Cyber Situational Awareness Using Live Hypervisor-Based Virtual Machine Introspection

Dustyn A. Dodge

Follow this and additional works at: https://scholar.afit.edu/etd

Part of the Information Security Commons, and the OS and Networks Commons

Recommended Citation
https://scholar.afit.edu/etd/1979

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact richard.mansfield@afit.edu.
CYBER-SITUATIONAL AWARENESS USING LIVE HYPERVERSOR-BASED VIRTUAL MACHINE INTROSPECTION

THESIS

Dustyn A. Dodge, 1st Lieutenant, USAF

AFIT/GCE/ENG/10-07

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.
CYBER-SITUATIONAL AWARENESS USING LIVE
HYPERVERVISOR-BASED VIRTUAL MACHINE INTROSPECTION

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command

In Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Engineering

Dustyn A. Dodge, B.S. Computer Engineering
1st Lieutenant, USAF

September 2010

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.
CYBER-SITUATIONAL AWARENESS USING LIVE
HYPERVISOR-BASED VIRTUAL MACHINE INTROSPECTION

Dustyn A. Dodge, B.S. Computer Engineering
1st Lieutenant, USAF

Approved:

//signed//

___________________________ _______________________
Dr. Barry E. Mullins (Chairman) (3 Sept 2010)

//signed//

___________________________ _______________________
Dr. Gilbert L. Peterson (Member) (3 Sept 2010)

//signed//

___________________________ _______________________
Maj. Ryan W. Thomas, Ph.D. (Member) (3 Sept 2010)
Abstract

Static memory analysis has been proven a valuable technique for digital forensics. However, the memory capture technique impacts the system by modifying important dynamic system data, such as logged on users, network status, active processes, active drivers, open files and open registry keys. As a result, live analysis techniques have emerged to complement static analysis. In this paper, a compiled memory analysis tool for virtualization (CMAT-V) is presented as a virtual machine introspection (VMI) utility to conduct live analysis during cyber attacks. CMAT-V leverages static memory dump analysis techniques to provide live system state awareness, which includes dynamic system data. Live analysis means that CMAT-V can continually parse live dynamic memory from an active guest operating system (OS). Unlike some VMI applications, CMAT-V bridges the semantic gap using derivation techniques. The semantic gap refers to the disconnect between raw data from dynamic memory and the OS-specific contextual meaning. CMAT-V detects Windows-based operating systems and uses the Microsoft Symbol Server to provide this context to the user. This technique provides increased CMAT-V compatibility for current and future Windows-based operating systems.

This research demonstrates the usefulness of CMAT-V as a situational awareness tool during cyber attacks, tests the detection of CMAT-V from the guest system level and measures its impact on host performance. During experimental testing, live system state information was successfully extracted from two simultaneously executing virtual
machines (VM’s) under four rootkit-based malware attack scenarios. For each malware attack scenario, CMAT-V was able to provide evidence of the attack. Furthermore, data from CMAT-V detection testing did not confirm detection of the presence of CMAT-V’s live memory analysis from the VM itself. This supports the conclusion that CMAT-V does not create uniquely identifiable interference in the VM itself. Finally, three different benchmark tests reveal an 8% to 12% decrease in the host VM performance while CMAT-V is executing.
Acknowledgments

I would first like to thank Jimmy Okolica for his outstanding work developing CMAT and for his technical mentoring and support throughout this whole project. Without your help none of this could have been accomplished. In addition, I would like to thank Bill Kimball for his help with Windows data structures and the Intel architecture. A sincere thank you also goes to Ryan Harris for such an outstanding job acquiring equipment and software for my research. I would also like to thank Bryan Payne for his outstanding work developing and documenting the XenAccess project. Last but certainly not least, I would like to thank my advisor, Dr. Barry Mullins, and my committee members, Dr. Gilbert Peterson and Maj Ryan Thomas, for guiding me in my research.
Table of Contents

Abstract .............................................................................................................................. iv

Acknowledgments .......................................................................................................... vi

List of Figures ................................................................................................................ x

List of Tables .................................................................................................................. xii

I. Introduction .................................................................................................................. 1

1.1. Goals ....................................................................................................................... 2
1.2. Assumptions .......................................................................................................... 4
1.3. Thesis Overview ..................................................................................................... 5

II. Background ................................................................................................................. 7

2.1. Virtualization ......................................................................................................... 7
  2.1.1. Protection Ring Architecture ................................................................ 9
  2.1.2. Virtualization Methods ........................................................................ 10
  2.1.3. Benefits of Virtualization ...................................................................... 17
2.2. Virtual Machine Introspection ............................................................................ 19
  2.2.1. Motivation ............................................................................................. 19
  2.2.2. The Semantic Gap ................................................................................ 22
  2.2.3. VMI Model .......................................................................................... 23
  2.2.4. VMI View-Generation .......................................................................... 25
2.3. Related Work ........................................................................................................ 29
  2.3.1. Introvirt .............................................................................................. 31
  2.3.2. Livewire ............................................................................................. 32
  2.3.3. XenAccess .......................................................................................... 33
  2.3.4. Lares .................................................................................................... 34
  2.3.5. VIX ..................................................................................................... 35
  2.3.6. VMWatcher ........................................................................................ 36
  2.3.7. Ant Farm .............................................................................................. 37
  2.3.8. LycosID ............................................................................................. 38
  2.3.9. Manitou ............................................................................................... 38
  2.3.10. Patagonix ............................................................................................. 39
  2.3.11. Revirt ................................................................................................. 40
  2.3.12. Wizard ................................................................................................. 41
III. Methodology ......................................................................................................................................... 43

3.1. Problem Definition .......................................................................................................................... 43
3.2. CMAT-V Software Developmental Approach .................................................................................... 44
   3.2.1. Overall Design .......................................................................................................................... 44
   3.2.2. Software Integration ................................................................................................................. 48
   3.2.3. Registry Search Optimization .................................................................................................... 53
3.3. Experimental Design ....................................................................................................................... 54
   3.3.1. Threat Awareness Testing ......................................................................................................... 54
   3.3.2. CMAT-V Detection and Multi-VM Testing .................................................................................. 65
   3.3.3. Host System Performance Testing .............................................................................................. 74

IV. Results .............................................................................................................................................. 79

4.1. Threat Awareness ............................................................................................................................. 79
   4.1.1. FU Results ................................................................................................................................. 80
   4.1.2. Hacker Defender Results .......................................................................................................... 81
   4.1.3. Vanquish Results ...................................................................................................................... 84
   4.1.4. HideProcess Results ................................................................................................................. 85
   4.1.5. Conclusion ................................................................................................................................. 86
4.2. CMAT-V Detection and Multi-VM Testing ....................................................................................... 87
   4.2.1. INTmem Benchmark .................................................................................................................. 87
   4.2.2. FLOATmem Performance ......................................................................................................... 95
   4.2.3. Conclusion ............................................................................................................................... 100
4.3. Host System Performance ................................................................................................................. 101
   4.3.1. INTmem Benchmark .................................................................................................................. 101
   4.3.2. FLOATmem Benchmark ............................................................................................................ 105
   4.3.3. FFmpeg Benchmark ................................................................................................................... 108
   4.3.4. Conclusion ............................................................................................................................... 111

V. Conclusions ........................................................................................................................................ 112

5.1. Executive Summary .......................................................................................................................... 112
   5.1.1. Create a VMI Prototype ............................................................................................................ 112
   5.1.2. Verify Live Analysis Under Cyber Attack ................................................................................... 114
   5.1.3. Evaluate the Detectability of VM Memory Latency ................................................................. 114
   5.1.4. Verify Multi-VM Live Analysis ................................................................................................. 115
   5.1.5. Measure System Overhead ...................................................................................................... 115
5.2. Future Work ..................................................................................................................................... 115
   5.2.1. Thread List Introspection ........................................................................................................... 116
   5.2.2. Response Time Analysis .......................................................................................................... 116
   5.2.3. Advanced Registry Search ....................................................................................................... 117
   5.2.4. Driver Detection ....................................................................................................................... 117
   5.2.5. IP Detection .............................................................................................................................. 117
5.3. Concluding Remarks .......................................................................................................................... 118
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Virtualization Architecture</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Ring Protection Levels</td>
<td>9</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Operating System Virtualization Adapted from [Gol08]</td>
<td>11</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Hardware Emulation Adapted from [Gol08]</td>
<td>12</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Full Virtualization Adapted from [Mat08]</td>
<td>14</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Xen Paravirtualization Adapted from [Gol08]</td>
<td>16</td>
</tr>
<tr>
<td>Figure 7</td>
<td>VMI Model Information Flow</td>
<td>25</td>
</tr>
<tr>
<td>Figure 8</td>
<td>IntroVirt System Structure</td>
<td>31</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Livewire Intrusion Detection System (IDS) Architecture</td>
<td>32</td>
</tr>
<tr>
<td>Figure 10</td>
<td>XenAccess VMI Using a Kernel Symbol</td>
<td>33</td>
</tr>
<tr>
<td>Figure 11</td>
<td>High-level View of the Lares Architecture and its Core Components</td>
<td>34</td>
</tr>
<tr>
<td>Figure 12</td>
<td>The VMWatcher Approach</td>
<td>36</td>
</tr>
<tr>
<td>Figure 13</td>
<td>The Patagonix Architecture</td>
<td>39</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Normal Behaviors for <code>read</code> Kernel Service Handler [Sri10]</td>
<td>41</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Abnormal Behaviors for <code>read</code> Kernel Service Handler [Sri10]</td>
<td>41</td>
</tr>
<tr>
<td>Figure 16</td>
<td>CMAT Memory Analysis Process Adapted from [Oko10b]</td>
<td>45</td>
</tr>
<tr>
<td>Figure 17</td>
<td>CMAT-V Architecture</td>
<td>47</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Enhancements to CMAT and XenAccess</td>
<td>48</td>
</tr>
<tr>
<td>Figure 19</td>
<td>XenAccess VM to Physical Memory Mapping</td>
<td>50</td>
</tr>
<tr>
<td>Figure 20</td>
<td>System Under Test: Malware Attack</td>
<td>54</td>
</tr>
<tr>
<td>Figure 21</td>
<td>VM System Configuration: Malware Attack</td>
<td>55</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Hacker Defender VM Configuration</td>
<td>60</td>
</tr>
<tr>
<td>Figure 23</td>
<td>System Under Test: VM Memory Latency and Multi-VM Analysis</td>
<td>66</td>
</tr>
<tr>
<td>Figure 24</td>
<td>System Configuration: VM Memory Latency and Multi-VM Analysis</td>
<td>66</td>
</tr>
<tr>
<td>Figure 25</td>
<td>System Under Test 3: Host Performance</td>
<td>75</td>
</tr>
<tr>
<td>Figure 26</td>
<td>CMAT-V Hidden Process Detection for FU Rootkit</td>
<td>80</td>
</tr>
<tr>
<td>Figure 27</td>
<td>CMAT-V Hidden Process Detection for Hacker Defender</td>
<td>81</td>
</tr>
<tr>
<td>Figure 28</td>
<td>CMAT-V Hidden File Detection for Hacker Defender</td>
<td>82</td>
</tr>
</tbody>
</table>
Figure 29. CMAT-V Hidden Port Detection for Hacker Defender ....................... 83
Figure 30. CMAT-V DLL Injection Detection for Vanquish .......................... 84
Figure 31. CMAT-V Hidden Process Detection for HideProcess .................... 85
Figure 32. Scatter Plot of VM INTmem Average Performance ......................... 87
Figure 33. Box Plot of VM INTmem Average Baseline Performance ................. 89
Figure 34. Boxplot of CMAT-V Impact on VM INTmem Performance ............... 90
Figure 35. VM Analysis: FLOATmem Scatter Plot of Average ......................... 95
Figure 36. Box Plot of VM FLOATmem Average Baseline Performance ............. 97
Figure 37. Boxplot of CMAT-V Impact on VM FLOATmem Performance .......... 98
Figure 38. Scatter Plot of Host INTmem Average Performance ...................... 102
Figure 39. Box Plot of Host INTmem Average Baseline Performance .............. 103
Figure 40. Box plot of CMAT-V Impact on Host INTmem Performance .......... 104
Figure 41. Scatter Plot of Host FLOATmem Average Performance ................... 105
Figure 42. Box Plot of Host FLOATmem Average Baseline Performance .......... 106
Figure 43. Boxplot of CMAT-V Impact on Host FLOATmem Performance .......... 107
Figure 44. Scatter Plot of Host FFmpeg Performance ...................................... 108
Figure 45. Box Plot of Host FFmpeg Baseline Performance .......................... 109
Figure 46. Boxplot of CMAT-V Impact on Host FFmpeg Performance ............... 110
List of Tables

Table 1. Comparison of View-Generation Properties................................. 29
Table 2. Executive Summary of Existing VMI Applications.......................... 30
Table 3. Antfarm Process Identification Techniques [Jon06]. ...................... 37
Table 4. Factors Under Test for VM Performance Experiment.......................... 68
Table 5. Configuration Sequence for VM Performance Experiment................. 72
Table 6. Factors Under Test for System Performance Test........................... 76
Table 7. Configuration Sequence for System Performance Test...................... 76
Table 8. Threat Awareness Results........................................................... 79
Table 9. ANOVA Results for VM INTmem Average Baseline Performance........ 89
Table 10. ANOVA of CMAT-V Impact on VM INTmem Performance.................. 91
Table 11. ANOVA Results for VM FLOATmem Average Baseline Performance..... 97
Table 12. ANOVA of CMAT-V Impact on VM FLOATmem Performance............. 99
Table 13. ANOVA Results for Host INTmem Average Baseline Performance...... 102
Table 14. ANOVA of Host INTmem Performance........................................ 104
Table 15. ANOVA Results for Host FLOATmem Average Baseline Performance.... 105
Table 16. ANOVA of Host FLOATmem Performance..................................... 107
Table 17. ANOVA Results for Host FFmpeg Baseline Performance................ 109
Table 18. ANOVA of Host FFmpeg Performance.......................................... 111
Table 19. CMAT-V Application Summary.................................................. 113
I. Introduction

From the dawn of military aviation, strategic positions in air and space have been leveraged for reconnaissance purposes. Whether obtained by hot air balloons or unmanned aerial vehicles (UAVs), timely and trustworthy intelligence remains paramount to execute strategic military operations. With the emergence of the cyberspace domain, such a privileged position is equally critical. General Robert Kehler, Commander Air Force Space Command (AFSPC) highlights the “move from situational awareness to situational comprehension” in cyberspace as a primary military objective [Keh09]. Strategies outlined to accomplish this are to “fuse cyber intelligence to deliver proactive, responsive operational cyber capabilities” and “develop, refine and apply data mining and visualization technologies” in the cyber fight [Keh09]. In addition, the Air Force Research Laboratory has identified the need to “detect and defeat threats through active defenses” as a focused long term challenge (FLTC) [Tur08]. As cutting edge virtualization technology continues to be utilized for military and commercial use, the hypervisor provides this trusted higher-ground to obtain cyber situational awareness.

Desktop and server virtualization has emerged as an efficient and cost effective alternative to traditional “one-box-per-user” systems for a given set of physical hardware. In addition, new microprocessor architectures have emerged that are specifically designed to support virtualization. Due to the hypervisors privileged view of the state of
its virtual machines (VMs), virtualization has been utilized as an effective environment for forensic analysis and intrusion detection systems. The virtual machine manager (VMM) located in the hypervisor provides a trusted view of system. As such, virtual machine introspection (VMI) utilities have been developed to collect and analyze VM state information. Because VMs are abstracted from the VMM itself, the hypervisor can collect raw memory data from the VM. A VMI utility must also provide or derive VM specific context in order to extract useful system state information [Hay08]. By leveraging this information, network operators can quickly detect and engage cyber threats.

1.1. Goals

The goal of this research is to investigate the feasibility of live VMI analysis and its effectiveness to provide a multidimensional view of the live system state. The tasks to accomplish this are divided into the categories of software development and performance testing.

Software Development

- *Create a prototype VMI tool that will access live VM memory, detect the OS executing in the VM and extract system state information.* This research covers new ground in VMI research by using innovative techniques to analyze dynamic VM memory in real time. The VMI prototype must be capable of extracting and interpreting system state data without pausing the VM itself. Not only this, but information about the OS running on the VM must not be explicitly provided to
the VMI utility. Instead, this information must be passively detected. Success is evaluated by the ability to engineer software that meets these conditions.

**Performance Testing**

- **Verify the dependability of live VM analysis while under cyber attack.** For a prototype VMI utility to be useful for network defense, it must be reliable while the VM is under attack. The VMI program must present the user with relevant and trustworthy information so that the health of the system can be determined. For example, the prototype might be able to show the active process list of a healthy system, but what if the VM is executing malicious software designed specifically to evade detection? Furthermore, the methods used by malware to hide its presence on the target system are numerous and are only limited by the imagination of the attacker. To prove the dependability of the VMI utility, it must show that it is capable of successfully providing evidence of several different types of malware.

- **Verify the ability to conduct live analysis on two simultaneously executing VMs.** When using virtualization, it is desirable to run multiple VMs supported by the same hypervisor. As such, using the VMI prototype, two different VMs executing simultaneously must be analyzed to show that the prototype is not limited to one VM, but can support multiple active VMs.

- **Evaluate the detectability of VM memory latency caused by live analysis.** The VMI prototype must also scan and analyze the VM’s dynamic memory without
the VM noticing unique changes in performance. If malware executing in the VM is able to detect the VMI sensor, it can react and cause additional damage to the system. This goal uses selected benchmarks to objectively evaluate whether or not the VMI prototype’s live analysis causes detectable memory latency noticeable from the VM.

- **Measure the system resource overhead of the VMI utility on the host.** This goal addresses the impact of CMAT-V to the hypervisor itself. A VMI utility must not excessively drain system resources such that the supported VMs cannot operate effectively. Running the VMI prototype likely results in longer CPU times and increased memory access time. These values must be quantified. Benchmarks are used while the VMI prototype is not executing and then while under the stress of the VMI prototype. The difference between these benchmark results is compared to quantify the change in performance. This information allows the user to know precisely what the impact the utility will have on the hypervisor.

1.2. **Assumptions**

This research is be conducted under several assumptions. First, for accurate VM analysis, the state of the VMM itself must be trustworthy. The VMM itself is assumed to be uncompromised throughout the experiment. As such, it is assumed that complete VM-to-VMM isolation exists. This also means that the VM cannot detect when the VMI application is in use. Second, Windows is the only VM operating system tested during this research effort. The United States Air Force (USAF) has entered a license agreement with Microsoft to make Microsoft Windows part of the USAF Standard Desktop
Configuration [Lan10]. This leads to the third assumption that the OS follows the software architecture and data structure format as defined by Microsoft. Once the Windows version is detected, the VMI test bed automatically looks up guest OS semantic information by accessing Microsoft’s symbol server. The VMI application uses techniques that leverage known OS data structures. For example, the EPROCESS linked list structure tracks active processes on the system [Rus09].

1.3. Thesis Overview

This chapter presents an introduction to the research effort. The introduction includes motivation, goals and assumptions made.

Chapter 2 provides background information on virtualization, virtual machine introspection and related work in the area of VMI development. First the foundational concepts behind virtualization are described and the differences between virtualization implementations are highlighted. Then, the motivation for VMI is presented. A formal model that characterizes VMI approaches is also discussed. Finally, using elements from the formal model, an overview of several existing VMI applications is presented.

