Abstract

The research performed under this publication will combine virtualization technology with current kernel debugging techniques to provide a more powerful set of debugging capabilities than those which are currently available. More specifically, the combination of current kernel debugging technology and virtualization technology can provide a kernel debugger with a more robust set of features to distinguish and breakpoint on critical points in low-level code execution. With these improved features, many new and powerful enhancements can be added to the current state-of-the-art in kernel debugging technologies.

Enhanced Debugging Capabilities through the Application of Virtualization Technology

SYRACUSE UNIVERSITY

Master's Thesis

December 2011

Submitted in partial fulfillment of the requirements for the graduate degree of Master of Science in Computer Engineering, in The Department of Electrical Engineering and Computer Science (EECS), at Syracuse University.

By:

Ryan M. Wilson

B.S. Clarkson University 2005

Jim Fawcett, Ph.D.

Approved by: ______________________

Date: ______________________

Table of Contents

Table of Figures

1. Chapter 1 - Introduction

1.1 Research Statement

The research performed under this publication will combine virtualization technology with current kernel debugging techniques to provide a more powerful set of debugging capabilities than those which are currently available. More specifically, the combination of current kernel debugging technology and virtualization technology can provide a kernel debugger with a more robust set of features to distinguish and breakpoint on critical points in low-level code execution. With these improved features, many new and powerful enhancements can be added to the current state-of-the-art in kernel debugging technologies.

1.2 Overview

1.2.1 Main Concept

Powerful enhancements can be provided to current state-of-the-art kernel debugging technologies through creative applications of virtualization technology, or more specifically, with the assistance of a virtual machine monitor (VMM). A VMM that makes use of Intel’s Virtualization Technology (VT) has the ability to set specific conditions and take control of the central processing unit (CPU) when these conditions are met. These conditions are called exit reasons, or VM-Exits, and they proceed to pass execution control to a software handler that executes inside of the VMM. The term exit reason is used because execution flow is exiting the guest virtual machine and entering the VMM. From a debugging standpoint, the ability to trap (or breakpoint) on these events can greatly increase traditional debugging capabilities beyond simple memory and I/O breakpoints.

1.2.2 Kernel Debugger

A typical kernel debugger provides the ability to utilize a set of internal CPU debug registers that are resident on Intel x86 processors. These debug registers are used to configure and enable up to four hardware-based breakpoint conditions. When one of these breakpoint conditions is executed, it is trapped on and control is passed to interrupt handler 0x01. When interrupt handler 0x01 handles the breakpoint, control is passed back to the code execution that caused the trap to interrupt handler 0x01. With the ability to break execution flow and trap to a software handler, the user of a debugger can analyze and manipulate the state of the system at any point in time where he/she was able to successfully place a breakpoint. Figure 1 – Kernel Debugger Breakpoint Trap - Execution Flow visually demonstrates this concept below:

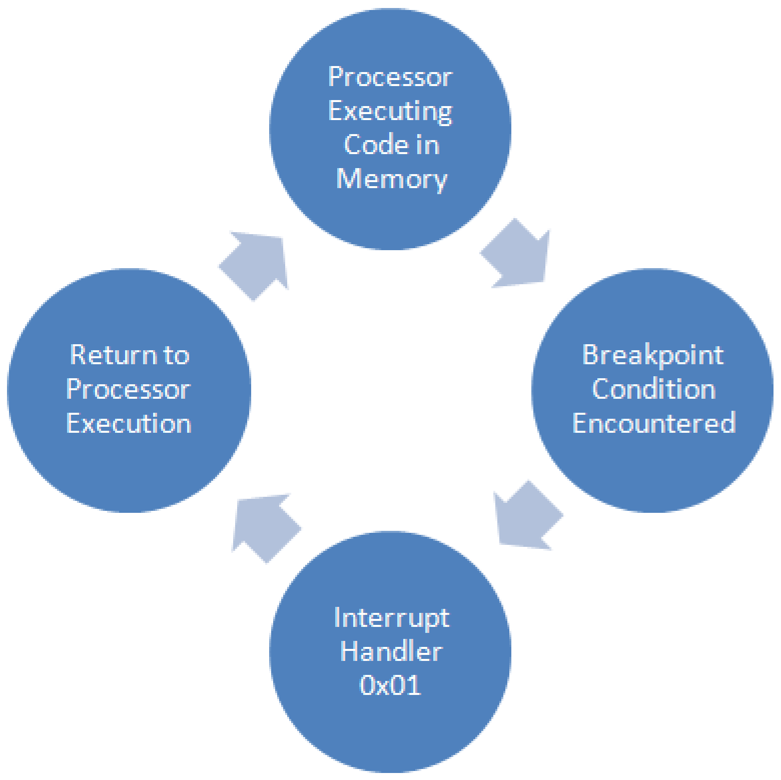

Processor Executing Code in Memory
Breakpoint Condition Encountered
Interrupt Handler 0x01
Return to Processor Execution

