LO-PHI: Low-Observable Physical Host Instrumentation for Malware Analysis

Chad Spensky*†, Hongyi Hu*§ and Kevin Leach*‡
*MIT Lincoln Laboratory *lophi@mit.edu*
†University of California, Santa Barbara *cspensky@cs.ucsb.edu*
§Dropbox *hongyihu@alum.mit.edu*
‡University of Virginia *kjl2y@virginia.edu*

***Abstract*—Dynamic-analysis techniques have become the linchpins of modern malware analysis. However, software-based methods have been shown to expose numerous artifacts, which can either be detected and subverted, or potentially interfere with the analysis altogether, making their results untrustworthy. The need for less-intrusive methods of analysis has led many researchers to utilize introspection in place of instrumenting the software itself. While most current introspection technologies have focused on virtual-machine introspection, we present a novel system, LO-PHI, which is capable of physical-machine introspection of both non-volatile and volatile memory, i.e., hard disk and system memory. We demonstrate that we are able to provide analysis capabilities comparable to existing solutions, whilst exposing *zero* software-based artifacts and minimal hardware artifacts. To demonstrate the usefulness of our system, we have developed a framework for performing automated binary analysis. We employ this framework to analyze numerous potentially malicious binaries using both traditional virtual-machine introspection and our new hardware-based instrumentation. Our results show that not only is our analysis on-par with existing software-based counterparts, but that our physical instrumentation is capable of successfully analyzing far more binaries, as it is not foiled by popular anti-analysis techniques.**

I. Introduction

With the rapid advancement of malware, the capabilities of existing analysis techniques have become obsolete. Tools that exist within the operating system or hypervisor are prone to creating artifacts that are visible to the malicious code. Malware authors can leverage these artifacts to conceal their true intentions by halting execution or potentially subverting the analysis technique all together. Even with clever designs, there remains no proven technique for developing low-artifact software-based analysis tools. Moreover, recent work by Kirat et al. [44] showed that at least 5% of the malware analyzed in their study employed anti-analysis techniques that successfully evade most existing analysis tools. Subsequently, numerous systems have been developed that attempt to detect and analyze these *environment-aware* malware (e.g., [13], [49]). Chen et al. [20] even provide a taxonomy of anti-analysis techniques and mitigations commonly employed. However, no bullet-proof solutions exist, and most existing solutions require continuous updating as they rely on emulation frameworks.

To address these problems we present LO-PHI (Low-Observable Physical Host Instrumentation), a novel system capable of analyzing software executing on commercial-off-the-shelf (COTS) bare-metal machines, without the need for any additional software on the 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 system under test (SUT). We have taken a two-pronged approach that is capable of instrumenting machines with actual hardware, to minimize artifacts, or with traditional software-based techniques utilizing hardware virtualization, to maximize scale. This permits the tradeoff between transparency, scale, and cost when appropriate as well as the potential for parallel analysis. Our architecture uses physical hardware and software-based sensors to monitor a SUT's memory, network, and disk activity as well as actuate its keyboard, mouse, and power. The raw data collected from our sensors is then processed with modified open source tools, i.e., Volatility [9] and Sleuthkit [18], to bridge the *semantic gap*, i.e., convert raw 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 be easily extended to support new or legacy operating systems and file systems as long as the hardware tap points are suitable for data acquisition.

LO-PHI also necessitated the development of novel introspection techniques. While numerous techniques exist for memory acquisition [19], [28], [72] of bare-metal systems, passively monitoring disk activity has only recently begun to be explored [50]. In this work we present a hardware sensor capable of passively sniffing the disk activity of a live machine while introducing minimal artifacts. Subsequently, we have also developed the required modules for parsing and reconstructing the underlying serial advanced technology attachment (SATA) protocol. 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.

While the potential applications for LO-PHI are vast, we focus on our ability to perform automated malware analysis on physical machines, and demonstrate its usefulness by showcasing the ability to analyze classes of malware that trivially

NDSS '16, 21-24 February 2016, San Diego, CA, USA
Copyright 2016 Internet Society, ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23121

evade existing dynamic analysis techniques. We first briefly summarize the current state of dynamic malware analysis to better highlight our contributions in Section II. Next, we describe the design and implementation of our system, including hardware sensors for memory and disk capture as well as actuators for controlling and reverting a system under test (SUT) in Section III. We then attempt to quantify the exposed hardware artifacts of our system Section IV and some of our inherent limitations in Section V. We discuss the design of our automated binary analysis framework in Section VI and present the findings from our analysis of various malware samples in Section VII. Finally, we compare LO-PHI to other related works in Section VIII and highlight areas that we feel are rich for future work in Section IX.

In summary we claim to make the following contributions to the field of dynamic analysis:

- Deployed and tested an extremely low-artifact, hardware-based, dynamic analysis environment capable of analyzing malware that avoids traditional software-based techniques
- Developed hardware capable of introspecting the communication between SATA devices as well as asynchronous memory acquisition
- Wrote a module capable of reconstructing SATA frames into high-level disk sector operations
- Modified open-source forensics tools to reconstruct file system and operating system states
- Constructed a framework, and accompanying infrastructure, for automating analysis of binaries on both physical and virtual machines
- Demonstrated the scalability of our system to execute and analyze thousands of samples with comparable fidelity to traditional VM-based solutions

II. Background and Threat Model

In this section, we introduce the vocabulary and basic concepts surrounding the LO-PHI system as well as the scope of our threat model.

a) Stealthy Malware: Recent malware detection and analysis tools rely on virtualization, emulation, and debugging tools. Unfortunately, these techniques are becoming obsolete with the growing interest in *stealthy malware*. Malware is stealthy if it makes an effort to hide its true behavior. This stealth can emerge in several ways.

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 program.

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

b) Artifacts: As mentioned above, stealthy malware evades detection by concluding whether an analysis tool is being used to watch its execution. This means that there must be some piece of evidence available to the malware that it uses to make this determination, commonly known as the *observer effect*. 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 refer to these bits of evidence as *artifacts*. LO-PHI seeks to make instrumentation and measurement of malware more transparent by reducing or eliminating the presence of these artifacts.

c) Malware Analysis: These stealth techniques have necessitated the development of increasingly sophisticated techniques to analyze them. For benign or non-stealthy binaries, numerous debuggers exist such as OllyDbg, Immunity, and gdb. These debuggers can be trivially detected in most cases (e.g., by using the `isDebuggerPresent()` function). These anti-analysis techniques led researchers to develop more transparent, security-focused analysis frameworks using virtual machines, which generally work by hooking system calls to provide an execution trace which can then be analyzed [7], [24], [25], [30], [64], [74]. System call interposition has its own inherent problems [34] (e.g., performance overhead), which led many researchers to decouple their analysis code even further from the execution path. Virtual-machine introspection (VMI) peeks at system state without any direct interaction with the control flow of the program, 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 [26], [35], [38], [42], [47], [63], [65]. Jain et al. [41] 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, which led to the development of numerous techniques for bridging this *semantic gap* in both memory [9], [27], [33], [42], [48] and disk [18], [45], [50] accesses. All VM-based techniques thus far have nevertheless been shown to reveal some detectable artifacts [20], [57], [59], [60] that could still be used to subvert analysis [31], [57].