Chapter 3 describes the experimental methodology. First the problem definition is stated. The next section outlines the approach used for VMI tool development. The last section describes the experimental design used to test the tool’s effectiveness for threat awareness as well as its system performance.

Chapter 4 presents the results of all experiments. This section includes all statistical analysis and describes the meaning of the results.
Chapter 5 contains a summary of the results, conclusions made from the study as well as recommendations for future work.
II. Background

The following sections give an overview of fundamental concepts related to virtualization and virtual machine introspection (VMI). Section 2.1 describes the concept of virtualization and highlights the differences between virtualization methods. In Section 2.2, the motivation for VMI is presented along with a formal model for characterizing VMI techniques. Finally, Section 2.3 presents an overview of several different VMI applications.

2.1. Virtualization

Virtualization is a technique in which hardware resources of a physical host computer are shared to allow multiple guest operating systems (OSs) to run on a single host machine. A general overview of a virtualized architecture is shown in Figure 1.

![Virtualization Architecture](figure1.png)

Figure 1. Virtualization Architecture.
Guest operating systems, known as virtual machines (VMs), run independently from one another, completely unaware that other VMs exist. Hardware resources are controlled by a hypervisor which interfaces between the physical hardware layer and guest VMs. The hypervisor, or virtual machine monitor (VMM), abstracts the physical hardware layer into virtualized hardware for VM use. The VM operates as if it is using physical hardware, unaware of the abstraction that has taken place. The hypervisor allocates either exclusive or shared host resources to the VM. Exclusive resources are used because either the virtualization technology doesn’t allow sharing or the particular application requires dedicated resources for certain VMs. Whether resources are exclusive or shared however, the number of ported VMs allowed on a particular system is limited to the physical hardware resources of the host computer. The minimum requirements for a particular VM are dictated by the OS distribution and the virtualization method used. At any time, the state of a VM can be saved to an image file which can be quickly restored on the same machine or can be ported to other host machines [Mat08][Gol08].

Virtualization is not a new concept. In fact, operating systems have been using virtualization for quite some time, allowing the applications executing in the operating system to simultaneously access and share hardware resources. In past implementations however, hypervisors constrained users to a particular operating system which further limited the system to only OS compatible applications. Current virtualization methods more efficiently utilize the x86 protection ring architecture to allow different operating systems to utilize system resources [Mat08][Gol08].
2.1.1. Protection Ring Architecture

The x86 architecture contains four privilege levels that allow regulated access to a system’s hardware resources. These privilege levels, as shown in Figure 2, are referred to as rings and range from ring 0 (most privileged) to ring 3 (least privileged) [Int10][Bar03].

![Protection Rings](image)

**Figure 2. Ring Protection Levels**
[Int10].

A higher level ring (e.g., level 3) must request access to data structures and routines from a lower level (e.g., level 0) [Sil10]. The main advantage of this structure is its usefulness in debugging software. Once lower ring level routines have been debugged, higher ring levels may reliably use its lower level calls. Consequently, if an error occurs, it can be assumed that the error was caused by the current layer under test.
In non-virtualized architectures, critical code modules like the operating system kernel typically reside in ring 0. This layer directly interacts with the hardware on the system. With virtualization however, the hypervisor often runs in ring 0 as well. The interaction between the hypervisor and the OS kernel depends on the method of virtualization used. The difference between these methods will now be discussed.

2.1.2. Virtualization Methods

The four main virtualization methods in use today are OS virtualization, hardware emulation, full virtualization and paravirtualization. The three main elements that differentiate virtualization methods are the location of the virtualization layer, the modification required for the guest operating systems and the performance impact of virtualization on the system. The similarities and differences of each method will be discussed in the following sections.

2.1.2.1. OS Virtualization

With operating system virtualization, the same OS as the host is installed on top of an existing operating system kernel. These isolated program execution environments are referred to as *virtual environments* (VE). Similar to a VM, the processes that run in a VE are isolated from each other (each having its own IP address, software configurations, etc); however, the resources they share are not isolated. OS virtualization does not use the VMM to control use of physical resources. If one VE decides to take resources for itself, it negatively affects the performance of the other VEs. Each VE is also referred to as a *container* that isolates its operation from the other VEs [Kol06][SWs05].
During OS virtualization, the host operating system executes virtualization software allowing multiple guest functionality. As shown in Figure 3, the container virtualization layer interacts with the host OS kernel to coordinate between the VEs and the underlying hardware. Each VE must be the same OS as the host; therefore no modification is required to the guest OS.

The benefit of this method is that it requires the least amount of hypervisor overhead to implement due to shared OS processes. Hypervisor overhead involves using system resources like CPU and memory. Despite the benefit of a thinner hypervisor, the VEs are still limited to same OS as the host which makes OS virtualization inflexible.
For many users, OS virtualization is insufficient because it cannot support different operating systems on the same physical system. Examples of OS virtualization include: OpenVZ [Kol06], Virtuozzo [Par10], Linux VServers [Pot09] and FreeBSD jails [Fre10].

2.1.2.2. Hardware Emulation

In hardware emulation, as shown in Figure 4, hypervisor emulation software creates emulated versions of the underlying physical hardware.
This emulated hardware environment acts as a VMM and is compatible with OS specific system calls. When a guest OS makes a system call, the VM interacts only with the emulated hardware provided by the hypervisor. The hypervisor then translates the calls from the emulated hardware and sends them to the physical hardware. This method allows the host to support a virtual machine of a foreign computing architecture [Bia06]. For example, if an Advanced Micro Devices (AMD) microprocessor is used, hardware emulation could still support an OS that might require an Intel microprocessor. As a result, hardware emulation offers great OS flexibility. However, due to the high overhead required to translate instructions from different architectures to that of the host, this method greatly decreases system performance [Sun10]. For this reason some hardware emulation software developers advertize decreased CPU performance ranging from 1/500 to 1/15 the speed of the host [Bia06]. Examples of hardware emulation include: PearPC [Bia06] and Bochs [Law09].

2.1.2.3. Full Virtualization

Full virtualization is similar to hardware emulation because it also allows different unmodified operating systems to run inside a virtual machine. With full virtualization however, instead of making calls to emulated hardware, the virtual machine guests run code directly on the physical hardware used by the host. Compatibility issues between a guest OS and the underlying architecture are handled by the hypervisor using a technique called binary translation. Binary translation is a technique used to intercept instructions calls made by an OS for a particular architecture and convert the instructions so that they can be recognized by a non-native architecture [Sit93].
As shown in Figure 5, rather than emulating the whole hardware architecture, full virtualization uses the hypervisor to conduct real time binary translation which makes VM instructions compatible with the architecture of the host. The VMM in this case, still regulates which VM’s have access to the host’s resources, but once given access, the guest VM’s have direct access to the physical hardware [Mat08][Gol08].

As virtualization has become more popular, hardware developers such as Intel and AMD have created the Intel VT and AMD-V architectures to better support virtualization [Int10][Adv10]. This technology is commonly referred to as hardware assisted
virtualization (HAV). In HAV, new hardware extensions and privilege sublevels are introduced to specifically manage VM control. Utilizing HAV improves binary translation performance or removes the need for it altogether [Int10]. Though full virtualization requires some overhead for the hypervisor layer controls, it allows VMs to run at near-native performance of the host [Sun10]. Examples of full virtualization include: *VMware Server* [VMw10b], *ESX Server* [VMw10a] and *VirtualBox* [Sun10].

2.1.2.4. *Paravirtualization*

All of the previously discussed virtualization methods required a host operating system kernel to execute priority system calls to the hypervisor which translated the calls for native hardware compatibility. Paravirtualization, however, presents a guest OS with a modified version of the actual physical hardware that allows VM’s direct access to lower level hardware. Xen, an open source paravirtualization package, allows *hypercalls* to be used rather than typical system calls to access privileged system resources. This allows the same memory to be accessed by two different processes. Paravirtualization also is particularly helpful in sharing information between VMs [Mat08].
As shown on the left half of Figure 6, in order for paravirtualization to take place each VM OS must be compatible with the architecture of the underlying VMM. In light of this requirement, the OS kernel for each VM must be slightly modified to ensure compatibility with the Xen hypervisor. Though only small changes to the OS kernel are necessary, this modification is not possible with closed-source operating systems such as Microsoft Windows. For this reason, Xen also supports full virtualization utilizing HAV technology as shown on the right half of Figure 6. These HAV VMs or hardware virtual machines (HVMs) can be built around both open source and closed source operating systems [Mat08].
Similar to typical ring protection, Xen paravirtualization utilizes a *domain* level architecture. The privileged domain (Dom0) is equivalent to ring 0 while user domain (DomU) refers to rings 1 through 3. Similar to previously discussed virtualization methods, a Dom0 hypervisor manages resource allocation of the underlying physical hardware. In the case of a Dom0 hypervisor, user level software resides in DomU with the operating system kernel in ring 1 and the user applications in ring 3 [Mat08].

In addition, Xen provides a special domain to hold the device drivers for the guest VMs. This domain is called the driver domain. The purpose of offloading driver complexity from Dom0 is to make the system more stable should a driver error occur. With this design, the driver can be stopped and restored without crashing the whole system.

The primary advantages of paravirtualization are its low overhead for the VMM layer and high performance due to direct access to the physical hardware. The main disadvantage with paravirtualization is its moderate inflexibility because it requires modification to the guest OS source code. This limitation has been remedied however with the introduction of HAV technology. Examples of paravirtualization software include: *Xen* [Mat08] and *User-mode Linux (UML)* [Dik10].

### 2.1.3. Benefits of Virtualization

There are many potential benefits that come with virtualization. These benefits include but are not limited to increased hardware efficiency, lower network maintenance costs, effective malware isolation and a more robust environment for software testing.
2.1.3.1. Increased Hardware Efficiency

As hardware performance increases according to Moore’s law, computing resources are often underutilized [Mat08][Gol08]. Virtualization has recently emerged as a solution to this problem. By allowing several VMs to utilize hardware resources on a single machine, virtualization provides network administrators with increased hardware efficiency. This is particularly advantageous as systems now include multiple cores and increased RAM sizes. As a result, fewer physical systems need to be purchased to accomplish equivalent tasks. For example, virtualization has been proposed as a low cost solution for creating realistic training environments for cyber security education [Ste09]. Virtualization allows organizations to leverage existing network PCs rather than upgrade to new hardware.

2.1.3.2. Lower Maintenance Cost

Another benefit of virtualization is decreased network maintenance cost. As the number of physical systems decrease, so does the workload for system maintenance personnel to replace parts, monitor resource usage, etc. Virtualization also allows remote updating of new software or VM images.

2.1.3.3. Effective Malware Isolation

Despite the fact that the security implications of virtualization are highly debated, virtualization has been shown to provide some degree security benefits [Mor09]. Because guest VMs are isolated from the host system, virtualization tools can be used against malware attacks to effectively sandbox the compromised VM from the rest of the
network. The compromised VM can be then ported to another host to be analyzed while
a clean copy of the VM image can then be loaded to restore the system to a safe state.

2.1.3.4. Robust Software Testing Environment

Virtualization provides developers a stable system in which to test new OS
software. Because each VM is isolated from the underlying host, an OS experiencing a
system crash in a VM does not take down the whole system. Developers can quickly
restore the VM without having to make a hard reboot of the system.

2.2. Virtual Machine Introspection

Virtual machine introspection (VMI) is defined as “… [An] approach of
inspecting a virtual machine from the outside for the purpose of analyzing the software
running inside it” [Gar03]. The following section describes the motivation for VMI, the
semantic gap challenge between the VM and the VMM, a formal model used to describe
VMI, methods for VMI detection and finally an overview of the capabilities of existing
VMI applications.

2.2.1. Motivation

The emergence of VMI is closely coupled to the increased interest in
virtualization over the last several years. The idea of live analysis of systems however is
not a new concept. The following sections describe some of the limitations of static
analysis that brought about the need for live analysis techniques.
2.2.1.1. Static Analysis

Static analysis refers to the traditional approach where a target system is halted so that all storage media can be copied and used for forensic analysis. Over time statistic analysis tools such as Encase [Gui10] and FTK [Acc10] have proven their value to locate and extract useful information. Static analysis alone, however, is limited in the information provided. Shutting down the system could cause system updates to install or the termination of running applications, both of which significantly affect storage media. Similarly, pulling the plug can cause significant data inconsistencies and synchronization issues [Hay09]. Halting the target system also results in a loss of all volatile system information. Data such as open ports, active network connections, running programs, temporary data, user interaction, encryption keys, RAM and cache are unrecoverable. If the encryption key is lost for protected volumes, recovery techniques must be used which are not always successful. Lastly, the downtime required to analyze the system poses a great inconvenience to the user. These limitations highlight the need to supplement traditional static analysis with live analysis techniques [Hay09].

2.2.1.2. Live Analysis

Nonquiescent or live analysis allows the forensic investigator to interrogate the system while it is running. Live analysis techniques include using installed user level applications [Sym10][Sou10][Wir10], using imported utilities (i.e., CD-ROM, USB, etc.) [DMZ10], implementing system modifications (i.e., dividing production and security processes on different processors) [Wil05] or using additional hardware (i.e., a PCI hardware expansion card memory scanner) [Car04]. With these approaches, investigators
have access to both static and dynamic memory as well as information about processes currently in execution. In spite of these benefits however, these non-virtualized live analysis methods have several drawbacks. Live analysis is susceptible to observer effects such that any probing done to the system inadvertently changes the state of the system itself. This makes preserving the integrity of the system state while using live analysis particularly difficult. In addition, if the system is compromised, the attackers can modify, hide or deny access to system data [Hay09]. However, using VMI for live analysis overcomes many of these challenges.

VMI inherently provides a more secure environment for live analysis to take place by allowing isolation, inspection and interposition [Gar03]. VMs are at a lower privilege than the VMM; therefore, the VM is not aware of or given access to the underlying hardware that supports its virtual hardware [Hay09]. This makes the VMM a prime candidate for system monitoring processes. The monitoring process is completely isolated and therefore is not susceptible to malicious modification. In addition, the VMM has complete access to inspect the state of the VM. The VMM can acquire the VM’s raw state which includes CPU state, all memory, I/O device states and I/O controller states [Gar03]. Finally, a VMI-based monitoring system allows a preconfigured VMM to interpose on certain virtual machine operations and flag when a VM attempts particular actions [Gar03].

Though the privileged level of the VMM allows complete oversight of the VM state, knowing where to look within OS specific data structures is not a trivial task and is discussed in the following section.
2.2.2. *The Semantic Gap*

A well known challenge in VMI research is the lack of specific knowledge of the guest VM’s state, commonly referred to as the semantic gap [Che01]. The word semantic is defined as, “of or relating to meaning in language” [Mer10]. Because the interface between the VMM and the VM is inherently designed to isolate the guest VM, it is difficult for the VMM to interpret meaningful state information from raw VM data. Semantic information includes but is not limited to hardware architectures, operating systems, data structures, running processes, system functions, performance goals and security policies. This semantic information is important for meaningful live analysis of VMs. For example, information obtained by probing VM system memory can be complex to interpret without knowledge of the VM’s OS specific architecture. As a result, many VMI applications require a priori semantic information as a road map to effectively monitor VM activity. This approach decreases VMI compatibility however because it closely couples the VMM to a specific operating system [Jon08].

Previous work has addressed the need for implicit semantic abstraction for both hardware [Bug97][Siv04][Wal02] and software [Pag09][Gar03]. For example, by analyzing a VM’s semantic performance at a hardware level, a VMM can reallocate least valuable pages in memory leading to more efficient memory utilization [Wal02]. Similarly, it has been shown that information about a particular OS running on a VM can also be extracted [Pag09]. Using known OS data structures, a VMM can detect the particular version of an operating system running in a guest VM. Knowing this, investigators can more strategically analyze VM activity. Specific implementations of
VMI methods will be discussed more in following sections, but first a more formal model for VMI is discussed to more precisely differentiate these techniques.

2.2.3. VMI Model

In [Pfo09], Pfoh, Schneider and Eckert highlight three main challenges VMI must overcome as (1) interpreting binary low-level data that comprise the system state to obtain abstract high level state information, (2) identifying the relevant parts of that state information and (3) classifying system state. With these challenges in mind, a formal model for VMI is proposed.

In the formal VMI model there are several terms used to describe VMI characteristics. First, $S$ is the set of all guest system states where $s \in S$ represents a particular system state. $S_{int}$ is the set of all VM states visible to the guest OS being monitored where $S_{int} \subseteq S$. This includes all introspection, like using a kernel debugger, which can be conducted inside the VM itself. $S_{ext}$ represents all possible states visible to an external source, like the VMM, where $S_{ext} \subseteq S$. In addition, $C$ is defined as the set of all possible classifications for a particular scenario. The ideal goal for a VMI implementation is to create some function $f : S_{int} \rightarrow C$ or $g : S_{ext} \rightarrow C$ that outputs a classification for a particular state. Because the characteristics that qualify as the system state are theoretically unbounded, a view must be generated to reduce the scope of the data. A view $V$ ideally contains only data relevant to determining classification. $V_{int}$ and $V_{ext}$ refer to the set of all possible views constructible from $S_{int}$ and $S_{ext}$ respectively [Pfo09].
The function that takes a state $s$ and outputs a corresponding view $v$ is made possible by a view generating function. This process uses knowledge about the guest system hardware and/or software architecture. A particular virtual hardware architecture description is defined as $\lambda$, and a guest system software architecture is defined as $\mu$. Additionally, a profile $p$ is defined as an aggregation of several consecutive views of a system run where $P$ represents the set of all possible profiles. The aggregation $a$ takes all profiles along with the current profile to create a new profile. The profile is then processed by function $d$ to determine the classification of the current state. The following functions are defined formally below [Pfo09].

$$f_{\lambda,\mu}: S_{int} \rightarrow V_{int}$$

$$g_{\lambda,\mu}: S_{ext} \rightarrow V_{ext}$$

$$a: V_{int} \times V_{ext} \times P \rightarrow P$$

$$d: P \rightarrow C$$

Where $f_{\lambda,\mu}$ describes internal view-generation; $g_{\lambda,\mu}$ describes external view-generation; $a$ describes aggregation; $d$ describes classification.

The information flow for the VMI model is shown in Figure 7 below. Using this model, unique VMI implementations can be more precisely described. As such, elements from this model will be referenced throughout the remainder of the paper.
2.2.4. **VMI View-Generation**

The stage of the VMI model of interest to this research is the view generating functions that begin with a state $s$ and output a corresponding view $v$. In this step the semantic gap is bridged, providing the investigator not only with raw data, but with meaning behind the data. The following describes three view-generation methods that differ based on where the view-generation takes place (internal or external) and the type of semantic information used in the generating function. These methods may be used individually or in combination [Pfo09].

