1091 unlabeled malware samples using LO-PHI to report general behavior of the samples in the corpus, ultimately demonstrating how LO-PHI can be used to analyze real malware. Fourth, we analyzed a stealthy malware prototype, `pafish`, using LO-PHI and two other malware analysis tools, ultimately showing how LO-PHI remains transparent to stealthy malware. Finally, we analyzed a set of 429 coarsely-labeled malware samples using LO-PHI, further demonstrating its ability to provide useful introspection data for malware analysis. In brief, our experiments show that LO-PHI is an integral component to our overall solution to the debugging transparency problem.

3.7 Concluding Remarks for LO-PHI

In this Chapter, we presented LO-PHI, a custom FPGA-based approach to transparent system introspection. We use the FPGA to spoof DMA packets over the PCI Express bus to rapidly acquiring snapshots of a physical SUT's memory during execution. We further interpose SATA traffic on a physical SUT to access its disk activity. Together, this FPGA component can acquire valuable introspection data from a SUT while maintaining timing transparency (i.e., it produces no timing artifacts.

In the next Chapter, we discuss an alternative approach to acquiring this introspection data from a physical SUT by using SMM. This alternative approach does not produce functional artifacts, but instead produces timing artifacts. In Chapter 5, we explore and discuss the tradeoffs between the two approaches. Together, these components form a cohesive solution to the debugging transparency problem.

明修栈道，暗度陈仓。

Build a walkway to attract your adversary, but send your soldiers through another path.

The Thirty-Six Strategems

4

Hardware Assisted System Introspection via System Management Mode

In the previous Chapter, we discussed an approach using a custom FPGA circuit to rapidly acquire snapshots of memory and disk activity from a live-running bare metal system. While it produces no measurable timing artifacts, there are two potential issues: 1) it potentially produces measurable artifacts from saturating the memory bus (and influencing a CPU performance counter as a result), and 2) it potentially misses transient activity that occurs between snapshots of memory (see Section 3.4). In this Chapter, we discuss an alternative approach using Intel System Management Mode (SMM) that sacrifices timing transparency to gain functional transparency.

In this chapter, we present MalT, a novel approach that helps to achieve transparent debugging by leveraging SMM (described in detail in Section 2.2) on bare metal systems. Our system is motivated by the intuition that malware debugging needs to be transparent—it should not leave functional artifacts introduced by the debugging functions. SMM is a special-purpose CPU mode in all x86 platforms as of 2016. The main benefit of SMM is to provide a distinct and easily-isolated processor environment that is transparent to the OS or running applications. By using SMM, we achieve a high level of transparency, which enables debugging and analyzing stealthy malware.

We briefly describe its basic workflow as follows: we run malware on one physical SUT and employ SMM to communicate with the analyst on a RS. While SMM executes, Protected Mode is essentially paused. The OS and hypervisor are therefore unaware of code executing in SMM. Because we run introspection code in SMM, we expose far fewer artifacts to the malware, enabling a more transparent execution environment for the debugging code than existing approaches.

The RS communicates with the SUT using a `gdb`-like protocol with serial messages. We implement the basic debugging commands (e.g., breakpoints and memory/register examination) in our prototype implementation of MalT. Furthermore, we implement four techniques to provide step-by-step debugging: (1) instruction-level, (2) branch-level, (3) far control transfer level, and (4) near return transfer level. We also provide an interface for MalT that can adapt to commonly-used tools such as `IDAPro` [126] and `gdb`. In addition, we implement a novel technique to completely restore the SUT to a clean state after analyzing a sample whose execution may have corrupted system state. We use SMM to perform this restoration process to remain immune to high-privilege malware and rootkits.

MalT runs the debugging code in SMM without using a hypervisor. Thus, it has a smaller

Trusted Code Base (TCB) than hypervisor-based debugging systems [15, 71, 187, 241], significantly reducing the attack surface of MalT. Moreover, MalT is OS-agnostic and immune to hypervisor attacks (e.g., VM-escape attacks [143, 237]). Compared to existing bare metal malware analysis [139, 233], SMM executes with the same privilege level as the hardware. Thus, MalT is capable of debugging and analyzing kernel and hypervisor rootkits as well [138, 194].

In our MalT prototype, we use SMM on a physical SUT to execute code to interact with an analyst on the RS. To demonstrate the efficiency and transparency of our approach, we test MalT with popular packing, anti-debugging, anti-virtualization, and anti-emulation techniques. The experimental results show that MalT remains transparent against these techniques. MalT introduces a reasonable overhead: It takes about 12μs on average to execute the introspection code. Moreover, we use popular benchmarks to measure the performance overhead for four types of step-by-step execution on Windows and Linux platforms. The overhead ranges from 1.46x to 1519x slowdown on the target system, depending on the user's selected instrumentation method.

The main contributions of this work are:

– We provide a bare metal debugging tool called MalT that leverages SMM for malware analysis. It produces few functional artifacts on SUT, providing a more transparent execution environment for the malware than existing approaches.

– We introduce a hardware-assisted malware analysis approach that uses neither the hypervisor nor OS code. MalT is OS-agnostic and is capable of conducting hypervisor rootkit analysis.

– We implement various debugging functions including breakpoints and step-by-step execution. Our experiments demonstrate that MalT induces moderate but manageable overhead on Windows and Linux environments.

- We implement a novel technique to completely and reliably restore the SUT to a clean state after analyzing a sample.
- Through testing MalT against popular packers, anti-debugging, anti-virtualization, and anti-emulation techniques, we demonstrate that MalT remains transparent to stealthy malware, ultimately remaining undetected.

4.1 System Architecture

Figure 4.1 shows the architecture of the MalT system. The RS is equipped with a simple `gdb`-like debugger. The user inputs basic debugging commands (e.g., *list registers*), and then the SUT executes the command and replies to the RS as required. When a command is entered, the RS sends a message containing the debugging comand via a serial cable to the SUT. While in SMM, the SUT transmits a response message containing the information requested by the command. Since the SUT executes the actual debugging command within the SMI, its operation remains transparent to the target application and underlying operating system.

As shown in Figure 4.1, the RS first sends a message to trigger the SMI on the SUT; MalT reroutes a serial interrupt to generate an SMI when the message is received by the SUT. Once the SUT enters SMM, the RS starts to send debugging commands to the SMI handler on the server. Finally, the SMI handler transparently executes the requested commands (e.g., list registers, set breakpoints at addresses) on the SUT and sends a response message back to the RS.

The SMI handler on the SUT inspects the Code Under Test at runtime. If the Code Under Test reaches a breakpoint, the SMI handler sends a *breakpoint hit* message to the RS and remains in SMM until further debugging commands are received. Once SMM has control of the SUT, we configure the next SMI to occur on the CPU via performance counters so

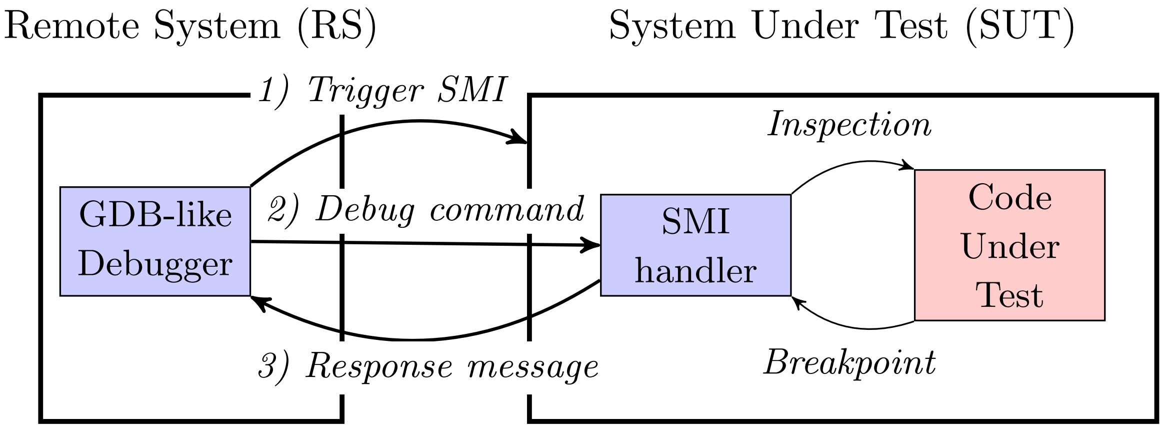


Figure 4.1: Architecture of MalT. The RS starts an analysis session by sending an initial *trigger SMI* message to the SUT. The SUT waits for the RS to provide a debugging command and then responds as required. Subsequent SMIs are triggered to repeat the process while the Code Under Test executes.

that the debugging process can continue during the life of execution of the Code Under Test. Next, we describe each component of the MalT system.

4.1.1 Remote System in MalT

The client on the RS can ideally implement a variety of popular debugging options. For example, we could use the SMI handler to implement the `gdb` protocol so that it would properly interface with a regular `gdb` client. Similarly, we might implement the necessary plugin for `IDAPro` to correctly interact with our system. However, this would require implementing a complex protocol within the SMI handler, which we leave for future work. Instead, we implement a custom protocol with which to communicate between the debugging client and the SMI handler. MalT implements a small `gdb`-like client to simplify our implementation. For the system restoration process, we store a clean disk image on the debugging client machine. Section 4.2.7 explains this in detail.

4.1.2 System Under Test in MalT

The SUT consists of two parts—the SMI handler and the Code Under Test. The SMI handler implements the critical debugging features (e.g., breakpoints and architectural state reports), thus restricting the execution of debugging code to SMM. The debugging target executes in Protected Mode or its other usual execution mode. Since the CPU state is saved within SMRAM when switching to SMM, we can reconstruct useful information and perform typical debugging operations each time a SMI is triggered.

SMRAM contains architectural state information (e.g., register values) of the process that was running when the SMI was triggered. Since the SMIs are produced regardless of the running thread, SMRAM is frequently populated with state information from another process besides the Code Under Test. To find relevant state information, we must solve the well-known semantic gap problem. By bridging the semantic gap within the SMI handler, we can ascertain the state of the process executing in Protected Mode. This is similar to VMI systems [129]. We continue our analysis in the SMI handler only if the SMRAM state corresponds to the process we are interested in debugging. Otherwise, we exit the SMI handler immediately. Note that MalT does not require Protected Mode; SMM can be initialized from other x86 modes (e.g., Real Mode), but the semantics of various structures would be different.

4.1.3 Communication between SUT and RS in MalT

We define a simple communication protocol used between the RS and SUT. The detailed protocol is described in Table 4.1. The commands are derived from basic `gdb` stubs, which are intended for debugging embedded software. The commands cover the basic debugging operations upon which the client can expand. The small number of commands greatly simplifies the process of communication within the SMI handler. For the disk restoration process, the SUT requests a chunk location from the RS, which responds with that chunk's value. Section 4.2.7 provides further details.

4.2 Design and Implementation

The MalT system is composed of two main parts discussed in Section 4.1. In this section, we provide implementation details such as specific memory addresses and x86 instructions that are relevant to our prototype. We provide these details in the interest of reproducibility.

4.2.1 Debugging Client on the RS

The RS consists of a simple command line application. A user can direct the debugger to perform useful tasks, such as setting breakpoints. For example, the user writes simple commands such as `b 0xdeadbeef` to set a breakpoint at address `0xdeadbeef`. The specific commands are described in Table 4.1. We did not implement features such as symbols.[1] The RS uses serial messages to communicate with the SUT.

4.2.2 System Under Test in MalT

The target machine consists of a computer with a custom `Coreboot`-based [67] BIOS. We changed the SMI handler in the `Coreboot` code to implement a simple debugging server on the SUT. This custom SMI handler is responsible for typical debugging functions found in other debuggers such as `gdb`. We implemented remote debugging functions via the serial protocol to achieve common debugging functions such as breakpoints, step-by-step execution, and state inspection and mutation.

[1] Resolving symbols is a standard feature included in `gdb`. If we fully implemented `gdb` stubs, symbol resolution would be taken care of by the `gdb` client.

Table 4.1: MALT Prototype Communication Protocol.

Message format	Description
`R`	A single byte, R is sent to request that all registers be read. This includes all the x86 registers. The order in which they are transmitted corresponds with the Windows trap frame. The response is a byte, `r`, followed by the registers $r_1r_2r_3r_4...r_n$.
`mAAAALLLL`	The byte `m` is sent to request a particular memory address for a given length. The address, A, is a 32-bit little-endian virtual address indicating the address to be read. The value L represents the number of bytes to be read.
`Wr1r2r3...rn`	The byte W is sent to request that the SMI handler write all of the registers. Each value r_i contains the value of a particular register. The response byte, $+$ is sent to indicate that it has finished.
`SAAAALLLLV...`	The command, S, is sent when the debugger wants to write a particular address. A is the 32-bit, little-endian virtual address to write, L represents the length of the data to be written, and V is the memory to be written, byte-by-byte. The response is a byte, `+`, indicating that the operation has finished, or a `-` if it fails.
`BAAAA`	The B command indicates a new breakpoint at the 32-bit little-endian virtual address A. The response is `+` if successful, or `-` if it fails (e.g., trying to break at an existing address). If the SMI handler is triggered by a breakpoint, it will send a status packet with the single character, `B`, to indicate that the program has reached a breakpoint and is ready for further debugging. The SMI handler will wait for commands from the client until the Continue command is received, whereupon it will exit from SMM.
`C`	The C command continues execution after a breakpoint. The SMI handler will send a packet with single character, `+`.
`X`	The X command clears all breakpoints and indicates the start of a new debugging session.
`KAAAA`	The K command removes the specified breakpoint if it was set previously. The 4-byte value A specifies the virtual address of the requested breakpoint. It responds with a single `+` byte if the breakpoint is removed successfully. If the breakpoint does not exist, it responds with a single `-`.
`SI, SB, SF, SN`	Each command changes the stepping mode: SI → single instruction, SB → branches, SF → far control transfers, SN → near return instructions. The SUT responds with one characters, `+`.

4.2.3 Bridging the Semantic Gap in MalT

As with VMI systems [104], SMM-based systems encounter the well-known semantic gap problem. In brief, code running in SMM cannot understand the semantics of raw memory. The CPU state saved by SMM only belongs to the process that was running when the SMI was triggered. During step-by-step execution, there is a chance that another process is executing when the SMI occurs. Thus, we must be able to identify the target process so that we do not interfere with the execution of unrelated process. This requires reconstructing OS semantics. Note that MalT has the same assumptions as traditional VMI systems [129].

In Windows, we start with the Kernel Processor Control Region (`KPCR`) structure associated with the CPU, which has a static linear address, `0xffdff000`. At offset `0x34` of `KPCR`, there is a pointer to another structure called `KdVersionBlock`, which contains a pointer to `PsActiveProcessHead`. The `PsActiveProcessHead` serves as the head of a doubly and circularly linked list of Executive Process (`EProcess`) structures. The `EProcess` structure is a process descriptor containing critical information for bridging the semantic gap in Windows NT kernels.

In particular, the Executive Process contains the value of the CR3 register associated with the process. The value of the CR3 register contains the physical address of the base of the page table of that process. We use the name field in the `EProcess` to identify the CR3 value of the target application when it executes first instruction. Since malware may change the name field, we only compare the saved CR3 with the current CR3 to identify the target process for further debugging. Bridging the semantic gap in Linux is a similar procedure, but there are fewer structures and thus fewer steps. Previous work [131, 249] describe the method, which MalT uses to debug applications on the Linux platform. Note that malware with ring 0 privilege can manipulate the kernel data structures to confuse the reconstruction process, and semantic gap solutions suffer from this limitation as of 2014 [129]. As with

VMI systems, MALT assumes that malware does not mutate kernel structures for correctly bridging the semantic gap (cf. Threat Model in Section 1.6).

4.2.4 Triggering an SMI

The system depends upon reliable assertions of SMIs because the debugging code is placed within the SMI handler.

We can assert an SMI via software or hardware. The software method writes to an Advanced Configuration and Power Interface (ACPI) port to trigger an SMI, and we can use this method to implement software breakpoints. We can place an `out` instruction in the malware's code so that when the malware's control flow reaches that point, SMM begins execution so that malware can be analyzed. The assembly instructions are:

```
mov $0x52f, %dx; out %ax, (%dx);
```

The first instruction moves the SMI software interrupt port number[2] into the `dx` register, and the second instruction writes the contents stored in `ax` to that SMI software interrupt port (the value stored in `ax` is inconsequential). In total, these two instructions take six bytes: `66 BA 2F 05 66 EE`. While this method is straightforward, it is similar to traditional debuggers using INT3 instructions to insert arbitrary breakpoints. Some malware samples can checksum their own code to detect such modifications (i.e., this approach produces functional artifacts). The alternative methods described below are harder to detect by such self-checking malware.

In MALT, we use two hardware-based methods to trigger SMIs. The first uses a serial port to trigger an SMI to start a debugging session. For the RS to interact with the SUT and start a session, we reroute a serial interrupt to generate an SMI by configuring the redirection table in the I/O Advanced Programmable Interrupt Controller (APIC). We use serial port COM1

[2] The SMI port number is `0x2b` on Intel and `0x52f` in our chipset [220]

on the SUT whose Interrupt Request (IRQ) number is 4. We configure the redirection table entry of IRQ 4 at offset `0x18` in I/O APIC and change the Delivery Mode (DM) to be SMI. Therefore, an SMI is generated when a serial message arrives. The RS sends a triggering message, causing the target machine to enter SMM. Once in SMM, the RS sends further debugging commands to which the SUT responds. In MalT, we use this method to trigger the first SMI and start a debugging session on the SUT. We trigger the first SMI before starting to execute the Code Under Test because our Threat Model (Section 1.6) assumes the first instruction can compromises the SUT.

The second hardware-based method uses performance counters to trigger an SMI. This method leverages two architectural components of the CPU: performance monitoring counters and the Local Advanced Programmable Interrupt Controller (LAPIC) [12]. First, we configure the Performance Counter Event Selection (`PerfEvtSel0`) register to select the counting event. There is an array of events from which to select; we use different events to implement various debugging functionalities. For example, we use the Retired Instructions Event (`C0h`) to single-step the whole system. Next, we set the corresponding performance counter (`PerfCtr0`) register to the maximum value. In this case, if the selected event occurs, it overflows the performance counter. Lastly, we configure the Local Vector Table Entry (LVTE) in the LAPIC to deliver SMIs when such an overflow occurs. Similar methods [22, 223] are used to switch from a guest VM to the hypervisor VMX root mode.