Figure 1 – Kernel Debugger Breakpoint Trap - Execution Flow

1.2.3 Virtual Machine Monitor

The virtualization technology that is detailed in this paper, Intel VMX technology, is made use of through a series of interactions with x86 assembly VMX commands that are provided by Intel processors that support VMX Virtualization Technology. The basic concept behind this technology is that the CPU can be placed into a state where it passes execution flow between a host VMM (Virtual Machine Monitor) and a single or multiple guest VMs (Virtual Machines). This execution flow is enforced in hardware by the processor. The main functionality of interest that is provided by a VMM is the ability to handle any instruction or event taking place within a VM Guest that generates a “VM-Exit”. A list of VM-Exits has been provided in Appendix A –Basic Exit Reasons. When a VM-Exit occurs within a VM Guest, execution flow exits the VM and is passed to the VMM for handling. The virtual machine monitor virtualizes the instruction or event that caused the VM-Exit and then passes control back to the Guest VM where execution continues after the instruction or event that caused the VM-Exit. This concept of VMMs, Guest VMs, and VM-Exits is illustrated below in Figure 2 - VMM / Guest VM Interaction Flow.

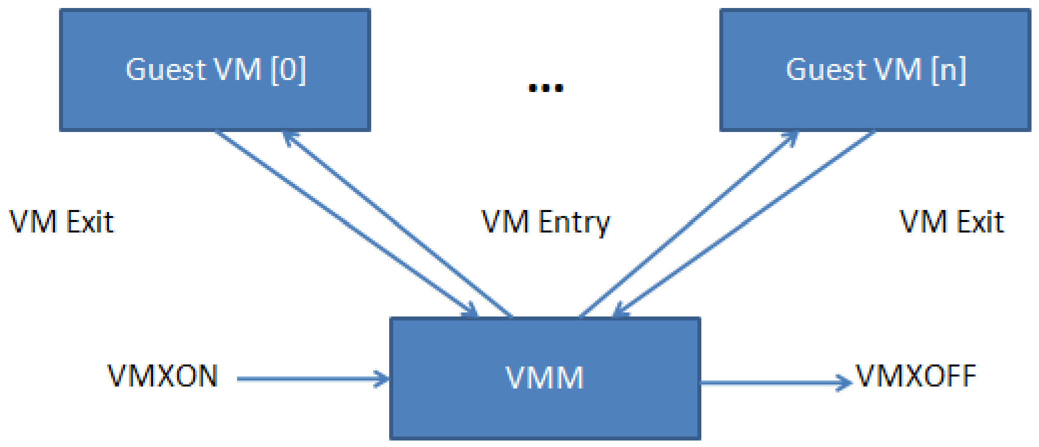


Figure 2 - VMM / Guest VM Interaction Flow

Please Note that the visual shown above in Figure 2 is a modification of a figure from the Intel Architecture Software Developer's Manuals [1].

1.2.4 Loading a VMM

It is important to note that the work performed for this thesis loads a VMM from an already running operating system. For instance, a device driver is loaded from the Windows operating system and is used to interface the processor to bring up a VMM. Once the VMM is configured and loaded, the loading operating system is now considered a guest operating system of the VMM. Therefore, once loaded, the VMM sits directly between the operating system and the CPU. The relationship between the operating system, VMM, and CPU is illustrated below in Figure 3 - Relationship between Guest OS, VMM, and CPU.