2.2.4.1. **Out-of-Band Delivery**

The most prevalently used approach to VMI view generation is referred to as out-of-band delivery which uses a priori semantic knowledge about the guest VM software architecture. With this advanced knowledge, the VMM queries the VM externally (out-of-band) to deliver specific state information. A formal description is shown below [Pfo09].
Implementing out-of-band delivery of VM state information, also referred to as explicit information [Jon08], has both advantages and disadvantages. Because the information is gathered by the VMM, the VMI application has an omniscient view of the VM state. Utilizing the VMM maintains isolation from attacks to the VMI itself. View consistency is also maintained by interrogating the VM while it is in a paused state. The main disadvantage of out-of-band delivery is that it inherently relies on semantic information about a particular OS architecture. If the assumptions made about the VM’s software architecture change (via patches, updates, etc.) the VMI method is often rendered invalid [Pfo09]. Any information gathered with this method would be considered nonbinding. This term describes the common occurrence where malware is not bound to maintain the semantics implied by the OS symbol information. When considering rootkits for example, “if the hypervisor uses non-binding information about the format or location of kernel data structures, the rootkit may evade detection by adding fields to the data structures or moving the data structures to a memory location that is not being monitored” [Lit08]. A VMI utility that doesn’t adapt to changes made in OS patch updates runs the risk of reporting an invalid view of the system state.
2.2.4.2. In-Band Delivery

In-band delivery of information originates from the guest VM. Using inherent OS knowledge of software architecture and function, the system state is reported to the VMM. A formal description is shown below [Pfo09].

\[ f_\mu : S_{int} \rightarrow V_{int} \]

\[ a : V_{int} \times P \rightarrow P \]

\[ d : P \rightarrow C \]

Though the guest OS has the most accurate semantic information about the software architecture, there are several disadvantages to this approach. First, this method relies heavily on the trustworthiness of the VM. Rootkits, for example, designed to hide applications could compromise data reported by the guest OS [Hog06]. This would mean that the function \( f_\mu \) would give an inaccurate view of the system state. In addition, the system cannot be paused which could cause inconsistency issues. By the time the view generation function reports the system state to the user, the VM system state might have changed. Finally, because the view-generating function resides within the VM, the function is likely to interfere with the system state itself [Pfo09].

In-band delivery is most commonly used in combination with other methods for view comparison. If two views report differing results, this informs the investigator that the VM is possibly compromised.
2.2.4.3. Derivation

Derivation, also referred to as implicit introspection [Jon08], uses semantic hardware architecture information to derive a view of the system state. This more passive approach relies on hardware specific activity such as interrupts, page faults, and I/O requests [Wil05]. The derivation method is described formally below [Pfo09].

\[ g_A : S_{ext} \rightarrow V_{ext} \]

\[ a : V_{ext} \times P \rightarrow P \]

\[ d : P \rightarrow C \]

Because this method uses semantic information about the system hardware architecture this approach overcomes the non-binding nature of the methods discussed previously. In addition, this method reports the true system state even if the guest OS has been compromised. Unless malware alters external interfaces, all running processes must function within the bounds of the virtual and physical hardware architecture. There are disadvantages to the derivation approach however. Interpretation of hardware level activity is a more difficult task than analysis at a software level. Low level activity provides little or no context for extract user level intentions. In light of this, the view generated from guest system state information is limited in scope [Jon08][Pfo09].

2.2.4.4. Combination

As mentioned previously, using a combination of in-band, out-of-band and/or derivation methods for view-generation can provide a more complete and accurate view
of the system state. For example cross view validation can be used to determine the differences between information reported by the VM and what is gathered by the VMM [Wan05]. A summary of the properties for each of these methods is shown in Table 1.

Table 1. Comparison of View-Generation Properties [Pfo09].

<table>
<thead>
<tr>
<th>Property</th>
<th>Delivery</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in-band</td>
<td>out-of-band</td>
</tr>
<tr>
<td>HW Portability</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Guest OS Portability</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Binding</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Isolation from Guest OS</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Inspection of Suspended VM</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Full State Visibility</td>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>

Though leveraging the strengths of each method discussed can be effective, applying multiple approaches will not necessarily produce the desired results. When selecting or combining view generation methods, suitability and unwanted interaction effects must also be considered [Pfo09].

2.3. Related Work

An executive summary of previous work in the area of VMI is shown in Table 2. Many VMI systems have been developed for general VM monitoring, intrusion detection and event replay. In the sections to follow, each VMI application is briefly discussed in further detail.
Table 2. Executive Summary of Existing VMI Applications.

<table>
<thead>
<tr>
<th>Sec.</th>
<th>Name</th>
<th>VMM(s) Tested</th>
<th>OS(s) Tested</th>
<th>VMI Pattern</th>
<th>Description</th>
<th>Limitations</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.1</td>
<td>Introvirt</td>
<td>UMLinux</td>
<td>Linux</td>
<td>In-band</td>
<td>Uses vulnerability-specific knowledge to invoke code on guest OS for system state reporting.</td>
<td>If compromised, VM cannot be trusted. Non-binding.</td>
<td>Jos05</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Livewire</td>
<td>VMWare</td>
<td>Linux</td>
<td>Out-of-Band</td>
<td>Interprets system state via VMM to access VM memory. Uses provided OS interface library for semantic context. Uses hooking for intrusion detection.</td>
<td>OS interface library must be provided. Requires VM interference. Non-binding.</td>
<td>Gar03</td>
</tr>
<tr>
<td>2.3.3</td>
<td>XenAccess</td>
<td>Xen</td>
<td>Windows</td>
<td>Out-of-Band</td>
<td>Uses user-provided OS semantic info to map VM memory.</td>
<td>OS information must be provided. Non-binding.</td>
<td>Pay07</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Lares</td>
<td>Xen</td>
<td>Windows</td>
<td>In &amp; Out-of-Band</td>
<td>Uses trampoline in VM to intercept events.</td>
<td>VM interference. Non-binding.</td>
<td>Pay08</td>
</tr>
<tr>
<td>2.3.5</td>
<td>VIX</td>
<td>Xen</td>
<td>Linux</td>
<td>Out-of-Band</td>
<td>Pauses VM, accesses VM memory, displays system data.</td>
<td>VM interference. Non-binding.</td>
<td>Hay08b</td>
</tr>
<tr>
<td>2.3.6</td>
<td>VMWatcher</td>
<td>VMWare, Xen, QEMU, UML</td>
<td>Windows Linux</td>
<td>Out-of-Band</td>
<td>Accesses VM state using user provided OS templates then uses COTS anti-malware software to make classification.</td>
<td>OS templates must be provided. Vulnerable to malware that targets COTS anti-malware software. Non-binding.</td>
<td>Jia07</td>
</tr>
<tr>
<td>2.3.7</td>
<td>AntFarm</td>
<td>Xen, Simics</td>
<td>Windows</td>
<td>Derivation</td>
<td>Tracks process creation, context switches and exits.</td>
<td>No user level context or VM-state classification.</td>
<td>Jon06</td>
</tr>
<tr>
<td>2.3.8</td>
<td>LycosID</td>
<td>Xen</td>
<td>Windows</td>
<td>Derivation</td>
<td>Uses Ant Farm data along with CPU inflation technique to compare the lengths of process lists at trusted and non-trusted levels.</td>
<td>VM interference.</td>
<td>Jon08</td>
</tr>
<tr>
<td>2.3.9</td>
<td>Manitou</td>
<td>Xen</td>
<td>Windows</td>
<td>Derivation</td>
<td>Assigns, tracks, and revokes code execution. Applications submit trusted code hashes and Manitou and disallows any untrusted code execution.</td>
<td>Dependant on hashes from applications.</td>
<td>Lit06</td>
</tr>
<tr>
<td>2.3.10</td>
<td>Patagonix</td>
<td>Xen</td>
<td>Windows</td>
<td>Derivation</td>
<td>Identifies covertly executing binaries. Administrator provides white-list of legitimate binaries and Patagonix disallows any untrusted code to execute.</td>
<td>Dependant on trusted binaries from administrator.</td>
<td>Lit08</td>
</tr>
<tr>
<td>2.3.11</td>
<td>Revirt</td>
<td>UMLinux</td>
<td>Linux</td>
<td>Derivation</td>
<td>Allows VM execution replay by creating VM checkpoints.</td>
<td>Requires knowledge of VM compromise. Does not allow live analysis.</td>
<td>Dun02</td>
</tr>
<tr>
<td>2.3.12</td>
<td>Wizard</td>
<td>Xen</td>
<td>Linux</td>
<td>Derivation</td>
<td>Implements training period to observe interaction between OS and VMM. Abnormal kernel handler functions are flagged.</td>
<td>Dependant on trustworthy training period results. No user-level context.</td>
<td>Sri10</td>
</tr>
</tbody>
</table>
2.3.1. IntroVirt

IntroVirt [Jos05] is a VMI system designed to monitor the execution of guest OS applications. The system uses a User Mode Linux VMM to support Linux guest and host VMs. IntroVirt uses in-band view generation by executing code that already exists in the guest.

An overview of the IntroVirt system architecture is shown in Figure 8. The system leverages specific OS semantic knowledge, including OS vulnerabilities, to determine the system state. This semantic knowledge is referred to as a predicate. These predicates must be provided to IntroVirt and are then used to detect the triggering of the vulnerability. During predicate execution, guest specific functions are used to present data back to the system. Classification of the system state can only occur after the vulnerability has already been discovered. With this method however, limitations of in-band view generation still apply. Data provided by a compromised guest cannot be trusted.
2.3.2. *Livewire*

As shown in Figure 9, Livewire [Gar03] uses an out-of-band view generation pattern by utilizing an OS interface library containing OS specific semantic information.

Figure 9. Livewire Intrusion Detection System (IDS) Architecture [Gar03].

Livewire can interpret the state of the VM by using the VMM to access pages from physical memory in combination with an OS interface library to provide context. Using VM hooks, the VMI intrusion detection system (IDS) communicates with the VMM to send event notification. The IDS then suspends the VM until given an administrative command to continue. Once the VM state has been retrieved, a policy engine component
determines the classification of the VM state. Livewire uses a VMWare VMM to support Linux guest and host VMs.

2.3.3. *XenAccess*

XenAccess [Pay07] monitors VM operating systems running on Xen virtualization software. XenAccess uses an out-of-bound view generation pattern. Using a priori semantic information about the VM operating system, XenAccess can map memory pages from domU to a local address range. An example of mapping using a kernel symbol is shown in Figure 10.

![Figure 10. XenAccess VMI Using a Kernel Symbol [Pay07].](image-url)
As shown in step 2 of Figure 10, a file `system.map` is used which contains OS specific kernel symbol information which is used to probe the guest VM. Using this information, XenAccess then enters the VM’s dynamic memory to gather raw system state data. In order to gather the raw data, XenAccess uses the Xen’s `libxc` and `libxenstore` libraries which provide functions to interface with the VM itself. For example, XenAccess uses the `xc_map_foreign_range()` function from `libxc` to view the memory of a guest VM. XenAccess supports VMI for both Linux and Windows based VMs.

### 2.3.4. *Lares*

Lares [Pay08] utilizes the functionality of XenAccess along with OS specific hooking locations to monitor process creation events. In-band and out-of-band view generation is implemented for the Lares IDS system. The overall Lares architecture is shown in Figure 11.

---

Figure 11. High-level View of the Lares Architecture and its Core Components [Pay08].
Through the use of a trampoline in the guest VM, the security VM intercepts events. This feature signifies an in-band technique. The security VM which is external to the guest VM then uses matching algorithms and heuristics to actively process the event. This feature signifies an out-of-band technique. Finally, a classification decision is made for the event. Though this is an effective approach it is not free from VM interference. The hooks installed in the guest VM inherently modify the VM state.

2.3.5. **VIX**

Virtual introspection for Xen (VIX) [Hay08] uses an out-of-band view generation pattern, where offset values based on the DomU system under examination are initialized upon VM creation. VIX utilizes the Xen Control Library to first pause the VM, access and decode memory, then unpause the VM to continue execution. This procedure is demonstrated by the pseudo-code example of the `vix-ps` utility shown below.

```
Pause DomU
Adr <- Address of Task List Head
Do
    Adr <- Adr.next_task_adr
    Map page(s) for Adr into Dom0
    Decode task_struct
    Display data
    Unmap page(s) with Adr
While (Adr != Address of Task List Head)
UnPause DomU
```
This utility lists the current processes being run in the VM. Using this utility enables VIX to conduct cross-view validation to detect basic rootkits. Though a classification is not made about the state of the VM, VIX gives the VMM contextual VM situational awareness. VIX has been shown to operate on Linux based VMs.

### 2.3.6. VMWatcher

VMWatcher [Jia07] is an IDS that uses an out-of-bounds view generation technique called *guest view casting*. This technique involves using guest OS semantic information as templates to interpret low-level VM states. The unique feature about VMWatcher is that it utilizes off-the-shelf anti-malware software to determine classifications on VM states.

![Figure 12. The VMWatcher Approach](Jia07).

As shown in Figure 12, these anti-malware systems run inside the host OS and in the VM itself. This allows VMWatcher to conduct cross-view validation by comparing anti-malware software results from the guest VM and the hypervisor host. Within the VM itself, the anti-malware systems can scan objects such as kernel modules (left circle),
processes (middle circle) and files (right). VMWatcher supports VMWare, Xen, QEMU and UML virtualization systems. Though this is an effective approach, this method is vulnerable to malware that targets commercially available anti-malware software.

2.3.7. Ant Farm

Antfarm [Jon06] uses a derivation view-generation pattern by observing how a guest uses a virtual memory management unit (MMU). Process creation, context switches and exits are monitored by tracking address spaces in which a process event occurs. The process identification techniques used by Antfarm are shown in Table 3.

Table 3. Antfarm Process Identification Techniques [Jon06].

<table>
<thead>
<tr>
<th></th>
<th>x86</th>
<th>SPARC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASID</td>
<td>Page directory PA</td>
<td>Context ID</td>
</tr>
<tr>
<td>Creation</td>
<td>New ASID</td>
<td>New ASID</td>
</tr>
<tr>
<td>Exit</td>
<td>No user mappings</td>
<td>Context demap</td>
</tr>
<tr>
<td></td>
<td>and TLB flushed</td>
<td></td>
</tr>
<tr>
<td>Context</td>
<td>CR3 change</td>
<td>Context ID change</td>
</tr>
<tr>
<td>switch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In order to track addresses spaces they are labeled with an address space identifier (ASID). By observing the ASID associated with an event it is possible to identify events such as process creation, process exit and context switching. To assign the ASID for the x86 architecture the physical address of the page directory is used; for the SPARC architecture, the virtual address space context ID is used. For each event of interest (creation, exit and context switch), Antfarm watches for certain activities to take place. These activities differ depending on the architecture used and are shown in Table 3. By
associating the activities with events of interest, Antfarm is able to draw conclusions about what how the MMU is being used.

Though no classification is made on the VM state, the VMM can extract useful system state information. Because Antfarm does not require explicit information about the layout or implementation of memory, the VMI technique is independent of the VM operating system. Antfarm supports Xen and Simics VMM’s and runs Linux or Windows as the guest OS.

2.3.8. LyosID

LyosID [Jon08] implements an IDS that uses derivation to obtain guest process information. LyosID leverages Antfarm’s ability to obtain a trusted view of the guest operating system then conducts cross-view validation to detect hidden processes. Finding hidden processes involves two steps. First, the difference $H$ in CPU time observed by the VMM and the VM is calculated. Then a technique called CPU inflation is used that inflates the CPU load of a given process. If $H$ increases as a result of the inflation, it is likely that the particular process is being hidden from the VM. This approach requires that the hidden process is not idle. Though CPU inflation is an effective technique, it is intrusive because it modifies the state of the VM itself.

2.3.9. Manitou

Manitou [Lit06] uses derivation at the microprocessor level for intrusion detection. Each new application must submit a signed list of code page hashes through a trusted path between the application and Manitou. A white list is created from the hashes
and is maintained by Manitou. Using semantic information about the underlying microprocessor, Manitou monitors each time a program tries to execute code from a specific page. If Manitou has not already marked the page executable, a page fault occurs. The page is then hashed and compared to the white list of allowable hashes. If the hash is found in the list, the code is allowed to execute, if not the unauthorized program is terminated. Manitou has been shown to run with the Xen hypervisor which supports Linux or Windows based operating systems [Lit06].

2.3.10. Patagonix

Patagonix [Lit08], similar to Manitou, uses derivation to look at microprocessor events; however, Patagonix focuses primarily on detecting hidden rootkits running in a VM. Figure 13 shows an overview of the Patagonix architecture.
As shown on the right side of the Patagonix VM in Figure 13, Patagonix keeps a white list of *identity oracles* which are created for each type of binary in the monitored VM. On the left side of the Patagonix VM, *control logic* compares executed code to the identity oracles and then presents the results with the *management console* interface between the user and Patagonix. Because Patagonix does not rely on the guest OS, an objective list of all running binaries is presented to the user. With this information, the user can kill malicious binaries running on the VM. There is also a *lie-detection* mode that uses cross-view validation between Patagonix and the VM OS to report hidden binaries. Similar to Manitou, Patagonix uses the Xen VMM and supports Linux and Windows based VMs.

### 2.3.11. Revirt

Revirt [Dun02] is an IDS that uses derivation techniques to enable event replay of a VM. A log is kept of every non-deterministic event that affects a process’s communication while the VM is running. Deterministic events include arithmetic computations, memory calculations, branch instructions that will re-execute the same way during replay. Non-deterministic events include the time an event occurred (to log interrupts for example) and any external input (such as input from a human). Revirt is designed to run on the UMLinux VMM. UMLinux uses software to emulate peripherals, system calls and interrupts. By tracking these emulation mechanisms, Revirt is able to identify the non-deterministic events. Periodic checkpoints of the system state are also created by suspending the VM. If an attack occurs, the VM state before, during and after
the attack can be restored, replayed and analyzed. This IDS approach is inherently reactionary and requires the administrator to know when the VM has been compromised.

2.3.12. Wizard

Wizard [Sri10] uses a derivation approach to detect kernel attacks. Developed for the Xen VMM, Wizard observes the interaction between the OS and the VMM rather than the memory state of the OS itself. Wizard enters a training period to record and characterize requests by applications running on the VM and the calls to the kernel. After the training period, Wizard checks for anomalies in the behavior of kernel handler functions. An example of the differences in normal and abnormal behavior is shown in Figures 14 and 15.

![Figure 14. Normal Behaviors for read Kernel Service Handler [Sri10].](image1)

![Figure 15. Abnormal Behaviors for read Kernel Service Handler [Sri10].](image2)
In Figure 14, each line shows the call sequence behaviors learned during the training period for the `read` system call executed when typing on the keyboard. Each call includes the system call with their parameter values and the associated interrupt handler. Figure 15 shows the behavior observed after the LVTES keylogger is installed. The lines in boldface indicate those behaviors that do not correlate. These abnormal behaviors can then be further investigated to determine if malware is present.
III. Methodology

The following sections detail the development of a compiled memory analysis tool for virtualization (CMAT-V) and the experimental methodology used during the research effort. Section 3.1 states the problem definition. Section 3.2 outlines the software development approach used to accomplish the goal of creating a CMAT-V. Recall from Section 1.2 that this goal requires the prototype to access live VM memory, detect the OS executing in the VM and extract system state information. Section 3.3 describes the experimental design for performance testing. This addresses the goals to test VMI cyber threat awareness, demonstrate multiple VM supportability, evaluate live analysis detection and measure host system overhead (see Section 1.2).