4.2.5 Breakpoints

Breakpoints can be implemented via software and hardware. Software breakpoints allow for unlimited breakpoints, but they must modify a program's code, typically placing a single interrupt or trap instruction at the breakpoint. Self-checking malware can easily detect or interfere with such changes (e.g., by checksumming its own code). On the other hand, hardware breakpoints do not modify code, but there can only be a limited number of hardware breakpoints as restricted by the CPU hardware. Moreover, ring 0 malware can detect the presence of hardware breakpoints by accessing the corresponding hardware registers. Techniques such as VMPiRE [219] aim to address the limitations of breakpoints, but they ultimately rely on the OS and so are not effective against ring 0 malware. We believe transparent breakpoint insertion with ring 0 malware is an open problem.

MalT tackles this problem by using performance counters to generate SMIs. Essentially, we compare the Extended Instruction Pointer (EIP) register of the currently-executing instruction with the stored breakpoint address during each instruction. We use 4 bytes to store the breakpoint address and 1 byte for a validity flag. Thus, we need only 5 bytes to store such pseudo-hardware breakpoints. For each Protected Mode instruction, the SMI handler takes the following steps: (1) Check if the Code Under Test is the running thread when the SMI is triggered, 2) check if the current EIP is in the set of stored breakpoint addresses, 3) start to count retired instructions in the performance counter, and set the corresponding performance counter to the maximum value, 4) configure the LAPIC so that the next performance counter overflow generates a subsequent SMI.

Breakpoint addresses are stored in SMRAM, and thus the number of active breakpoints we can have is limited by the size of SMRAM. In our system, we reserve a 512-byte region from `SMM_BASE+0xFC00` to `SMM_BASE+0xFE00`. Since each hardware breakpoint takes 5 bytes, we can store a total 102 breakpoints in this region. If necessary, we can expand the total region of SMRAM by taking advantage of a region called `TSeg`, which is configurable via the `SMM_MASK` register [12]. In contrast to the limited number of hardware breakpoints on the x86 platform, MalT is capable of storing more breakpoints in a transparent manner.

4.2.6 Step-by-Step Execution Debugging

As discussed above, we break the execution of a program by using different performance counters. For instance, by monitoring the Retired Instruction event, we can achieve instruction-

Table 4.2: Stepping Methods Available in MalT

Performance Counter	**Description** [12]
Retired instructions	Counts retired instructions, plus exceptions and interrupts (each count as one instruction)
Retired branch	Includes all types of architectural control flow changes, including exceptions and interrupts
Retired mispredicted branch	Counts the number of branch retired that were not correctly predicted
Retired taken branches	Counts the number of taken branches that were retired
Retired taken branch mispredicted	Counts number of retired taken branch instructions that were mispredicted
Retired far control transfers	Includes far calls/jumps/returns, `IRET`, `SYSCALL` and `SYSRET` instructions, exceptions and interrupts
Retired near returns	Counts near return instructions (`RET` or `RET Iw`) retired

level stepping in the system. Table 4.2 summarizes the performance counters we used in our prototype. First, we assign the event to the `PerfEvtSel0` register to indicate that the event of interest will be monitored. Next, we set the value of the counter to the maximum value (i.e., a 48-bit register is assigned $2^{48} - 2$). Thus, the next event to increase the value will cause an overflow, triggering an SMI. Note that the -2 term is used because the Retired Instruction event also counts interrupts. In our case, the SMI itself will cause the counter to increase as well, so we account for that change accordingly. Incorrect configuration of this value can cause the SUT to become deadlocked.

Vogl and Eckert [223] also proposed the use of performance counters for instruction-level monitoring. Their system delivers a Non-Maskable Interrupt (NMI) to force a VM Exit when a performance counter overflows. However, their work is implemented on a hypervisor. MalT leverages SMM and does not employ any virtualization, which provide a more transparent execution environment. In addition, Vogl and Eckert incur a time gap between the occurrence of a performance event and the NMI delivery, while MalT does not encounter this problem. Note that the SMI has priority over both NMIs and maskable interrupts. Among the stepping methods listed in Table 4.2, instruction-by-instruction stepping achieves fine-grained tracing (and therefore high functional transparency), but at the cost of a significant performance overhead (and therefore higher timing artifacts. Using the Retired Near Returns event causes low system overhead, but it only provides coarse-gained debugging (thus potentially missing transient malware behavior as discussed in Section 3.4).

4.2.7 System Restoration

Restoring a system to a clean state after each debugging session is criticial to ensuring safe malware analysis on bare metal systems. In general, there are two approaches to restore a system: *reboot* and *rebootless*. The reboot approach needs only to reimage the non-volatile devices (e.g., hard disk or BIOS), but this is time-consuming. The rebootless approach must manually reinitialize the whole system state, including memory and disks, but takes less time. For the rebootless approach, hardware devices must be restored in addition to the disk and memory. Modern I/O devices as of 2016 now have their own processors and memory (e.g., GPU and NIC); quickly and efficiently reinitializing these hardware devices remains a challenging problem.

BareBox [139] used a rebootless approach to restore the memory and disk of the analysis machine. However, BareBox only focuses on user-level malware; it disables loading new kernel modules and prevents user-mode access to kernel memory. In other words, ring 0 malware can easily detect the presence of BareBox using a memory scan and manipulate the restoration process, while MalT can successfully operate in the presence of kernel and hypervisor rootkits. Additionally, BareBox does not fully restore the I/O devices (e.g., the internal memory of GPU). BareCloud [140] used LVM-based copy-on-write to restore a remote storage disk. MalT also supports a similar approach to restore the disk. However, this method introduces artifacts (e.g., configurations for the remote disk) that malware can detect, which violates the transparency goal.

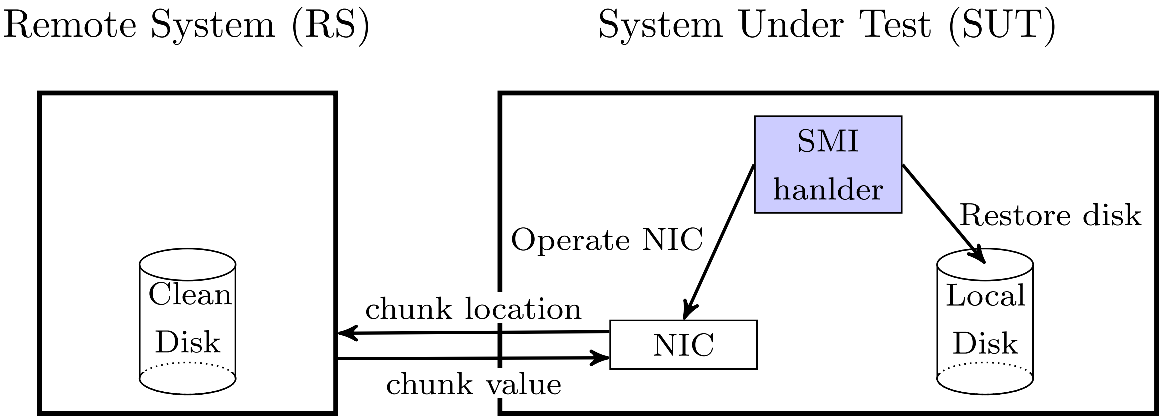


Figure 4.2: Workflow for Disk Restoration in MalT. When the SUT must restore its disk, it sequentially requests chunk locations from the RS, which stores a clean disk image. The RS responds with the clean values stored in the requested chunks until the entire disk is reimaged on the SUT.

System Restoration in MalT

To completely restore the SUT, we consider four components in MalT: (1) volatile memory (i.e., RAM), (2) I/O devices, (3) hard disks, and (4) the BIOS. For the first and second components, MalT uses the aforementioned reboot approach to restore them. Since we reboot the debugging server, the memory and I/O devices are reset to clean states. This addresses the problem of system restoration for malware analysis—I/O device and kernel memory can be reinitialized when ring 0 malware is present.

For the third component (hard disks), we reimage the disk by using SMM. Since the debugged malware is assumed to have ring 0 privilege in our Threat Model (Section 1.6), we cannot use the disk restoration tools provided by the OS or hypervisor. One simple solution is to remove the disk and restore it using another system. However, this is not convenient for users. In MalT, we use SMM as a trusted execution environment to remotely reimage the disk over the network. Figure 4.2 shows the workflow for disk restoration in MalT. The RS stores a copy of the clean disk. We use the SMI handler to copy the clean image from the RS to the SUT chunk-by-chunk over the network. As a result, this approach is OS-agnostic.

Since we do not trust any code in the OS (including device drivers), we write two simple device drivers in the SMI handler: one for the hard disk and the other for the network card. We use SATA-based hard disks. The I/O base address of the primary SATA controller is `0x38a0` on our SUT. We can access disk data by performing SATA read/write operations.[3] The BIOS and OS initializes the network card when booting, so we only need to write the transmit/receive descriptors to enable network communication.

Since replacing the whole disk image is time consuming (roughly 8 hours to replace a 500GB disk sector-by-sector on our platform), we only restore the modified contents on the disk. In this case, we use a bitmap to record the modified sectors. If we use a single bit for a sector, the bitmap requires 128MB for a 500GB disk with a sector size of 512-byte. To reduce the size of the bitmap, we group sectors into a 32KB chunk. This method is similar to BareBox [139]. To record the modified chunks, we trigger an SMI for each SATA operation. Specifically, we configure IRQ 14 in the I/O APIC to reroute a SATA controller interrupt to become an SMI. Next, we check the SATA operation to see if it is a write. If it is, we mark the bitmap at the corresponding location. This bitmap is stored in the trusted SMRAM. When the SUT requires disk restoration, we can use the bitmap to request clean versions of the chunks that were modified by the SUT from the RS.

Finally, for restoring the BIOS, we use the tool `Flashrom` [93] to flash firmware. We need to flash the BIOS twice per debugging session; once after the debugging session to inject the modified SMM code into the BIOS; and once before the subsequent debugging session to remove the presence of the modified SMM code to maintain transparency. Section 4.5 discusses this further.

[3] In addition, SATA access is performed at the block-level with 512-byte sectors on our platform.

4.3 Experimental Evaluation of MalT

In this section, we describe the details of the testing setup (i.e., the specifications of the SUT and RS. We describe our experimental parameters and present results. In particular, we present three targeted experiments:

1. We analyze the time taken per invocation of SMM to determine the time taken to perform our introspection task.
2. We examine how much overhead is incurred on a system-wide basis as a result of using MalT on indicative workloads.
3. We measure the overhead incurred by transparently restoring the SUT's disk to a known clean state.
4. We analyze the transparency of MalT when applied to analyzing stealthy malware.

4.3.1 Testbed Specification and Code Size

We evaluated MalT using two physical machines. The SUT used an ASUS M2V-MX_SE motherboard with an AMD K8 northbridge and a VIA VT8237r southbridge. We used a 2.2 GHz AMD LE-1250 CPU and 2GB Kingston DDR2 RAM. The SUT used Windows XP SP3, CentOS 5.5 with kernel 2.6.24, and Xen 3.1.2 with CentOS 5.5 as domain 0. To simplify the installation, they were installed on three separate hard disks using SeaBIOS to manage the boot process. We used Seagate Barracuda 7200 RPM hard disks with 500GB capacity. The RS is a Dell Inspiron 15R laptop with Ubuntu 12.04 LTS, with a 2.4GHz Intel Core i5-2430M CPU and 6 GB DDR3 RAM. We use a USB 2.0 to Serial (9-Pin) DB-9 RS-232 converter cable to connect two machines for communication.

We use `cloc` [62] to compute the number of lines of source code. `Coreboot` and its SeaBIOS payload contain 248,421 lines. MalT adds about 1900 lines of C code in the SMI hander.

After compiling the Coreboot code, the size of the image is 1MB, and the SMI hander contains 3749 bytes. The debugger client consists of 494 lines of C code.

4.3.2 Breakdown of Operations in MalT For Timing Analysis

To understand the performance of our debugging system, we measure the time elapsed during particular operations in the SMI handler. We use the Time Stamp Counter (TSC) to measure the number of CPU cycles elapsed during each operation; we multiplied the clock frequency by the delta in TSCs to obtain total time elapsed in seconds. After a performance counter triggers an SMI, the system hardware automatically saves the current architectural state into SMRAM and begins executing the SMI handler. The first operation in the SMI handler is to identify the last running process in the CPU. If the last running process is not the target malware, we only configure the performance counter register for the next SMI and exit from SMM. Otherwise, we perform several checks. First, we check for newly-received messages and whether a breakpoint has been reached. If there are no new commands and no breakpoints to evaluate, we reconfigure the performance counter registers for the next SMI. Table 4.3 shows a breakdown of the operations in the SMI handler if the last running process is the target malware. This experiment shows the mean, standard deviation, and 95% confidence interval of 25 runs. The time taken to switch into SMM is about 3.29μs. Command and breakpoint checking take about 2.19μs in total. Configuring performance monitoring registers and SMI status registers for subsequent SMI generation takes about 1.66μs. Lastly, resuming from SMM takes 4.58μs. Thus, when examining the target malware, MalT takes about 12μs to execute an instruction without debugging command communication. Note that the SMM switching and resume times are dictated by the hardware vendor's implementation of SMM. We would need to cooperate with hardware vendors to lower the time elapsed by these operations. This 12μs is assessed every time a performance counter overflows in MalT.

Table 4.3: Breakdown of SMI Handler (Time: μs)

Operations	**Mean**	**STD**	**95% CI**
SMM switching	3.29	0.08	[3.27,3.32]
Command and BP checking	2.19	0.09	[2.15,2.22]
Next SMI configuration	1.66	0.06	[1.64,1.69]
SMM resume	4.58	0.10	[4.55,4.61]
Total	11.72		

4.3.3 Step-by-Step Debugging Overhead

To demonstrate the efficiency of our system, we measure the performance overhead (i.e., timing artifacts) for each of the stepping methods shown in Table 4.2 on both Windows and Linux platforms. We use a benchmark program, `SuperPI` [213], version 1.8 on Windows and version 2.0 on Linux. `SuperPI` is a single-threaded benchmark that calculates the value of π to a specific number of digits and outputs the calculation time. This tightly written, arithmetic-intensive benchmark is suitable for evaluating CPU performance. We configure `SuperPI` to calculate 64K digits of π, which takes 1.610s and 1.898s on Windows and Linux, respectively, without instrumentation. Note that the speed of π calculation varies depending on the selected algorithm. Additionally, we use a popular Linux Command, `gzip`, to compress 4M digits of π computed by `SuperPi` to measure the performance overhead. On Windows, we install `Cygwin` to execute `gzip` version 1.4. On Linux, we use `gzip` version 1.3.5. The compression operation takes 1.875s on Windows and 1.704s on Linux without instrumentation. We then enable each of the stepping methods separately and record the runtimes to compute system overhead incurred by MalT. We run each experiment 5 times and show the average results in Table 4.4.

Table 4.4 shows the performance slowdown (and thus the extent of timing artifacts) introduced by step-by-step debugging. The first column specifies the stepping method; the following four columns show the running time of the `SuperPI` and `gzip`; and the last four columns represent the slowdown of the programs, which is calculated by dividing the instrumented running time by the base running time. The table shows that far control transfer (e.g., `IRET` instruction) stepping only introduces a 1.5x slowdown on Windows and Linux, which facilitates coarse-grained tracing for malware analysis. As expected, fine-grained stepping methods introduce more overhead. The instruction-by-instruction debugging causes about 1519x slowdown on Windows for `gzip` compressing 4M digits of π, which demonstrates the worst-case performance degradation among our debugging methods. One way to improve the performance is to reduce the time elapsed while switching to and resuming from SMM. Note that MalT is twice as fast as the state-of-the-art Ether approach [71, 241] in the single-stepping mode.

Despite a three order-of-magnitude slowdown on Windows, the SUT is still usable and responsive to user interaction. In particular, the instruction-by-instruction debugging is intended for use by a human operator from the RS, and we argue that the user would not notice this overhead while interacting with the RS.[4] We believe that achieving high functional transparency at the cost of sacrificing timing transparency is necessary for certain types of malware analysis. Note that the overhead in Windows is larger than that in Linux. This is because 1) the semantic gap problem is solved differently in each platform, and 2) the implementations of the benchmark programs are different for each platform.

4.3.4 System Restoration Overhead

To measure the overhead of a complete SUT restoration in MalT, we measure the time taken by each restoration step, including rebooting the SUT, restoring its disk, and flashing its BIOS. To calculate the time taken to reboot the SUT, we start an external timer when the OS executes the reboot command (i.e., `reboot` on Linux and `shutdown -h now` on Windows), and we stop the timer when the OS GUI is displayed after rebooting. Note that the boot time is OS-dependent. Since we only restore the changed chunks in the disk, the time taken

[4] To visualize the performance slowdown, we record a video (`https://youtu.be/NP6Bb4CdqN0`) that shows MalT operating in the instruction-stepping mode in Windows (cf. highest overhead in Table 4.4).

Table 4.4: Stepping overhead on Windows and Linux

Stepping method	**Runtime (Seconds)**				**Slowdown**			
	Windows		Linux		Windows		Linux	
	π	*gzip*	π	*gzip*	π	*gzip*	π	*gzip*
Without MalT	1.610s	1.875s	1.898s	1.704s	1.00x	1.00x	1.00x	1.00x
far control transfers	2.230s	2.564s	2.495s	2.432s	1.38x	1.36x	1.46x	1.42x
near returns	74.43s	73.14s	61.56s	59.08s	46.2x	39.1x	36.1x	34.7x
taken mispredicted	155.3s	145.7s	68.42s	138.3s	96.5x	40.2x	77.7x	81.2x
taken branches	1020s	1754s	476.6s	1538s	634x	935x	280x	903x
mispredicted branches	160.3s	280.0s	77.31s	236.3s	99.6x	149x	45.4x	138x
branches	1200s	2243s	494s	1760s	745x	1196x	290x	1033x
instructions	1645s	2849s	839s	2333s	1021x	1519x	492x	1369x

to restore the disk depends on the number of the modified chunks. In the experiment, we copy a 10MB file on the disk. Then, we use SMM to restore the disk. We use the TSC to measure the time elapsed for disk restoration and flashing the BIOS. Table 4.5 shows the breakdown of the system restoration process. Rebooting the system takes about 25 seconds. This includes shutting down the OS, initializing the BIOS, executing the boot loader, and loading the OS. Since we only need to restore the modified contents in the disk (i.e., 10 MB file and OS system logs), restoring the hard disk takes only 21 seconds. Recall that we need to flash the BIOS twice. Each flash operating takes about 28 seconds, yielding a total of about 56 seconds. Therefore, the total SUT restoration process takes about 102 seconds. Compared to BareBox [139], MalT requires more time to complete the system restoration. However, BareBox relies on a Meta-OS and only works for user-level malware, while MalT is capable of analyzing ring 0 code so that privileged malware cannot tamper with the restoration process. We believe our system provides a more transparent approach for system restoration in malware analysis.