Because the techniques used to bridge the semantic gap rely only on raw data and are in no way tied to the method of acquisition, newer techniques further decouple the analysis code from the SUT by moving the analysis portion into system management mode (SMM) mode, an isolated execution mode available on x86 processors, [10], [70], [72], [75] or onto a separate processor altogether [12], [52], [53], [56], [76]. Similarly, our work fueled the development of an array of methods for acquiring a system's memory and disk state, while introducing even fewer artifacts. In decreasing order of artifacts, the most popular techniques for acquiring memory use either specialized software [21], [62] or hardware to exploit direct memory access (DMA) over FireWire [28], [51] or using a peripheral component interconnect (PCI) card [19], [68]. Molina et al. [52] used a PCI card to obtain an out-of-band method for accessing the hard disk. While a few techniques have been proposed to defeat introspection and semantic-gap based approaches [11], they tend to be very fragile in practice and are not likely to be widely deployed. With LO-PHI, we hope to help bridge the gap between these low-artifact data acquisition methods and the semantically-rich emulated analysis frameworks to provide similar output with far less overhead.

d) Threat Model: While LO-PHI has an almost-complete view of the SUT, and introduces very few artifacts, we still make a few necessary assumptions about the malware that we are analyzing. Specifically, we assume that the malware can interact with the SUT without restriction, but that any malicious modifications made to the system will be visible either in main memory or on the disk drive. Malware capable of infecting peripherals or other onboard chips are currently out of scope for our system. Similarly, we assume that the malware in question is not actively trying to thwart semantic-gap reconstruction or avoid our particular hardware through signature-based means. Finally, we assume that our instrumentation was in place before the malicious code was executed, to ensure that our exposed artifacts cannot be fingerprinted. That is, the malware has no chance to analyze the SUT without LO-PHI in place, and thus cannot distinguish our analysis system from a system without our instrumentation. Otherwise, we assume that malware may employ any exploitation, anti-analysis, or anti-debugging techniques. More precisely, LO-PHI is specifically designed to detect highly-sophisticated stealthy malware.

III. System Implementation

LO-PHI leverages various sensors, actuators, and software analysis tools combined into a simple and scalable framework. For the purpose of our framework we generally define a *sensor* as any data collection component (e.g., memory, disk, or network) and an *actuator* as any component which provides inputs for the system (e.g., power, keyboard, or mouse). Our architecture allows for simple one-off experiments on a single target SUT as well as much larger analyses running in parallel on multiple SUTs. The hardware sensors support high-speed, low-artifact collection of various data from a SUT, such as memory or disk activity. Similarly, we employ hardware actuators, which automatically drive a SUT to set up and run experiments, as well as clean up afterwards. The software analysis tools run on a separate analysis machine, capable of aggregating and analyzing the collected data in real time.

In addition to the hardware, LO-PHI also supports traditional virtual-machine introspection using software-based sensors and actuators within our framework. While the major contribution of this paper is our hardware instrumentation, much of our framework's power stems from its duality. We have implemented all of our capabilities in both virtual and physical environments within the same abstracted software interface written in Python. This permits the development of tools that will seamlessly work in either environment, physical or virtual, depending on their instantiation. For example, analysis scripts need only implement high-level functionality (e.g., `memory_read()`, `power_on()`, `disk_revert()`), which our framework will execute appropriately for the given machine type. Similarly, we devised a scripting language for keyboard and mouse actions, along with an appropriate parser for each instantiation.

A. Physical Instrumentation

While much work has been done in instrumenting and introspecting virtual machines, we are only aware of a few systems [43], [44], [73] that have had the goal of bare-metal instrumentation. We feel that the lack of existing solutions is likely due to a lack of motivation and the high barrier of entry, i.e., it is more costly in both human effort and resources to instrument physical machines. In this work, we hope to highlight the usefulness of these techniques and advance the state-of-the-art in malware analysis.

One of our major design goals was to create tools that could be utilized on a wide range of existing commercial-off-the-shelf (COTS) hardware with minimal modification. While implementations with fewer artifacts are possible with specialized hardware (e.g., memory interposers) or modifications of existing hardware (e.g., firmware modifications), developing more robust sensors permits a wider range of potential analyses and helps future-proof our techniques. For these reasons, we chose to focus on two major sources of data for our physical sensors: main memory and disk activity. In particular, we employed Xilinx field-programmable gate array (FPGA) boards to interface with both the peripheral component interconnect express (PCIe) interface, for memory acquisition, and the Serial ATA (SATA) interface, for disk introspection. All of our sensors and actuators communicate over gigabit Ethernet via the user datagram protocol (UDP).

1) Memory (Physical): Our memory sensor is implemented on a Xilinx ML507 development board. This particular board has single-lane PCIe connector, which we utilize as an end-point to read the SUT's memory using direct memory access (DMA). We instantiate the card as a bus master by enabling the *Bus Master Enable* bit in the card's configuration register. While each peripheral is designated a distinct memory region for DMA, there were traditionally no enforcement mechanisms to stop a peripheral from reading and writing arbitrary memory locations. This method has been widely studied [19], [61], [72] and exploited [8], [28], [29], [39], [46], [61], [67] over the years. Subsequently, PCIe can achieve very rapid polling rates, making it an ideal candidate for reliable memory acquisition.

2) Disk (Physical): We also employ the ML507 board for our disk analysis by using its two onboard SATA connectors. An Intelliprop SATA bridge core (Part number: IPP-SA110A-BR) provides the ability to passively monitor, and potentially manipulate, all of the traffic over the SATA interface between the host and device. To receive this data on our remote analysis host, we implemented the logic to package the SATA frames into UDP packets and send them over the gigabit Ethernet connection. This proved a difficult engineering feat, as the data rates of SATA exceed the capacity of our gigabit Ethernet link, and thus necessitated numerous data-flow integrity guarantees. This interface is completely passive and is essentially invisible to the SUT, aside from the occasional throttling of frames.

3) Actuation (Physical): For many of our intended applications, it is convenient, and sometimes necessary, to actuate the SUT from our analysis scripts. For this purpose, we employ the Arduino Leonardo, which is driven by a ATmega32u4. It has numerous general-purpose input/ouput (GPIO) pins, as well as the ability easily emulate a keyboard and mouse through the universal serial bus (USB) interface. We use an external power source for the Arduino to permit functions such as powering on the SUT through the GPIO pins attached to the SUT's motherboard. Integrating the Leonardo into our software framework was relatively straightforward, given the simplistic development environment provided for Arduino platforms. That said, correctly emulating mouse movements required some additional effort. However, once implemented, these mouse movements provide LO-PHI with the capability to move the mouse as a human user would (e.g., continuously move the mouse) and also click buttons presented by the software. While numerous commercial solutions already exist for some of this instrumentation (e.g., Intelligent Platform Management Interface, Active Management Technology, Dell Remote Access Card), we wanted to ensure that LO-PHI would be usable on the widest-range of systems possible, and thus opted for the lower-level interfaces, specifically USB.