3.1. Problem Definition

Existing VMI applications are designed to leverage specific activities and architectures found in hardware or software to extract dynamic system state information about a guest VM. VMI techniques that target microprocessor-level hardware activities are effective at collecting data; however, the results they provide are conceptually separate from user-level context. This separation limits the scope of system state information that can be derived. VMI tools that target software characteristics are privy to user-level context which allows the tool to provide a more detailed perspective of the system state. As a consequence however, many software-based VMI tools require a priori OS semantic information to effectively extract useful forensic information. This limits the portability of such tools to operating systems of different distributions or versions. Also, such
techniques are often strictly tied to specific data structures in dynamic memory to extract information. This can provide an incomplete view of the system state by limiting the range of dynamic memory investigated. Other techniques have the potential to interfere with the state or execution of the VM itself. This interference could contaminate the VM itself or allow the inspection to be detected by the guest. This research investigates the feasibility of developing a non-interfering VMI tool that conducts a complete scan of dynamic memory. In addition, OS detection techniques are applied in an effort to objectively extract contextual VM system state information.

3.2. CMAT-V Software Developmental Approach

This section describes the overall software design of CMAT-V, modifications required for software integration and modifications to improve live analysis performance.

3.2.1. Overall Design

CMAT-V builds upon CMAT, a compiled memory analysis tool for static forensic analysis [Oko10a][Oko10b]. CMAT parses through a memory dump file to extract current users, open network ports, active processes, driver information, open files and registry keys.

Figure 16 shows the CMAT memory analysis process.
First, the addressing mode is detected as well as the location of the kernel page directory table base. Using this information, the location of the kernel executable and tcpip.sys are found. Then, the kernel executable’s globally unique identifier (GUID) and age are found. This GUID is a 32 character hexadecimal character string designed to uniquely identify a particular OS installation. The age is a one byte ID that denotes the linking level during compilation. Next, using the GUID and age, CMAT looks up semantic information by downloading a program database (PDB) file from Microsoft Symbol Server. The PDB file acts as a road map to give context to the data read from the memory dump file. Then this process is repeated for tcpip.sys so that key data structures can be identified which provide information about UDP and TCP network activity.
Finally, with the PDB information acquired, CMAT can extract process, registry user and network information. This is accomplished using previously developed techniques [Sch06][Dol08]. Schuster describes search patterns that can be used to scan a memory dump for process objects. Once found, these objects reveal active processes information [Sch06]. Dolan-Gavitt presents tools that also use a memory dump to locate register hives and use cell indices to locate specific key addresses in kernel memory. In addition, the Configuration Manager can be located which provides information about which keys are being accessed and what processes are accessing them [Dol08]. Using these same techniques, the registry can also be searched to find the users logged on to the system. For example, the registry keys located in the Windows registry directory

\Microsoft\Windows NT\CurrentVersion\ProfileList

contain information on about all users on the system. After these final steps, CMAT then presents the user with an interactive user interface to review the results. This process forms the foundation of CMAT-V.

Rather than use a memory dump file, CMAT-V is a prototype VMI application designed to conduct live forensic analysis of Windows-based guest VMs. Though the static analysis techniques used by CMAT are applicable for most virtualization software packages, CMAT-V is designed for compatibility with Xen [Mat08] virtualization software. Xen supports both paravirtualization and full virtualization modes. Because CMAT-V targets proprietary Windows-based guests, Xen is run in full virtualization mode. Figure 17 shows the overall CMAT-V architecture.
As shown on the left of Figure 17, CMAT-V utilizes Xen’s built in Hypervisor Management API (HM-API) to manage and monitor VM guests. The interface for the HM-API resides in a trusted Dom0 VM and uses the CentOS 5 operating system. CMAT-V also uses a modified version of XenAccess [Pay07] as a framework to interface between Xen and CMAT functions. Both Xen and XenAccess were chosen due their open source availability. CMAT, XenAccess and Xen are all written in the C programming language. This allows custom modifications to be made to achieve desired VMI functionality.

All software development and experiments are conducted on a Dell Latitude D630 laptop with an Intel Core 2 Duo T7300 processor, 2 GB of memory and a 120 GB hard drive. The processor includes Intel-VT technology which allows HAV mode operation. The OS used for VM guests during the malware attack scenarios and benchmarks is Windows XP SP3 with all available updates installed as of 6/10/2010. Each VM is configured with 512 MB of RAM, 10 GB hard disk memory and is allocated one CPU processor. The Dom0 OS is CentOS 5 which is allocated the RAM not being
used by VMs and the HM-API is allocated both CPUs. All Windows-based automatic update features, screensavers or any processes that could possibly interrupt the experiment are turned off in the VM and HM-API. Detailed procedures on how to install and set up Xen and the Windows XP virtual machines are described in Appendix A; instructions for installing CMAT-V are in Appendix B.

### 3.2.2. Software Integration

In order to create CMAT-V, this research made several enhancements to both CMAT and XenAccess. These enhancements are shown in yellow in Figure 18 and are further described.

![Figure 18. Enhancements to CMAT and XenAccess.](image-url)
First, CMAT-V must ensure that all functions do not assume a particular Windows distribution. Therefore, the default XenAccess initialization of a structure called `xa_instance` was modified. The `xa_instance` structure contains VM specific configuration information like VM number, VM name, size of memory, etc and is passed to nearly all XenAccess application programming interface (API) functions. By default, XenAccess uses two important user provided files called `System.map` (or `exports file`) and `xenaccess.conf` to initialize the `xa_instance` structure. `System.map` contains symbol information unique to the target VM. This requires that the user of XenAccess has prior knowledge of this OS semantic information. In addition, they must populate and create the file ahead of time. The configuration file `xenaccess.conf` includes the location of the `System.map` file as well as other VM specific information such as VM domain name and OS specific offsets.

To create CMAT-V, XenAccess was modified by removing the dependence on these user provided files. This involved removing all dependencies on these files in all XenAccess source files. Once this was completed, CMAT was used to derive semantic information from the memory itself. As discussed previously, OS specific symbol information is downloaded from the Microsoft Symbol Server. Care was taken to synchronize CMAT and XenAccess such that CMAT would initialize the `xa_instance` structure with all necessary symbol information before XenAccess required the use of the structure. XenAccess function calls that required `xa_instance` were delayed until after CMAT had populated the data structure. Once all these
modifications were accomplished, XenAccess no longer required any user provided configuration files.

Next, CMAT-V must integrate CMAT with XenAccess and Xen to give CMAT-V its ability access the live dynamic memory of a VM instead of a previously captured memory dump file. CMAT-V modifies CMAT so that it can access live VM memory by calling the XenAccess function \( xa\_access\_pa() \). The remainder of this section describes this function and how it is used to modify CMAT.

Given a physical address, \( xa\_access\_pa() \) memory maps the page of memory from the DomU VM that contains that physical address. The function returns the address to the mapped page and a page offset to the desired physical address. This memory mapping process is shown in Figure 19.

Figure 19. XenAccess VM to Physical Memory Mapping.
Parameters to `xa_access_pa()` include the desired physical address and the pre-initialized `xa_instance` data structure. In particular, `xa_instance` includes a value called `xc_handle` which is used to access Xen’s built in API. Using this handle, the `xa_access_pa()` function can subsequently call the Xen function `xc_map_foreign_range()` which executes the memory mapping and returns the page base address and offset to the desired physical memory location.

For live VMI, CMAT is modified to use data returned by `xa_access_pa()` instead of CMAT file pointers that read from a memory dump file [Oko10b]. Most of the analysis techniques used by CMAT are built upon the two functions `fimove()` and `figetc()`. The first function, `fimove()`, was originally designed to move to a particular location in the memory dump file. The FILE object that identifies the stream to the opened memory dump file and the desired physical address are passed as parameters to the `fimove()` function. Then the file stream pointer is moved to the requested location in the file. Once the file stream pointer is in the desired location, the `figetc()` function is called to grab one byte from the location of the file pointer. Once `figetc()` is called, it returns an 8 byte unsigned integer and increments the file stream pointer point to the next byte in the memory dump file. This function only grabs one byte at a time; therefore if you wanted five bytes for example, you would call `figetc()` five times. CMAT-V enhances CMAT by modifying these two functions to support VM introspection.

Similar to a file stream pointer, CMAT-V uses a variable called `memory_index` to keep track of the currently targeted location in physical memory. When calling the
fimove() function, the memory_index is updated to the desired physical address location. After updating the memory_index address variable, figetc() can be called in conjunction with xa_access_pa() to access the VM’s dynamic memory. In order to do this efficiently however, figetc() is modified to keep track of the last mapped page by the xa_access_pa(). When data is requested from a given physical address figetc() first checks to see if the address is located within the page that was last mapped by the function. If the address is within the current page, then only the offset into the page needs to be changed and no mapping is necessary. The principle of locality dictates there is a high chance that the next memory request will be located close to the previously accessed memory. Using this principle, an algorithm was developed to significantly reduce the paging overhead. Psuedo code for this algorithm is shown below.

```plaintext
if requested physical address is lower than memory_index then
    if falls below of range of last accessed page then
        map new page
    else
        update offset by subtracting appropriate value
else
    if falls above of range of last accessed page then
        map new page
    else
        update offset by adding appropriate value
```

If the requested physical address is located outside the range of the last mapped page, then a new page is mapped. Once these changes are made, the remaining CMAT functions are now fully compatible using live virtual machine memory instead of a memory dump file.
3.2.3. Registry Search Optimization

This research also modified CMAT’s registry search function `get_key()`. This function returns the location of the requested registry key value. When CMAT searches the registry to extract user information, as described previously, it uses a linear search to find the key value of interest. This linear search achieves O(n) performance. During this research, it was observed that Windows arranges the registry keys in alphabetical order. To leverage this convention, CMAT was modified to use a binary search. Using the function `strcmp(current_value,target_value)` the algorithm was able to determine whether the alphabetical value of the string variable `target_value`, containing the key of interest, was greater than, equal to or less than `current_value`, the key value currently pointed to in the search. If `strcmp` returns a negative value, then `target_value` is further down alphabetically; if a positive value then `target_value` is locate earlier alphabetically in the list; if zero then the key has been found. Using this capability, a binary search was implemented. If the value is not found (negative or positive return value), then the binary search moves `current_value` to the middle (alphabetically) of the remaining entries to be searched. This is repeated until the `target_value` is found. CMAT-V’s modified version of CMAT with a binary search achieves O(log n) performance which performs better than the previous O(n) performance.
3.3. **Experimental Design**

The objectives of the following experiments are divided among three tests. The first test, described in Section 3.3.1, evaluates CMAT-V’s ability to provide VM state information to detect different malware attacks running in a VM. The second test, described in Section 3.3.2, measures any detectable memory latency within the VM itself caused by CMAT-V’s live analysis. This test also demonstrates CMAT-V’s ability to scan two simultaneously executing VMs. The third test, described in Section 3.3.3, measures CMAT-V’s impact on host performance within the Xen’s Dom0 Hypervisor Management API.

3.3.1. **Threat Awareness Testing**

The objective of this test is to evaluate the effectiveness of CMAT-V for providing evidence of malware attacks. With this evidence, network operators can make classifications about the state of the system and determine the defensive actions required.

![CMAT-V Program Diagram](image)

**Figure 20. System Under Test: Malware Attack.**
The hypothesis for these tests is that CMAT-V can effectively provide evidence of malware attacks. An overview of the system under test for is shown in Figure 20.

The tested CMAT-V subroutines include functions to load debug data, get active user information, load registry and process data, link process data with users and load tcpip data. The left side of Figure 20 shows that the workload to CMAT-V includes selected malware attack scenarios. The malware attacks for this experiment focus on the rootkit class of malware because they are known to hide information from the user. A rootkit is “a set of programs and code that allows a permanent or consistent, undetectable presence on a computer” [Hog06]. Many rootkits are designed to gain access to administrator level privileges without being detected [Sko03]. Once system access is attained, payloads can be delivered to the target system. The right side of Figure 20 shows that the metric for this test is evidence of the malware presence. The nature of the evidence needed for successful detection is dependent on the type of malware being executed on the system. These malware attack scenarios are later described in detail Sections 3.3.1.1 through 3.3.1.4.

![Figure 21. VM System Configuration: Malware Attack.](image-url)
Unless otherwise specified, only one VM is used to represent the system under attack. The VM system configuration for this test is shown in Figure 21. CMAT-V is used to conduct live analysis on the target virtual machine. The target virtual machine is then subjected to the malware attack and the changes in CMAT-V results are observed. During the malware attacks, CMAT-V is run in its default -virt_live live analysis mode, which provides the user with interactive menus to further analyze the results.

For this series of tests, the performance of CMAT-V is evaluated for four different malware scenarios. The names of the malware used for each scenario are FU, Hacker Defender, Vanquish and HideProcessHookMDI [Var10b]. These rootkits are selected because they are open source and are well documented. Though malware in the “wild” is often unpredictable and its behavior unknown, well-known malware is chosen so that their existence can be clearly identified. Descriptions and implementations of each attack are outlined in Sections 3.3.1.1 through 3.3.1.4. The last entry in the description lists the evidence-of-interest to detect the associated attack. Whether or not these elements of the attack are identified by CMAT-V serve as the metric for these tests.

3.3.1.1. Attack Scenario 1: FU Rootkit [Var10b]

**Malware Description:** This is a direct kernel object manipulation (DKOM) rootkit. DKOM aims to gain kernel level privilege (Ring0) access, then leverage and modify known OS specific architectures. **FU.exe** gains access to kernel level privilege by loading a false driver called **msdirectx.sys**. Instructions given via the command line to **FU.exe** are passed to the driver. With the driver’s escalated privilege level, it is able
to print process information, hide processes, list driver information, hide drivers, set user security identifier (SID), list available privileges and set privilege levels [Var10b] [CAT10].

**Additional Software Used:**

- Process Explorer
- DriverView
- InstDrv

These programs are freely available. See Appendix D for details on obtaining the software.

**Attack Scenario Procedure:** With this attack scenario, CMAT-V is used to conduct a scan of the uncompromised VM. Then the FU rootkit is installed and used to hide a process. Once the process has been confirmed as hidden from the VM, CMAT-V is executed again and the results are analyzed in attempt to find the hidden process. To accomplish this, the following procedures are followed:

1) Open Process Explorer, DriverView and Windows Event Viewer.

2) Execute a single CMAT-V scan.

3) Open InstDrv and install the driver msdirectx.sys.

4) Open up a command prompt and launch regedit.exe

5) Launch Windows calc.exe.

6) Launch a command shell.

7) Use FU to hide calc.exe by process identifier (PID). >fu.exe -ph <#PID>

8) Execute another CMAT-V scan and compare results to first scan.
**CMAT-V Evidence-of-interest:**

- Evidence of the existence of hidden process. Success if CMAT-V is able to detect the hidden process; failure if hidden process is not found.

### 3.3.1.2. Attack Scenario 2: Hacker Defender [Var10b]

**Description:** This is a backdoor program that is designed to exploit vulnerabilities in a program to gain access to a system. Hacker Defender rewrites a few memory segments in all running processes while maintaining the stability of the system. Once this is accomplished it can hide files, processes, system services, system drivers, registry keys and open ports. The program also installs a hidden backdoor allowing remote exploitation [Var10b][CAT10].

**Additional Software Used:**

- Process Explorer
- DriverView
- NetCat
- RegScanner

These programs are freely available. See Appendix D for details on obtaining the software.

**Attack Scenario Procedure:** The actions of Hacker Defender are controlled by an initialization file (ini-file). The contents of the file used for this scenario is shown below.

<table>
<thead>
<tr>
<th>[Hidden Table]</th>
<th>[Hidden Processes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>hxdef*</td>
<td>hxdef*</td>
</tr>
<tr>
<td>calc.exe</td>
<td>calc.exe</td>
</tr>
</tbody>
</table>
[Root Processes]  [Hidden RegValues]
hxdef*          [Hidden Ports]
calc.exe        TCPI: 100
                TCPO: 100

[Hidden Services]  [Hidden RegKeys]
HackerDefender*  HackerDefender100
[Hidden RegKeys]  LEGACY_HACKERDEFENDER100

[Hidden RegKeys]  HackerDefenderDrv100
                  LEGACY_HACKERDEFENDERDRV100

The [Hidden Table] section hides all files and directories which start with the strings listed below it. Similarly, the [Hidden Processes], [Hidden Services], [Hidden RegKeys], [Hidden Ports] sections hide processes, service and driver names, registry keys and ports respectively. [Startup Run] is a list of programs the rootkit will run after startup. The configuration listed hides all processes starting with “hxdef” and those named calc.exe. The service and driver names that start with “HackerDefender” will be hidden as well as any registry keys containing HackerDefender100, LEGACY_HACKERDEFENDER100, HackerDefenderDrv100, or LEGACY_HACKERDEFENDERDRV100. Finally, open connections on port 100 are also hidden by the malware. Under [Hidden Ports], TCPI is for all inbound TCP traffic; TCPO is for outbound traffic. The ini-file must be named hxdef100.ini to use hxdef100.exe in default mode. The VM configuration for this scenario uses two VMs as shown in Figure 22.
The target VM executes `calc.exe` and the Hacker Defender payload `hxdef100.exe`. The attacker VM executes NetCat and the Hacker Defender back door program (`bcdli100.exe`) to gain access to the target VM. The target VM must have all additional software installed as listed previously. With the ini-file created, the following procedures will be followed. Which VM is being referenced is included each step as needed.

1) (Target VM) Open Process Explorer.

2) (Target VM) Run the Windows `calc.exe` program.

3) (Target VM) In Process Explorer, verify the existence of `calc.exe`.

4) (HM-API) Execute a single CMAT-V scan.

5) (Target VM) Execute NetCat with

   ```
   >nc -l -p 100 -t -e cmd.exe
   ```

   This will open up port 100 to listen for a remote connection.
6) (Target VM) In the command line execute `hxdef100.exe`

This will load the ini-file `hxdef100.ini` and execute the pre-configured payload.

7) (Target VM) Verify that calc.exe has disappeared from the Process Explorer list.

This confirms that the Hacker Defender payload has been executed.

8) (Attacker VM) Execute the file `bdcli100.exe` and follow the on screen instructions with the IP address of the target VM, port 100 and the default password “hxdef-rulez” to the command line. This will open up a remote command shell.

9) (Target VM) Use Windows explorer to verify that any files with “hxdef” in the name have disappeared. This might take a few moments or involve refreshing the window view to observe their removal.

10) (Target VM) Use the Windows `netstat` command to verify that port 100 does not show up as an open connection.

11) (Target VM) Verify with RegScanner that the listed registry keys in the ini-file cannot be found.

12) (HM-API) Execute a second CMAT-V scan and compare results to first scan.

**CMAT-V Evidence-of-interest:**

- Evidence of the existence of hidden processes `hxdef100.exe`. Success if CMAT-V is able to detect the hidden process; failure if hidden process is not found.
• Evidence of existence of executable file location for hxdef100.exe. Success if CMAT-V is able to detect the executable; failure if hidden executable is not found.

• Evidence of hidden registry keys as described in the ini-file. Success if CMAT-V is able to detect any of the registry keys; failure if none are found.

• Evidence of hidden port 100. Success if CMAT-V is able to detect the open port; failure if open port is not detected.