Table 4.5: Breakdown of System restoration Process (Time: s)

Steps	**Mean**	**STD**	95% CI
System reboot	25.03	1.01	[24.01, 26.12]
Hard disk restoration	20.75	2.33	[17.39, 22.34]
BIOS flash	56.23	1.34	[54.55, 57.97]
Total	102.01		

4.4 Discussion and Limitations

MalT uses SMM as its foundation to implement various debugging functions. Before 2006, computers did not lock their SMRAM in the BIOS [86], and researchers used this flaw to implement SMM-based rootkits [45, 79, 86]. Since then, computers lock the SMRAM in the BIOS so that SMRAM is inaccessible from any other CPU modes after booting. Wojtczuk and Rutkowska demonstrated bypassing the SMRAM lock through reclaiming memory [196] or cache poisoning [236]. The memory reclaiming attack is addressed by locking the remapping registers and Top of Low Usable DRAM (TOLUD) register. The cache poisoning attack forces the CPU to execute instructions from the cache instead of SMRAM by manipulating the Memory Type Range Register (MTRR). Duflot also independently discovered this architectural vulnerability [80], but this issue was fixed with Intel's introduction of the System Management Range Register (SMRR) [127]. Furthermore, Duflot *et al.* [81] listed some design issues of SMM, but they are addressed by correcting configurations in BIOS and careful implementation of the SMI handler. Wojtczuk and Kallenberg [234] presented an SMM attack by manipulating a UEFI boot script that allows attackers to bypass the SMM lock and modify the SMI handler with ring 0 privilege. Fortunately, as discussed in Wojtczuk and Kallenberg [234], the BIOS update around the end of 2014 fixed this vulnerability. In MalT, we assume that SMM is trusted (cf. trusted hardware in the Threat Model in Section 1.6.

Butterworth et al. [49] demonstrated a buffer overflow vulnerability in the BIOS updating process in SMM, but this is not an architectural vulnerability and is specific to that particular BIOS version (our SMM code does not contain that vulnerable code). Since MalT adds 1500 lines of C code in the SMI handler, it is possible that our code has bugs that could be exploited. Fortunately, SMM provides a strong isolation from other CPU modes (i.e., sealed memory). The only inputs from a user are through serial messages (see Section 4.1), making it difficult for malicious code to be injected into our system. We implement MalT on a single-core processor for compatibility with `Coreboot`, but SMM also works on multi-core systems [127]. Each core has its own set of Manufacturer Specific Registers (MSRs), which define the SMRAM region. When an SMI is generated, all the cores will enter SMM with their own SMI handler. One simple way is to let one core execute our debugging code and spin the other cores until the first has finished. SMM-based systems such as HyperSentry [22] and SICE [23] are implemented on multi-core processors. In a multi-core system, MalT can debug a process by pinning it to a specific core while allowing the other cores to execute the rest of the system normally. This will change thread scheduling for the Code Under Test by effectively serializing its threads, which may be detectable by an adversary (i.e., a functional artifact). Intel introduced its SMM-Transfer Monitor (STM) technology, which virtualizes the SMM code [127]. It addresses attacks against Trust Execution Technology (TXT) [235]. Unfortunately, the use of an STM involves blocking SMIs, which potentially prevents our system from executing. However, we can modify the STM code in SMRAM, which executes in SMM, to provide our transparent debugging functionality without affecting our system.

4.4.1 Evaluating Transparency Against Packing Malware

Packing is used to obfuscate the binary code of a program. It is typically used to protect the executable from reverse-engineering. As of 2016, malware writers also use actively-studied packers for obfuscation [29, 132]. Packed malware is more difficult for security researchers

Table 4.6: Running packed `Notepad.exe` under different environments.

Packing Tool	**MalT**	**OllyDbg**	**DynamoRIO**	**VMware Fusion**
UPX v3.08	✓	✓	✓	✓
Obsidium v1.4	✓	✗ (access violation)	✗ (segfault)	✓
ASPack v2.29	✓	✓	✓	✓
Armadillo v2.01	✓	✗ (access violation)	✗ (crash)	✗(crash)
Themida v2.2.3.0	✓	✗ (exception)	✗ (exception)	✗(no VM)
RLPack v1.21	✓	✓	✓	✓
PELock v1.0694	✓	✗	✗ (segfault)	✓
VMProtect v2.13.5	✓	✗	✓	✗ (crash)
eXPressor v1.8.0.1	✓	✗	✗ (segfault)	✗ (crash)
PECompact v3.02.2	✓	✗ (access violation)	✓	✓

to reverse-engineer than binary code. In addition, many packers contain anti-debugging and anti-VM features (see Table 1.1 from Chapter 1, further increasing the challenge of reverse-engineering packed malware).

To demonstrate the transparency of MalT, we use popular packing tools to pack the `Notepad.exe` application in a Windows environment and run this packed application in MalT with near return stepping mode. We also compare against `OllyDbg` [244] v1.10, `DynamoRIO` [84] v4.2.0-3, and a Windows XP SP3 guest in `VMware Fusion` [221] v6.0.2. Ten packing tools are used, including `UPX` v3.08 [6], `Obsidium` v1.4 [3], `ASPack` v2.29 [1], `Armadillo` v2.01 [239], `Themida` v2.2.3.0 [169], `RLPack` v1.21 [192], `PELock` v1.0694 [4], `VMProtect` v2.13.5 [7], `eXPressor` v1.8.0.1 [54], and `PECompact` v3.02.2 [38]. All these packing tools were run with anti-debugging and anti-VM functions where appropriate. Running the packed `Notepad.exe` successfully executes if the `Notepad` window appears after unpacking. Table 4.6 lists the results. All the packing tools except `UPX`, `ASPack`, and `RLPack` can detect `OllyDbg`. `Obsidium`, `Armadillo`, `Themida`, `PELock`, and `eXPressor` are able to detect `DynamoRIO`, and the VM can be detected by `Armadillo`, `Themida`, `VMProtect`, and `eXpressor`. In contrast, MalT remains transparent to all these packing tools as we expected.

4.5 Transparency Analysis

In this section, we consider the transparency from four perspectives: 1) virtualization, (2) emulation, (3) SMM, and (4) debuggers.

Virtualization: Transparent virtualization is difficult to achieve. For instance, Red Pill [193] uses an unprivileged instruction, `SIDT`, to read the interrupt descriptor (IDT) register to determine the presence of a virtual machine. To work on multi-processor systems, Red Pill needs to use the `SetThreadAffinityMask()` Windows API call to limit thread execution to one processor [186]. nEther [10] detects hardware virtualization using CPU design defects. Furthermore, there are many footprints introduced by virtualization such as well-known strings in memory [57], magic I/O ports [139], and invalid instruction behaviors [24]. Moreover, Garfinkel *et al.* [102] argued that building a transparent virtual machine is impractical.

Emulation: Researchers have used emulation to analyze malware. QEMU simulates all the hardware devices, including the CPU, and malware runs atop the emulated software. Because of the emulated environment (i.e., functional artifacts), malware can detect it. For example, accessing a reserved or unimplemented MSR causes a general protection exception, but QEMU does not raise an exception [188]. The underlying problem is that the emulator was not designed specifically for transparent malware analysis (e.g., the emulator architect may not implement CPU errata). Table 1.1 in Chapter 1 shows more anti-emulation techniques. In theory, these defects could be fixed, but it is impractical to patch all of them in a timely manner.

SMM: As explained in Section 2.2 of Chapter 2, SMM is a hardware feature existing in all x86 machines. Regarding its transparency, the Intel manual [127] specifies the following mechanisms that make SMM transparent to application programs and operating systems: 1) the only way to enter SMM is by means of an SMI, 2) the processor executes SMM code in a separate address space (SMRAM) that is inaccessible from the other operating modes; 3) upon entering SMM, the processor saves the context of the interrupted program or task; 4) all interrupts normally handled by the operating system are disabled upon entry into SMM; and 5) the `RSM` instruction can be executed only in SMM. Note that SMM effectively steals CPU time from the Code Under Test, which introduces a potentially-measurable timing artifact. Even so, SMM is still more functionally transparent than virtualization and emulation.

Debuggers: An array of debuggers have been proposed for transparent debugging. These include in-guest [126, 219], emulation-based [15, 203], and virtualization-based [70, 71] approaches. MalT is an SMM-based system. As for transparency, we only consider the artifacts introduced by debuggers themselves, not the environments (e.g., hypervisor or SMM). Ether [71] proposes five formal requirements for achieving transparency, including 1) high privilege, 2) no non-privileged side effects, 3) identical basic instruction execution semantics, 4) transparent exception handling, and 5) identical measurement of time. MalT satisfies the first requirement by running the analysis code in SMM with ring -2. We enumerate all the artifacts introduced by MalT in Section 4.5.1 and attempt to meet the second requirement in our system. Since MalT runs on the bare metal, it immediately meets the third and fourth requirements. Lastly, MalT partially satisfies the fifth requirement by adjusting the local timers in the SMI handler to reduce the prevalence of timing artifacts. We further discuss the timing artifacts below.

4.5.1 Artifacts Introduced by MalT

MalT transparently analyzes malware with minimal functional artifacts. In this section, we enumerate artifacts introduced by MalT and show how we mitigate them. Note that achieving the highest level of transparency requires MalT to run in single-stepping mode.

CPU: We implement MalT in SMM, which provides an isolated environment for executing code. After recognizing the SMI assertion, the processor saves almost the entirety of its state to SMRAM. As previously discussed, we rely on the performance monitoring registers and the LAPIC to generate SMIs. Although these registers are inaccessible from user-level malware, attackers with ring 0 privilege can read and modify them. LAPIC registers in the CPU are memory-mapped[5] In MALT, we relocate LAPIC registers to another physical address by modifying the value in the 24-bit base address field of the `IA32_APIC_BASE` Model Specific Register (MSR) [127]. To find the LAPIC registers, attackers would need to read `IA32_APIC_BASE` MSR first, which we can intercept. Performance monitoring registers are also MSRs. `RDMSR`, `RDPMC`, and `WRMSR` are the only instructions that can access the performance counters [12] or MSRs. To mitigate the artifacts exposed from these MSRs, we run MALT in the per-instruction mode and adjust the return values seen by these instructions before resuming Protected Mode. If we find a `WRMSR` attempting to modify the performance counters, the debugger client on the RS is notified.

Memory and Cache: MALT uses an isolated memory region (SMRAM) from normal memory in Protected Mode. Any access to this memory in other CPU modes will be redirected to VGA memory. Note that this memory redirection occurs in all x86 machines, even without MALT; this is not unique to our system. In 2009, Intel introduced System Management Range Registers (SMRR) [127] that limits cache references of addresses in SMRAM to code running in SMM to address cache poisoning attacks [236]. The AMD64 architecture does not have SMRR, but the processor internally keeps track of SMRAM and system memory accesses separately and properly handles situations where aliasing occurs (i.e., main-memory locations as aliases for SMRAM locations) [13]. MALT does not flush the cache when entering and exiting SMM to avoid cache-based side-channel detection.

I/O Configurations and BIOS: MALT reroutes a serial interrupt to generate an SMI to initialize an analysis session, and the modified redirection table entry in the I/O APIC can be read by malware with ring 0 privilege. We revert the redirection table entry to its original value to remove this functional artifact in the first SMI handler. Once SMM has

[5] LAPIC memory-mapper registered reside at base address `0xFEE00000`.

control of the system, the SMIs are triggered by configuring performance counters. MalT uses `Coreboot` to program the SMM code. An attacker with ring 0 privilege can checksum the BIOS to detect the presence of our system. We flash the BIOS with the original image before executing the Code Under Test to address this functional artifact. At that time, the SMI handler, including the MalT code, has been loaded into SMRAM and locked. Note that we also need to reflash the `Coreboot` image for the next system restart.

Timing: There are many timers and counters on the motherboard and chipsets, such as the Real Time Clock (RTC), the Programmable Interval Timer (8253/8254 chip), the High Precision Event Timer (HPET), the ACPI Power Management Timer, the APIC Timer, and the Time Stamp Counter (TSC). Malware can read a timer and calculate its running time. For configurable timers, we record their values after switching into SMM. When SMM exits, we set the values back using the recorded values minus the SMM switching time. Thus, malware is unaware of the time spent in the SMI handler using interally-present timers. However, some of the timers and counters cannot be changed even in SMM. To address this problem, we adjust the return values of these timers in the instruction-level stepping mode. For example, the `RDTSC` instruction reads the TSC register and writes the value to the `EAX` and `EDX` registers. While debugging, we check if the current instruction is `RDTSC`, and adjust the values of `EAX` and `EDX` before leaving the SMI handler. This approach provides the illusion that the SMI handler is not consuming CPU time based on cycle count alone.

Unfortunately, MalT cannot eliminate timing artifacts involving an external timer. For instance, malware can send a packet to a remote server to get correct timing information (e.g., Network Time Protocol (NTP) service). One potential solution to address this type of timing artifact is to intercept the instruction that reaches out for timing information and prepare a fake time for the OS. Naturally, this would not be foolproof as an attacker could retrieve an encrypted time from a remote location. Such artifact are difficult to eliminate because we cannot always know when a particular packet contains timing information. To the best of our knowledge, all existing debugging systems with any measurable performance slowdown suffer from this attack. We believe that malware will avoid using external timing artifacts precisely because it wants to minimize its footprint on the victim's computer. If a malware sample uses an external timing artifact, we could consider using an alternative approach such as LO-PHI (see Chapter 3).

4.5.2 Analysis of Anti-debugging, -VM, and -emulation Techniques

To analyze the transparency of the MalT component, we employ anti-debugging, anti-virtualization and anti-emulation techniques from established work [24, 57, 88, 186, 188] to verify this component. Since MalT runs on a bare metal system, these anti-virtualization techniques fail to work in this component. Additionally, MalT does not change any code nor does it affect the running environment, operating system, or applications so that normal anti-debugging techniques cannot work against it. For example, the debug flag in the `PEB` structure on Windows will not be set while MalT is running. Table 1.1 in Chapter 1 summarizes popular anti-debugging, anti-virtualization, and anti-emulation techniques. Through rigorous experimentation, we have verified that MalT can evade all these detection techniques.

4.5.3 Concluding Remarks on the Transparency of MalT

Our experimentation has demonstrated that MalT provides a transparent malware analysis tool that introduces few functional artifacts while introducing timing artifacts. In this section, we discuss potential future issues that may arise with MalT as a result of the ongoing cat-and-mouse-game in security research.

Functions and Code Added by MalT: Sections 4.5.2 and 4.4.1 show that existing anti-debugging, anti-VM, anti-emulation, and packing techniques cannot detect the presence of

MalT. This is because the current techniques do not target MalT's functions or code, so it is possible that future malware could detect MalT due to the ever-present cat-and-mouse-game between attackers and defenders. As for 'tomorrow's malware,' we enumerate and mitigate the artifacts introduced by MalT in Section 4.5.1. Note that mitigating all artifacts requires the high fidelity associated with single-stepping instructions. As with other debugging systems, MalT cannot eliminate external timing artifacts.

Running Environment Used by MalT: MalT is built on SMM, so its transparency depends on the implications of SMM usage. Since SMM is not intended for debugging, the hardware and software on the system may not expect this usage, which may introduce artifacts for attackers to detect MalT (e.g., performance slowdown and frequent switching to and from SMM). However, we believe using SMM is more transparent than using virtualization or emulation as done in previous approaches due to its minimal TCB and attack surface.

4.6 Concluding Remarks for MalT

In this Chapter, we presented MalT, a functionally transparent malware analysis component. We used a custom SMI handler in SMM to achieve per-instruction introspection granularity. In addition, MalT can transparently reimage a SUT's disk to a clean state to ensure safe malware analysis. MalT introduces a 12μs timing cost each time we execute our SMI handler code, resulting in slowdowns from 1.5x to 1519x on indicative workloads in Windows and Linux environments. We evaluated MalT against several packers equipped with anti-analysis techniques, and demonstrated that it is transparent to such behavior where existing state-of-the-art tools fail. In brief, MalT is an integral component in our stealthy malware analysis tool.

This Chapter and Chapter 3 discuss two alternatives to hardware-assisted introspection supporting the analysis of stealthy malware. Chapter 3 presented LO-PHI, a technique that incurs low timing artifacts but potentially misses transient malicious behavior and introduces functional artifacts. In this Chapter, we presented an alternative that achieves low functional artifacts while introducing timing artifacts. In the next Chapter, we discuss the tradeoff space that exists between maintaining transparency while sacrificing fidelity of introspection data.

假痴不癫。

Feign madness but keep your balance.

The Thirty-Six Strategems

5

Tradeoffs Between Transparency and Fidelity

Chapters 3 and 4 discussed two hardware-assisted approaches for transparently introspecting a physical system in support of analyzing stealthy malware. LO-PHI (Chapter 3) used a custom FPGA to rapidly acquire snapshots of SUT memory by spoofing DMA packets over the PCI Express bus, while MALT (Chapter 4) used SMM to instrument a physical SUT on a per-instruction basis. Each approach comes with its own benefits and drawbacks with respect to transparency. In particular, LO-PHI can acquire snapshots of memory every 3200 PCI Express bus cycles (see Chapter 3). As a result, some transient malicious behavior may be missed by LO-PHI if it completes within this 3200 cycle window. In brief, maintaining transparency comes at the cost of the fidelity of introspection data. In this Chapter, we discuss an approach to exploring this transparency-fidelity tradeoff space.

5.1 Approach

We present Hops, an approach for hardware-supported low-overhead asynchronous debugging and analysis. Hops is a technique that iteratively inspects snapshots of physical memory (recorded asynchronously via LO-PHI from Chapter 3 and MalT from Chapter 4) and recovers semantic information. Using a combination of operating systems and programming languages techniques, we locate local, global, and stack-allocated variables and their values. In addition, we determine the current call stack. Because we may operate on COTS optimized code and only assume access to memory (e.g., and not the CPU), our approach may not be able to report the values of some variables (such as those stored in registers) or some stack frames (such as those associated with inlined functions). Additionally, the limited rate at which we may poll for memory may impact availability of correct variable or stack trace information. This approach provides a rich set of information for common security analysis tasks in practice, which serves as the basis for determining the extent to which transparency can be preserved to achieve a certain degree of introspection fidelity.

We envision two use cases for this dissertation. First, in automated malware analysis and triage systems, we want to analyze a large corpus of malware samples as quickly as possible. Unfortunately, existing solutions to this problem depend on virtualization. For example, the popular Anubis [15] framework, which analyzes binaries for malicious behavior, depends on Xen to run its analysis. This dependency on virtualization allows stealthy or VM-aware malware to subvert the triage system (see Table 1.1 for a list of techniques stealthy malware can use to subvert analysis).