4) Infrastructure (Physical): While our sensors and actuators achieve all of our low-level requirements for instrumentation, numerous actuation functions (e.g., reverting the disk or checking the OS's boot status) required the development of specialized infrastructure. While reverting the disk for a virtual machine is as easy as overwriting a file, reverting a physical machine quickly becomes much more involved. Our requirement of unmodified hardware eliminates options such as network booting or specialized drives, which would also produce significant detectable artifacts.

To achieve the desired outcome, we implemented our own preboot execute environment (PXE) server as well as an accompanying trivial file transfer protocol (TFTP) server, dynamic host configuration protocol (DHCP) server and domain name service (DNS) server. This enables us to temporarily permit a given media access control (MAC) address to boot a Clonezilla [16] instance, which restores the disk to a previously saved state. By doing this, we make our system more flexible and scalable as the hardware is no longer tied to a particular operating system or installation configuration. Similarly, by hosting our own DNS and DHCP server, we are able to simplify our scripts and create a richer atmosphere for malware analysis (e.g., we can trivially lookup the IP of a given machine since our infrastructure assigns it). We also use gigabit Netgear GS108T switches, one per physical machine, and utilize virtual local-area networks (VLANs) to ensure that our control and sensor traffic do not interfere with each other.

B. Virtual Instrumentation

Since a great deal of work has already been done using virtual-machine introspection (VMI), we choose to leverage existing capabilities when possible by simply incorporating them into our software framework with minimal amounts of glue code. Because of our desire to use existing solutions, and source code availability, we chose to use open-source hypervisor implementations, namely QEMU/KVM [14], [37].

1) Memory (Virtual): For live memory acquisition, we use techniques similar to those employed by *LibVMI* [55]. We obtain access to the guest's physical memory by means of a UNIX socket. This socket then permits arbitrary memory read and write commands, which perform the appropriate action on the guest's memory using `cpu_physical_memory_map` and `cpu_physical_memory_unmap`.

2) Disk (Virtual): To incorporate disk introspection into our virtual environment, we inserted hooks into the QEMU block driver. These hooks intercept all disk operations and copy the relevant data to a separate thread, which then exports them over a UNIX socket to a subscription sever. By spawning a new thread for every access, we should have negligible impact on the system performance, especially if the host system has underutilized resources. The server then allows our clients to connect and subscribe to the disk activity of any guest. We also ensured that our implementation works properly with copy-on-write disks, greatly increasing performance when an experiment requires frequently resetting the disk state.

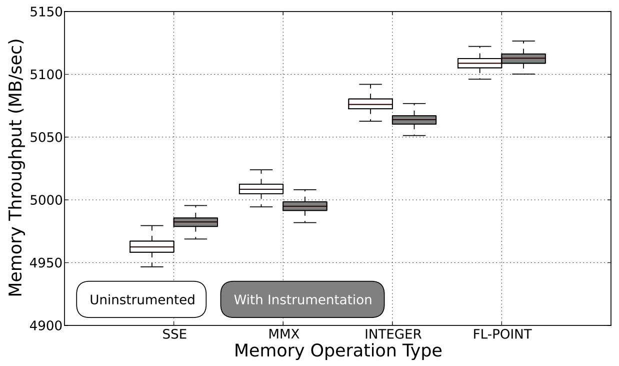
3) Actuation (Virtual): For actuation, we leverage libvirt [2], an open-source tool for interacting with hypervisors. Again, mouse movement proved to be less straightforward, and necessitated the development of a custom virtual network computing (VNC) client.

IV. Artifacts

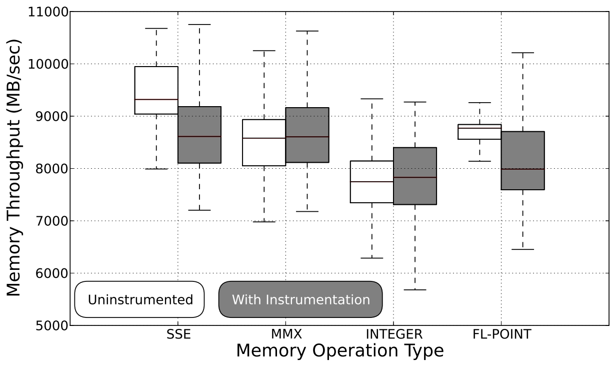
While hardware-level introspection provides numerous desirable security guarantees that are not available for software-based solutions (e.g., hardware segregation of analysis code), it is still critically important to introduce minimal artifacts and, in the ideal case, none at all. We emphasize that the artifacts produced by LO-PHI are likely unusable by malware for subversion, because the malware would lack a baseline for comparison. Minimizing artifacts not only improves the fidelity of our data and the performance of our system, but also reduces the number of "tells" on a SUT that can be used to evade or hinder our analysis. Nevertheless, we enumerate the artifacts introduced by our instrumentation and attempt to address any shortcomings in both virtual and physical machines. All of our experiments were run using our infrastructure to automate the execution of our benchmarking applications on both physical and virtual machines both with and without instrumentation. In our performance experiments, the physical SUT was a Dell T7500 equipped with an 6-core Xeon X5670 and 2GB of RAM and a WD3200AAKX disk drive, and our virtual machine was instantiated on a Dell T7600 equipped with dual 8-core Xeon E5-2665 processors and 68GB of RAM, with the VM itself allocated one core, 1GB of RAM and a 10GB hard disk.

A. Memory Artifacts

In both of our memory-introspection techniques, i.e., physical and virtual, there is likely to be a performance impact on the SUT, as we are accessing a shared resource (main memory). We attempted to quantify this performance impact by utilizing *RAMSpeed*, a popular RAM benchmarking application. We ran RAMSpeed on the same system with and without our instrumentation. 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 sub-experiments, 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 had our sensors continuously read the entire memory space, which should have also introduced the largest performance impact on the system. The memory polling rates were dictated by the hardware and our particular implementation.



(a) Physical machine (Polling at 14MB/sec)



(b) Virtual machine (Polling at 160MB/sec)

Fig. 1: Average memory throughput comparison as reported by RAMSpeed, with and without instrumentation on both physical and virtual machines. (500 samples for each box plot)

At first glance, Figure 1 may seem to indicate that our instrumentation has a discernible effect on the system; however, the deviation from the uninstrumented median is only 0.4% in the worst case (SSE in Figure 1a). Despite our best efforts to create a controlled experiment, i.e., running RAMSpeed on a fresh install of Windows with no other processes, we were unable to to definitively attribute any deviation to our polling of memory. While our memory instrumentation certainly has 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 appears to indicate that systems like ours could poll at significantly higher rates while still remaining undetectable. For example, PCIe implementations can achieve DMA read speeds of 3 GB/sec [6], which could permit a new class entirely of introspection capabilities. To this end, we have successfully achieved rates as fast as 60 MB/sec using SLOTSCREAMER [32]; however, the implementation is not yet stable enough to incorporate into our framework. Nevertheless, in order to detect any deviation in performance, the software being analyzed would need to have the ability to baseline our system, which is not feasible in our experiments.