3.3.1.3. Attack Scenario 3: Vanquish [Var10b]

**Description:** This is a DLL-Injection based rootkit designed to hide files, folders, registry entries and log passwords. Vanquish uses two files, a Vanquish Autoloader (vanquish.exe) and a Vanquish DLL (vanquish.dll). Upon execution of vanquish.exe, the DLL is injected into running applications. The DLL then executes the exploits to hide any files, folders or registry keys that contain the string “vanquish.” In addition, vanquish installs a key logger to record passwords from the user log on screen [Var10b][CAT10].

**Additional Software Used:**

• None

**Attack Scenario Procedure:** The installation of this rootkit simply involves starting the executable and verifying that the rootkit is active. This attack scenario only requires the use of one VM. On this VM, the following procedures are completed:

1) Execute a preliminary CMAT-V scan.
2) Execute *setup.cmd* and follow the on-screen instructions. These instructions will simply have you press any key to install the rootkit and then click **Yes** when the Windows registry editor asks if you want to add to the registry. Once completed, all files that have the string “vanquish” in them will disappear. This confirms that the rootkit is working.

3) Execute a secondary CMAT-V scan and compare results to first scan.

**CMAT-V Evidence-of-interest:**

- Evidence of the existence of DLL injection. Success if CMAT-V is able to detect the DLL in any running process; failure if the DLL is not found.
- Evidence of hidden registry keys. Success if CMAT-V is able to detect any of the registry keys; failure if none are found.

### 3.3.1.4. Attack Scenario 4: HideProcessHookMDI [Var10b]

**Description:** This is a basic hook of the system service dispatch table (SSDT). As the name suggests, the SSDT is a table used to locate the call address of a particular system function in memory. The hide process rootkit hooks this table by replacing the *ZwQuerySystemInformation* function in the SSDT. *ZwQuerySystemInformation* is used by programs such as *Taskmgr.exe* to get a list of processes executing on the system. A new function called *NewZwQuerySystemInformation* replaces the original and filters out selected processes and adds the running times of the process to the Idle process [Hog06].
**Additional Software Used:**

- Process Explorer
- InstDrv
- Windows Driver Kit (WDK)

These programs are freely available. See Appendix D for details on obtaining the software.

**Attack Scenario Procedure:** This attack scenario only requires the use of one VM. This particular test involves building and installing a driver on the target VM system. First, modify the file `basic_mdl_flags.c` to indicate the process to hide. Open the source file in a C program in a code editor. At approximately line 142 you will see code that looks like the following:

```c
if(0 == memcmp(curr->ProcessName.Buffer, L"_root_", 12))
```

By default, the rootkit is programmed to hide processes starting with "_root_" in the name. For this test, the program calc.exe is the target process to hide. Therefore change "_root_" too "calc.exe" as shown:

```c
if(0 == memcmp(curr->ProcessName.Buffer, L"calc.exe", 12))
```

After making these changes, save and close the file. Now the driver must be built. To do this, open a command shell for the WDK checked-build environment. Change to the `HideProcessHookMDI` directory where the `MAKEFILE` and `SOURCES` files are contained. The `HideProcessHookMDI` must be in a directory that does not contain any spaces. Then type the command "build" in the command line. This will create the file
hideprocess.sys in the sys\i386 folder, which is the rootkit driver to be installed. Once the driver has been built, the following procedures must be completed:

1) Run the Windows calc.exe program.
2) Open Process Explorer and verify the presence of calc.exe.
3) Execute a preliminary CMAT-V scan.
4) Open InstDriver.
5) Type the location of the driver hideprocess.sys.
6) Click install, then click start to start the driver.
7) Verify in Process Explorer that calc.exe has disappeared. This indicates that the rootkit is has successfully been installed.
8) Execute a secondary CMAT-V scan.

CMAT-V Evidence-of-interest:

- Evidence of hidden process calc.exe. Success if CMAT-V is able to detect the hidden process; failure if hidden process is not found.

3.3.2. **CMAT-V Detection and Multi-VM Testing**

The primary objective of this test is to measure the performance impact of the DomU guest machine caused by running CMAT-V within the Dom0 HM-API. In other words, this experiment tests the assumption that CMAT-V’s live analysis does not interfere by causing unique memory latency within VM itself. If it does, the presence of CMAT-V could be detectable by the VM and any malware executing within it. In
conjunction with interference testing however, this test is also designed to demonstrate CMAT-V’s ability to scan two different VMs supported by the same hypervisor.

Figure 23. System Under Test: VM Memory Latency and Multi-VM Analysis.

Figure 24. System Configuration: VM Memory Latency and Multi-VM Analysis.
Two simultaneously active VMs are used for this test. An overview of the system under test is shown in Figure 23 and the VM system configuration is shown in Figure 24. As shown in Figure 23, the system under test for this experiment is a VM called the benchmark virtual machine. As the name suggests, the benchmark VM runs the benchmarks during the tests. The subsystems that make up this system include all Xen-allocated resources such as CPU, memory and hard disk as well as load balancing algorithms used by Xen to manage the use of the VM’s shared resources. As shown in the top of Figure 23, the only system parameter that is varied is the memory analysis mode. The memory analysis modes represent different configurations of CMAT-V live analysis. Figure 24 reveals that these modes are called baseline, direct and indirect. These modes are described in more detail in Section 3.3.2.1. As shown the left side of Figure 23, the workload profiles to the target virtual machine include CMAT-V as well as two benchmarks programs. The benchmarks used are FLOATmem and INTmem by RAMspeed [Hol02]. These benchmarks were chosen because they are open source and are advertised to be highly sensitive to memory latencies [Hol02]. One benchmark is used for each test run. These benchmarks are described in more detail in Section 3.3.2.2. Finally, the right side of Figure 23 shows that the metrics evaluated are CMAT-V’s ability to support multi-VM analysis and the data transfer rates reported by the RAMspeed benchmarks.

The factors under test, shown in Table 4 below, include the RAMspeed benchmarks and the memory analysis modes.
Table 4. Factors Under Test for VM Performance Experiment.

<table>
<thead>
<tr>
<th>RAMspeed Benchmark</th>
<th>Memory Analysis Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTmem</td>
<td>Baseline</td>
</tr>
<tr>
<td>FLOATmem</td>
<td>Indirect</td>
</tr>
<tr>
<td></td>
<td>Direct</td>
</tr>
</tbody>
</table>

Both VMs are installed with Windows XP service pack 3 with all updates installed as of 6/10/2010. The benchmark VM and a secondary VM are the only VMs running within the Xen hypervisor in addition to the Dom0 management OS. The VMs are allocated with 512 MB RAM and 10 GB hard disk.

3.3.2.1. Memory Analysis Modes

Recall from Figure 24 that three memory analysis modes used for this test: baseline, indirect and direct. Baseline mode does not execute any memory analysis at all, and therefore CMAT-V is not running in the Dom0 HM-API. As the name suggests, this mode is used for baseline measurements. Indirect mode uses CMAT-V to conduct static memory analysis on the secondary VM. This VM is used to simulate the workload of CMAT-V running in Dom0. This secondary VM does not execute any benchmarks. From the perspective of the benchmark VM, it experiences the effects of Xen load balancing from the CMAT-V workload, yet the memory of the benchmark VM itself is not being probed. Direct mode uses CMAT-V to access live VM memory on the benchmark VM throughout the execution of the benchmark. These modes require live CMAT-V analysis of two simultaneously running VMs. As shown in the right side of Figure 23, the success of implementing all memory analysis modes is
evaluated as a metric. This multi-VM capability is implicitly demonstrated by accomplishing the procedures for the VM detection testing that follow.

In direct mode, benchmarks are executed in the benchmark VM (DomU) while CMAT-V is simultaneously executed in the HM-API (Dom0). Because CMAT-V is using the Xen hypervisor to access the benchmark VM’s dynamic memory, it is possible that the benchmark VM itself could encounter memory latencies as VM applications and CMAT-V negotiate with Xen to access the same memory locations. To evaluate the possible impact of CMAT-V to the benchmark VM’s memory access performance, the benchmarks chosen specifically exercise the system’s memory read/write functions. Any decrease in benchmark performance observed while in CMAT-V direct mode would suggest that CMAT-V is interfering with the VM. It is hypothesized that due to Xen’s abstraction of the guest resources from the host and associated API, this interference will not be observed.

Memory latency is not the only factor that might decrease benchmark performance within the benchmark VM while CMAT-V is running. A lower benchmark data transfer rate due to the Xen hypervisor sharing hardware resources, such as the CPU, between DomU and Dom0 processes could be observed as well. This particular slowdown would not be specifically caused by CMAT-V, but would occur for any program that is competing for execution time on the same CPU. Since this test is designed to specifically target memory latency, measures must be taken to isolate system sensitivity to that caused strictly by CMAT-V’s live VM memory access. For this reason, the indirect mode becomes necessary.
Recall that indirect mode creates a similar CMAT-V workload to the Xen hypervisor as experienced in direct mode, but the benchmark VM is not actively being probed by CMAT-V. This means decreased benchmark performance within the benchmark VM is strictly caused by Xen’s resource sharing features and not by memory latencies caused by CMAT-V. The benchmark performance from CMAT-V indirect and direct modes is then compared. Using these tests, any decreased benchmark performance within the benchmark VM specifically caused by CMAT-V accessing its dynamic memory can be further isolated.

In addition, it is important to correctly configure Xen’s CPU management. Xen allocates each VM a certain number of virtual CPUs (VCPUs). For multi-processor systems it is possible to restrict certain VMs to only use certain processors. The default VCPU configuration used for this test is shown below.

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
<th>VCPUs</th>
<th>CPU State</th>
<th>Time(s)</th>
<th>CPU Affinity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain-0</td>
<td>0</td>
<td>0</td>
<td>r--</td>
<td>1836.6</td>
<td>0</td>
</tr>
<tr>
<td>Domain-0</td>
<td>0</td>
<td>1</td>
<td>r--</td>
<td>985.3</td>
<td>1</td>
</tr>
<tr>
<td>WINXP_BenchVM</td>
<td>1</td>
<td>0</td>
<td>---</td>
<td>7.8</td>
<td>any cpu</td>
</tr>
<tr>
<td>WINXP_SecondVM</td>
<td>2</td>
<td>0</td>
<td>-b-</td>
<td>9.8</td>
<td>any cpu</td>
</tr>
</tbody>
</table>

The host (Domain-0) is allocated two VCPUs and is assigned two processors (0 and 1). WINXP_BenchVM and WINXP_SecondVM are configured to allow Xen to allocate any available processor. This configuration is preferable because it is less affected by Xen’s load balancing which restricts the performance of VM guests [Mat08]. This restriction is necessary to prevent misbehaving guest VMs from consuming too many resources. For this series of tests however, it is important that the VM under test is not significantly affected by load balancing. For this reason, the VMs are configured to use any processor
available. In addition, this default mode is more robust for determining CMAT-V
detection because it does not assume a specific CPU/VCPU assignment.

3.3.2.2. RAMspeed Benchmarks

This test uses the RAMspeed benchmark suite [Hol02]. RAMspeed is a
command line benchmark that measures the effective bandwidth of both dynamic cache
and memory. In particular, the INTmem and FLOATmem benchmarks are chosen as the
workloads for this experiment. Both integer and floating point benchmarks are used to
represent real life workloads that conduct both integer and float based calculations.

**INTmem Benchmark** - This benchmark consists of the following four subtests: copy,
scale, add and triad. Copy simply transfers data from one memory location to another (A
= B). Scale modifies the value before transferring the data by multiplying the value by a
constant (A = m*B). Add reads two memory locations adds them together and stores the
result into a third location (A = B + C). Triad merges all three instructions together by
reading from two memory locations, multiplying one by a constant, adding them together
and then storing the result (A = m*B +C). The last result provided by the benchmark is
an average of the performance of all four subtests. INTmem uses double (32 bit) words
and can be configured to transfer a variable amount of data per pass. The result of this
benchmark is the data transfer rate in megabytes per second (MB/s).

**FLOATmem Benchmark** - Similar to INTmem, FLOATmem executes the copy, scale,
add and triad subtests. The last value averages the performance from all four subtests
together. Unlike INTmem however, FLOATmem uses quad (128 bit) words. The result
of this benchmark is the data transfer rate in megabytes per second (MB/s).
Each benchmark configuration is run 100 times for both direct and indirect memory analysis modes. For both benchmarks, 50 measurements are taken both before and after the active CMAT-V measurements, to ensure baseline consistency throughout the test. For example, when running either benchmark the configuration sequence is shown in Table 5.

Table 5. Configuration Sequence for VM Performance Experiment.

<table>
<thead>
<tr>
<th>Number of Runs</th>
<th>Memory Analysis Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>Baseline</td>
</tr>
<tr>
<td>100</td>
<td>Indirect</td>
</tr>
<tr>
<td>50</td>
<td>Baseline</td>
</tr>
<tr>
<td>100</td>
<td>Direct</td>
</tr>
<tr>
<td>50</td>
<td>Baseline</td>
</tr>
</tbody>
</table>

This results in a total of 350 runs for each benchmark. There is nothing inherently special about choosing 100 runs for the analysis modes and 50 for each baseline. These numbers were selected as starting points because they were hypothesized to be enough runs for the data collected to stabilize at a certain range or for any trends to be noticed. In addition, it is advantageous for the number of baseline measurements to meet or exceed that of the memory analysis modes because the baseline is used as a standard to compare with other results. Also, by conducting baseline measurements before and after the configuration under test, any changes in overall system performance throughout the experiment can be observed. Ultimately, however, the data dictates if more precision (and therefore more runs) is necessary. For example, if the results of the baseline and direct/indirect modes are very close together, more runs might be required to increase the precision of the test. For this experiment, these run numbers mentioned are used as an
initial starting point for analysis and discretion is used once the data is collected to determine whether more runs are necessary.

The default data size for each pass of the benchmarks of 8 gigabytes is used for each run. RAMspeed is run using the provided batch mode, which allows repeated runs to be executed. An example command line for the RAMspeed benchmark is shown below.

```
<current_directory>ramspeed-win32 -b 3 -l 100 > data.log
```

The ‘-b’ indicates the benchmark ID. In the above example, ID 3 is selected which is the INTmem benchmark. The ‘-l’ indicates that the benchmark will be run in batch mode and the example shows this benchmark will run 100 times. The output from the benchmark is then stored in the file `data.log`.

When running the benchmarks, CMAT-V is run continuously by using a bash script that repeatedly executes CMAT-V. An example bash script is shown below.

```
for(( ; ; ))
do
    /xenaccess/examples/cmat -virt_live_bench 1
done
```

The `-virt_live_bench` mode configures CMAT-V to immediately exit after one iteration through the program. Using this bash script, CMAT-V is then immediately restarted.
One-way analysis of variance (ANOVA) tests that use student’s t-test are used to determine whether there are significant differences in performance between the baselines, between the baselines and active CMAT-V modes and between the CMAT-V direct and indirect modes. Real-world user workloads on a given system often vary over time, therefore when determining the baseline variation only the average RAMspeed subtest is considered. The averaged data combines the performance of the other four subtests. The variation baseline measurements based on this subtest provides a generalization of overall system state variation. ANOVA tests are conducted using a 95% confidence interval assuming normal distributions.

### 3.3.3. Host System Performance Testing

The objective of these experiments is to measure the host system overhead required to run CMAT-V within the Xen’s HM-API. This test is different from the previous test because benchmarks are run in the HM-API rather than a particular VM. Because the HM-API has complete access to the hypervisor, the results from these benchmarks characterize the performance of all host hypervisor resources, not just those allocated to a particular VM. The hypothesis for this series of experiments is that executing CMAT-V will have a low to moderate impact (>30% decrease) on host system performance using the selected laptop and hardware configuration. These tests use the Phoronix Test Suite (PTS) [Pho10] to conduct several Linux-based benchmarks. Phoronix was chosen because it is freely available and contains a wide variety of benchmark workloads. Installation procedures for the PTS can be found in Appendix C.

An overview of the system under test is shown in Figure 25
The Dom0 HM-API has privilege to use and control all system resources such as CPU, memory and hard disk. This Dom0 OS must also use Xen’s load balancing algorithms to share its resources with VMs running on the hypervisor. Similar to previous tests, CMAT-V is a workload to the system under test. In addition to the CMAT-V workload, benchmarks used in this series of tests target memory read/write and CPU performance. The PTS includes Linux-based versions of the RAMspeed benchmarks INTmem and FLOATmem described in Section 3.3.2. As shown in Figure 25, these benchmarks are used as workloads to the system for testing memory read/write performance. In addition to the RAMspeed benchmarks, the FFmpeg [Var10a] benchmark is used as a workload to the system to specifically test CPU performance. The metrics for the RAMspeed benchmarks are the same as previous tests. Each RAMspeed benchmark reports the data transfer rate for the copy, scale and triad subtests as described previously. For the FFmpeg benchmark however, encoding time is the metric observed. The details of this benchmark are described next.
**FFmpeg** - This benchmark uses FFmpeg to test the system’s audio/video encoding performance. FFmpeg converts an audio video interleave (AVI) video file to a National Television System Committee (NTSC) video compact disc (VCD) file. The results for this benchmark are reported as an average of no less than three runs and are reported seconds to complete the file encoding. The results of this benchmark are in seconds.

To install this benchmark in the PTS, the following command must be executed:

```
> phoronix-test-suite install ffmpeg
```

All benchmarks are run within one active Windows XP SP3 VM. This VM is the only VM running within the Xen hypervisor in addition to the HM-API. The VM is allocated with 512 MB RAM and 10 GB hard disk. If one VM is instantiated, this leaves the Dom0 guest 1.45 GB RAM. The Dom0 HM-API is configured with two VCPUs, one assigned to each processor. The VM is configured to use only one processor but allows Xen to select any available processor. The factors under test are shown in Table 6 below and the configuration sequence is shown in Table 7.

Table 6. Factors Under Test for System Performance Test.

<table>
<thead>
<tr>
<th>RAMspeed Benchmark</th>
<th>Memory Analysis Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTmem</td>
<td>Baseline</td>
</tr>
<tr>
<td>FLOATmem</td>
<td>Direct</td>
</tr>
<tr>
<td>Ffmpeg</td>
<td></td>
</tr>
</tbody>
</table>

Table 7. Configuration Sequence for System Performance Test.

<table>
<thead>
<tr>
<th>Number of Runs</th>
<th>Memory Analysis Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>Baseline</td>
</tr>
<tr>
<td>100</td>
<td>Direct</td>
</tr>
<tr>
<td>50</td>
<td>Baseline</td>
</tr>
</tbody>
</table>

76
Fifty baseline benchmark measurements run without CMAT-V in execution are run before and after each benchmark test as shown in Table 7. This results in a total of 200 runs for each benchmark. Using the same reasoning mentioned in Section 3.3.2, 100 runs for direct mode and 50 for each baseline mode are used as a starting point and discretion is used to determine if more runs are necessary. The FFmpeg benchmark uses the Phoronix batch mode so that the any user prompts are automated. The batchmode setup is run by executing `phoronix-test-suite batch-setup` indicating not to save test results after each run and to run all test options. After this has been setup, a bash script is run to automate execution. Below is a bash example that runs FFmpeg benchmark 100 times.

```bash
for((1;1;100))
do
   /working_directory/phoronix-test-suite/phoronix-test-suite batch-run ffmpeg
done
```

The output from running the bash script is written to a log file. By conducting baseline measurements before and after the configuration under test, any changes in HM-API system performance throughout the experiment can be observed.