In contrast, we assume the presence of low-overhead sampling hardware (i.e., LO-PHI from

Chapter 3) that enables fast access to a host's memory. This dissertation, therefore, is charged with introspecting the process while it is running in the triage system. For such a triage system, we want to understand the SUT's behavior in part by reasoning about the values of variables and producing a dynamic stack trace during execution. For example, the system might inspect control variables and function calls to determine how a stealthy malware sample detects virtualization, or might inspect snapshots of critical but short-lived buffers for in-memory cryptographic keys. In this use case, the primary metric of concern is accuracy with respect to variable values and stack trace information.

Additionally, this use case may reduce the manual effort involved in reverse engineering state-of-the-art stealthy malware samples. This dissertation enables debugging-like capabilities that are transparent to the sample being analyzed. This power allows analysts to save time when reverse engineering the anti-VM and anti-debugging features employed by current malware so that they can focus on understanding the payload's behavior.

The second use case is for deployments of benign code that may contain vulnerabilities that could be exploited by a stealthy adversary. In this case, we have vulnerable software running and, if an exploit occurs, we want to know what memory locations are implicated in the exploit. Here, we have a limited amount of time between when malicious data is placed into program memory and when malicious behavior begins[1](this is a relaxation of the Threat Model in Section 1.6 for the sake of experimentation). Thus, we want to study which buffers may be implicated by malicious exploits in COTS software. In this use case, an additional metric is the relationship between the speed and accuracy of our asynchronous debugging approach: for example, we may require that accurate information about potentially-malicious data be available quickly enough to admit classification by an anomaly intrusion detection system [122, 137, 149].

In both use cases, we begin with a binary that we want to study. If the source code to

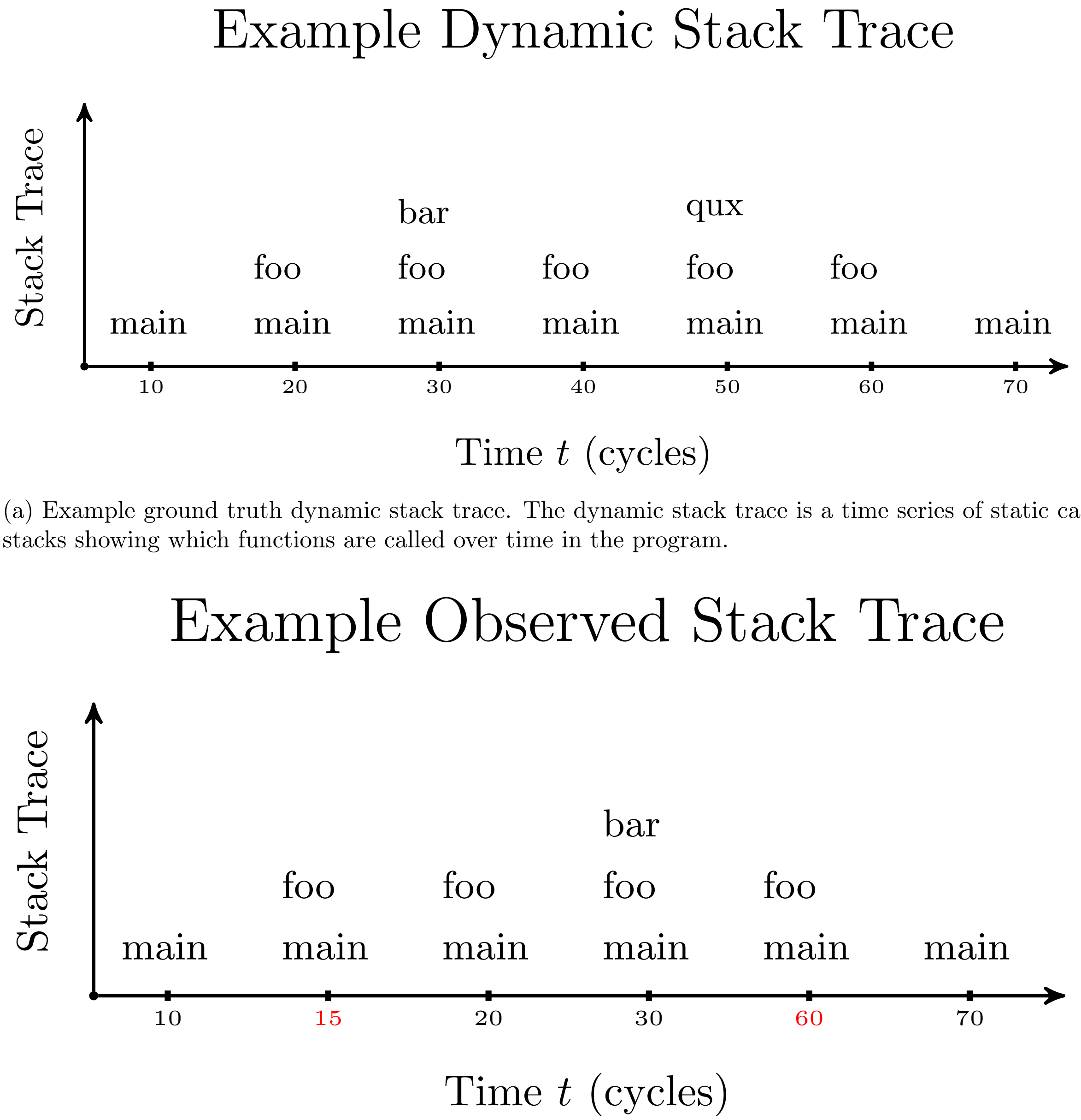
[1] Careto [136] is an example of malware that cleans up buffers during execution—without high-fidelity introspection data, we may miss observing this transient malicious behavior.

this binary is available (as is likely in the second use case but unlikely in the first), we take advantage of it to formulate hypotheses about variable locations. In any case, we desire to find and report 1) variables of interest in program memory, and 2) a dynamic stack trace of activation records to help understand the semantics of the program.

We assume that we have a binary compiled with unknown optimizations or flags (the *deployed* binary). For the purpose of experimentation, we also assume access to the source code for the binary to gather ground truth by compiling the binary with detailed variable and function call information (called the *instrumented binary*). This ground truth information is meant to simulate what an analyst would produce if given enough time to manually annotate a stripped binary. We compile the instrumented binary with symbol information to formulate the runtime memory locations of global and local variables and formal parameters. We then use this information as a guide for finding the same variables in the deployed binary. For instance, if global a is stored at offset x from the data segment in the instrumented binary, then a should be near x in the deployed binary.

We also assume the binary makes use of a standard stack-based calling convention (i.e., continuation passing style [16] is out of scope). Thus, we can reconstruct a stack trace by finding the stack in our memory snapshots and looking for the return address. We can then map the return address to the nearest enclosing entry point in the code segment using the symbol table generated in our test binary. Because the hardware assisted introspection components are based on polling for memory, reconstructing the stack trace in this manner could fail. If we do not poll frequently enough, we may miss activations altogether (e.g., if a function is called and quickly returns before we can poll). Figure 5.1 explains this tradeoff visually.

HOPS provides transparent introspection capabilities by asynchronously acquiring snapshots of program memory at runtime. Given such snapshots, we bridge the semantic gap (i.e., reconstruct important program data), converting from operating system information to process



(a) Example ground truth dynamic stack trace. The dynamic stack trace is a time series of static call stacks showing which functions are called over time in the program.

(b) This example set of stack trace observations demonstrates the tradeoffs between accuracy and transparency. Sampling too frequently may require additional resources or introduce artifacts without yielding additional information (e.g., sampling at $t = 15$ and $t = 20$). Conversely, sampling too coarsely may cause miss function calls (e.g., between $t = 30$ and $t = 60$).

Figure 5.1: Visual representation of (a) an idealized stack trace and (b) a stack trace generated by this component.

address spaces to variables, buffers, and stack frames of interest to the analyst. We build our transparent program introspection framework atop the hardware-assisted introspection tools discussed in Chapters 3 and 4. With the advent of commodity hardware and virtual machine techniques capable of quickly reading system memory as of 2016, we believe a transparent program introspection framework is now possible. In brieft, HOPS is a lightweight, native, accurate, asynchronous introspection technique that usefully supports analysis tasks. We use HOPS to investigate the tradeoffs between transparency and introspection fidelity.

We consider the transparency of our system and the artifacts it might produce, with a particular focus on timing artifacts. We measure the accuracy of our system's variable value and stack trace information against ground truth. Finally, we conduct a human study of 30 participants to evaluate our system's ability to support standard debugging tasks, as well as an expert case study to evaluate our system's ability to support a malware reverse engineering task. We find that HOPS is generally accurate for variable values and stack traces, is able to support conventional maintenance tasks as well as `gdb` can, and supports domain-specific reverse engineering tasks. In addition, because we build atop low-overhead approaches from Chapters 3 and 4, HOPS can be used in systems where traditional debuggers cannot apply.

5.2 Approach

5.2.1 Input Assumptions

We describe the assumptions of our approach that delineate its applicability.

Assembly Code

We assume access to the assembly code of the target executable, but not knowledge of the exact assembler or compiler flags or options used to produce that executable. For example, we may know that the target machine is running a particular version of the Apache HTTP Server, but not whether the deployed version was compiled with "–O2" or "–O3". We can make use of the source code, when applicable.

Memory Snapshots

Most critically, we assume access to periodic samples or snapshots of physical memory. Such snapshots could be provided via hardware-based approaches as discussed in Chapters 3 and 4. (e.g., [37, 51, 59, 211, 243]), or through VMI. In Section 5.4, we evaluate our approach using both LO-PHI and MalT techniques to gather memory snapshots.

We focus on LO-PHI (described in Chapter 3) as the ideal candidate for memory introspection because of its promising throughput rates. The rate at which we can obtain snapshots is limited theoretically by peak PCI Express transfer speeds (i.e., to 15.754GB/s, or roughly to 3.85 million page samples per second). In practice, however, most implementations can expect to achieve an effective transfer rate of 250MB/s per PCI lane, or 4GB/s total using 16 lanes [14]. This equates to 1.05 million page samples per second, or roughly 3200 cycles per page acquisition on our experimental platform (see Section 5.4).

Alternatively, we can acquire snapshots via SMM on x86 machines using MalT (Chapter 4). SMM remains available on extant x86 systems as of 2016, admitting broad applicability. Using SMM allows the capture of individual pages of memory at a time, albeit with high overhead (roughly 12μs to capture a page of memory).

In either case, we seek to reconstruct useful semantic information from these acquired snapshots, and subsequently evaluate the tradeoff between transparency and fidelity of semantic

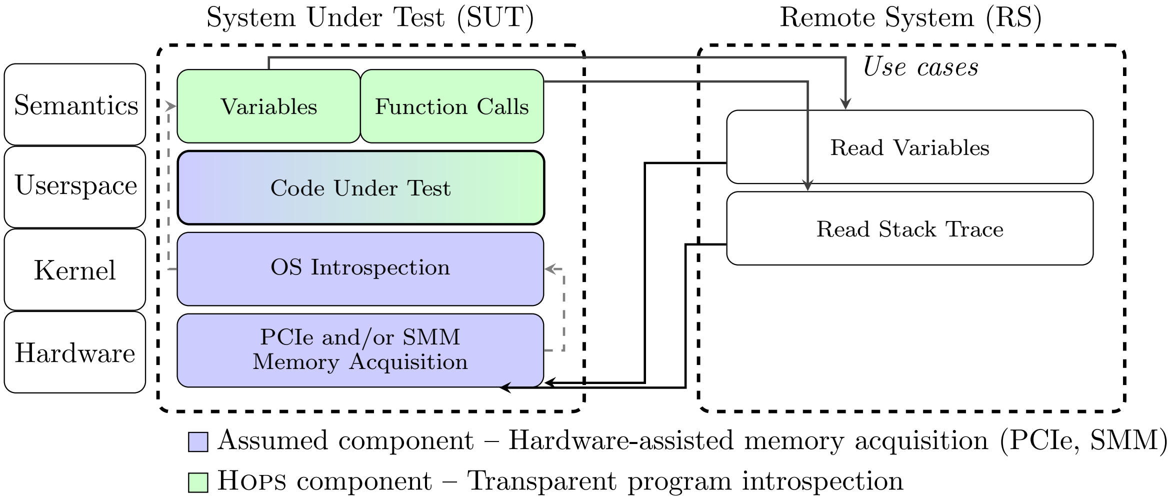


Figure 5.2: Architecture of the Hops system. We assume access to hardware (discussed in Chapters 3 and 4) that can transparently acquire snapshots of system memory from the SUT. Combined with OS introspection techniques [20, 249] to find the program being tested (Code Under Test), we use these snapshots to reason about variable and stack trace information in the Code Under Test on the RS.

information.

Normal Systems

We do not consider any modification to the target machine or the target executable beyond the hardware required to log memory snapshots. That is, one plug-in PCI device for LO-PHI or an unchanged x86 processor for SMM via MalT. In particular, we do not replace, patch, or otherwise change the target executable or the OS on the SUT.

5.2.2 Architecture

Our approach follows a pipelined architecture in which raw snapshots of physical memory are passed through a number of analyses, each of which refines or approximates the information available. The raw snapshots are collected at regular intervals. We focus on reporting the values of variables (local, global, or stack-allocated) as well as determining the dynamic stack trace. Figure 5.2 illustrates the approach. We start with raw snapshots and combine existing OS introspection techniques (e.g., Spectre [249], `Volatility` [20]) to reason about variable and stack trace information in a particular program being tested (denoted Code Under Test in the figure).

5.2.3 Physical Memory Snapshots

First, we direct the memory snapshot hardware (see Chapters 3 and 4) to log physical pages related to the target executable. This is a matter of bridging the semantic gap between raw bit patterns and logical program data and code. This is done by inspecting in-kernel data structures in physical memory to find the virtual address space mapping for the target process. From the virtual address space mapping, we can obtain a sequence of physical pages that correspond to the process's virtual address space. In addition, from in-kernel data structures, such as the process control block, we note the memory ranges associated with the stack segment, data segment, and code segment. Interpretation of relevant kernel data structures is well-described in the semantic gap literature [97,129,150,175,249]. Hops can use any such black-box analysis to bridge the semantic gap.

5.2.4 Reporting Variables

Given a correctly-ordered snapshot of a process's virtual address space as well as the locations of the various segments, our analysis proceeds by enumerating hypotheses about the locations of variables. These hypotheses are heuristically ordered by decreasing level of confidence. We start with the standard techniques used by debuggers.

For global variables, we use information from the symbol table (e.g., PE in memory for Windows executables, ELF for Linux). For locals, we focus on variables that were stack allocated (by the writer of the assembly code or by the compiler). We use debugging information when

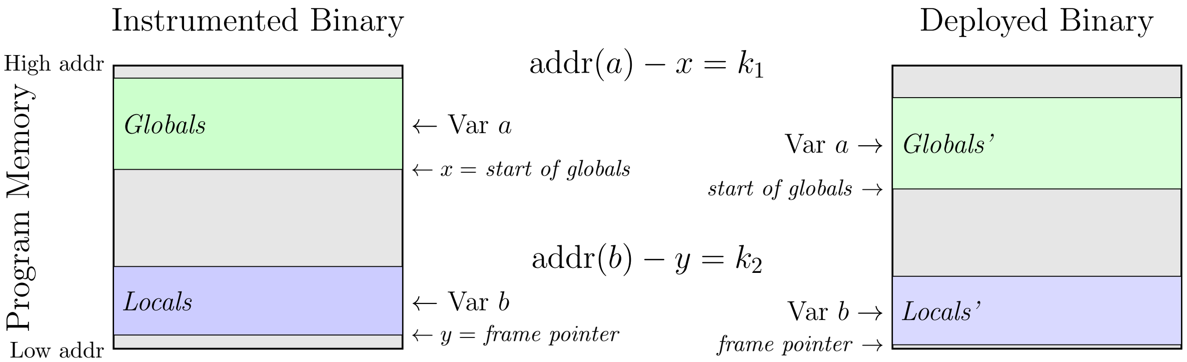


Figure 5.3: We hypothesize for some binaries that two variables exist at the same offsets between two different compiled version. In both versions, we hypothesize that a variable a exists at the same offset from the start of the globals (x in the figure). Similarly, we also hypothesize that stack allocated variables (b) exist at a fixed offset from the frame pointer (y).

available.

Some variables do not admit localization in this manner (e.g., variables that are stored in registers at certain optimization levels). In such cases, our second hypothesis is based on the relative location of the those variables in an instrumented binary on a test machine. Recall that we do not assume knowledge of the exact assembler, linker or compiler flags used to produce the target executable. Thus, we track the relative locations of variables and hypothesize that those offsets will be the same for the target executable. This hypothesis does not always hold, but it allows us to recover information for additional variables in practice. Figure 5.3 explains this hypothesis visually.

5.2.5 Reporting Stack Traces

In addition to the values of particular local and global variables, we also produce snapshots of the stack trace and the values of variables in caller contexts. We consider a number of standard calling conventions (e.g., `stdcall`, `cdecl`, `fastcall`) and attempt to locate the chain of activation records on the stack following standard debugger techniques [244]. For example, the name of the calling function is determined by finding the return address on the stack and mapping it to the nearest enclosing entry point in the code segment via the symbol table.

We also consider samples for which there is no source available. We can operate on labeled disassembly, as produced by `IDA Pro` or a similar tool. In such cases, synthetic label names or unique heuristic names are used [126] (e.g., "`printf`-like function #2" may label a function that behaves like the `printf` function). While walking the stack, we gather hypotheses about the locations and values of stack-allocated variables (such as the actual arguments or locals in-scope in parent contexts), as above.

Figures 5.1a and 5.1b illustrate our notion of a dynamic stack trace and the potential tradeoffs between transparency and introspection accuracy. We refer to the sequence of activation records (i.e., stack frames) present at a particular point in a program's execution as a static call stack. A *dynamic stack trace* of a program is a time series of static call stacks. Ideally, a dynamic stack trace includes a new static stack frame every time the code under test calls or returns from a procedure. Recording the static call stack after every instruction would observe every call and return, but would likely be resource- and timing-intensive and introduce anti-analysis artifacts (cf. 1500x overhead with MALT single-stepping in Section 4.3.3), Conversely, recording too few static call stacks results in a dynamic stack trace that may miss important behavior (cf. missing transient behavior in Section 3.4).

5.2.6 Output

One basic output of our technique is a dynamic stack trace: A sequence of static call stacks, one call stack per snapshot of physical memory. Each call stack lists the name of each called function and the values of its actual arguments. In addition, the analyst can request to inspect the value of a particular local, global, or stack-allocated variable. Both call stack reports and variable value inspections are asynchronous, as in MALT (Chapter 4).