While performance concerns are a universal problem with instrumentation, adding hardware to a physical configuration has numerous additional artifacts that must also be addressed. To utilize PCIe, we must enumerate our card on the bus, which means that the BIOS and operating system are able to see our hardware. This inevitably reveals our presence on the machine; nevertheless, mitigations do exist (e.g., masquerading as a different device). To avoid detection, our card could trivially use a different hardware identifier every time to avoid signature-based detection. Even with a masked hardware identifier however, Stewin et al. [66] 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 could necessitate the need for more sophisticated acquisition techniques, some of which are proposed in Section IX.

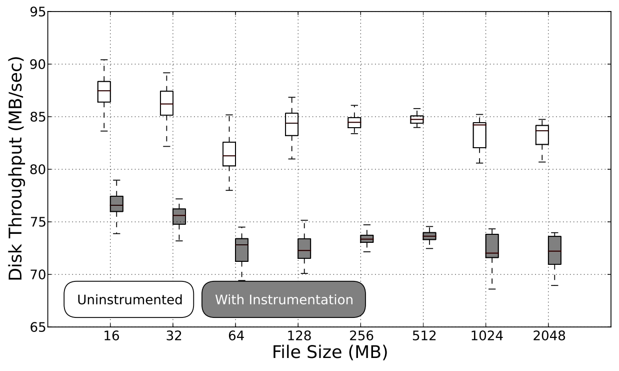
B. Disk Artifacts

To quantify the performance impact of our disk instrumentation, we similarly employed a popular disk benchmarking utility, *IOZone* [1]. 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 and ran IOZone 50 times for each case, i.e., with and without our instrumentation, monitoring the read and write throughput with a record size of 16MB and file sizes ranging from 16MB to 2GB (the total amount of RAM on SUT).

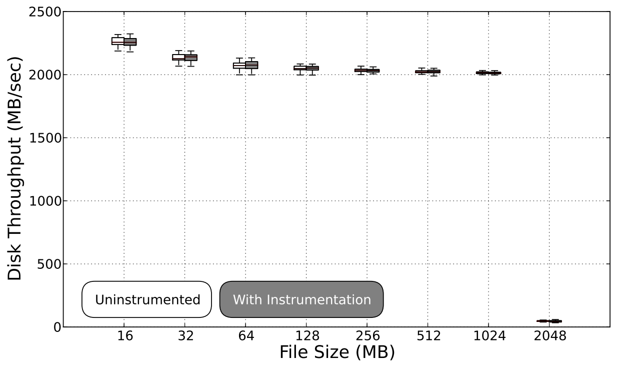
Our hardware should only be visible when we intentionally delay SATA frames to meet the constraints of our gigabit Ethernet link. We designed the system with this delay to minimize our packet loss since UDP does not guarantee delivery of packets. In practice, we rarely observed the system cross this threshold; however, IOZone is explicitly made to stress the limits of a file system. For smaller files, caching masks most of our impact as these cache hits limit the accesses that 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).

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 2b 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 2a, 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



(a) File writes



(b) File reads

Fig. 2: File system throughput comparison as reported by IOZone on Windows XP, with and without instrumentation on a physical machine. (50 samples for each box plot)

aside from the presence of a kernel virtual machine (KVM). 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.

V. Limitations

All of our techniques have some inherent limitations. We attempt to enumerate the most prominent of those below.

a) Input/Output Memory Management Unit (IOMMU): Newer chipsets are equipped with IOMMUs, which, when properly configured to disable DMA from peripherals, would render our current memory acquisition approach ineffective. While this limits us from instrumenting arbitrary systems, it does not thwart our approach in the analysis case, i.e., where we have complete control of the system that we are instrumenting, because we could simply disable this functionality or purchase chipsets that do not contain IOMMUs. In the long term however, we will likely have to migrate our techniques to employ a different memory acquisition method.

b) Asynchronous Memory Access: Wang et al. [71] provide a good analysis of the inherent limitations of polling-based systems for malware detection and potential evasion techniques. However, memory polling may also create issues with *smearing* and caching as well. Smearing occurs when the state of memory of a SUT changes during acquisition, resulting in an imperfect memory capture over a time window rather than a specific instant in time. The memory contents of live SUTs are very dynamic, so smearing is likely to occur. Nevertheless, we have only rarely had this effect cause problems in practice, and faster polling rates would help minimize the smearing effect. Similarly, there may be rare cases where data never leaves a cache; however, this can be easily mitigated with cache coherency.

c) Filesystem Caching: For disk monitoring, OS-level disk caching may cause our disk sensor to miss SATA frames that hit the cache and are overwritten before being flushed to disk. While this effect is likely to be minor during continuous disk monitoring, it is conceivable that malware could drop a file, execute it, and delete it before the cache is ever flushed to disk, completely evading detection. However, we do not see this as a major issue, as attempts of persistence will eventually have to write to disk. Additionally, the effects of the malware would also likely be detectable in memory.

d) No Internet Access: Our experiments also had a few limitations that were beyond our control. For example, the network policies within our organization currently forbid us from running these malware samples on the live Internet. Similar studies have concluded that most malware from the wild will appear to do nothing, aside from network activity, without the presence of its command-and-control infrastructure from which to retrieve a payload. Because of this, we do not present our results as representative of the presence of VM-aware malware in the wild, but instead highlight our capabilities and the ability to detect particularly sophisticated payloads once they are executed.

VI. 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 **machines**, both physical and virtual SUTs, that serve as its worker pool. Upon a job submission, the **controller** first downloads the script, which describes the actions to perform on the SUT, and submits the job to a scheduler. This scheduler then waits for a **machine** 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. All of our malware samples, analyses, and results were stored in a MongoDB database. Samples were submitted using a custom 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**.

```
 1 # Reset our disk using PXE
 2 machine.machine_reset()
 3 machine.power_on()
 4 # Wait for OS to appear on network
 5 while not machine.network_get_status():
 6     time.sleep(1)
 7 # Allow time for OS to continue loading
 8 time.sleep(OS_BOOT_WAIT)
 9 # Start disk capture
10 disk_tap.start()
11 # Send key presses to download binary
12 machine.keypress_send(ftp_script)
13 # Dump memory (clean)
14 machine.memory_dump(memory_file_clean)
15 # Start collection network traffic
16 network_tap.start()
17 # Get a list of current visible buttons
18 button_clicker.update_buttons()
19 # Start our binary and click any buttons
20 machine.keypress_send('SPECIAL:RETURN')
21 # Move our mouse to imitate a human
22 machine.mouse_wiggle(True)
23 # Allow binary to execute (Initial)
24 time.sleep(MALWARE_START_TIME)
25 # Dump memory (interim)
26 machine.memory_dump(memory_file_interim)
27 # Take a screenshot (Before clicking buttons)
28 machine.screenshot(screenshot_one)
29 # Click any new buttons that appeared
30 button_clicker.click_buttons(new_only=True)
31 # Allow binary to execute (3 min total)
32 time.sleep(MALWARE_EXECUTION_TIME-elapsed_time)
33 # Take a final screenshot
34 machine.screenshot(screenshot_two)
35 # Dump memory (Dirty)
36 machine.memory_dump(memory_file_dirty)
37 # Shutdown Machine
38 machine.power_shutdown()
```

Fig. 3: Python script for running a malware sample and collecting the appropriate raw data for analysis.