One-way ANOVA tests are used to determine whether there is a significant difference in benchmark performance between the baselines as well as between the baselines and with CMAT-V in execution. Similar to the test in Section 3.3.2, real-world user workloads on a given system often vary over time, therefore when determining the baseline variation only the average RAMspeed subtest is considered. The averaged data
combines the performance of the other four subtests. The variation baseline measurements based on this subtest provides a generalization of overall system state variation. ANOVA tests are conducted using a 95% confidence interval assuming normal distributions.
IV. Results

The following sections detail the results of all experiments. Section 4.1 describes results from the threat awareness tests. Section 4.2 describes the results from the VM performance testing. Section 4.3 describes the results from the host system performance testing. Section 4.4 summarizes and discusses any general conclusions made from the results.

4.1. Threat Awareness

A summary of the results from the threat awareness testing are shown in Table 8.

Table 8. Threat Awareness Results.

<table>
<thead>
<tr>
<th>Malware Attack</th>
<th>Evidence of Interest</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU</td>
<td>Hidden Processes</td>
<td>Success</td>
</tr>
<tr>
<td>Hacker Defender</td>
<td>Hidden Processes</td>
<td>Success</td>
</tr>
<tr>
<td></td>
<td>Hidden Files</td>
<td>Success</td>
</tr>
<tr>
<td></td>
<td>Hidden Registry Keys</td>
<td>Inconclusive</td>
</tr>
<tr>
<td></td>
<td>Hidden Port</td>
<td>Success</td>
</tr>
<tr>
<td>Vanquish</td>
<td>DLL injection</td>
<td>Success</td>
</tr>
<tr>
<td></td>
<td>Hidden Registry Keys</td>
<td>Inconclusive</td>
</tr>
<tr>
<td>HideProcess</td>
<td>Hidden Processes</td>
<td>Success</td>
</tr>
</tbody>
</table>

For each test a successful result indicates that CMAT-V provided evidence of the malware attack. For those results that were inconclusive, CMAT-V did not confirm nor
deny the presence of the malware being tested. These results are described further in Sections 4.1.1 through 4.1.4 and final conclusions are discussed in Section 4.1.5.

4.1.1. FU Results

The evidence-of-interest for this attack scenario was to determine if CMAT-V could detect the process `calc.exe` after it has been hidden from the target VM. CMAT-V was successful in detecting the hidden process. A screenshot of the process detection is shown in Figure 26.

![Figure 26. CMAT-V Hidden Process Detection for FU Rootkit.](image)

The right side of Figure 26, shows the active processes reported from the target VM via Process Explorer. The left side of Figure 26 shows the active processes reported by CMAT-V. The process `calc.exe` with the process identifier (PID) of 884 is absent from the Process Explorer application. This process however is captured by CMAT-V, as shown by the red box. CMAT-V also reports a “No” result under the InProcList
column for `calc.exe`. This means that CMAT-V has identified this process as missing from the Windows executive process (EPROCESS) list.

### 4.1.2. Hacker Defender Results

The first evidence-of-interest for this attack scenario was the detection of the hidden executable `hxdef100.exe`. CMAT-V was successful in detecting the hidden process. A screenshot of the process detection is shown in Figure 27.

![Figure 27. CMAT-V Hidden Process Detection for Hacker Defender.](image)

Like in Figure 26, the right side of Figure 27 shows the active process list reported by the VM (via Windows Task Manager), while the left side shows the same reported by CMAT-V. The PID of the `hxdef100.exe` process is 1640. The screenshot shows no evidence of `hxdef100.exe` in the task manager of the target VM; however, CMAT-V was able to detect this process (as shown in the red box).
The second evidence-of-interest was CMAT-V’s ability to detect hidden files on the target VM. Using the CMAT-V interface, the process `hxdef100.exe` was selected and process environment information was requested. This description includes information about the location of the executable, command line arguments used to execute, the title of any windows open and the DLL path(s) used by the executable. The process environment information for `hxdef100.exe` is shown in the bottom half of Figure 28.

![Figure 28. CMAT-V Hidden File Detection for Hacker Defender.](image)

The particular information of interest for this test is the location of the executable. The red box in Figure 28 shows that the detected executable originated from the path `C:\Documents and Settings\ddodge\My Documents\ROOTKITS\hxdef100r\`. 
According to the VM file system however (as shown at the top of Figure 28), this folder does not exist. This successfully demonstrates how CMAT-V can be used to detect files and folders hidden from the VM itself.

The third evidence-of-interest for this scenario are hidden registry keys used by the rootkit. The only registry information shown by CMAT-V is `REGISTRY/MACHINE/` which only indicates the root-level registry key. Due to the large number of registry keys and values, CMAT-V is not currently designed to search for and display all registry keys for each process. In light of this, the presence of the malware cannot be confirmed or denied by observing the registry information provided by CMAT-V.

![Image of CMAT-V Hidden Port Detection for Hacker Defender](image)

**Figure 29.** CMAT-V Hidden Port Detection for Hacker Defender.

The final desired piece of evidence is the presence of the open port 100. A screenshot of the results is shown in Figure 29. As shown at the top half of Figure 29 the VM `netstat` command reports no open connections; however, CMAT-V accurately detects
a connection at port 100 for the NetCat process `nc.exe` (shown in the red box). Though the port connection is detected, the local and remote internet protocol (IP) address information is not presented. For user datagram protocol (UDP) connections both the local and remote addresses are detectable by CMAT-V. For transmission control protocol (TCP) connections however, CMAT-V is currently unable to retrieve IP address information and is described as a focus for future development in Chapter 5. Currently, for both TCP and UDP, CMAT-V provides evidence of an open port hidden from the VM guest. In doing so, CMAT-V provides user information to guide further investigation on the guest machine.

### 4.1.3. Vanquish Results

The attack scenario that includes the Vanquish rootkit first tests CMAT-V’s ability to provide evidence of DLL injection and hidden registry keys.

![Figure 30. CMAT-V DLL Injection Detection for Vanquish.](image)
For this test, the user-owned process `explorer.exe` is selected. For each process, CMAT-V provides a list of all currently used DLLs. The Vanquish rootkit injects its DLL into every user-owned process. As shown in the red box of Figure 30, `VANQUISH.DLL` is detected by CMAT-V. This demonstrates CMAT-V’s ability to provide uncompromised evidence of DLL injection on the guest system.

The second test investigated CMAT-V’s ability to reveal hidden registry keys. Similar to the test in Section 4.1.2, CMAT-V was only able to provide root-level descriptions of the registries used by a process. Consequently, the presence of the Vanquish rootkit can be neither confirmed nor denied with this information.

### 4.1.4. HideProcess Results

The final attack scenario uses the HideProcess rootkit which tests CMAT-V’s ability to detect a process when hidden using a modified SSDT.

![Figure 31. CMAT-V Hidden Process Detection for HideProcess.](image-url)
The results of this test are shown in Figure 31. The rootkit is used to hide the process `calc.exe`. As shown in on the right in Figure 31, the task manager by the guest VM does not report the existence of the process `calc.exe`. The left side of Figure 31, however, shows that CMAT-V is able to detect this hidden process with a PID of 3816. For this case, CMAT-V reports a “Yes” in the `InProcList` column which signifies that `calc.exe` is still contained within the VM’s EPROCESS list. This is consistent with the behavior of HideProcess, as it does not remove targeted processes from the list but rather filters them out when the list is queried.

### 4.1.5. Conclusion

For all of the attack scenarios conducted, CMAT-V was able to provide uncompromised system state information to allow successful malware detection. Of each evidence-of-interest type, most were demonstrated as successfully detected by CMAT-V. The registry key detection was the only inconclusive evidence provided by CMAT-V and future work to improve this capability is discussed in Chapter 5. It is important to note that CMAT-V is currently designed for VM state view-generation only, providing only evidence of the system state. CMAT-V does not make claims about the classification of the VM state itself. In order for a classification to be made about whether the VM guest is safe or compromised, VM guest requires user interpretation of the results provided.
4.2. CMAT-V Detection and Multi-VM Testing

This section contains the results from the VM performance tests to measure the performance impact to the DomU guest machine caused by running CMAT-V within the Dom0 HM-API. In addition, CMAT-V’s multi-VM analysis is demonstrated. Sections 4.2.1 and 4.2.2 describe the results from the INTmem and FLOATmem benchmarks respectively. Section 4.2.3 contains concluding remarks based on the benchmark results and multi-VM analysis.

4.2.1. INTmem Benchmark

The first step in analyzing the performance of the INTmem benchmark is to ensure that the data collected reflects normal system behavior. Any uncharacteristic behavior must be identified and handled accordingly. The scatter plot in Figure 32 shows the behavior of the INTmem benchmark for the sequential 350 runs.

Figure 32. Scatter Plot of VM INTmem Average Performance.
The data shown is the average of the four tests copy, add, scale and triad for each run. These results show that the first several runs of each set encounter distinctively high data transfer rates. After this initial period, the performance appears to stabilize at a lower range for the remaining runs. This initial spike in performance could likely be caused by the time delay for the Xen load balancing routines to evaluate and balance the workload. These initial runs appear abnormal compared to subsequent runs. The data collected while the performance is stabilized best represents normal program performance. As a result, the first 10 runs from each set of runs are not considered.

Recall from Section 3.3.2 that baseline mode does not execute any memory analysis at all, and therefore CMAT-V is not running in the Dom0 HM-API. In direct mode, benchmarks are executed in the benchmark VM while CMAT-V is simultaneously probes the VM from the HM-API. Indirect mode creates a similar CMAT-V workload to the Xen hypervisor as experienced in direct mode, but the benchmark VM is not actively being probed by CMAT-V. This means decreased benchmark performance within the benchmark VM is strictly caused by Xen’s resource sharing features and not by memory latencies caused by CMAT-V.

With these memory analysis modes in mind, next step is to compare the baseline performance sets to determine if any factors affect the system state over time. A baseline is run before the indirect mode testing, between tests and after the direct mode testing. If any significant increases or decreases in baseline performance over time exist compared to the modes under test, the effects of the variations must be taken into account. ANOVA
statistics and box plot of the three baselines are shown in Table 9 and Figure 33 respectively.

Table 9. ANOVA Results for VM INTmem Average Baseline Performance.

<table>
<thead>
<tr>
<th>Baseline Test</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>2684.48</td>
<td>2676.59</td>
<td>2681.9</td>
</tr>
<tr>
<td>StDev</td>
<td>7</td>
<td>8.02</td>
<td>9.17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tests Compared</th>
<th>1-2</th>
<th>1-3</th>
<th>2-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-Value [Diff = 0]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Est. Difference [First - Second]</td>
<td>7.89</td>
<td>2.58</td>
<td>-5.31</td>
</tr>
<tr>
<td>Percent Change</td>
<td>0.29%</td>
<td>0.10%</td>
<td>-0.20%</td>
</tr>
</tbody>
</table>

1 = Before Indirect Test
2 = Intermediate
3 = After Direct Test

Figure 33. Box Plot of VM INTmem Average Baseline Performance.

The ANOVA results in Table 9 show that the P-value over a 95% confidence interval is zero when each of the baselines is compared. The hypothesis being tested is that the difference between each sample sets is zero (the sample means are not statistically different). Considering the data collected, the P-value reveals that the probability of this being true is 0%. In other words, the results obtained support the
conclusion that the baselines measured are statistically different. As shown in the box plot in Figure 33, it is possible that over time there could be a slight linear decrease in performance over time, but this is ultimately inconclusive. A more precise characterization of the performance trend over time would require further testing which is out of the scope of this study. However, the data collected does provide valuable information about the magnitude of the change in system behavior over the course of the experiment. As shown in the second-to-last row in Table 9, the estimate in magnitude of the difference between each baseline ranges from 2.58-7.89 MB/s. Because these baseline measurements are combined for later analysis, once final results are obtained these variations in performance must be taken into consideration when drawing conclusions about the data.

Figure 34. Boxplot of CMAT-V Impact on VM INTmem Performance.
Table 10. ANOVA of CMAT-V Impact on VM INTmem Performance.

<table>
<thead>
<tr>
<th>INTmem Subtest</th>
<th>Memory Analysis Mode</th>
<th>Mean (MB/s)</th>
<th>StDev (MB/s)</th>
<th>Est Difference (MB/s) [Baseline - X]</th>
<th>P-Value</th>
<th>Decrease in Performance</th>
<th>P-Value Ind vs Dir Δ Performance [Diff = 0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>Baseline</td>
<td>2503.6</td>
<td>20.5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Indirect</td>
<td>2406.1</td>
<td>18</td>
<td>97.5</td>
<td>0</td>
<td>3.89%</td>
<td>0.247</td>
</tr>
<tr>
<td></td>
<td>Direct</td>
<td>2403.3</td>
<td>14.3</td>
<td>100.31</td>
<td>0</td>
<td>4.01%</td>
<td>-</td>
</tr>
<tr>
<td>Scale</td>
<td>Baseline</td>
<td>2477.3</td>
<td>15.9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Indirect</td>
<td>2381.8</td>
<td>17.4</td>
<td>95.5</td>
<td>0</td>
<td>3.86%</td>
<td>0.003</td>
</tr>
<tr>
<td></td>
<td>Direct</td>
<td>2373.9</td>
<td>17.9</td>
<td>103.41</td>
<td>0</td>
<td>4.17%</td>
<td>-</td>
</tr>
<tr>
<td>Add</td>
<td>Baseline</td>
<td>2879.1</td>
<td>11.6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Indirect</td>
<td>2765.7</td>
<td>20.3</td>
<td>113.41</td>
<td>0</td>
<td>3.94%</td>
<td>0.205</td>
</tr>
<tr>
<td></td>
<td>Direct</td>
<td>2762</td>
<td>19.1</td>
<td>117.14</td>
<td>0</td>
<td>4.07%</td>
<td>-</td>
</tr>
<tr>
<td>Triad</td>
<td>Baseline</td>
<td>2864</td>
<td>9.7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Indirect</td>
<td>2755.5</td>
<td>20.2</td>
<td>108.46</td>
<td>0</td>
<td>3.79%</td>
<td>0.163</td>
</tr>
<tr>
<td></td>
<td>Direct</td>
<td>2751.5</td>
<td>17.5</td>
<td>112.41</td>
<td>0</td>
<td>3.93%</td>
<td>-</td>
</tr>
<tr>
<td>Average</td>
<td>Baseline</td>
<td>2681</td>
<td>8.9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Indirect</td>
<td>2577.3</td>
<td>10.8</td>
<td>103.72</td>
<td>0</td>
<td>3.87%</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>Direct</td>
<td>2572.7</td>
<td>9.1</td>
<td>108.32</td>
<td>0</td>
<td>4.04%</td>
<td>-</td>
</tr>
</tbody>
</table>

After the initial characterization of normal system behavior and baseline performance has been conducted, the performance of the benchmark under indirect and direct memory analysis modes is compared. A box plot of the performance results is shown in Figure 34 and ANOVA analysis is shown in Table 10. Each performance measurement is grouped by benchmark subtest copy, scale, add and triad as well as the average of the four subtests. These box plots reveal outliers for some of the data sets. After further investigation, removing these outliers does not affect the outcomes from the data; therefore they are retained. This convention is used throughout the remainder of
Chapter 5. Outliers that are significant to the outcome from the data are explained and handled accordingly.

An initial look at the box plot reveals that for both indirect and direct analysis modes, the data transfer rate is reduced compared to the baseline. In addition, the magnitude of the reduction in performance of the direct mode appears to be similar or slightly lower than that of the indirect mode. Further ANOVA analysis, as shown in Table 10, reveals that these conclusions are accurate.

For each memory analysis mode, the mean and standard deviation is calculated for each INTmem subtest. Within each subtest the sets of performance data are then compared. The estimate of the difference between the baseline and the indirect or direct memory analysis modes are calculated. The column labeled “Decrease in Performance” in Table 10 shows that while CMAT-V is running (indirect or direct mode) the decrease in performance ranges from 3.86% to 4.17%. This supports the expected conclusion that executing CMAT-V impacts VM benchmark performance.

The next step in the analysis is to attempt to determine if there is a significant difference in the decrease in performance between indirect and direct modes. If such a significant difference exists, it is possible that this could be attributed to the live-memory introspection, suggesting that it is detectable by the VM itself. The estimated difference compared to the baseline between indirect and direct modes varies from 2.81 to 7.91 MB/s. To more accurately quantify the difference, or lack thereof, the mean baseline for each subtest is used to calculate the change in performance for each run conducted. This
calculation assumes a constant baseline for comparison purposes. The calculation of the difference in performance for each subtest is shown in (1) and (2).

$$\mu_B - [I_1, I_2, \ldots I_n] = [\Delta I_1, \Delta I_2, \ldots \Delta I_n] \quad (1)$$

$$\mu_B - [D_1, D_2, \ldots D_n] = [\Delta D_1, \Delta D_2, \ldots \Delta D_n] \quad (2)$$

The variable $\mu_B$ is the mean baseline performance for the subtest, $I$ is the performance with indirect mode and $D$ is the performance in direct mode for each run $n$. The change in performance results is calculated for both indirect and direct memory analysis modes represented as $\Delta_I$ and $\Delta_D$ respectively.

As a simple example, consider the case where a copy subtest baseline mean is $\mu_B = 2300$ MB/s. Further consider the case where the data set from the indirect benchmark results for the copy subtest was $I = \{2200, 2150, 2253, 2215, 2126\}$ MB/s; for the direct benchmark results $D = \{2290, 2157, 2200, 2101, 2036\}$ MB/s. To calculate the change in performance for each memory analysis mode, calculations would be made for the indirect mode as shown in 3-6.

$$\Delta_I = [\Delta I_1, \Delta I_2, \Delta I_3, \Delta I_4, \Delta I_5] \quad (3)$$

$$= [(\mu_B - I_1), (\mu_B - I_2), (\mu_B - I_3), (\mu_B - I_4), (\mu_B - I_5)] \quad (4)$$

$$= [(2300 - 2200), (2300 - 2150), (2300 - 2235), (2300 - 2215), (2300 - 2126)] \quad (5)$$

$$= [100, 150, 47, 85, 174] \quad (6)$$
Similarly, calculations are made for direct mode as shown in 7-10.

\[ \Delta_D = [\Delta_{D1}, \Delta_{D2}, \Delta_{D3}, \Delta_{D4}, \Delta_{D5}] \]  \hspace{1cm} (7)

\[ = [(\mu_B - D_1), (\mu_B - D_2), (\mu_B - D_3), (\mu_B - D_4), (\mu_B - D_5)] \]  \hspace{1cm} (8)

\[ = [(2300 - 2290), (2300 - 2157), (2300 - 2200),
(2300 - 2101), (2300 - 2036)] \]  \hspace{1cm} (9)

\[ = [10, 143, 100, 99, 264] \]  \hspace{1cm} (10)

Now that both as \( \Delta_t \) and \( \Delta_D \) have been calculated, ANOVA testing is be used to compare these two data sets. Using a two-sample t-test reveals that the P-value is 0.808 which means there is an 80.8% probability that the sample population means are not statistically different.