Note that any presented information may be incorrect. For example, no information may be available about variables optimized away by the compiler or stored in registers. However, we hypothesize that optimized-away variables are not implicated in standard maintenance or security use-cases. For example, if the variable `x` is optimized away after `x=4;`, a developer with access to the source code may be able to reason about conditions involving `x` even if Hops cannot report its value. In security settings, if a variable in the source is optimized away and not present in the deployed code, it is unlikely that that variable could be used maliciously. Alternatively, we may miss variables due to the limited polling nature of our approach—for example, if we can only poll every 3200 cycles for a page of memory, we may miss changes to variable values within a 3200-cycle window.

Similarly, a hand-crafted leaf function that uses a non-standard calling convention (e.g., a custom hybrid of callee-saves and caller-saves for registers without pushing the return address) may not show up on a heuristic stack trace. However, such behavior is not commonly observed in general software systems (e.g., it is not easy to express in standard C), and in particular is not common in malware in the wild: stack and heap overflow, heap spray, and return-oriented programming (ROP [183]) and jump-oriented programming (JOP [39]) attacks depend on the traditional stack model for both Linux and Windows platforms.

5.3 Use Cases and Protocols

In this section, we describe intended use cases for Hops that set the stage for restrictions about the environment used by our technique. We describe the general implementation of the algorithm and protocol used to conduct our human study. Experiment-specific details are described in Section 5.4.

In both use cases, we begin with a binary that we want to study. If the source code to this binary is available (as is likely in the second and second cases but unlikely in the first), we take advantage of it to formulate additional hypotheses about variable locations (see Section 5.2.4). In all cases, we desire to find and report 1) variables of interest in program memory, and 2) a dynamic stack trace of activation records to help understand the semantics of the program, then use this information to evaluate the tradeoffs between transparency and fidelity of the reconstructed semantic data.

5.3.1 Security Analysis of Malicious Binaries

In automated malware triage systems, we desire to analyze a large corpus of malware samples as quickly as possible. Unfortunately, existing solutions to this problem depend on virtualization. For example, the common `Anubis` [15] framework, which analyzes binaries for malicious behavior, depends on Xen for virtualization, which allows stealthy or VM-aware malware to subvert the triage system. In contrast, our system assumes the presence of low-overhead sampling hardware that enables fast access to a host's memory. Our algorithm is thus charged with introspecting the malware process when running in the triage system. In such a triage system, we want to understand a sample's behavior in part by knowing the values of variables and producing a dynamic stack trace as the sample executes. For example, the system might inspect control variables and function calls to determine how the malware sample detects virtualization or might inspect snapshots of critical but short-lived buffers for in-memory keys. In this use case, the primary metric of concern is the fraction of critical malware aspects (e.g., artifacts used to evade analysis) an analyst can identify while supported by information from Hops. A secondary metric is accuracy with respect to variable values and stack trace information.

Similarly, we envision an extension of this use case for reducing the manual effort involved in reverse engineering state-of-the-art stealthy malware samples. Our system enables debugging-like capabilities that are transparent to the sample being analyzed. This power allows analysts to save time reverse engineering the anti-VM and anti-debugging features employed by current malware so that they can focus on understanding the payload's behavior.

5.3.2 Maintenance and Security Analysis of Benign Binaries

In addition to analyzing stealthy malware, Hops supports the analysis and maintenance of non-malicious software by providing information about variable values and stack traces. In particular, for benign software that may have vulnerabilities, we are interested in understanding how a hypothetical attacker compromises such software. We consider standard maintenance tasks such as fault localization, refactoring, or debugging that would normally be supported by a tool such as `gdb`. In this use case, the source code is likely to be available, but heisenbugs, timing dependencies, or similar issues still require the use of a transparent analysis technique. The primary metric is the fraction of maintenance questions the analyst is able to answer correctly when supported by information from Hops. Secondary metrics include Hops's accuracy when reporting variable values and stack traces, and Hops's transparency.

5.3.3 Human Study Protocol

The goal of our human study is to measure how well humans can perform debugging and maintenance tasks when supported by Hops—exploring our first use case, the maintenance analysis of benign programs. Simply put, we presented each participant with a snippet of code and the output from either Hops or `gdb`. We then measured participant accuracy on maintenance questions regarding that snippet. Multiple snippets were shown to each participant in a survey.

Participants were presented instructions for completing the survey, as well as example questions and possible answers. This instructive introduction helps address mistakes attributed to confusion or training effects. Each snippet was shown with corresponding output from a debugging tool—either from Hops or `gdb`—randomly selected for each participant on each question. Additionally, each snippet was shown with a corresponding question meant to test understanding of the snippet during execution. Participants were asked to answer the question in free form text. Finally, participants were presented with an exit survey asking for personal opinions on the debugging tools and experience. This study falls under IRB#2013-0466-00, which covers human assessments of artifacts and activities associated with debugging and maintenance. We describe participant selection, snippet selection, and question selection in detail.

Participant Selection

We required participants that have at least novice software development skills. We solicited responses from 24 third- and fourth-year undergraduate students enrolled in a computer security course and 6 graduate students. Participants were kept anonymous and were offered a chance to win one of two $50 Amazon gift cards or class extra credit (via randomized completion codes). We removed participants from consideration if they scored more than one standard deviation below the average score or if they failed to provide responses to all questions. We imposed this restriction due to the difficulty of controlling for C development and debugging experience. Participants were made aware of these requirements and that their potential reward depended on it.

Snippet Selection

The goal of the human study was to simulate debugging or maintenance in a controlled environment. We selected snippets of code from the two open source projects, `nullhttpd` 0.5.0 (1861 LOC) and `wu-ftpd` 2.6.0 (29,167 LOC). To create a snippet, we first randomly selected a function defined in the source code of each project. Only functions that were at least 10 lines long and were reachable by one of our test cases were considered [48, 96]. Similarly, functions longer than 100 lines were truncated to their first 100 lines. We then chose a random reachable point within that function.

The snippet thus consisted of that function, with the particular reachable point visibly marked as a breakpoint—as if the participant had placed a breakpoint on that line in a debugger and run the program until the breakpoint was reached and execution paused. As every snippet corresponded to a point in the test suite, debugging information was obtained by running the program on the test suite and invoking `gdb` or HOPS at that point. Ultimately, 23 snippets were created. Snippet counts and size limits were selected to ensure a reasonable completion time by the participants.

Software Maintenance Questions

This study measured how the information provided by our technique aids a developer when reasoning about code. We required participants to answer questions that are indicative of debugging activities that developers might ask during the maintenance process. Sillito *et al.* [200] identify several different types of questions real developers ask during maintenance tasks. Following previous human study protocols involving software maintenance [96], we used these three human study questions (HSQ):

HSQ1 What conditions must hold true to reach line X during execution?

HSQ2 What is the value of variable "`y`" on line X?

HSQ3 At line X, which variables are in scope?

Many questions discussed by Sillito *et al.* were general in nature and would not have been applicable for gauging participants' understanding of the snippets used in the study (e.g., one question reads "Does this type have any siblings in the type hierarchy?", which is not applicable to our subject C programs). Questions were randomly assigned to each snippet.

5.4 Experimental Evaluation of Hops

We consider four primary research questions (RQ) when evaluating Hops.

RQ1. On average, what fraction of local, global, and stack-allocated variable values can our system correctly report under multiple hardware regimes?

RQ2. How accurately can our system correctly report dynamic stack traces as a function of the asynchronous sampling rate of memory snapshots?

RQ3. Is the information provided useful for reasoning about debugging tasks compared to the state of the art?

RQ4. Could the information provided by our system help analysts reason about stealthy malware?

At any given point in time, some subset of the target program's variables are available. For RQ1, we measured success at each time step in terms of the fraction of those variables for which our technique reports the correct value (w.r.t. ground truth). Similarly, at any given point in time during execution, there is a particular stack of activation records. We further evaluated the performance of our technique when implemented atop both LO-PHI (Chapter 3) and MalT (Chapter 4) to establish our approach's feasibility on current hardware. For RQ2, we introduce a metric that requires functions to be reported and correctly ordered (w.r.t. ground truth). We then evaluated Hops in terms of this metric as a function of the sampling rate (i.e., how often asynchronous memory snapshots are made available). For RQ3, we conducted a human study in which 30 participants answer debugging questions about snippets of code using information from Hops and `gdb`. Finally, for RQ4, we considered a case study involving a VM-aware program sample that checks a number of different artifacts

Table 5.1: Description of test cases used in Hops experiments.

	# calls	Description	# calls	Description
Test		**nullhttpd**		**wu-ftpd**
1	239	GET standard HTML page	407	Change directory
2	239	GET empty file	453	Get /etc/passwd, text
3	239	GET invalid file	457	Get /bin/dmesg, binary
4	240	GET binary data (image)	22	Attempt executing binary
5	245	GET directory	267	Login with invalid user
6	385	POST information	22	Exploit: format string [8]
7	180	Exploit: buffer overrun [9]		

to detect analysis and report the fraction of those artifact queries that can be identified using Hops.

5.4.1 Experimental Setup and Benchmarks

We evaluate Hops using two indicative security-critical systems, `nullhttpd` 0.5.0 and `wu-ftpd` 2.6.0, each of which has an associated security vulnerability and publicly-available exploit. For `nullhttpd`, we consider a remote heap-based buffer overflow [9], while for `wu-ftpd`, we consider a format string vulnerability [8]. In addition to these exploits, for each program we consider non-malicious indicative test cases taken from previous research [147]. For example, one of the web server test cases retrieves a webpage, while one of the FTP server test cases logs in anonymously and transfers a file. Table 5.1 summarizes the test cases used in our experiments. As in Section 5.3, when the source code is available, our approach uses it to construct additional hypotheses about variable locations, but we do not assume that the compiler flags used in the deployed executable are known or the same. In these experiments, we simulate that disparity by gathering hypotheses from programs compiled with "–O2" but evaluating against different binaries produced with "–O3".

Evaluating RQ1 and RQ2 requires that we establish ground truth values of variables and stack traces at every program point. For the purposes of evaluation only, we employ source-level instrumentation [165] to gather these values. Because our approach is based on memory snapshots and local, global, and stack-allocated variables, we instrument and evaluate at all points where a variable enters or leaves scope or is placed on the stack, including all function calls, function entry points, and loop heads. We also separately instrument for timing information, using the Intel `rdtsc` instruction. Recording the timing information for instrumentation points introduces a small overhead (2% on average on these benchmarks). Note that any instrumentation overhead applies only to gathering ground truth information for our experimental evaluation and is not part of our general approach. Moreover, because we seek to analyze the tradeoffs associated with maintaining transparency and maximizing fidelity of introspection data, the overhead incurred in this stage does not apply to general transparent debugging tools.

We also note that, as with a standard debugger [126, 244], heap-allocated variables are accessed in our system by traversing expressions that start with local or global variables ("roots"). For example, after `glob_ptr = malloc(...)`, if an analyst wishes to inspect `glob_ptr->x`, the request is handled in four steps: 1) locate the (constant) address of `glob_ptr` in the data segment; 2) read the (dynamic) value stored there from the memory snapshot; 3) add the (constant) offset associated with `x`; and 4), read the (dynamic) value stored there from the memory snapshot. Accuracy on heap-allocated variable expressions thus reduces to accuracy on local, global, or stack-allocated variables.

In addition, for RQ4, we also consider Paranoid Fish (`pafish`, first introduced in Chapter 3) v04, an open source program for Windows that uses a variety of common artifacts to determine the presence of a debugger or VM, printing out the results of each check. Evaluating RQ4 requires that we know the ground truth set of artifacts that `pafish` considers to detect analysis. We obtain this set by manual inspection of the `pafish` code and comments.

Software and LO-PHI experiments concerned with RQ1 and RQ2 were conducted on a 3.2GHz Intel Xeon X5672 machine with 48GB RAM. This system uses 64-bit Linux 3.2.

SMM-based experiments were conducted using 32-bit Linux 2.6 on a system with an AMD Sempron CPU and 4GB RAM. In each case, we consider two versions of the same binary—ultimately, this means the runtime locations of variables will change due to address randomization. To facilitate experimental reproducibility and determinism, we disabled Address Space Layout Randomization (ASLR). However, our architecture admits accounting for ASLR by bridging the semantic gap in kernel memory to find the offset used for randomization.

For RQ4, we ran `pafish` on a 64-bit Windows 7 system with two Intel Xeon E5-2660 CPUs at 2.2GHz and 48GB RAM.

5.4.2 RQ1 — Variable Value Introspection

We evaluate the accuracy of Hops with respectto asynchronous requests for variable values: what fraction of variables will our technique accurately report, averaged over every variable in scope at every function call, function entry, and loop head? To admit a more fine-grained analysis of our technique, we partition the set of all in-scope program variables into local, global, and stack-allocated. For this experiment, the set of *local* variables for a function is all locally-declared variables including those in various nested local scopes, as allowed in C. The set of *global* variables for a function is all global variables in scope at that function (e.g., not including global variables declared after that function or `static` variables in other modules). The set of *stack-allocated* variables of a function includes the function's arguments as well as the local and stack-allocated variables of all (transitive) callers of the function. We do not consider variables that are optimized away by the compiler because variables that are not present at run-time are unlikely to be implicated by exploits and may admit maintenance reasoning via context clues (see Section 5.2 for a qualitative explanation, as well as Section 5.4.4 for a quantitative justification). At each point, we query the value of each variable using our system and report the result. The result is correct if it matches the

Table 5.2: Variable introspection accuracy.

	nullhttpd			**wu-ftpd**		
	Software	SMM	PCI	Software	SMM	PCI
Locals	43%	66%	41%	46%	48%	45%
Stack	65%	13%	58%	56%	31%	54%
Globals	100%	83%	96%	92%	93%	90%
Overall	69%	54%	65%	65%	57%	76%

ground truth: for strings, this takes the form of a string comparison while all other variables (e.g., integers, pointers) are compared numerically.

Table 5.2 reports the results. The "Software" columns record our accuracy via software simulation (rather than any special hardware). The "SMM" columns report our accuracy using MalT via SMM. The "PCI" columns report our accuracy using LO-PHI via PCI Express hardware.

For each program, the results are averaged over all test inputs (non-malicious indicative tests and one malicious exploit) and all relevant points (all function calls, all function entries, all loop heads). Specifically, these results help address the question, "if an analyst were to ask at a random point to introspect the value of a random variable, what fraction of such queries would our system be able to answer correctly?"

These results show 83–100% accuracy for global variables. This high introspection accuracy is because many of these variables are available at a constant location described in the subject program's symbol table. However, our approach still produces reasonable introspection accuracy for local and stack-allocated variable queries. For these variables, the values are not necessarily unavailable, but the hypotheses considered by our system do not account for differences caused by dynamic allocation of structures (or, indeed, whether compiler optimizations change the structures or layout altogether). Conversely, some values are not available to our technique based on its design assumptions (e.g., variables that live exclusively in registers). Over the three snapshot-gathering techniques, Hops answered 54–76%

of variable introspection queries correctly. We consider what these accuracy results mean in the context of supporting software maintenance questions in Section 5.4.4.

5.4.3 RQ2 — Stack Trace Introspection and Sampling Rate

For this research question we evaluated the portion of dynamic stack trace information that our technique can accurately report given a memory snapshot every k cycles. We report a single activation record as a tuple consisting of a function name and a list of actual argument values. A single static stack trace (at a given point in time) is thus a sequence (stack) of activation records.

We are ultimately interested in changes to the stack over time: a full dynamic stack trace is a sequence of static stack traces (each one corresponding to a point in time). For simplicity of presentation, we elide variable values (for which our accuracy is evaluated in RQ1) and denote a dynamic stack trace as a sequence of tuples (t, s) where t is the time in cycles and s the static call stack (e.g., $f_1 \rightarrow f_2 \rightarrow f_3$) corresponding to the activation records live at time t.

Our ground truth answer is equivalent to having a full memory snapshot available at every cycle. That is, a ground truth dynamic stack trace corresponds to 100% of the function calls that were invoked or returned from during that execution. For this experiment, we consider function calls only in userspace (e.g., program functions like `main` and library functions like `printf` are included, but actions taken by the kernel on behalf of system calls like `write` are not).

The metric used for this experiment is the number of function calls missed in our output stack trace that were present in the ground truth stack trace. For example, if the ground truth sample contains $\langle (1, f_1), (2, f_1 \rightarrow f_2), (3, f_1 \rightarrow f_2 \rightarrow f_3) \rangle$ and we sample at cycles $1, 3, 5, \ldots$, then our approach would report a stack trace missing the call to f_2 at $t = 2$. Our metric

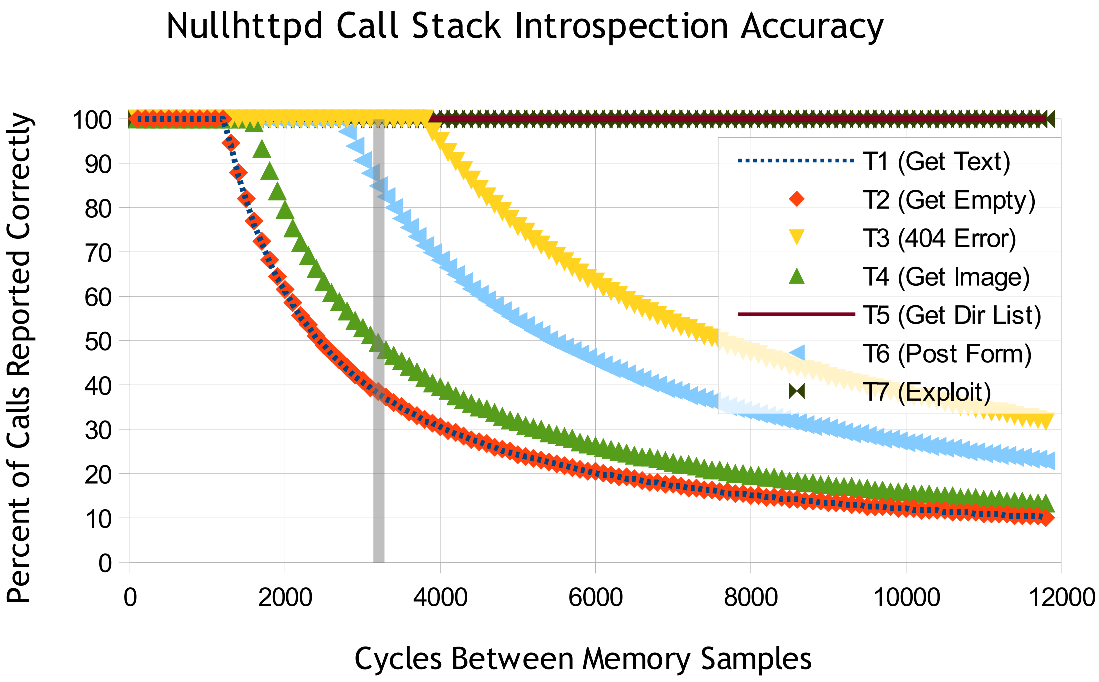


Figure 5.4: Call stack introspection accuracy for `nullhttpd` as a function of the number of machine cycles between memory samples. The reference line at 3200 corresponds to current hardware. On all tests sampling every 1200 cycles yields perfect accuracy.

counts the total number of such omissions. Because the stack trace length differs among test cases and programs, we normalize this value to 100%. Thus, a stack trace identical to the ground truth corresponds to an accuracy of 100%, while the example above, missing program behavior at $t = 2$, has an accuracy of 66%. In other words, the final value we report is $\frac{f-m}{f}$, where m is the number of misses and f is the number of function calls in the ground truth data. While other evaluation metrics are possible for dynamic stack traces (e.g., edit distance [164], largest common subsequence [33]), we prefer this metric because it is conservative and corresponds to our algorithmic framework (see Figure 5.1b).