Because of the duality of our framework we were able to write one simple script (see Figure 3) that will: 1) reset our machine to a clean state, 2) take a memory image before and after execution, 3) attempt to click any graphical buttons, 4) capture screenshots, and 5) capture all disk and network activity throughout the execution. To download and execute an arbitrary binary (Figure 3, line 12), our implementation uses hotkeys to open a command line interface, executes a recursive file-transfer protocol (FTP) download to retrieve the files to be analyzed, and then runs a batch file to execute the binary. From this data, we reconstruct the changes in system memory, in addition to a complete capture of disk and network activity generated by the binary. To identify any graphical buttons that the malware may present, we use the Volatility "windows" module to identify all visible windows that have an atom class of 0xc061 or an atom superclass of 0xc017, which indicate a button, and then use our actuator to move the mouse to the appropriate location and click it. 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, i.e., the system is not hung, 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 10 GB partition for the operating system and 1 GB of volatile memory. The operating system 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 images before storing them in our databases. While this adds a significant amount of time to our analysis (approximately 2 minutes), it significantly reduces the storage requirement. Finally, the virtual machine's networks were logically divided to ensure that samples did not interfere with each other, and the physical environment consisted of only one machine.

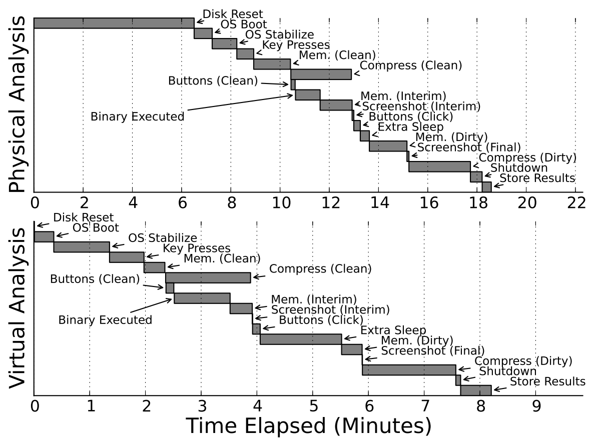
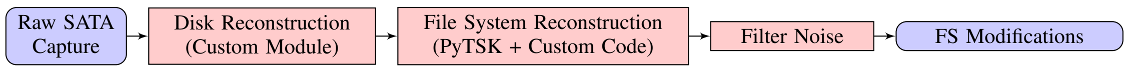


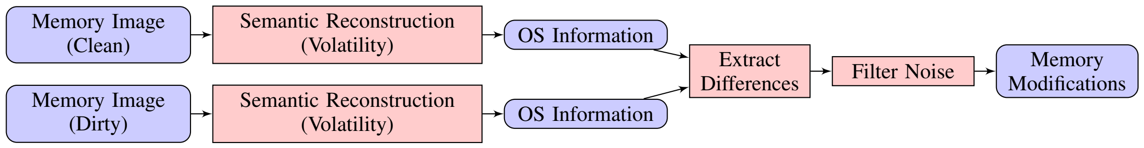
Fig. 4: Time spent in each step of binary analysis. Both environments were booting a 10 GB Windows 7 (64-bit) hibernate image and were running on a system with 1 GB of volatile memory.

The respective runtimes for each portion of our analysis can be seen in Figure 4. We ensured that every binary executed for at least 3 minutes before retrieving our final memory image and resetting the system. Screenshots were obtained using Volatility's *screenshot* module on physical machines and were extracted from the captured memory images. Note that most of the time taken in the physical case is due to our resetting of the system state using Clonezilla, waiting for the system to boot, and memory acquisition. The resetting and boot process could be decreased significantly by writing a custom PXE loader, or completely mitigated by implementing copy-on-write into our FPGA. Similarly, the memory acquisition times could be more comparable to the virtual case, if not faster, by optimizing our PCIe implementation. Finally, *system snapshots* could reduce the time spent setting up the environment to mere seconds. While snapshots are trivial with virtual machines, it is still an open problem for physical machines.

We note that LO-PHI may miss any temporal memory modifications made by the binary between our clean and dirty memory images. To analyze the transient behavior of a binary, LO-PHI could be used to continuously poll the systems memory during execution. However, while this has the potential to produce a lot more 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 hope to explore this area of research in more detail in the future as we improve our memory capture capabilities.


Raw SATA Capture
Disk Reconstruction (Custom Module)
File System Reconstruction (PyTSK + Custom Code)
Filter Noise
FS Modifications

(a) Disk Reconstruction



(b) Memory Reconstruction

Fig. 5: Binary analysis workflow. (Rounded nodes represent data and rectangles represent data manipulation.)

VII. Evaluation and Analysis

In this section, we explain our methodology for semantic gap reconstruction (Section VII-A) and demonstrate the practicality of LO-PHI with three targeted experiments. The experiments were constructed to demonstrate the following:

- The ability of LO-PHI to detect the behaviors elicited by real malware, confirmed with ground truth (Section VII-C)
- The ability to scale and extract meaningful results from unknown malware samples (Section VII-D)
- The ability to analyze malware samples that employ anti-analysis and evasion techniques (Section VII-E)

For each binary, we determine the system changes that occurred during execution by forensically comparing the resulting *clean* and *dirty* states. Each such pair of datasets contains a clean and a dirty raw memory snapshot respectively as well as a log of raw disk and network activity that occurred between clean and dirty states. We exclude the network trace analysis from much of our discussion since it is a well-known technique and not the focus of our work. Our analysis of a binary's execution involves four steps: 1) bridging the semantic gap for both the clean and dirty states, 2) computing the delta between the two states, 3) filtering out actions that are not attributed to the binary, and 4) comparing the delta for physical execution and virtual execution to determine if the sample employs VM-detection techniques (if applicable). The process taken for each binary is illustrated Figure 5. When appropriate, we also compare our results to those produce by Anubis [7] and Cuckoo Sandbox [36].

A. Semantic Gap Reconstruction

As previously mentioned, before any analysis can be conducted, we must first bridge the semantic gap, i.e., translate our memory snapshots and SATA captures, which contain low-level, raw, data into high-level, semantically-rich, information.

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.

2) Disk: The first step in our disk analysis is to first convert the raw capture of the SATA activity into a 4-tuple containing the disk operation (e.g., READ or WRITE), starting sector, total number of sectors, and data. Our physical drives, as with most modern drives, used an optimization in the SATA specification known as Native Command Queuing (NCQ) [23]. NCQ reorders SATA Frame Information Structure (FIS) requests to achieve better performance by reducing extraneous 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 in order 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 [22]. 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 5a. Unlike previous work [50], which was designed for NTFS, our approach is generalizable to any filesystem supported by Sleuthkit. A sample output from creating the file *LO-PHI.txt* on the desktop can be seen below:

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)

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.