Using the data from the experiment, a similar ANOVA test is conducted for each subtest to determine whether or not there is a significant difference in the change in performance. The results of this analysis are shown in the far right column in Table 10. By convention, any P-value above 0.1 when testing for a difference of zero suggests that the sample populations are not statistically different. As such, the results indicate that the scale and average subtests are statistically different while the copy, add and triad subtests are not statistically different. However, the change in system state over time must now be considered. The change in performance observed between memory analysis modes (2.81 to 7.91 MB/s) is within the same magnitude as the variation in baseline
performance measured over the course of the experiment (2.85 to 7.89 MB/s). As a consequence, it is highly likely that the difference in memory analysis mode performance is heavily influenced by change in system state. This means that changes in performance strictly due to alternating memory analysis modes could be masked or do not exist. In summary, when taking changes in baseline performance in account, the data does not reveal any statistically significant differences in performance between indirect and direct analysis modes.

4.2.2. FLOATmem Performance

Similar to Section 4.2.1, the first step in analyzing the performance of the FLOATmem benchmark is to ensure that the data collected reflects normal system behavior. The results for the 350 runs completed are shown in Figure 35.

![Figure 35. VM Analysis: FLOATmem Scatter Plot of Average.](image)

Figure 35. VM Analysis: FLOATmem Scatter Plot of Average.
Similar to the INTmem analysis, these results show that the first several runs of each set encounter distinctively high data transfer rates. This initial spike in performance could likely be caused by the time delay for the Xen load balancing routines to evaluate and balance the workload. As such, the first 10 runs of each set are not considered in later analysis.

Another anomaly in this data set occurs during runs 116 to 122 while testing in indirect memory analysis mode. During these runs there is a distinct increase in data transfer rate. Further investigation reveals that, on occasion, when CMAT-V is retrieving data from the Microsoft symbol server, the request to the server times out. During this time, CMAT-V halts further execution as and waits for a response from Microsoft. While CMAT-V is halted, Xen detects this reduction in resource requirements and reallocates them to other processes, namely the VM running the benchmark. It is no surprise then, that when CMAT-V is halted the benchmark performance is similar to that of the baseline measurements which are taken without CMAT-V executing. This stall in CMAT-V execution is an uncommon occurrence and not representative of typical CMAT-V behavior. As a result, runs 116 to 122 are also not considered in later analysis.

Next, the baseline performance sets are compared to determine if any factors affect the system state over time. A baseline is run before the indirect mode testing, between tests and after the direct mode testing. If any significant increases or decreases in baseline performance over time exist compared to the modes under test, the effects of the variations must be taken into account. ANOVA statistics and box plot of the three baselines are shown in Table 11 and Figure 36.
The ANOVA test reveals that there is no difference observed between the first and the third (P-value = 0.903) baseline sets while a moderate difference is observed between the first and third sets and the second intermediate baseline set (P-value = 0.04 and 0.056). This is similar to the trend observed during the INTmem test in Section 4.2.1, where the intermediate baseline test performed significantly lower than the other two. With this test however, the estimated difference in baseline sets range from 0.17 to
3.58 MB/s which is significantly less than in the previous section. Again, it is possible that over time a slight decrease in baseline performance could exist. Though the characterization of this trend would require further testing, the data provided indicates that an inherent variation exists independent of the memory analysis mode. This variation must be taken into account during subsequent performance evaluation.

After the initial characterization of normal system behavior and baseline performance has been conducted, the performance of the benchmark under indirect and direct memory analysis modes is compared. As shown in the box plot in Figure 37, it is apparent that during both indirect and direct analysis mode, a significant decrease in benchmark performance exists.

Figure 37. Boxplot of CMAT-V Impact on VM FLOATmem Performance.
To further quantify the difference between indirect and direct performance, ANOVA tests are conducted and the results are shown in Table 12.

Table 12. ANOVA of CMAT-V Impact on VM FLOA\textsuperscript{t}m\textsuperscript{em} Performance.

<table>
<thead>
<tr>
<th>INT\textsuperscript{t}m Subtest</th>
<th>Memory Analysis Mode</th>
<th>Mean (MB/s)</th>
<th>StDev (MB/s)</th>
<th>Est Difference (MB/s) [Baseline - X]</th>
<th>P-Value [Diff = 0]</th>
<th>Decrease in Performance</th>
<th>P-Value Ind vs Dir Δ Performance [Diff = 0]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Baseline</td>
<td>2564.2</td>
<td>18.8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Indirect</td>
<td>2482.4</td>
<td>19.9</td>
<td>81.84</td>
<td>0</td>
<td>3.19%</td>
<td>0.302</td>
</tr>
<tr>
<td></td>
<td>Direct</td>
<td>2478.8</td>
<td>26.1</td>
<td>85.47</td>
<td>0</td>
<td>3.33%</td>
<td>-</td>
</tr>
<tr>
<td>Scale</td>
<td>Baseline</td>
<td>2557.1</td>
<td>12.9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Indirect</td>
<td>2474.7</td>
<td>22.6</td>
<td>82.44</td>
<td>0</td>
<td>3.22%</td>
<td>0.150</td>
</tr>
<tr>
<td></td>
<td>Direct</td>
<td>2470.2</td>
<td>17.7</td>
<td>86.9</td>
<td>0</td>
<td>3.40%</td>
<td>-</td>
</tr>
<tr>
<td>Add</td>
<td>Baseline</td>
<td>2925.8</td>
<td>10.7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Indirect</td>
<td>2827.2</td>
<td>21.7</td>
<td>98.59</td>
<td>0</td>
<td>3.37%</td>
<td>0.844</td>
</tr>
<tr>
<td></td>
<td>Direct</td>
<td>2826.6</td>
<td>24.0</td>
<td>99.27</td>
<td>0</td>
<td>3.39%</td>
<td>-</td>
</tr>
<tr>
<td>Triad</td>
<td>Baseline</td>
<td>2922.3</td>
<td>10.7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Indirect</td>
<td>2821.2</td>
<td>20.9</td>
<td>101.8</td>
<td>0</td>
<td>3.46%</td>
<td>0.693</td>
</tr>
<tr>
<td></td>
<td>Direct</td>
<td>2822.5</td>
<td>21.3</td>
<td>99.82</td>
<td>0</td>
<td>3.42%</td>
<td>-</td>
</tr>
<tr>
<td>Average</td>
<td>Baseline</td>
<td>2742.4</td>
<td>7.4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Indirect</td>
<td>2651.4</td>
<td>9.0</td>
<td>90.99</td>
<td>0</td>
<td>3.32%</td>
<td>0.235</td>
</tr>
<tr>
<td></td>
<td>Direct</td>
<td>2649.5</td>
<td>11.7</td>
<td>92.87</td>
<td>0</td>
<td>3.39%</td>
<td>-</td>
</tr>
</tbody>
</table>

For each memory analysis mode, the mean and standard deviation is calculated for each FLOA\textsuperscript{t}m\textsuperscript{em} subtest. Within each subtest the sets of performance data are then compared. The estimate of the difference between the baseline and the indirect or direct memory analysis modes are calculated. The decrease in performance while CMAT-V is running (indirect or direct mode) ranges from 81.84 to 101.8 MB/s which equates to 3.19% to 3.46%. This supports the expected conclusion that the benchmark performance running CMAT-V significantly impacts VM system performance.
Furthermore, when observing the estimated difference compared to the baseline, for each subtest the difference between indirect and direct modes varies from 0.68 to 4.46 MB/s (99.27 - 98.59 and 86.9 - 82.44 respectively). To more accurately quantify the difference, or lack thereof, the mean baseline for each subtest is used to calculate the change in performance for each run conducted. This calculation assumes a constant baseline for comparison purposes. The calculation of the difference in performance for each subtest is the same as described in Section 4.2.1 and with (1) and (2). The results indicate that for all subtests the performance of indirect and direct modes are not statistically different.

Furthermore, change in performance observed between memory analysis modes is within the same magnitude as the variation in baseline performance measured over the course of the experiment (0.17 to 3.58 MB/s). It is highly likely that the any observed difference in performance is heavily influenced by unidentified changes in system state. This means that changes in performance strictly due to alternating memory analysis modes could be masked or do not exist. In summary, a difference in memory analysis modes is not observed within the VM for the FLOATmem benchmark.

4.2.3. Conclusion

For all tests the chosen number of runs were sufficient to draw accurate conclusions about the data; therefore, no additional runs were conducted. Within the VM itself, a decrease of approximately 3% to 4.5% was observed by executing CMAT-V within the Dom0 HM-API. In order to completely isolate changes caused by CMAT-V’s live-memory analysis one must ensure that they have a baseline to compare to that does not change over time as the system changes. Achieving these conditions however is very
difficult, if not improbable, to achieve for any system with real-life workloads. Even if these conditions are met, the existence of VM interference unique to CMAT-V live analysis remains inconclusive. However, if the system state changes over time are taken into account, the data collected under the experimental conditions described in this research support the hypothesis that direct analysis does not uniquely interfere with VM performance. Therefore, when under realistic workloads, the accurate detection of unique interference caused by direct CMAT-V introspection is predicted as highly unlikely. Finally, CMAT-V was able to accommodate all memory analysis modes. This demonstrates that CMAT-V can scan two different simultaneously running VMs.

4.3. Host System Performance

This section contains the results from the performance tests to measure the overhead of running CMAT-V on the HM-API. Sections 4.3.1 to 4.3.3 describe the results from the INTmem, FLOATmem and FFmpeg benchmarks respectively. Section 4.3.4 contains concluding remarks based on the results.

4.3.1. INTmem Benchmark

It is first important to note that for this series of tests, there was no abnormal behavior of interest such that any runs were required to be removed from later analysis. The performance during the entire experiment is shown in Figure 38.
Baseline measurements are made both before and after running the INTmem benchmark. The results of these measurements are shown in Table 13 and Figure 39. These results show that there is a statistical difference in the baselines measured before and after the test (P-value = 0). The estimate of the difference between the sample populations is approximately 21.58 MB/s. This indicates that any calculated increase or decrease in performance calculated using compiled baseline data will have an associated error of approximately +/- 10.79 MB/s.

Table 13. ANOVA Results for Host INTmem Average Baseline Performance.

<table>
<thead>
<tr>
<th>Baseline Test</th>
<th>1</th>
<th>2</th>
<th>Tests Compared</th>
<th>1-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>2971.5</td>
<td>2949.9</td>
<td>P-Value [Diff = 0]</td>
<td>0</td>
</tr>
<tr>
<td>StDev</td>
<td>19.3</td>
<td>18.5</td>
<td>Est. Difference [First - Second]</td>
<td>21.58</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Percent Change</td>
<td>0.73%</td>
</tr>
</tbody>
</table>

1 = Before Direct Test
2 = After Direct Test
The relevance of this difference in baseline in regards to the experiment being conducted is determined after analyzing CMAT-V performance results. If the difference in baseline performance is significantly lower in magnitude than the performance difference running CMAT-V, this trend could be considered negligible.

A comparison of the host performance results between the baseline and direct modes is shown in the box plot in Figure 40. As expected, it appears there is a significant decrease in benchmark performance while CMAT-V is executing. Further analysis to quantify this decrease in performance is shown in Table 14.
The mean and standard deviation for each set of runs is shown in rows 3 and 4 of Table 14. As shown in row 6, the samples for the baseline and direct mode measurements are statistically different for each INTmem subtest (P-value = 0). Furthermore, for each INTmem subtest, a decrease of approximately 9.51% to 10.20% is observed. These performance results have an approximate error of 0.34% to 0.40% due
to variations in baseline measurements calculated previously. Because the error due to baseline variation is less than 0.5% these errors are considered negligible.

### 4.3.2. FLOATmem Benchmark

Similar to Section 4.3.1, there was no abnormal behavior of interest such that any outliers needed to be removed for later analysis.

![Figure 41. Scatter Plot of Host FLOATmem Average Performance.](image)

The performance during the entire experiment is shown in Figure 41. Baseline measurements are made both before and after running the FLOATmem benchmark. The results of these measurements are shown in Table 15 and Figure 42.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mean</strong></td>
<td>3005.7</td>
<td>2997.2</td>
</tr>
<tr>
<td><strong>StDev</strong></td>
<td>18.9</td>
<td>15.2</td>
</tr>
</tbody>
</table>

1 = Before Direct Test  
2 = After Direct Test

<table>
<thead>
<tr>
<th>Tests Compared</th>
<th>1-2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P-Value [Diff = 0]</strong></td>
<td>0.015</td>
</tr>
<tr>
<td><strong>Est. Difference [First - Second]</strong></td>
<td>8.47</td>
</tr>
<tr>
<td><strong>Percent Change</strong></td>
<td>0.28%</td>
</tr>
</tbody>
</table>

Table 15. ANOVA Results for Host FLOATmem Average Baseline Performance.
The results in Table 15 show that there is a statistical difference in the baselines measured before and after the test (P-value = 0.015 < 0.1). The estimate of the difference between the sample populations is approximately 8.47 MB/s. This indicates that any calculated increase or decrease in performance calculated using compiled baseline data will have an associated error of approximately +/- 4.24 MB/s. The relevance of this difference in baseline in regards to the experiment being conducted is determined after analyzing CMAT-V performance results. If the difference in baseline performance is significantly lower in magnitude than the performance difference running CMAT-V, this trend could be considered negligible.

A comparison of the host performance results between the baseline and direct modes is shown in the box plot in Figure 43. As expected, it appears there is a significant decrease in benchmark performance while CMAT-V is executing.
Figure 43. Boxplot of CMAT-V Impact on Host FLOATmem Performance.

Further analysis to quantify this decrease in performance is shown in Table 16.

Table 16. ANOVA of Host FLOATmem Performance.

<table>
<thead>
<tr>
<th>FLOATmem Subtest</th>
<th>Memory Analysis Mode</th>
<th>Copy</th>
<th>Scale</th>
<th>Add</th>
<th>Triad</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean (MB/s)</td>
<td>Baseline</td>
<td>Direct</td>
<td>Baseline</td>
<td>Direct</td>
<td>Baseline</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2741.9</td>
<td>2500.3</td>
<td>2726.9</td>
<td>2498.6</td>
<td>3270.4</td>
</tr>
<tr>
<td></td>
<td>StDev (MB/s)</td>
<td>22.7</td>
<td>57</td>
<td>23.3</td>
<td>58.6</td>
<td>36.4</td>
</tr>
<tr>
<td>Est Difference (MB/s) [Baseline - CMAT-V]</td>
<td>241.66</td>
<td>228.39</td>
<td>309.26</td>
<td>307.82</td>
<td>271.78</td>
<td></td>
</tr>
<tr>
<td>P-Value [Diff = 0]</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Decrease in Performance</td>
<td>8.81%</td>
<td>8.37%</td>
<td>9.46%</td>
<td>9.43%</td>
<td>9.06%</td>
<td></td>
</tr>
<tr>
<td>Error (+/-)</td>
<td>0.15%</td>
<td>0.16%</td>
<td>0.13%</td>
<td>0.13%</td>
<td>0.14%</td>
<td></td>
</tr>
</tbody>
</table>

The mean and standard deviation for each set of runs is shown. As suspected, the samples for the baseline and direct mode measurements are statistically different for each
FLOATmem subtest (P-value = 0). Furthermore, for each FLAOTmem subtest, a decrease of approximately 8.81% to 9.46% is observed. These performance results have an approximate error of 0.13% to 0.16% due to variations in baseline measurements calculated previously. Because the error due to baseline variation is less than 0.5% these errors are considered negligible.

4.3.3. FFmpeg Benchmark

The performance of the FFmpeg benchmark over the course of the experiment is shown in Figure 44.

![Figure 44. Scatter Plot of Host FFmpeg Performance.](image)

Recall, the metric for this benchmark is the number of seconds it takes for the benchmark to complete. Therefore, higher values indicate a decrease in performance. The first two baseline runs show a mildly uncharacteristic decrease in performance. The
first baseline set quickly reaches a steady state while all remaining sets show no abnormal behavior. Because these outliers are rare and limited in overall impact, they do not affect any outcomes. As such, they are included in the benchmark data for later analysis.

Baseline measurements are made both before and after running the FFmpeg benchmark. The results of these measurements are shown in Table 17 and Figure 45.

Table 17. ANOVA Results for Host FFmpeg Baseline Performance.

<table>
<thead>
<tr>
<th>Baseline Test</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>29.12</td>
<td>29.08</td>
</tr>
<tr>
<td>StDev</td>
<td>0.21</td>
<td>0.16</td>
</tr>
</tbody>
</table>

Tests Compared | 1-2
---           ---|
P-Value [Diff = 0] | 0.342
Est. Difference [First - Second] | 0.04
Percent Change | 0.12%

1 = Before Direct Test  
2 = After Direct Test

Figure 45. Box Plot of Host FFmpeg Baseline Performance.

The results in Table 17 show that there is not a statistical difference in the baselines measured before and after the test (P-value = 0.342 > 0.1). This means that
intermediate execution of CMAT-V did not significantly affect the performance of the baseline measurements. This is likely different from previous RAMspeed tests because FFmpeg is not designed with specific sensitivity towards memory performance. Because the baseline sets are not different, this indicates limited change in system state over time which better isolates the cause of performance variation when changing memory analysis modes.

A comparison of the host performance results between the baseline and direct modes is shown in the box plot in Figure 46.

![Boxplot of CMAT-V Impact on Host FFmpeg Performance](image)

As expected, it appears there is a significant decrease in benchmark performance while CMAT-V is executing. Further analysis to quantify this decrease in performance is shown in Table 18.
Table 18. ANOVA of Host FFmpeg Performance.

<table>
<thead>
<tr>
<th>Memory Analysis Mode</th>
<th>Baseline</th>
<th>Direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean (s)</td>
<td>29.098</td>
<td>32.583</td>
</tr>
<tr>
<td>StDev (s)</td>
<td>16.5</td>
<td>64.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Est Difference (s) [Baseline - CMAT-V]</th>
<th>-3.4852</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-Value [Diff = 0]</td>
<td>0</td>
</tr>
<tr>
<td>Decrease in Performance</td>
<td>11.98%</td>
</tr>
</tbody>
</table>

The mean and standard deviation for each set of runs is shown. The samples for the baseline and direct mode measurements are statistically different (P-value = 0). Furthermore, a decrease of approximately 11.98% is observed.

4.3.4. Conclusion

For all tests the chosen number of runs were sufficient to draw accurate conclusions about the data; therefore, no additional runs were conducted. In summary, an 8% to 12% decrease in performance was observed while running CMAT-V within the Dom0 HM-API. For the RAMspeed benchmarks, variations in the baseline measurements were also observed, but were determined as negligible.
V. Conclusions

This chapter provides a summary of key findings of this research. Section 5.1 contains an executive summary of the results. Section 5.2 gives recommendations for future follow-on research.

5.1. Executive Summary

The following sections describe how each goal of the research effort was met, summarizes the experimental results and draws final conclusions based on the data.

5.1.1. Create a VMI Prototype

The first goal identified was to create a prototype VMI tool that accesses live VM memory, detect the OS executing in the VM and extract system state information. This objective was achieved by developing CMAT-V, which utilizes CMAT, XenAccess and Xen APIs. During experimental testing, strategic system state information was successfully extracted from two live VM’s. This information includes kernel base address, active process lists, open network ports, registry keys, and system files and directories. CMAT-V uses out-of-bound view generation. Recall from Section 2.2 that this is formally described as shown below.

\[ g_\mu : S_{ext} \rightarrow V_{ext} \]

CMAT-V uses software architecture information \( \mu \) to create a view \( V_{ext} \) of the system state \( S_{ext} \) observed from the VMM or hypervisor level. Furthermore, the information in
Table 19 can be added to the VMI utilities summary shown in Table 2 of Section 2.3 to include CMAT-V.