Figure 5.4 reports the results for stack trace introspection for `nullhttpd`. As discussed in Section 5.2, current LO-PHI hardware can read roughly 1 million pages per second or 1 page every 3200 cycles. Thus, current hardware approximately corresponds to 3200 cycles between memory snapshots in these figures. Our introspection system loses accuracy when functions execute faster than the chosen inter-sample cycle count. Each test case causes

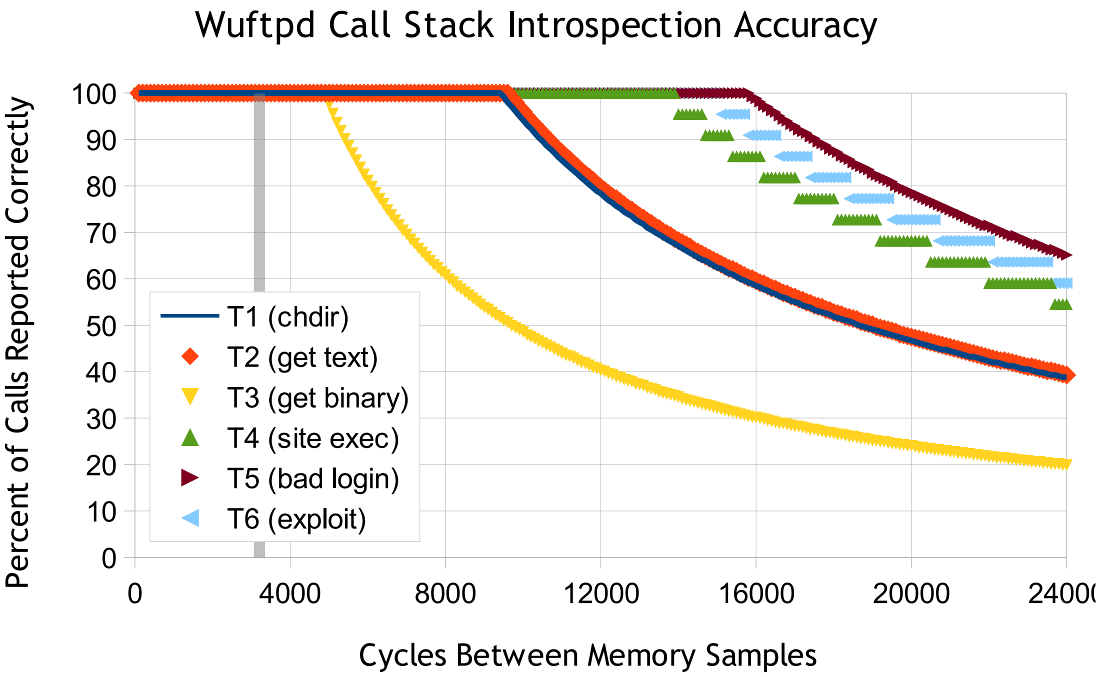


Figure 5.5: Call stack introspection accuracy for `wu-ftpd` as a function of the number of machine cycles between memory samples. The reference line at 3200 corresponds to current hardware. On all tests sampling every 4800 cycles yields perfect accuracy.

a different execution path to be taken, thus explaining the difference in results between test cases. For `nullhttpd`, we remain 100% accurate until the inter-sample cycle counter reaches approximately 1800 cycles. After this point, the accuracy steadily declines until the inter-sample cycle count exceeds the total execution time of the program—at that point, the accuracy is 0%. Note that with the 3200 cycle sample rate, we observe a stack trace accuracy over 50% for all test cases.

Figure 5.5 reports the accuracy for stack trace introspection for `wu-ftpd`. This program, which contains longer-running procedures, admits perfect call stack introspection up to a sampling interval of 4800. With available LO-PHI hardware, Hops would report 100% accurate stack traces for all test cases. For such programs and workloads, a faster sampling rate (i.e., a smaller inter-cycle rate) may allow for even greater introspection transparency.

5.4.4 RQ3 — Human Study

We conducted a human study to measure how helpful or informative Hops is to humans in practice. The study involved 30 participants (24 undergraduate and 6 graduate students). Each participant was shown 23 snippets of code and corresponding debugging output from either Hops or `gdb`, and then asked a debugging question. Each question was randomly selected from a pool of three questions (HSQ1–3 in Section 5.3.3). We measured the accuracy of each participant on each question as well as the time taken to answer each question. We calculate accuracy by manually grading each participant's responses: each answer given by each participant is assigned a score from 0.0 to 1.0. For HSQ1 and HSQ3, the correct answer may consist of multiple parts (e.g., multiple conditions may be required to reach a particular line of code). For these cases, the participant's score is the fraction of correctly identified conditions or variables. For HSQ2, the participant's score is either 0 or 1.

We divide our human study results into two groups: the `gdb` group (control) and the Hops group (treatment). We find that the control and treatment groups answered questions with an average accuracy of 59.2% and 68.0%, respectively.[2] The Hops treatment group is at least as accurate as the control group with statistical significance ($p < .01$ using the Wilcoxon rank-sum test). Additionally, the control group took an average of 105±6.4 seconds to answer each question while the treatment group took an average of 118±8.2 seconds on each question. Differences in timing were not statistically significant.

Because humans are at least as accurate using Hops when answering indicative software maintenance questions, we conclude that Hops could usefully support standard software maintenance analysis tasks. We do not claim that Hops is generally better than `gdb`—indeed, `gdb` has many features, such as changing variable values or poking memory, that Hops does not support. However, Hops is transparent, allowing it to be used in heisenbug or security-analysis tasks where `gdb` is inapplicable. Without such a tool, developers had

[2] Human study materials and anonymized responses are available at `http://church.cs.virginia.edu/hops-materials/all.tar.gz`.

little to no information in such situations; HOPS transparently provides information that is accurate enough to usefully support analysis tasks.

5.4.5 RQ4 — `pafish` Case Study

This case study evaluates the utility of the information provided by our approach in assessing the artifacts inspected by stealthy malware. We manually annotated `pafish` to collect ground truth data, as in RQ1 and RQ2.

`pafish` is particularly useful in evaluating HOPS because 1) it is is amenable to complete ground-truth annotation (unlike a wild malware sample, for which we could entirely miss a stealthy check and thus have false negatives) and 2) it helps answer RQ4 in a general manner (because it contains a large number of indicative artifact checks), which ultimately gives confidence that our tool applies to our considered use cases.

Using HOPS, we can introspect visible variables and dynamic stack traces as in RQ1 and RQ2. We consider the question: "are the variables and stack traces values that HOPS reports accurate enough to conclude which anti-debugging techniques `pafish` is employing?" While analyst skill plays a role in such tasks, for this evaluation we used a conservative criterion, indicating success only for cases in which variables and function calls directly implicating the artifact were introspected correctly. For example, calling the `OutputDebugString` method in Windows would cause an error if a debugger is *not* attached. HOPS reports the call to `OutputDebugString` (ultimately culminating in a `write`-like system call), as well as its parameter (a *stack* variable in RQ1). From this, an analyst could accurately determine the artifact being employed in this scenario (i.e., `OutputDebugString`'s conditional behavior).

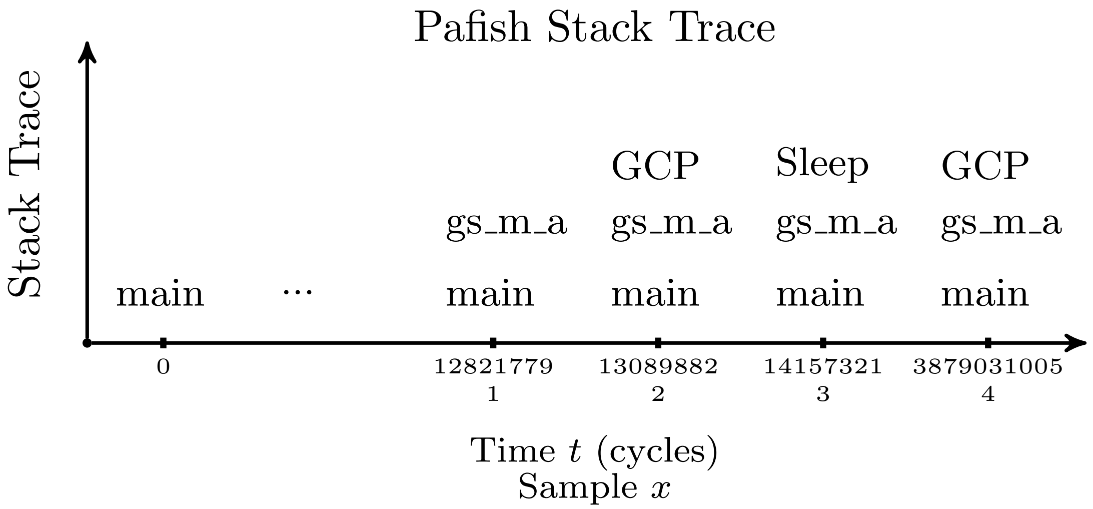
Table 5.3 summarizes which of the 22 artifacts considered by `pafish` can be detected with our stack tracing and variable introspection technique. A ✓ indicates that an analyst could

Table 5.3: List of artifacts used by `pafish`.

Method or Artifact	**Hops Success**
Debuggers	
IsDebuggerPresent	✗
CheckRemoteDebuggerPresent	✗
OutputDebugString	✓
General Sandboxes	
GetCursorPos	✓
GetUserName	✗
GetModuleFileName	✓
Disk legitimacy	✓
Disk size	✓
GetTickCount	✓
QEMU Registry Keys	
Device names	✓
BiosVersion	✓
Sandboxie	
sbiedll.dll	✗
VirtualBox	
Registry information	✓
Drivers	✓
MAC Address	✓
Window	✗
Processes	✓
VMWare	
Device names	✓
VMWare Tools	✓
Drivers	✓
Wine	
kernel32.dll features	✓
Hooking	
various	n/a

use introspection information from HOPS to determine that `pafish` is using the given anti-analysis method or artifact. A ✗ indicates that perfect variable or stack trace introspection information would allow the analyst to determine that the given anti-analysis method is being used, but in practice HOPS does not provide accurate information about the relevant variables or stack frames (i.e., we cannot sample quickly enough with current hardware). For example, the `IsDebuggerPresent` API call is very fast. As a result, our current sampling rate is too coarse to capture the calls to this function. In fact, all five of the failing cases result from too coarse a sampling rate. In these situations, HOPS could acquire accurate stack traces, and thus implicate the artifacts, with a faster sampling rate (i.e., improved hardware). Finally, some methods or artifacts are beyond the scope of our technique. For instance, checking for hooked functions does not require calling any functions at all (instead, it scans virtual addresses of API functions for particular signatures, using values in registers). However, HOPS requires activation records created by function calls or variables stored in memory, so even with perfect memory introspection accuracy, HOPS could not reveal such artifact usage. We refer to these types of artifacts as being not applicable to our approach and denote them with an "n/a" in Table 5.3.

Overall, Table 5.3 shows that we can accurately discern when `pafish` attempts to use 16 of the 22 artifacts in its suite. That is, for these 16 artifacts, our technique can be applied to provide useful information to an analyst who can conclude that a specific artifact was used. As an example, `pafish` uses the `GetCursorPos` API call to determine the position of the mouse cursor. It calls this twice and concludes it is being analyzed if the mouse does not move at all after 2 seconds. Figure 5.6 shows a run of our system against `pafish`. We elide other function calls and focus specifically on the code near its use of `GetCursorPos`. We show that HOPS allows an analyst to see the calls to `GetCursorPos` and `Sleep` (samples 2, 3, and 4 in the Figure). At these points, we can also correctly introspect the values of the stack allocated variables `p1` and `p2`, which are pointers to `POINT` structures with x and y fields that correspond to the cursor's position. In summary, our system permits the analyst



Code around sample 1

```
t = ...      int gensandbox_mouse_act(){
12821779       POINT p1, p2;
13089882       GetCursorPos(&p1);
14157321       Sleep(2000);
3879031005     GetCursorPos(&p2);
             if (p1.x==p2.x && ...)
               traced("found");
             else
3879559528     nottraced("not found");
```

Figure 5.6: Stack trace gathered against `pafish`, specifically focused on the `GetCursorPos` artifact. The top of the figure shows the stack trace acquired by Hops over time as a function of CPU clock cycles. At the bottom, the source code is annotated by the timestamp at which the line runs. At these points of time, we are capable of acquiring the values of structures `p1` and `p2`. The `traced` line is never executed during this run because the mouse was moved.

to see 1) the stack trace including the two `GetCursorPos` invocations, and 2) the variables in which the cursor's x and y positions are stored. This information implicates the exact artifact used (i.e., lack of mouse movement over time).

5.4.6 Evaluation Conclusions

This evaluation of our system against three programs (i.e., `nullhttpd`, `wu-ftpd`, and `pafish`) provides promising results in terms of accuracy and is indicative for the types of programs and workloads in our use cases. Our empirical results measure the tradeoff between introspection accuracy and transparency (sampling rate) for our low-artifact analysis technique. Essentially, Hops constructs meaningful debugging information (variables and stack traces) from raw memory dumps provided via low-artifact hardware (e.g., via LO-PHI). Recall that the ultimate goal of this system is to assist program analysis, whether for software maintenance, manual analysis, or automated triage. In this regard, Hops is 84% accurate over all variables and test cases considered (cf. Table 5.2, but note that there are more globals and stack-allocated variables than local variables). Third, in a human study involving 30 participants, information provided by Hops was no worse than information provided by `gdb` when supporting debugging questions with statistical significance. Lastly, we again demonstrate that Hops is useful in practice by testing it against `pafish`. Hops is capable of detecting when `pafish` uses 16 out of 22 artifacts during its execution.

By implementing Hops on two different hardware platforms, we demonstrated its generalizability under multiple hardware regimes. However, both hardware implementations have restrictions in terms of transparency. First, using MalT-based introspection incurs high overhead (roughly 12μs to access one page), meaning that a malware sample could measure time elapsed during execution. Secondly, LO-PHI potentially influences performance counters (see Chapter 3, which could be measured by stealthy malware with ring 0 privilege. Similarly, in systems where the PCI Express bus is under high load, the use of LO-PHI could adversely affect throughput.

5.5 Concluding Remarks for Hops

Many software systems, from embedded devices to virtualization to security, cannot make use of standard debuggers. The act of analyzing a system can change that system, leading to heisenbugs in benign software and admitting anti-analysis by stealthy malware. We thus focus on zero-overhead approaches that leave no functional artifacts or traces that a program could use to behave differently when analyzed. We presented Hops, an approach to program introspection that infers and reports variable values and dynamic stack traces from hardware-provided memory snapshots. Our approach is based on two key observations. First, it is possible, using existing hardware from Chapters 3 and 4, to log snapshots of memory pages with low to no overhead. Second, it is possible to bridge the semantic gap between raw memory snapshots and software semantics using a combination of program analysis, operating system, and security techniques.

Our approach formulates hypotheses about the locations of variables and stack frames, allowing analysts to introspect malicious and non-malicious programs. In our experiments, Hops was 84% accurate, overall, at reporting the values of local, stack-allocated, and global variables. We also implemented Hops using SMM-based MalT and PCI Express-based LO-PHI FPGA to test our hypotheses on real hardware. In addition, it was over 50% accurate at reporting entire dynamic stack traces using conservative memory timings associated with available hardware. Third, in a human study involving 30 participants with statistical significance, Hops was no worse than `gdb` at supporting the analysis of standard maintenance questions. Finally, we examined 22 methods or artifacts that can be used by stealthy malware to detect analysis and observed that the introspection information provided by Hops was sufficient to reveal 16 of 22 artifacts used by `pafish` (and could reveal 5 more with faster hardware). Overall, we see Hops as an approach towards transparent process introspection as well as a thorough investigation as to the tradeoffs that exist between maintaining transparency during analysis while obtaining high-fidelity introspection data that can be used in real malware analysis and software maintenance tasks.

我想不出别的典故了。

I ran out of quotes. Time to end it.

Kevin Leach

6

Conclusion

The proliferation of malware has increased dramatically in the past few years, seriously eroding user and corporate privacy and trust in computer systems [30, 133, 134, 158, 159]. This growing volume of malware requires analysis to build cogent defenses against it. Unfortunately, novel malware employs a variety of anti-analysis techniques by measuring artifacts introduced by the analysis tool. Such stealthy malware requires significant manual effort to analyze, sometimes taking as long as a person month to fully understand [205]. Additionally, automated techniques for triage and analysis such as `Cuckoo` and `Anubis` fail to execute some stealthy malware samples because they detect the underlying analysis framework. With the rapidly-growing volume of malware and the increasing effort required to analyze stealthy malware, there is a need to produce malware analysis tools that do not produce any artifacts and thus are transparent against detection by stealthy malware.

In this dissertation, we presented an end-to-end malware analysis system that admits the analysis of stealthy malware. We presented two approaches to minimize functional and timing artifacts, as well as an analysis of the tradeoff space that exists between maintaining transparency and fidelity of analysis data.

- In Chapter 3, we presented LO-PHI, which employs a custom FPGA circuit that uses DMA over PCI Express to rapidly acquire snapshots of a SUT's memory. We showed that LO-PHI produces no measurable timing artifacts (memory throughput within a margin of error), but potentially introduces functional artifacts (e.g., DMA access performance counter). We showed that LO-PHI was capable of analyzing large corpora of malware, including hundreds of stealthy malware samples that would evade existing approaches.
- In Chapter 4, we presented MalT, which employs a custom SMI handler to acquire snapshots of a SUT's memory on a per-instruction basis. We showed that MalT can eliminate functional artifacts as well as some internal timing artifacts (0 of 10 stealthy packers detected MalT), but causes significant system overhead (1514x in the worst case).
- In Chapter 5, we presented Hops, which helps to analyze the tradeoffs that exist between transparency and fidelity of useful semantic data produced by tools like LO-PHI and MalT. We showed, in a human study, that LO-PHI is capable of providing useful semantic data no worse than `gdb` for standard maintenance and analysis tasks ($p < .01$), but with the added desirable transparency property.