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)

Fig. 6: Post-filtered semantic output from rootkit experiment (Section VII-C1).

B. 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 (10 independent executions in our case) using a benign binary (*rundll32.exe* with no arguments). We then use our analysis framework to extract all of the system events for those trials and created a filter based on the events that frequently occurred in this benign dataset. Two of our memory analysis modules, i.e., *filescan* and *timers*, had 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.

C. Experiment 1: High-fidelity Output

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. In our first case study, we evaluated a rootkit that we developed utilizing techniques from *The Rootkit Arsenal* [15] (Section VII-C1). Similarly, we were able to obtain a set of 213 malware samples that were constructed in a cleanroom 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.

1) Homemade Rootkit: Our rootkit stealths itself by adding hooks to the Windows Global Descriptor Table (GDT) and System Service Dispatch Table (SSDT) that will 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 6 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 6f), load the kernel model (Figure 6d), hook the kernel (Figure 6e and Figure 6c), and then execute our FTP server (Figure 6a and Figure 6b). 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 6f 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, we were unable to find our listening socket, or our GDT and SSDT hooks from analyzing 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 lack of execution for such a simple rootkit also emphasizes the importance of having a realistic software environment as well as a hardware environment. We attempt to address this issue for our analysis in Section VII-E.

2) Labeled Malware: For the analysis of our 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 interesting findings are shown.

a) VM-detection: We found that 66 of these samples were labeled as employing 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.

b) 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 dropped 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 4 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, a debugger included in Windows, but also appeared to be injected with malicious code. The 4 samples each created 2 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 3 samples that executed the SUT's legitimate *firefox.exe* browser, but loaded with a suspicious library *needful.dll* that they dropped to disk. The *firefox.exe* process opened TCP listening sockets on ports 1044 and 1045 in 2 of the 3 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.

c) 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 2 particular samples because they appeared to be exfiltrating data over 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.

d) Worms and Network Scanning: We detected approximately 30 labeled samples having worm propagation and network scanning behavior, which was also confirmed by the ground truth data. These samples contacted a significant number of IP addresses and opened up a large number of network sockets in our five minute window. For example, 8 of the samples contacted over 140 IP addresses, and 13 samples opened more than 2000 sockets. The same 13 samples appeared to target external IPs over port 135, which is associated with Microsoft RPC, a service that has had remote exploitable vulnerabilities targeted by worms in the past.

e) Command and Control (C2) and DNS: 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 C2 mechanism for remotely controlling malware, which was the case for these samples as confirmed by the ground truth data. 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*, *asdf.it*, etc.

f) Kernel Modules: We detected 3 samples that unloaded the *ipnat.sys* driver and appeared to gain persistence by replacing it with a malicious version.

D. Experiment 2: 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 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 before running out of available storage for our data on our development server. We present the general types of behaviors detected by LO-PHI in this section. Without ground truth data or manual reverse engineering, we are unable to verify any strong claims from our findings—however, we feel that the findings clearly demonstrate the usefulness of our system. Basic statistics for our analysis of these unlabled samples are shown in Table I.

TABLE I: Overall statistics for unlabeled malware (Section VII-D).

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

g) New Processes: A large majority (70%) of the wild samples created new processes that persisted until the end of our analysis. The most common names are shown in Table II. 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 II: Top processes created by wild malware (Section VII-D).

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

h) Sockets: About 19% of the wild samples opened at least one network socket. The most commonly opened sockets are shown in Table III. 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 III: Top 6 sockets (by port and protocol) created by wild malware (Section VII-D).

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

i) 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.

E. Experiment 3: Evasive Malware

In this section, we exhibit LO-PHI's ability to analyze evasive malware, which thwart existing analysis frameworks. Because we aim to analyze modern malware samples, 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 [5]. Specifically, Acrobat 9.4.0[1], 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 done 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 Figure IV). It is worth noting that the ssdt and driverip modules did not return any findings in our dataset.

1) Paranoid Fish: First, we highlight our ability to analyze evasive binaries with a ground truth sample. We chose Paranoid Fish (*pafish v054*) [54], 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, when run against Anubis [7], pafish dropped the following files:

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

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

Similarly, Cuckoo Sandbox had the following artifacts:

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

On the contrary, when executed in our physical analysis environment, the only artifacts that pafish detected were:

hi_sandbox_physicalmemory_less_1Gb
hi_sandbox_drive_size2

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

2) Labeled Malware: To evaluate LO-PHI's ability to analyze real malware samples, we obtained a set of coarsely-labeled evasive malware samples, generated by Kirat et al. [44] 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 had ground truth that 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 VII-C2. Similarly, because of our aforementioned networking restriction, we expect that numerous samples will produce uninteresting behavior without access their command-and-control infrastructure. Thus, we are unable to make any definitive claims as to specific intent of the malware. 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 V. A summary of our findings is presented in Table VI.

TABLE IV: 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.

TABLE V: Evasive malware dataset.

Technique Employed	# Samples
Wait for keyboard	3
Bios-based	6
Hardware id-based	28
Processor feature-based	62
Exception-based	79
Timing-based	251

a) 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 them 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.

b) 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 connections 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`.

c) 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`, 1 attempted to connect to `62.75.235.238:443`, and 2 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 2 of them set particularly interesting environment variables `9Yy9Y9YYy9YYy` and `YYY9YYY9YYY99`, which both had the value of `E4EC4E2160D8E128C919C56915BFED6C`.

d) 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 4 of them once again loading `TrustedInstaller.exe`, 3 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 8 of the samples querying a variation of `boxonline`, and 7 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.

TABLE VI: 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

e) Exception-based: The exception-based malware samples also exhibited similar behavior, with all but 3 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.

f) 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. The most interesting process names being: `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 4 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 5 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 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 that 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.

VIII. Related Work

VAMPiRE [69] is a software breakpoint framework running within the operating system. It runs in kernel mode, meaning it is safe for debugging ring 3 (user mode) malware. Rootkits can gain kernel-level privileges to circumvent VAMPiRE. However, as LO-PHI does not rely on the operating system, it can be used to safely debug rootkits.

Ether [25] is a malware analysis framework based on hardware virtualization extensions (e.g., Intel VT). It runs outside of the guest operating systems, i.e., in the hypervisor, by relying on underlying hardware features. BitBlaze [64] and Anubis [7] are QEMU-based malware analysis systems. They focus on understanding malware behavior, instead of achieving better transparency. V2E [74] combines both hardware virtualization and software emulation. HyperDbg [30] uses the hardware virtualization that allows the late launching of VMX modes to install a virtual machine monitor, and run the analysis code in the VMX root mode. SPIDER [24] uses Extended Page Tables to implement invisible breakpoints and hardware virtualization to hide its side-effects. Compared to our system, Ether, BitBlaze, Anubis, V2E, HyperDbg and SPIDER all rely on easily detected emulation or virtualization technology [20], [57], [59], [60] and make the assumption that virtualization or emulation is transparent from guest-OSes. In contrast, LO-PHI provides memory access directly from the PCI bus, greatly reducing the potential attack surface. In addition, traditional debugging techniques often add varying degrees of execution overhead. LO-PHI employs specialized hardware that is fast enough to decrease visible timing artifacts otherwise introduced by emulation.