Table 19. CMAT-V Application Summary.

<table>
<thead>
<tr>
<th>Name</th>
<th>VMM(s) Tested</th>
<th>OS(s) Tested</th>
<th>VMI Pattern</th>
<th>Description</th>
<th>Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMAT-V</td>
<td>Xen</td>
<td>Windows</td>
<td>Out-of-Band</td>
<td>Detects OS and uses semantic information downloaded from Microsoft Symbol</td>
<td>Depends on Microsoft semantics and the availability of the Microsoft Symbol</td>
</tr>
</tbody>
</table>

For Windows XP operating systems, CMAT-V is binding. In other words, because CMAT-V detects the Windows version, malware cannot leverage vulnerabilities introduced with new patches and updates.

The success of the CMAT-V prototype shows that VMI can be conducted on Windows-based VMs using static memory analysis techniques. These techniques do not require a priori OS semantic information and do not interfere with VM operation by using pausing techniques. One limitation to this approach, however, is that the system state of the VM itself can change during the course of the CMAT-V memory scan. Any changes that occur between when the scan starts and completes are not guaranteed to be reflected in the results reported by CMAT-V. If a history record of system state data is kept, however, CMAT-V is ideal for continual scanning over time without stalling the VM.
5.1.2. Verify Live Analysis Under Cyber Attack

The second goal was to verify live analysis functionality while under cyber attack. Four cyber attack scenarios were run to determine CMAT-V’s ability to provide evidence of the threat. These scenarios focused on rootkit-class malware which are known for hiding information. For all four scenarios, while the VM itself often reported no evidence of compromise, CMAT-V was successful in providing one or more distinguishing characteristics of the existence of the threat. This confirms that sensors which leverage the hypervisor’s privileged position can provide uncompromised situational awareness information for network defense operators.

5.1.3. Evaluate the Detectability of VM Memory Latency

This set of experiments focused on VMI performance within the VM itself. These tests revealed a 3% to 4.5% decrease in VM benchmark performance when executing CMAT-V within the Dom0 HM-API. Furthermore, under the experimental conditions described, the data was not able to confirm VM detection of the presence of CMAT-V’s live memory analysis. If a stable system state can be maintained throughout all baseline benchmarks, it is possible that a detectable presence of CMAT-V could be seen. However, establishing a consistent baseline with memory sensitive benchmarks could prove difficult because they are particularly sensitive to varying user workloads over time. Unless this challenge is overcome, detection of CMAT-V is highly unlikely.
5.1.4. **Verify Multi-VM Live Analysis**

Another goal was to demonstrate that two different VMs executing simultaneously could be analyzed to show that the prototype is not limited to one VM at a time, but can support multiple active VMs. This was demonstrated by using CMAT-V in both indirect and direct memory analysis modes. In each mode, two VMs are simultaneously executing. By successfully implementing these modes, this research demonstrated that CMAT-V could conduct live analysis on both of the VMs.

5.1.5. **Measure System Overhead**

The final goal was to evaluate system overhead. This set of experiments focused on the impact of CMAT-V on the host system. The results from these experiments, which used three different benchmarks, revealed an 8% to 12% decrease in performance if CMAT-V is executing within the Dom0 HM-API. The significance of this decrease in performance will depend on the user’s allowable threshold for the desired application.

5.2. **Future Work**

The following sections contain suggestions for future work. These suggestions include evaluation and enhancements to the edition of CMAT-V used in this research.
5.2.1. *Thread List Introspection*

CMAT and CMAT-V currently scan the EPROCESS linked list to identify discrepancies between what CMAT detects and what the target system reports. A similar approach could be used for the executive thread (ETHREAD) block that contains thread information for a given process [Rus09]. Each ETHREAD block contains a pointer back to the parent EPROCESS block. While EPROCESS blocks can be unlinked and hidden from the OS, the process threads must execute on the CPU, therefore their existence is known. Future work should involve displaying running threads for each process. In addition, cross validation can be conducted by detecting threads executing on the CPU that point to a parent process not in the EPROCESS block.

5.2.2. *Response Time Analysis*

In this study, tests were conducted to determine the interference of CMAT-V on the DomU guest as well as the overhead on the Dom0 HM-API. Another important metric to consider is the amount of time it takes a user of CMAT-V to retrieve relevant system state information. For example, if it takes CMAT-V too long to report the results, then it is likely that it will be impractical to use in the field. In addition, the faster CMAT-V is able to report results, the less time is allowed for the system state itself to change while introspection is taking place. Performance of CMAT-V should be evaluated by taking timing measurements and comparing the performance of CMAT-V to similar VMI applications.
5.2.3. **Advanced Registry Search**

Currently, CMAT-V only reports root-level registry key information for a particular process. As discussed previously, rootkits use and often hide registry key information. If the user were looking for a particular registry key, it would be valuable for CMAT-V to include a registry key search capability. Similar programs such as *Active Registry Monitor* allow users to conduct searches for specific registry keys. This search capability, however, might take a long time to execute. This will need to be taken into consideration and optimized as appropriate.

5.2.4. **Driver Detection**

Many rootkits rely on custom drivers in order to gain access to kernel level functions. Adding the functionality for CMAT-V to detect system drivers would provide a more complete picture of the VM’s system state. One possible detection method is to search for driver objects which represent an individual driver connected to a system process. *Process Explorer* is an example of a program that lists the loaded drivers, their names, version information and load address [Rus09]. Similar techniques could be used to enhance CMAT-V functionality.

5.2.5. **IP Detection**

Currently CMAT-V is able to report open port information. Future CMAT-V development should focus on determining IP addresses of the open ports. With this IP information, users will be able to determine where the connection is being made and if necessary, conduct appropriate counter measures.
5.3. Concluding Remarks

There is no silver bullet in the area of cyber defense. The cyber domain requires a multi-faceted approach where network policies, firewalls, sensors, antivirus, etc. must all be leveraged to deliver synergistic effects. As part of this “cyber-toolkit,” hypervisor-based VMI provides an unparalleled strategic position to achieve cyber situational awareness. This research, however, is just the tip of the iceberg. There is a well of untapped potential in the area of hypervisor-based virtual machine introspection. More research and development needs to be conducted to further this technology. These sensors are essential to military network defense networks now and will be even more critical in the future.
Appendix A: Installing CentOS 5 Using Xen Hypervisor

A-1. Downloading CentOS 5 & Making Installation Boot DVD

1. Visit http://mirror.centos.org/centos/5/isos/ for CentOS 5 download and select either i386 (for 32-bit) or x86_64 (for 64 bit).\(^1\)

2. Select a download location (ex. http://mirrors.rit.edu/centos/5.4/isos/i386/).

3. Download CentOS-5.4-i386-bin-DVD.iso (~3.7 GB).

4. To ensure the ISO has been downloaded correctly, it may necessary to conduct a hash test of the ISO file. To do this, execute the following in the terminal:

   `md5sum [FILE]` where `[FILE]` is the path to the file CentOS-5.4-i386-bin-DVD.iso. This will perform a checksum of the file to verify if there are any errors. This will take a few minutes to complete. Once the checksum is complete it returns a stream of hex numbers. Take this checksum stream and check it with checksum in a txt file usually located at the same mirror site used in step 2 (e.g., md5sum.txt). If it does not match, try repeating the download (step 3) or change your mirror site (step 2).

5. Burn ISO to a DVD using CD/DVD burning software (e.g., Disc Burner for Linux; IMG Burn for Windows).

A-2. Installing CentOS 5

1. Place ISO disk into drive and reboot the computer.

2. Follow the on screen installation instructions.

3. When prompted to select additional tasks for CentOS to support, click the **Virtualization** check box. This will install the Xen hypervisor as well as all dependencies (including libvirt, and virtual machine manager).

If you are installing Xen after CentOS has already been installed, see for installation instructions [http://www.howtoforge.com/centos_5.0_xen](http://www.howtoforge.com/centos_5.0_xen)¹.

4. Continue following on screen installation instructions until CentOS install is complete. When prompted, the system will reboot.

---

5. Upon reboot you might see the Grand Unified Bood Loader (GRUB) menu to select which kernel you would like to boot with. Select the xen kernel which is identified by some version number followed by el5xen. The screenshot below shows two different version examples. The version installed may or may not match those shown below.

6. Once you have booted CentOS you can verify the current kernel being used by typing and executing `uname -r` in the terminal.
You should see the Xen kernel displayed. You are now running in the paravirtualized Dom0 operating system.

A-3. Switching to HAV Mode

In order to enable full virtualization with use of HAV such as Intel-VT or AMD-V. It may be necessary to manually enable virtualization within the system BIOS. This allows operating systems such as Windows to be installed as a fully virtualized guest in the Virtual Machine Manager without access to the OS source code. Follow the following procedures to enable virtualization in the BIOS with the Dell Latitude D630 laptop.

1. Reboot the system.

2. Upon reboot enter the BIOS setup by pressing the appropriate initiation key. For Dell Latitude D630, press the F2 key.

3. Once in the BIOS setup use the arrow keys to expand the POST Behavior dropdown. Under POST Behavior, arrow down to Virtualization settings.
Hit ENTER to modify the settings and change the setting to Enabled.

4. Hit ESC and select save settings and exit. At this point your PC is now configured to enable HAV.

A-4. Installing Windows Using Xen HAV

The following procedure guides you through creating a Windows XP virtual machine using hardware assisted virtualization. To complete this lab you must have completed CentOS 5 installation as described previously and have a Windows XP installation CD.

1. Place the Windows XP installation CD/DVD in the CD/DVD-ROM drive.
2. In CentOS 5 click on the **Applications** menu bar and go to **System Tools>Virtual Machine Manager**. This will launch the virtual machine manager that guides you through creating a virtual machine.

3. In the virtual machine manager click on the local host with ID **xen** and then click on the button.

4. VMM will then give you instructions on how to create a virtual machine. Click the button.

5. In the virtual machine name dialog box type in **Windows_XP_1**. And click the button.

6. Select the radio button for **Fully Virtualized**, then click the button. 

   *NOTE: If this selection is grayed out, see Section A-3.*
7. Check and make sure the Local install media (ISO image or CDROM) is selected. Set the OS type to Windows and the OS Variant to Microsoft Windows XP x86. Click forward.
8. Click the radio button for **CD-ROM or DVD** install and select device mount for your CD-ROM drive. Click Forward.
9. In the storage dialog box select the **File (disk image)** radio button and set the path to install your image file (e.g., `/var/lib/xen/images/Windows_XP_1`). Set the **Size** to be 10,000 MB to allocate 10 GB of hard drive for your virtual machine. Ensure the **Allocate entire virtual disk now** is checked. Click **Finish**.

10. In the Network setup dialog, make sure that the Virtual Network radio button is selected and the Network dropdown menu is set to **default**. Click **Forward**.
11. In the Memory and CPU Allocation dialog set Max Memory and Startup memory to 512 and set Virtual CPUs to 1. Click Forward.

12. To finish the virtual machine creation and begin installation, click Finish.
13. This should launch your virtual machine and begin installation from your Windows XP disk. Follow the on screen instructions to begin installation. HINT: To return mouse functionality to the host OS (CentOS 5) enter Ctrl + Alt.
14. Once setup is complete your VM will shut down. You should now see the following in your virtual machine manager window:
To restart your computer double click Windows_XP_1 to bring up the VM window (if the window is not open already) then click **Run**. This will reboot your VM and launch Windows XP.

15. Follow the on screen instructions to complete Windows XP installation.
Appendix B: Installing and Using CMAT-V

B-1. Preparing Your System

1. First, install CentOS 5 development tools. These tools will allow you to edit/compile the C program. In the CentOS 5 host, click Applications>Add Remove Software and select Development. Check Development Tools and click Apply. This will download and install all packages. [pic]

2. Close Package Manager and then open up a shell terminal and install the xen development tools by typing and executing “yum install xen-devel”.

3. Reopen Package Manager open and install SVN server module for Subversion server by searching for “svn” and checking the package starting with mod_dav_svn then click Apply and Continue.
4. Search for “curl” in the Package Manager. Verify that curl and curl-devel are installed; if not, select them and click Apply. Curl is used to allow CMAT-V to connect to the Internet.

5. Close Package Manager. Go to the cabextract website [www.cabextract.org.uk](http://www.cabextract.org.uk) and download the RPM for Linux (e.g., Cabextract-1.2-1.i386.rpm). This will allow CMAT-V to extract Microsoft cabinet files.

6. Run the RPM and install the package. If a message box appears stating it is unable to verify the software, click Install Anyway.
7. You are now ready to load and compile CMAT-V.

**B-2. Compiling CMAT-V v 2.7**

1. Copy the file `XA_CMAT_2_7.zip` to system. All software is available by contacting Dr. Barry Mullins at [barry.mullins@afit.edu](mailto:barry.mullins@afit.edu).

2. Extract the zip file to a desired directory.

3. Open up a shell terminal. Change to the directory `~/XA_CMAT_2_7/xenaccess`. Once in this directory type the command “make install”.

4. Once this completes, in the same directory, execute “./configure”.
5. Now change to the directory 

   /.../XA_CMAT_2_7/xenaccess/xenaccess

   and execute the command “make”.

\[ 135 \]

\[ 108 \]

**B-3. Executing CMAT-V v 2.7**

Change to the directory 

   /.../XA_CMAT_2_7/xenaccess/xenaccess/examples

Once in this directory there are several modes of operation to choose from:

   ./cmat [mode] [parameter]...[parameter]

**Static Analysis Mode:**   ./cmat [file]

This mode implements the legacy CMAT functionality. In this mode a memory dump file must be provided.

**Live VM Mode:**   ./cmat -virt_live [VM ID#]

This mode uses live VM memory to gather VM state information.

**VM Dump Mode:**   ./cmat -virt_dump [VM ID#] [output file]

This mode takes a memory dump of the live VM, then conducts static analysis on the memory dump.

**VM Benchmark Mode:**   ./cmat -virt_live_bench [VM ID#]

This mode is similar to Live VM mode except the program will immediately exit after data has been extracted by introspection. This mode is used when running benchmarks. Using a bash script with CMAT-V execution in a for loop allows for uninterrupted execution.
Appendix C: Installing Phoronix Test Suite

1) This describes the procedure to install the Phoronix Test Suite on CentOS5.

From the terminal, switch to root and install Phoronix by typing the command

```
>yum install phoronix-test-suite
```

Alternatively you may download the suite and install it.

2) You will also need to install PHP 5 CLI hypertext preprocessor by typing the command

```
>yum install php-cli
```

Once installed, run Phoronix Test Suite to display available options by typing the command

```
>phoronix-test-suite
```
Appendix D: Downloading Software Utilities

The following contains download locations for software utilities used in this research.

**Process Explorer**

Process Explorer shows a list of the currently active processes, including the names of their owning accounts. Additionally, if Process Explorer is in DLL mode, the DLLs and memory-mapped files that the process has loaded are visible. Process Explorer also has a search capability that will show which processes have particular handles opened or DLLs loaded [Rus10].


**Driver View**

“DriverView utility displays the list of all device drivers currently loaded on your system. For each driver in the list, additional useful information is displayed: load address of the driver, description, version, product name, company that created the driver, and more” [Nir10].

**Download Location:** [http://www.nirsoft.net/utils/driverview.html](http://www.nirsoft.net/utils/driverview.html)

**InstDrv**

GUI tool that allows a driver to be registered, started, stopped and removed.
NetCat for Windows

“Netcat is a featured networking utility which reads and writes data across network connections, using the TCP/IP protocol” [Gia06].

Download Location: http://joncraton.org/blog/netcat-for-windows

RegScanner

“RegScanner is a small utility that allows you to scan the Registry, find the desired Registry values that match to the specified search criteria, and display them in one list. After finding the Registry values, you can easily jump to the right value in RegEdit, simply by double-clicking the desired Registry item. You can also export the found Registry values into a .reg file that can be used in RegEdit” [Nir10].

Download Location: http://www.nirsoft.net/utils/regscanner.html

Windows Driver Kit (WDK)

“The Windows Driver Kit (WDK) Version 7.1.0 is an update to the WDK 7.0.0 release and contains the tools, code samples, documentation, compilers, headers and libraries with which software developers create drivers for Windows 7, Windows Vista, Windows XP, Windows Server 2008 R2, Windows Server 2008, and Windows Server 2003. This development kit does not contain device drivers for your personal computer. If you are looking for drivers for your personal computer, go to Microsoft Update for
downloads, or visit Windows Hardware Help for more information to find device drivers and hardware. A working knowledge of C programming is necessary to use this kit to develop Windows drivers” [Mic10].

Bibliography

http://www.accessdata.com/forensictoolkit.html


http://user-mode-linux.sourceforge.net/

http://fire.dmzs.com/index.php?section=main


http://netcat.sourceforge.net/


http://www.guidancesoftware.com/


http://www.intel.com/products/processor/manuals/


[Lit06] L. Litty and D. Lie, "Manitou: A Layer-Below Approach to Fighting
Malware," in ASID’06, San Jose, 2006, pp. 6-11.

[Lit08] L. Litty, H. Lagar-Cavilla, and D. Lie, "Hypervisor Support for Identifying
Covertly Executing Binaries," in SS'08: Proceedings of the 17th conference

2008.

http://www.merriam-webster.com/dictionary/semantic


Conference (DFRWS), 2010.

[Pag09] B. A. Pagel, "Automated Virtual Machine Introspection for Host-Based

[Par10] Parallels. (2010, Mar.) OS Virtualization Solution for Windows and Linux -
Parallels Virtuozzo Containers. [Online].
http://www.parallels.com/products/pvc45/

Virtual Machines," in Annual Computer Security Applications Conference,
ACSAC, Miami Beach, 2007, pp. 385-397.


http://www.virtualbox.org/wiki/Virtualization


   http://www.vmware.com/products/server/


   http://www.wireshark.org/
In this research, a compiled memory analysis tool for virtualization (CMAT-V) is developed as a virtual machine introspection (VMI) utility to conduct live analysis during cyber attacks. CMAT-V leverages static memory dump analysis techniques to provide live dynamic system state data. Unlike some VMI applications, CMAT-V bridges the semantic gap using derivation techniques. CMAT-V detects Windows-based operating systems and uses the Microsoft Symbol Server to provide this context to the user. This research demonstrates the usefulness of CMAT-V as a situational awareness tool during cyber attacks, tests the detection of CMAT-V from the guest system level and measures its impact on host performance. During experimental testing, live system state information was successfully extracted from two simultaneously executing virtual machines (VM’s) under four rootkit-based malware attack scenarios. For each malware attack scenario, CMAT-V was able to provide evidence of the attack. Furthermore, data from CMAT-V detection testing did not confirm detection of the presence of CMAT-V’s live memory analysis from the VM itself. This supports the conclusion that CMAT-V does not create uniquely identifiable interference in the VM. Finally, three different benchmark tests reveal an 8% to 12% decrease in the host VM performance while CMAT-V is executing.