Table 6.1 lists several peer-reviewed publications that support the findings presented in this dissertation. The work in this dissertation addresses the unwieldy burden associated with

Table 6.1: Publications supporting this dissertation.

IEEE S&P 2015	Using Hardware Features for Increased Debugging Transparency [250] (Chapter 4)
TDSC 2016	Towards Transparent Debugging [248] (Chapter 4)
Patent	Towards Transparent Debugging [247]
NDSS 2015	LO-PHI: Low-Observable Physical Host Instrumentation [206] (Chapter 3)
SANER 2016	Towards Transparent Introspection [148] (Chapter 5)

analyzing stealthy malware in two ways: 1) making manual analysis faster by removing the need to consider the mechanism for stealth used by a sample of malware, and 2) enhancing the expressive power of automated analysis and triage systems by introducing low-artifact analysis techniques so that automated analyses can apply to stealthy malware samples. To that end, we have presented and rigorously evaluated two hardware-assisted introspection techniques that apply to stealthy malware and examined the tradeoffs associated with maintaining transparency while providing useful semantic information. We believe that the increasing volume of malware and the growing body of stealthy malware will require tools akin to LO-PHI or MalT to efficiently analyze for years to come. Additionally, we believe such systems will shift focus to include transparently instrumenting the CPU in addition to memory and disk. Moreover, with the growth of mobile devices, the ARM instruction set will become increasingly important to analyze transparently.

Bibliography

[1] ASPack. http://www.aspack.com. Retrieved November 2016.

[2] IOZone filesystem benchmark. http://www.iozone.org/.

[3] Obsidium. https://www.obsidium.de/show/download/en. Retrieved November 2016.

[4] PELock. https://www.pelock.com. Retrieved November 2016.

[5] pytsk, Python Bindings for the SleuthKit. https://github.com/py4n6/pytsk. Retrieved November 2016.

[6] UPX: The Ultimate Packer for eXecutables. https://upx.github.io. Retrieved November 2016.

[7] VMProtect. http://www.vmpsoft.com. Retrieved November 2016.

[8] CVE-2000-0573: Format string vulnerability. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2000-0573, 2000.

[9] CVE-2002-1496: Heap-based buffer overflow. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2002-1496, 2002.

[10] nEther: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the 4th European Workshop on System Security (EUROSEC '11)* (2011).

[11] Aderholdt, F., Han, F., Scott, S. L., and Naughton, T. Efficient checkpointing of virtual machines using virtual machine introspection. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on* (2014), IEEE, pp. 414–423.

[12] Advanced Micro Devices, Inc. BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors.

[13] Advanced Micro Devices, Inc. AMD64 Architecture Programmer Manual Volume 2: System Programming. http://support.amd.com/TechDocs/24593.pdf, June 2015.

[14] Altera Corporation. PCI Express High Performance Reference Design. http://www.altera.com/literature/an/an456.pdf, 2014.

[15] Anubis. Analyzing unknown binaries. http://anubis.iseclab.org.

[16] Appel, A. W. *Compiling with continuations.* Cambridge University Press, 2007.

[17] ARM. DSTREAM. `http://ds.arm.com/ds-5/debug/dstream/`.

[18] Asset Intertech. Processor-Controlled Test. `http://www.asset-intertech.com/Products/Processor-Controlled-Test`.

[19] Aumaitre, D., and Devine, C. Subverting windows 7 x64 kernel with dma attacks. *HITBSecConf 2010 Amsterdam 29* (2010).

[20] Auty, M., Case, A., Cohen, M., Dolan-Gavitt, B., Ligh, M. H., Levy, J., and Walters, A. Volatility framework - volatile memory extraction utility framework.

[21] Azab, A. M., Ning, P., Wang, Z., Jiang, X., Zhang, X., and Skalsky, N. C. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 38–49.

[22] Azab, A. M., Ning, P., Wang, Z., Jiang, X., Zhang, X., and Skalsky, N. C. HyperSentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)* (2010).

[23] Azab, A. M., Ning, P., and Zhang, X. SICE: A Hardware-level Strongly Isolated Computing Environment for x86 Multi-core Platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)* (2011).

[24] Bachaalany, E. Detect if your program is running inside a Virtual Machine. `http://www.codeproject.com/Articles/9823/Detect-if-your-program-is-running-inside-a-Virtual`.

[25] Baecher, P., and Koetter, M. Libemu-x86 shellcode emulation library. *See http://libemu. carnivore. it* (2007).

[26] Baig, M. B., Fitzsimons, C., Balasubramanian, S., Sion, R., and Porter, D. E. Cloudflow: Cloud-wide policy enforcement using fast vm introspection. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on* (2014), IEEE, pp. 159–164.

[27] Baliga, A., Ganapathy, V., and Iftode, L. Automatic inference and enforcement of kernel data structure invariants. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual* (2008), IEEE, pp. 77–86.

[28] Baliga, A., Ganapathy, V., and Iftode, L. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing 8*, 5 (2011), 670–684.

[29] Bat-Erdene, M., Park, H., Li, H., Lee, H., and Choi, M.-S. Entropy analysis to classify unknown packing algorithms for malware detection. *International Journal of Information Security* (2016), 1–22.

[30] Bauer, J. M., van Eeten, M. J. G., and Chattopadhyay, T. Itu study on the financial aspects of network security: Malware and spam. http://www.itu.int/ITU-D/cyb/cybersecurity/docs/itu-study-financial-aspects-of-malware-and-spam.pdf, July 2008.

[31] Bayer, U., Moser, A., Kruegel, C., and Kirda, E. Dynamic analysis of malicious code. *Journal in Computer Virology 2*, 1 (2006), 67–77.

[32] Bellard, F. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track* (2005), pp. 41–46.

[33] Bergroth, L., Hakonen, H., and Raita, T. A survey of longest common subsequence algorithms. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on* (2000), IEEE, pp. 39–48.

[34] Bhunia, S., Hsiao, M. S., Banga, M., and Narasimhan, S. Hardware trojan attacks: threat analysis and countermeasures. *Proceedings of the IEEE 102*, 8 (2014), 1229–1247.

[35] Bianchi, A., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. Blacksheep: Detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 341–352.

[36] Bianchi, A., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. Blacksheep: Detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 341–352.

[37] Biedermann, S., and Szefer, J. Systemwall: An isolated firewall using hardware-based memory introspection. In *Information Security*. Springer, 2014, pp. 273–290.

[38] bitsum. PECompact—Windows (PE) Executable Compressor. https://bitsum.com/pecompact/. Retrieved November 2016.

[39] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (2011), ACM, pp. 30–40.

[40] Blunden, B. *The Rootkit Arsenal*, 2 ed. Jones and Barlett Learning, 2013.

[41] Bowling, J. Clonezilla: build, clone, repeat. *Linux journal 2011*, 201 (2011), 6.

[42] Branco, R., Barbosa, G., and Neto, P. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies. In *Black Hat* (2012).

[43] Bruening, D., Kiriansky, V., Garnett, T., and Amarasinghe, S. Dynamorio: An infrastructure for runtime code manipulation.

[44] Bruening, D., Zhao, Q., and Amarasinghe, S. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)* (2012).

[45] BSDaemon, coideloko, and D0nAnd0n. System Management Mode Hack: Using SMM for 'Other Purposes'. *Phrack Magazine* (2008).

[46] Bulygin, Y., and Samyde, D. Chipset based approach to detect virtualization malware a.k.a. DeepWatch. *Blackhat USA* (2008).

[47] Burdach, M. Digital forensics of the physical memory. `http://forensic.seccure.net/pdf/mburdach_digital_forensics_of_physical_memory.pdf`, 2005.

[48] Buse, R. P., and Weimer, W. R. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis* (2008), ACM, pp. 121–130.

[49] Butterworth, J., Kallenberg, C., and Kovah, X. BIOS Chronomancy: Fixing the Core Root of Trust for Measurement. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)* (2013).

[50] Carrier, B. The Sleuth Kit. http://www.sleuthkit.org/sleuthkit/desc.php.

[51] Carrier, B. D., and Grand, J. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation 1*, 1 (2004), 50–60.

[52] Case, A., Cristina, A., Marziale, L., Richard, G. G., and Roussev, V. Face: Automated digital evidence discovery and correlation. *digital investigation 5* (2008), S65–S75.

[53] Case, A., Marziale, L., and Richard, G. G. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation 7* (2010), S32–S40.

[54] CGSoftLabs. eXpressor. `http://www.cgsoftlabs.ro/`. Retrieved November 2016.

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

[56] Chen, T. M., and Robert, J.-M. The evolution of viruses and worms. *Statistical methods in computer security 1* (2004).

[57] Chen, X., Andersen, J., Mao, Z., Bailey, M., and Nazario, J. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN '08)* (2008).

[58] Chen, X., Garfinkel, T., Lewis, E., Subrahmanyam, P., Waldspurger, C., Boneh, D., Dwoskin, J., and Ports, D. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (2008), ACM, pp. 2–13.

[59] Chen, Y., Wang, Y., Ha, Y., Felipe, M. R., Ren, S., and Aung, K. M. M. saes: A high throughput and low latency secure cloud storage with pipelined dma based pcie interface. In *Field-Programmable Technology (FPT), 2013 International Conference on* (2013), IEEE, pp. 374–377.

[60] Chiang, J.-H., Li, H.-L., and Chiueh, T.-c. Introspection-based memory deduplication and migration. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 51–62.

[61] Chisnall, D. *The definitive guide to the Xen hypervisor*. Prentice Hall Press Upper Saddle River, NJ, USA, 2007.

[62] CLOC. Count lines of code. `http://cloc.sourceforge.net/`.

[63] CMU Software Engineering Institute. Michelangelo pc virus warning. `http://www.cert.org/historical/advisories/CA-1992-02.cfm`, September 1997.

[64] CMU Software Engineering Institute. Nimda worm. `http://www.cert.org/historical/advisories/CA-2001-26.cfm`, September 2001.

[65] CMU Software Engineering Institute. “code red” worm exploiting buffer overflow in iis indexing service dll, January 2002.

[66] Contagio. An overview of exploit packs. `http://contagiodump.blogspot.com/2010/06/overview-of-exploit-packs-update.html`, May 2015.

[67] Coreboot. Open-Source BIOS. `http://www.coreboot.org/`.

[68] Courbon, F., Loubet-Moundi, P., Fournier, J. J., and Tria, A. Semba: A sem based acquisition technique for fast invasive hardware trojan detection. In *Circuit Theory and Design (ECCTD), 2015 European Conference on* (2015), IEEE, pp. 1–4.

[69] Dees, B. Native command queuing-advanced performance in desktop storage. *IEEE Potentials 24*, 4 (2005), 4–7.

[70] Deng, Z., Zhang, X., and Xu, D. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'13)* (2013).

[71] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)* (2008).

[72] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 51–62.

[73] Distler, D. *Malware Analysis: An Introduction.* SANS Institute, December 2007. Available via `https://www.sans.org/reading-room/whitepapers/malicious/malware-analysis-introduction-2103`.

[74] Dolan-Gavitt, B., Leek, T., Hodosh, J., and Lee, W. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 839–850.

[75] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), IEEE, pp. 297–312.

[76] Dolan-Gavitt, B., Payne, B., and Lee, W. Leveraging forensic tools for virtual machine introspection.

[77] Dornseif, M. 0wned by an ipod. *Presentation, PacSec* (2004).

[78] Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., Barham, P., and Neugebauer, R. Xen and the art of virtualization. In *In Proceedings of the ACM Symposium on Operating Systems Principles* (2003).

[79] Duflot, L., Etiemble, D., and Grumelard, O. Using CPU system management mode to circumvent operating system security functions. In *Proceedings of the 7th CanSecWest Conference (CanSecWest'04)* (2004).

[80] Duflot, L., Levillain, O., Morin, B., and Grumelard, O. Getting into the SMRAM: SMM Reloaded. In *Proceedings of the 12th CanSecWest Conference (CanSecWest'09)* (2009).

[81] Duflot, L., Levillain, O., Morin, B., and Grumelard, O. System Management Mode Design and Security Issues. `http://www.ssi.gouv.fr/IMG/pdf/IT_Defense_2010_final.pdf`, 2010.

[82] Duflot, L., Perez, Y.-A., Valadon, G., and Levillain, O. Can you still trust your network card. *CanSecWest/core10* (2010), 24–26.

[83] Dykstra, J., and Sherman, A. T. Acquiring forensic evidence from infrastructure-as-a-service cloud computing: Exploring and evaluating tools, trust, and techniques. *Digital Investigation 9* (2012), S90–S98.

[84] DynamoRIO. Dynamic Instrumentation Tool Platform. http://dynamorio.org/.

[85] Egele, M., Scholte, T., Kirda, E., and Kruegel, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR) 44*, 2 (2012), 6.

[86] Embleton, S., Sparks, S., and Zou, C. SMM rootkits: A New Breed of OS Independent Malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks (SecureComm'08)* (2008).

[87] Facebook Security. Facebook whitehat. `https://www.facebook.com/whitehat`, March 2014.

[88] Falliere, N. Windows anti-debug reference. `http://www.symantec.com/connect/articles/windows-anti-debug-reference`, 2010.

[89] Farmer, D., and Venema, W. *Forensic Discover*. Addison-Wesley, 2005.

[90] Fattori, A., Paleari, R., Martignoni, L., and Monga, M. Dynamic and Transparent Analysis of Commodity Production Systems. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)* (2010).

[91] Ferrie, P. Attacks on more virtual machine emulators. *Symantec Technology Exchange* (2007).

[92] FitzPatrick, J., and Crabill, M. NSA Playset: PCIE. In *DEFCON 22* (2014).

[93] Flashrom. Firmware flash utility. `http://www.flashrom.org/`.

[94] Forte, D., Bao, C., and Srivastava, A. Temperature tracking: An innovative run-time approach for hardware trojan detection. In *Proceedings of the International Conference on Computer-Aided Design* (2013), IEEE Press, pp. 532–539.

[95] Fox, S. 51% of u.s. adults bank online. `http://www.pewinternet.org/2013/08/07/51-of-u-s-adults-bank-online/`, 2013.

[96] Fry, Z. P., Landau, B., and Weimer, W. A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (2012), ACM, pp. 177–187.

[97] Fu, Y., and Lin, Z. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P'12)* (2012).

[98] Fu, Y., and Lin, Z. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 586–600.

[99] Garber, L. The challenges of securing the virtualized environment. *Computer 45*, 1 (2012), 17–20.

[100] Garfinkel, T. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS* (2003), vol. 3, pp. 163–176.

[101] Garfinkel, T., Adams, K., Warfield, A., and Franklin, J. Compatibility is not transparency: Vmm detection myths and realities. In *HotOS* (2007).

[102] Garfinkel, T., Adams, K., Warfield, A., and Franklin, J. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HotOS'07)* (2007).

[103] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 193–206.

[104] Garfinkel, T., and Rosenblum, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium (NDSS'03)* (2003).

[105] Garfinkel, T., Rosenblum, M., et al. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium* (2003).

[106] Gates, G., Ewing, J., Russell, K., and Watkins, D. Explaining volkswagen's emissions scandal. http://www.nytimes.com/interactive/2015/business/international/vw-diesel-emissions-scandal-explained.html, September 2016.

[107] Girault, E. Volatilitux—memory forensics framework to help analyzing linux physical memory dumps. http://code.google.com/p/volatilitux.

[108] GNU. GDB: GNU Project Debugger. www.gnu.org/software/gdb.

[109] Google. Vulnerability assessment reward program. www.google.com/about/appsecurity/reward-program, 2014.

[110] Gross, D. Millions of accounts compromised in snapchat hack. http://www.cnn.com/2014/01/01/tech/social-media/snapchat-hack, January 2014.

[111] Guarnieri, C., Tanasi, A., Bremer, J., and Schloesser, M. The cuckoo sandbox, 2012.

[112] Habib, I. Virtualization with kvm. *Linux Journal 2008*, 166 (2008), 8.

[113] HackerOne. Yahoo bug bounty program rules. https://hackerone.com/yahoo, 2014.

[114] Happ, C., Melzer, A., and Steffgen, G. Trick with treat–reciprocity increases the willingness to communicate personal data. *Computers in Human Behavior 61* (2016), 372–377.

[115] Hasan, S. R., Mossa, S. F., Elkeelany, O. S. A., and Awwad, F. Tenacious hardware trojans due to high temperature in middle tiers of 3-d ics. In *2015 IEEE 58th International Midwest Symposium on Circuits and Systems (MWSCAS)* (2015), IEEE, pp. 1–4.

[116] Hay, B., and Nance, K. Forensics examination of volatile system data using virtual introspection. *ACM SIGOPS Operating Systems Review 42*, 3 (2008), 74–82.

[117] Heasman, J. Implementing and detecting a pci rootkit. *Retrieved February 20*, 2007 (2006), 3.

[118] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*, 3 ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[119] Hizver, J., and Chiueh, T.-c. Automated discovery of credit card data flow for pci dss compliance. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on* (2011), IEEE, pp. 51–58.

[120] Hizver, J., and Chiueh, T.-c. Real-time deep virtual machine introspection and its applications. In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 3–14.

[121] Hofmann, O. S., Dunn, A. M., Kim, S., Roy, I., and Witchel, E. Ensuring operating system kernel integrity with osck. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 279–290.

[122] Hofmeyr, S. A., Forrest, S., and Somayaji, A. Intrusion detection using sequences of system calls. *Journal of computer security 6*, 3 (1998), 151–180.

[123] Holllander, R., and Bolotoff, P. RAMSpeed, a cache and memory benchmarking tool. http://alasir.com/software/ramspeed, 2011.

[124] Hotten, R. Volkswagen: The scandal explained. http://www.bbc.com/news/business-34324772, 2015.

[125] Ibrahim, A. S., Hamlyn-Harris, J., Grundy, J., and Almorsy, M. CloudSec: A security monitoring appliance for Virtual Machines in the IaaS cloud model.

[126] IDA Pro. www.hex-rays.com/products/ida/.

[127] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual.

[128] Intel. Intel hall of fame. http://www.intel.com/content/www/us/en/company-overview/intel-museum.html, July 2007. See the July 7, 2006 archive via https://web.archive.org/web/20070706032836/http://www.intel.com/museum/online/hist_micro/hof/.

[129] Jain, B., Baig, M. B., Zhang, D., Porter, D. E., and Sion, R. Sok: Introspections on trust and the semantic gap. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 605–620.