BareBox [43] is a malware analysis framework based on a bare-metal machine without any virtualization or emulation techniques. However, it only targets the analysis of user-mode malware, while LO-PHI can be used for debugging hypervisor rootkits and kernel-mode device drivers. BareCloud [44] is more similar to our approach, as it utilizes mostly uninstrumented machines and executes the binaries with a small software-based loader. Nevertheless, BareCloud requires a network-based storage device and only has information about the disk state before and after execution of the binary, whereas we are able to reconstruction the entire stream of file-system operations. Futhermore, BareCloud has no memory instrumentation and presents numerous detectable artifacts (e.g., malware loader software, networked-drive). Willems et al. [73] used branch tracing to record all the branches taken by a program execution. As pointed out in the paper, the data obtainable by branch tracing is rather coarse, and this approach still suffers from a CPU register attack against branch tracing settings. However, LO-PHI provides fine-grained memory access over the PCI bus, and is thus resistant to CPU register mutation.

Virt-ICE [58] is a remote debugging framework. It leverages virtualization technology to debug malware in a VM and communicates with a debugging client over a TCP connection. However, since it uses a VM, a malware may refuse to unpack itself in the VM. LO-PHI accesses the raw host memory very rapidly, so we can transparently detect when this type of execution occurs.

There is a vast array of popular debugging tools. For instance, IDA Pro [40] and OllyDbg [3] are popular debuggers running within the operating system that focus on ring 3 malware. DynamoRIO [17] is a process virtualization system implemented using software code cache techniques. It executes on top of the OS and allows users to build customized dynamic instrumentation tools. Similar to LO-PHI, WinDbg [4] uses a remote machine to connect to the target machine using serial or network communications. However, these options require special booting configuration or software running within the operating system, which is easily detected by malware. LO-PHI requires a PCI slot, but is intended to run on out-of-the-box consumer hardware, where debugging facilities may not be desirable.

IX. Future Work

We have identified numerous areas that we feel are critical to eventually achieve a more transparent and robust framework. As previously discussed, our current approach has numerous limitations with smearing and incomplete views of the system state. CPU debuggers could alleviate these pains by either completely halting the system during memory acquisition or simply providing more insight into the internal register values. We have been experimenting with Intel's eXtended Debug Port (XDP) and ARM's DSTREAM debugger and found them to extremely powerful devices. Utilizing these hardware debugging technologies in the context of malware analysis could provide very high fidelity data while maintaining the transparency that we require. We plan to explore these techniques further and incorporate any useful developments into our framework.

In this work, we limited our scope to the system-level analysis that was provided by Volatility (e.g., process list, services, sockets). While these modules are more than sufficient for the experiments proposed in this paper, we feel that expanding this scope and introspecting into an individual process's memory space to monitor process-level data (e.g., stack, heap, call trace) could prove invaluable when analyzing advanced malware.

Since LO-PHI currently employs the lowest-level of instrumentation that we are aware of, we feel that continuing to push this boundary is going to be critical for the analysis of more sophisticated future malware. To this end, we feel that our disk, memory, and CPU introspection capabilities positions us well to begin investigating malware that attempts to infect the BIOS or peripherals on a SUT for persistence and plan on continuing to develop these capabilities.

X. Conclusion

We presented LO-PHI, a novel framework capable of instrumenting physical and virtual machines without any software on the system, using a set of sensors and actuators.

Furthermore, we developed a supporting framework capable of automating dynamic analysis of arbitrary binaries by introspecting into the memory, disk, and network activity, reconstructing the semantic operations that occurred, and outputting them as concise events (e.g., process appeared, file written). We show that the sensors used to collect the necessary data produce minimal artifacts to any software running on the machine (Section IV) and that our lack of artifacts enables LO-PHI to analyze particularly sophisticated malware samples with relative ease. As malware continues to advance and evade detection, we expect hardware-based analysis frameworks to become increasingly important. We believe this work exhibits the usefulness of physical-machine introspection and instrumentation, as well as the value of forensic-based malware analysis. We demonstrated that LO-PHI provides valuable analytical capabilities that are unavailable using existing tools. To this end, we hope to engage the community by open-sourcing the project to help advance the state of the art in malware analysis.

Acknowledgment

We would like to thank Brendon Chetwynd for his hardware development efforts and general guidance; Charles V. Wright for his invaluable mentorship, insights, and leadership in the early phases of this work; Joshua Hodosh and Ryan Whelan for their contributions to our memory introspection capabilities and early proofs-of-concept; and finally Lee Rossey, Doug Stetson, and John Wilkinson for their oversight and support throughout.

This work was sponsored by the Assistance Secretary of Defense for Research and Engineering under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

References

[1] "Iozone filesystem benchmark," http://www.iozone.org/.

[2] "libvirt: The Virtualization API," http://libvirt.org/.

[3] "OllyDbg," www.ollydbg.de.

[4] "Windbg," www.windbg.org.

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

[6] Altera Corporation, "PCI Express High Performance Reference Design," http://www.altera.com/literature/an/an456.pdf, October 2015.

[7] Anubis, "Analyzing Unknown Binaries," http://anubis.iseclab.org.

[8] D. Aumaitre and C. Devine, "Subverting windows 7 x64 kernel with dma attacks," *HITBSecConf 2010 Amsterdam*, vol. 29, 2010.

[9] M. Auty, A. Case, M. Cohen, B. Dolan-Gavitt, M. H. Ligh, J. Levy, and A. Walters, "Volatility Framework - Volatile memory extraction utility framework," http://www.volatilityfoundation.org/.

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

[11] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "Dksm: Subverting virtual machine introspection for fun and profit," in *Proceedings of the 29th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2010, pp. 82–91.

[12] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic inference and enforcement of kernel data structure invariants," in *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2008, pp. 77–86.

[13] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, "Efficient detection of split personalities in malware." in *In Proceedings of the 17th Annual Network & Distributed System Security Conference (NDSS)*. The Internet Society, 2010.

[14] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

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

[16] J. Bowling, "Clonezilla: build, clone, repeat," *Linux journal*, vol. 2011, no. 201, p. 6, 2011.

[17] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent Dynamic Instrumentation," in *Proceedings of the 8th Conference on Virtual Execution Environments (VEE)*. ACM SIGPLAN/SIGOPS, 2012.

[18] B. Carrier, "The Sleuth Kit," http://www.sleuthkit.org/.

[19] B. D. Carrier and J. Grand, "A hardware-based memory acquisition procedure for digital investigations," *Digital Investigation*, vol. 1, no. 1, pp. 50–60, 2004.

[20] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *Proceedings of the 38th annual International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2008, pp. 177–186.

[21] M. Cohen, D. Bilby, and G. Caronni, "Distributed forensics and incident response in the enterprise," *Digital Investigation*, vol. 8, pp. S101–S110, 2011.

[22] M. Cohen and J. Metz, "PyTSK," https://github.com/py4n6/pytsk.