[130] Jiang, X., Wang, X., and Xu, D. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 128–138.

[131] Jiang, X., Wang, X., and Xu, D. Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)* (2007).

[132] Kancherla, K., Donahue, J., and Mukkamala, S. Packer identification using byte plot and markov plot. *Journal of Computer Virology and Hacking Techniques 12*, 2 (2016), 101–111.

[133] Kaspersky Lab. Kaspersky Security Bulletin 2014. http://securelist.com/analysis/kaspersky-security-bulletin/68010/kaspersky-security-bulletin-2014-overall-statistics-for-2014/.

[134] Kaspersky Lab. Kaspersky Security Bulletin 2015. https://securelist.com/files/2015/12/Kaspersky-Security-Bulletin-2015_FINAL_EN.pdf.

[135] Kaspersky Lab. Financial Cyberthreats. http://cdn.securelist.com/files/2015/02/KSN_Financial_Threats_Report_2014_eng.pdf, 2014.

[136] Kaspersky Lab. Kaspersky Lab Uncovers "The Mask": One of the Most Advanced Global Cyber-espionage Operations to Date Due to the Complexity of the Toolset Used by the Attackers. http://kaspersky.com/, 2014.

[137] Kemmerer, R., and Vigna, G. Intrusion detection: a brief history and overview. *Computer 35*, 4 (2002), 27–30.

[138] King, S., and Chen, P. SubVirt: Implementing malware with Virtual Machines. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P'06)* (May 2006).

[139] Kirat, D., Vigna, G., and Kruegel, C. BareBox: Efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)* (2011).

[140] Kirat, D., Vigna, G., and Kruegel, C. Barecloud: bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX conference on Security Symposium (SEC'14). USENIX Association, Berkeley, CA, USA* (2014), pp. 287–301.

[141] Kollar, I. *Forensic RAM dump image analyser.* Charles University in Prague, 2010. Master's Thesis available via `https://is.cuni.cz/webapps/zzp/download/120124298`.

[142] Kopytov, A. Draugr—live memory forensics on linux. `http://code.google.com/p/draugr`.

[143] Kortchinsky, K. CLOUDBURST: A VMware Guest to Host Escape Story. In *Black Hat USA* (2009).

[144] Krebs, B. Hackers break into virginia health professions database, demand ransom. `http://voices.washingtonpost.com/securityfix/2009/05/hackers_break_into_virginia_he.html`, September 2009.

[145] Krishnan, S., Snow, K. Z., and Monrose, F. Trail of bytes: efficient support for forensic analysis. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 50–60.

[146] Ladakis, E., Koromilas, L., Vasiliadis, G., Polychronakis, M., and Ioannidis, S. You can type, but you can't hide: A stealthy gpu-based keylogger. In *Proceedings of the 6th European Workshop on System Security (EuroSec)* (2013).

[147] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on 38*, 1 (2012), 54–72.

[148] Leach, K., Spensky, C., Weimer, W., and Zhang, F. Hops: Towards transparent introspection. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)* (Osaka, Japan, March 2016). Acceptance rate: 37%.

[149] Lee, W., and Stolfo, S. J. Data mining approaches for intrusion detection. In *Usenix security* (1998).

[150] Leek, T., Zhivich, M., Giffin, J., and Lee, W. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P'11)* (2011).

[151] Lengyel, T. K., Neumann, J., Maresca, S., Payne, B. D., and Kiayias, A. Virtual machine introspection in a hybrid honeypot architecture. In *CSET* (2012).

[152] Lin, Z., Rhee, J., Wu, C., Zhang, X., and Xu, D. Dimsum: Discovering semantic data of interest from un-mappable memory with confidence. In *Proc. ISOC Network and Distributed System Security Symposium* (2012).

[153] Lobosco, K. Michaels hack hit 3 million. `http://money.cnn.com/2014/04/17/news/companies/michaels-security-breach/`, April 2014.

[154] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm Sigplan Notices* (2005), vol. 40, ACM, pp. 190–200.

[155] Madden, M. More online americans say they have experienced a personal data breach. `http://www.pewresearch.org/fact-tank/2014/04/14/more-online-americans-say-theyve-experienced-a-personal-data-breach/`, 2014.

[156] Mankin, J., and Kaeli, D. Dione: a flexible disk monitoring and analysis framework. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2012, pp. 127–146.

[157] Martin, A. Firewire memory dump of a windows xp computer: a forensic approach. *Black Hat DC* (2007), 1–13.

[158] McAfee. Threats Report: Fourth Quarter 2014. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2014.pdf`.

[159] McAfee. Threats Report: March 2016. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-mar-2016.pdf`.

[160] Microsoft. Dr. watson overview. `https://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/drwatson_overview.mspx?mfr=true`. Retrieved November 2016.

[161] Molina, J., and Arbaugh, W. Using independent auditors as intrusion detection systems. In *Information and Communications Security*. Springer, 2002, pp. 291–302.

[162] MongoDB Inc. Mongodb. `https://mongodb.com`. Retrieved November 2016.

[163] Moon, H., Lee, H., Lee, J., Kim, K., Paek, Y., and Kang, B. B. Vigilare: toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 28–37.

[164] Navarro, G. A guided tour to approximate string matching. *ACM computing surveys (CSUR) 33*, 1 (2001), 31–88.

[165] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction* (2002), Springer, pp. 213–228.

[166] of Homeland Security, D. National cyber security awareness month. https://www.dhs.gov/national-cyber-security-awareness-month, 2016.

[167] Olmstead, K., Lampe, C., and Ellison, N. B.

[168] Oracle. VirtualBox. http://www.virtualbox.com, 2007.

[169] Oreans Technologies. Themida. http://www.oreans.com/themida.php. Retrieved November 2016.

[170] Ortega, A. Paranoid fish.

[171] Pagliery, J. Adobe has an epically abysmal security record. http://money.cnn.com/2013/10/08/technology/security/adobe-security/index.html, October 2014.

[172] Pagliery, J. Aol hack causes zombie spam. http://money.cnn.com/2014/01/23/news/companies/neiman-marcus-hack/, April 2014.

[173] Pagliery, J. Ebay customers must reset passwords after major hack. http://money.cnn.com/2014/05/21/technology/security/ebay-passwords/index.html, May 2014.

[174] Pagliery, J. Target hack is a wake-up call on privacy. http://money.cnn.com/2014/01/11/technology/security/target-hack-privacy/index.html, January 2014.

[175] Payne, B. D. Libvmi: Simplified virtual machine introspection.

[176] Payne, B. D., Carbone, M., Sharif, M., and Lee, W. Lares: An architecture for secure active monitoring using virtualization. *IEEE Symposium on Security and Privacy* (2008), 233–247.

[177] Perez, S. 66% of employees use 2 or more devices at work, 12% use tables. https://techcrunch.com/2012/10/10/forrester-66-of-employees-use-2-or-more-devices-at-work-12-use-tablets/, 2012.

[178] Perlroth, N., and Sanger, D. E. Nations buying as hackers sell flaws in computer code. http://www.nytimes.com/2013/07/14/world/europe/nations-buying-as-hackers-sell-computer-flaws.html, July 2013.

[179] Perrin, A. Social media usage: 2005–2015. http://www.pewinternet.org/2015/10/08/social-networking-usage-2005-2015/, 2015.

[180] Petroni, N. L., Aaron, J., Timothy, W., William, F., and Arbaugh, A. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation 3* (2006).

[181] Petroni Jr, N. L., Fraser, T., Molina, J., and Arbaugh, W. A. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium* (2004), San Diego, USA, pp. 179–194.

[182] Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., and Ioannidis, S. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security* (2014), ACM, p. 5.

[183] Prandini, M., and Ramilli, M. Return-oriented programming. *Security & Privacy, IEEE 10*, 6 (2012), 84–87.

[184] Puri, R. Bots & botnet: An overview. `http://www.sans.org/reading-room/whitepapers/malicious/bots-botnet-overview-1299`, August 2003.

[185] Quist, D., Smith, V., and Computing, O. Detecting the presence of virtual machines using the local data table. *Offensive Computing* (2006).

[186] Quist, D., and Val Smith, V. Detecting the Presence of Virtual Machines Using the Local Data Table. `http://www.offensivecomputing.net/`.

[187] Quynh, N., and Suzaki, K. Virt-ICE: Next-generation Debugger for Malware Analysis. In *In Black Hat USA* (2010).

[188] Raffetseder, T., Kruegel, C., and Kirda, E. Detecting system emulators. In *Information Security.* Springer Berlin Heidelberg, 2007.

[189] Rajendran, J., Kanuparthi, A. K., Karri, R., Zahran, M., Addepalli, S. K., and Ormazabal, G. Securing processors against insider attacks. *IEEE Design and Test 30*, 2 (2013), 35–44.

[190] Ranadive, A., Gavrilovska, A., and Schwan, K. Ibmon: monitoring vmm-bypass capable infiniband devices using memory introspection. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing* (2009), ACM, pp. 25–32.

[191] Reuben, J. S. A survey on virtual machine security. *Helsinki University of Technology 2* (2007), 36.

[192] Reversing Labs. RLPack. `https://reversinglabs.com`. Retrieved November 2016.

[193] Rutkowska, J. Red Pill. `http://www.ouah.org/Red_Pill.html`.

[194] Rutkowska, J. Blue Pill. `http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html`, 2006.

[195] Rutkowska, J. Beyond the cpu: Defeating hardware based ram acquisition. *Proceedings of BlackHat DC 2007* (2007).

[196] Rutkowska, J., and Wojtczuk, R. Preventing and detecting Xen hypervisor subversions. *Black Hat Briefings USA* (2008).

[197] Schuster, A. Searching for processes and threads in microsoft windows memory dumps. *digital investigation 3* (2006), 10–16.

[198] Seltzer, L. The morris worm: Internet malware turns 25. `http://www.zdnet.com/article/the-morris-worm-internet-malware-turns-25/`, November 2013.

[199] Silberschatz, A., Galvin, P. B., and Gagne, G. *Operating System Concepts*, 8th ed. Wiley Publishing, 2008.

[200] Sillito, J., Murphy, G. C., and De Volder, K. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (2006), ACM, pp. 23–34.

[201] Smith, M. New malware threat hides from researchers in 'cat-and-mouse game'. `http://business-reporter.co.uk/2016/05/25/new-malware-threat-hides-researchers-cat-mouse-game/`, May 2016.

[202] Snow, K. Z., Krishnan, S., Monrose, F., and Provos, N. Shellos: Enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium* (2011).

[203] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)* (2008).

[204] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. Bitblaze: A new approach to computer security via binary analysis. In *Information systems security*. Springer, 2008, pp. 1–25.

[205] Spensky, C. Analysis Time for Malware Samples. Email correspondence with author, 2015.

[206] Spensky, C., Hu, H., and Leach, K. LO-PHI: Low observable physical host instrumentation. In *Networks and Distributed Systems Security Symposium 2016 (NDSS 2016)* (San Diego, CA, February 2016). Acceptance rate: 15.8%.

[207] Srinivasan, D., Wang, Z., Jiang, X., and Xu, D. Process out-grafting: an efficient out-of-vm approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 363–374.

[208] Srivastava, A., and Giffin, J. Tamper-resistant, application-aware blocking of malicious network connections. In *International Workshop on Recent Advances in Intrusion Detection* (2008), Springer, pp. 39–58.

[209] Stewin, P. A primitive for revealing stealthy peripheral-based attacks on the computing platform's main memory. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2013, pp. 1–20.

[210] Stewin, P., and Bystrov, I. Understanding dma malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2013, pp. 21–41.

[211] Stüttgen, J., and Cohen, M. Anti-forensic resilient memory acquisition. *Digital Investigation 10* (2013), S105–S115.

[212] Suneja, S., Isci, C., Bala, V., De Lara, E., and Mummert, T. Non-intrusive, out-of-band and out-of-the-box systems monitoring in the cloud. In *ACM SIGMETRICS Performance Evaluation Review* (2014), vol. 42, ACM, pp. 249–261.

[213] SuperPI. http://www.superpi.net/.

[214] Swanson, D. The cat-and-mouse game: The story of malwarebytes chameleon. https://blog.malwarebytes.com/cybercrime/2012/04/the-cat-and-mouse-game-the-story-of-malwarebytes-chameleon/, April 2012.

[215] Systems, C. RPC Interface Vulnerabilities. https://tools.cisco.com/security/center/viewAlert.x?alertId=6638, September 2003.

[216] Tang, A., Sethumadhavan, S., and Stolfo, S. Unsupervised anomaly-based malware detection using hardware features. In *Research in Attacks, Intrusions and Defenses*, A. Stavrou, H. Bos, and G. Portokalidis, Eds., vol. 8688 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 109–129.

[217] Tehranipoor, M., and Koushanfar, F. A survey of hardware trojan taxonomy and detection. *IEEE Design and Test of Computers 27*, 1 (2010), 10–25.

[218] The Linux Information Project. Ms-dos: A brief introduction. http://www.linfo.org/ms-dos.html, September 2006.

[219] Vasudevan, A., and Yerraballi, R. Stealth breakpoints. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)* (2005).

[220] VIA Technologies, Inc. VT8237R South Bridge, Revision 2.06, December 2005.

[221] VMware, Inc. VMWare Fusion. https://www.vmware.com/products/fusion. Access time: July 2015.

[222] VMWare, Inc. Vmware server. http://www.vmware.com/products/server, 2008.

[223] Vogl, S., and Eckert, C. Using hardware performance events for instruction-level monitoring on the x86 architecture. In *Proceedings of the 2012 European Workshop on System Security EuroSec* (2012), vol. 12.

[224] Waddell, K. The computer virus that haunted early aids researchers. http://www.theatlantic.com/technology/archive/2016/05/the-computer-virus-that-haunted-early-aids-researchers/481965/, March 2016.

[225] Walker, J. Animal source code. http://www.fourmilab.ch/documents/univac/animalsrc.html, August 1996.

[226] Wallace, G. Neiman marcus hack hit 1.1 million customers. http://money.cnn.com/2014/01/23/news/companies/neiman-marcus-hack/, January 2014.

[227] Wang, J., Stavrou, A., and Ghosh, A. Hypercheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection* (2010), Springer, pp. 158–177.

[228] Wang, J., Zhang, F., Sun, K., and Stavrou, A. Firmware-assisted memory acquisition and analysis tools for digital forensics. In *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on* (2011), IEEE, pp. 1–5.

[229] Wang, Y., Chen, P., Hu, J., and Rajendran, J. J. The cat and mouse in split manufacturing. In *Proceedings of the 53rd Annual Design Automation Conference* (2016), ACM, p. 165.

[230] Wedum, P. L. *Malware Analysis: A Systematic Approach.* Norwegian University of Science and Technology, 2008. Master's Thesis: Available via https://brage.bibsys.no/xmlui//bitstream/handle/11250/261770/-1/347719_FULLTEXT01.pdf.

[231] Wei, S., Li, K., Koushanfar, F., and Potkonjak, M. Provably complete hardware trojan detection using test point insertion. In *2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2012), IEEE, pp. 569–576.

[232] Whelan, R., Leek, T., and Kaeli, D. Architecture-independent dynamic information flow tracking. In *Compiler Construction* (2013), Springer, pp. 144–163.

[233] Willems, C., Hund, R., Fobian, A., Felsch, D., Holz, T., and Vasudevan, A. Down to the bare metal: Using processor features for binary analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'12)* (2012).

[234] Wojtczuk, R., and Kallenberg, C. Attacking UEFI Boot Script. 31st Chaos Communication Congress (31C3), http://events.ccc.de/congress/2014/Fahrplan/system/attachments/2566/original/venamis_whitepaper.pdf, 2014.

[235] Wojtczuk, R., and Rutkowska, J. Attacking Intel Trust Execution Technologies. http://invisiblethingslab.com/resources/bh09dc/Attacking2009.

[236] Wojtczuk, R., and Rutkowska, J. Attacking SMM Memory via Intel CPU Cache Poisoning, 2009.

[237] Wojtczuk, R., Rutkowska, J., and Tereshkin, A. Xen 0wning Trilogy. In *Black Hat USA* (2008).

[238] Wood, T., Shenoy, P. J., Venkataramani, A., and Yousif, M. S. Black-box and gray-box strategies for virtual machine migration. In *NSDI* (2007), vol. 7, pp. 17–17.

[239] WoodMann. Packers and Unpackers. http://www.woodmann.com/crackz/Packers.htm. Retrieved November 2016.

[240] Xiao, J., Xu, Z., Huang, H., and Wang, H. Security implications of memory deduplication in a virtualized environment. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)* (2013).

[241] Yan, L.-K., Jayachandra, M., Zhang, M., and Yin, H. V2E: Combining hardware virtualization and software emulation for transparent and extensible malware analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)* (2012).

[242] Yason, M. The art of unpacking. *Black Hat Briefings USA, Aug 2007* (2007).

[243] Yu, P., Bo, L., Datong, L., and Xiyuan, P. A high speed dma transaction method for pci express devices. In *Testing and Diagnosis, 2009. ICTD 2009. IEEE Circuits and Systems International Conference on* (2009), IEEE, pp. 1–4.

[244] Yuschuk, O. OllyDbg. `www.ollydbg.de`.

[245] Zelster, L. Mastering 4 stages of malware analysis. `https://zeltser.com/mastering-4-stages-of-malware-analysis/`, February 2015.

[246] Zetter, K. An unprecedented look at stuxnet, the world's first digital weapon. `https://www.wired.com/2014/11/countdown-to-zero-day-stuxnet/`, November 2014.

[247] Zhang, F., Leach, K., Stavrou, A., and Wang, H. Using hardware features for increased debugging transparency, July 2015. Provisional Patent Application 62/170,155.

[248] Zhang, F., Leach, K., Stavrou, A., and Wang, H. Towards transparent debugging. *IEEE Transactions on Dependable and Secure Computing* (January 2016). Impact factor: 1.592.

[249] Zhang, F., Leach, K., Sun, K., and Stavrou, A. SPECTRE: A Dependable Introspection Framework via System Management Mode. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)* (2013).

[250] Zhang, F., Leach, K., Wang, H., Stavrou, A., and Sun, K. Using Hardware Features to Increase Debugging Transparency. In *Proceedings of the 36th IEEE Symposium on Security and Privacy* (2015).

[251] Zhang, X., van Doorn, L., Jaeger, T., Perez, R., and Sailer, R. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop* (2002), ACM, pp. 239–242.

[252] Zhou, B., Zhang, W., Thambipillai, S., and Teo, J. K. J. A low cost acceleration method for hardware trojan detection based on fan-out cone analysis. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis* (New York, NY, USA, 2014), CODES '14, ACM, pp. 28:1–28:10.