[23] B. Dees, "Native command queuing-advanced performance in desktop storage," *Potentials, IEEE*, vol. 24, no. 4, pp. 4–7, 2005.

[24] Z. Deng, X. Zhang, and D. Xu, "SPIDER: Stealthy Binary Program Instrumentation and Debugging Via Hardware Virtualization," in *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2013, pp. 289–298.

[25] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware Analysis via Hardware Virtualization Extensions," in *Proceedings of the 15th Annual Conference on Computer and Communications Security (CCS)*. ACM, 2008.

[26] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: mining memory accesses for introspection," in *Proceedings of the 20th conference on Computer and communications security (CCS)*. ACM, 2013, pp. 839–850.

[27] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proceedings of the 32nd Symposium on Security and Privacy (Oakland)*. IEEE, 2011, pp. 297–312.

[28] M. Dornseif, "0wned by an ipod," *Presentation, PacSec*, 2004.

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

[30] A. Fattori, R. Paleari, L. Martignoni, and M. Monga, "Dynamic and Transparent Analysis of Commodity Production Systems," in *Proceedings of the 25th International Conference on Automated Software Engineering (ASE'10)*. IEEE/ACM, 2010.

[31] P. Ferrie, "Attacks on more virtual machine emulators," *Symantec Technology Exchange*, 2007.

[32] J. FitzPatrick and M. Crabill, "NSA Playset: PCIe," in *DEF CON 22*, 2014.

[33] Y. Fu and Z. Lin, "Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *In Proceedings of the 33rd Annual Symposium on Security and Privacy (Oakland)*. IEEE, 2012, pp. 586–600.

[34] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools." in *In Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS)*, vol. 3. The Internet Society, 2003, pp. 163–176.

[35] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 193–206, 2003.

[36] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser, "The cuckoo sandbox: Automated malware analysis," www.cuckoosandbox.org/.

[37] I. Habib, “Virtualization with kvm,” *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.

[38] B. Hay and K. Nance, “Forensics examination of volatile system data using virtual introspection,” *ACM SIGOPS Operating Systems Review*, vol. 42, no. 3, pp. 74–82, 2008.

[39] J. Heasman, “Implementing and detecting a pci rootkit,” *Retrieved February*, vol. 20, no. 2007, p. 3, 2006.

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

[41] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, “Sok: Introspections on trust and the semantic gap,” in *Proceedings of the 35th Symposium on Security and Privacy (Oakland)*. IEEE, 2014, pp. 605–620.

[42] X. Jiang, X. Wang, and D. Xu, “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)*. ACM, 2007, pp. 128–138.

[43] D. Kirat, G. Vigna, and C. Kruegel, “BareBox: Efficient Malware Analysis on Bare-metal,” in *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2011.

[44] D. Kirat, G. Vigna, and C. Kruegel, “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.

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

[46] E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis, and S. Ioannidis, “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.

[47] T. K. Lengyel, J. Neumann, S. Maresca, B. D. Payne, and A. Kiayias, “Virtual machine introspection in a hybrid honeypot architecture.” in *In Proceedings of the 5th Workshop on Cyber Security Experimentation and Test (CSET)*. USENIX, 2012.

[48] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu, “Dimsum: Discovering semantic data of interest from un-mappable memory with confidence,” in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2012.

[49] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, “Detecting environment-sensitive malware,” in *Proceedings of the 14 International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, 2011, pp. 338–357.

[50] J. Mankin and D. Kaeli, “Dione: a flexible disk monitoring and analysis framework,” in *Proceedings of the 15th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Springer, 2012, pp. 127–146.

[51] A. Martin, “Firewire memory dump of a windows xp computer: a forensic approach,” *Black Hat DC*, 2007.

[52] J. Molina and W. Arbaugh, “Using independent auditors as intrusion detection systems,” in *In Proceedings of the 4th International Conference on Information and Communications Security (ICICS)*. Springer, 2002, pp. 291–302.

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

[54] A. Ortega, “Paranoid fish,” http://github.com/a0rtega/pafish.

[55] B. D. Payne, “Libvmi: Simplified virtual machine introspection,” https://github.com/bdpayne/libvmi.

[56] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, “Copilot-a coprocessor-based kernel runtime integrity monitor.” in *In Proceedings of the 13 Security Symposium (SEC)*. USENIX, 2004, pp. 179–194.

[57] D. Quist and V. Smith, “Detecting the presence of virtual machines using the local data table,” *Offensive Computing*, 2006.

[58] N. A. Quynh and K. Suzaki, “Virt-ICE: Next-generation Debugger for Malware Analysis,” in *Black Hat USA*, 2010.

[59] T. Raffetseder, C. Kruegel, and E. Kirda, “Detecting System Emulators,” in *Information Security*. Springer, 2007.

[60] J. Rutkowska, “Red Pill,” http://www.ouah.org/Red_Pill.html.

[61] J. Rutkowska, “Beyond the cpu: Defeating hardware based ram acquisition,” *Proceedings of BlackHat DC 2007*, 2007.

[62] B. Schatz, “Bodysnatcher: Towards reliable volatile memory acquisition by software,” *Digital Investigation*, vol. 4, pp. 126–134, 2007.

[63] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos, “SHELLOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks,” in *In Proceedings of the 20th Security Symposium (SEC)*. USENIX, 2011.

[64] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “BitBlaze: A New Approach to Computer Security via Binary Analysis,” in *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*. Springer, 2008.

[65] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, “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 (CCS)*. ACM, 2011, pp. 363–374.

[66] P. Stewin, “A primitive for revealing stealthy peripheral-based attacks on the computing platform's main memory,” in *Proceedings of the 13th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Springer, 2013, pp. 1–20.

[67] P. Stewin and I. Bystrov, “Understanding dma malware,” in *In Proceedings of the 10th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2013, pp. 21–41.

[68] J. Stüttgen and M. Cohen, “Anti-forensic resilient memory acquisition,” *Digital Investigation*, vol. 10, pp. S105–S115, 2013.

[69] A. Vasudevan and R. Yerraballi, “Stealth Breakpoints,” in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, 2005.

[70] J. Wang, A. Stavrou, and A. Ghosh, “Hypercheck: A hardware-assisted integrity monitor,” in *In Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, 2010, pp. 158–177.

[71] J. Wang, K. Sun, and A. Stavrou, “A dependability analysis of hardware-assisted polling integrity checking systems,” in *In Proceedings of the 42nd Annual International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2012, pp. 1–12.

[72] J. Wang, F. Zhang, K. Sun, and A. Stavrou, “Firmware-assisted memory acquisition and analysis tools for digital forensics,” in *In Proceedings of the 6th International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE)*. IEEE, 2011, pp. 1–5.

[73] C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan, “Down to the bare metal: Using processor features for binary analysis,” in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2012, pp. 189–198.

[74] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, “V2E: Combining Hardware Virtualization and Software Emulation for Transparent and Extensible Malware Analysis,” in *Proceedings of the 8th SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*. ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2151024.2151053

[75] F. Zhang, K. Leach, K. Sun, and A. Stavrou, “Spectre: A dependable introspection framework via system management mode,” in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE, 2013, pp. 1–12.

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