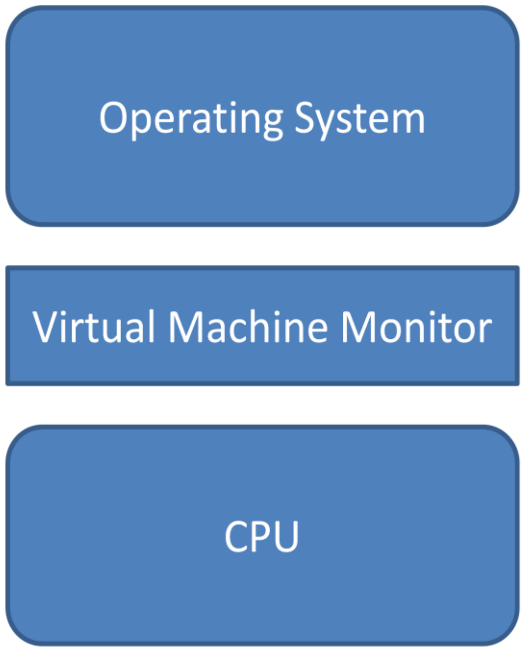


Figure 3 - Relationship between Guest OS, VMM, and CPU

1.2.5 Core Idea for Thesis

The ability for a VMM to trap on a specific system-level event and pass control to a software handler is analogous to the ability of a kernel debugger to trap on a specific breakpoint and pass control to the interrupt 0x01 handler. In contrast to a debug breakpoint, which can only be set to trap on a specified memory or I/O location, a VMM can trap on a multitude of conditions, as listed in Appendix A –Basic Exit Reasons. By combining the vast set of exit reasons provided by a VMM with the traditional capabilities of a kernel debugger, the kernel debugger can be enhanced to provide more specific fine-grained control over a system or application that is being manipulated by the debugger. With this greater degree of control a debugger can benefit from the enhancements described in section 1.3.

1.3 Contributions

This section provides an overview of the contributions that this thesis will make to existing capabilities and research within the community. Each of these contributions will be covered in greater detail in chapters 3 and 4. A summary of the enhancements described in this sections is as follows:

Enhanced Debugging Capabilities:

- More than four breakpoints can be set at once - A VMM can trap or set a virtual breakpoint for every possible exit reason. A handler can then further elaborate on the reason that the virtual breakpoint has chosen to trap.

- More fine-grained control can be obtained using a virtual breakpoint rather than a standard debug breakpoint. This control can be as detailed as the low-level events associated with each exit condition.
- Full view and control of the system state is inherently available to a VMM, greatly simplifying complex interactions with low-level aspects on a system being debugged. A VMM contains a set of data structures representing the state of the system in detail.

Applications of Enhanced Debugging Capabilities:

- Enhanced basic debugger capabilities - Large extension to the number of available breakpoints and the ability to trap on a multitude of system events, beyond traditional memory and I/O breakpoints.
- Smart Filtering of Data During Dynamic Analysis - Areas of interest can be isolated much easier and faster.
- Analysis of complex debug environments - Dynamic detection and analysis of exceptions and crashes.

1.3.1 Enhanced Kernel Debugging Capabilities

Typical hardware based kernel debugging support on an x86-based system provides the ability to set breakpoints on either virtual memory addresses or port I/O addresses. Intel processors support up to four of these breakpoints to be set at one time via the processor’s debug registers. Having the ability to trap to a breakpoint handler on any given virtual memory or I/O address can be very powerful, but it can also be very limiting to only have four simultaneous breakpoints and two categories of breakpoints (memory address or port I/O address).

Although not necessarily intended for debugging purposes, Intel VMX technology provides the ability to set “virtual breakpoints” to trap execution flow to a hypervisor handler, on a large variety of conditions that go far beyond virtual memory addresses or Port I/O locations. These virtual breakpoints can also be set in nearly unlimited quantities, as opposed to the four hardware breakpoints provided by an x86-based processor’s debug registers[1]. The term “virtual breakpoint” is being used as a programming, or debugging, analogy for an Intel VT-based VM-Exit, or exit reason. VM exits allow for the ability to filter the execution flow of a large number of x86-based events and commands. The ability to filter a specific command or event, as well as providing and trapping to a handler for this filtered command is comparable to the concept of a debugger breakpoint, where virtual memory and port I/O addresses are filtered out and forced to pass execution control to an interrupt handler. When filtering and trapping to a VM-Exit handler based on a specified command or event, even more power and control is available than with a traditional debug breakpoint. This is because the command or event can be handled before the actual execution occurs, the state of the guest system can be retrieved without direct interaction against the guest system's resources, and the option exists to prevent the command or event from ever occurring. A list of exit conditions that an Intel VMX-based VMM can handle is listed in Appendix A –Basic Exit Reasons. Also, the CPU state of the guest VM is under complete control of the VMM, and can be monitored and/or modified on every VM-Exit.

1.3.2 Applications of Enhanced Debugging Capabilities

The aforementioned enhancements can greatly improve the capabilities that a debugger can provide. Some improvements that could be provided to a debugger with the use of these enhancements are described in this section.

One of the novel capabilities that can be created as a result of the combination of VMX and Kernel debugging technologies is an ability to isolate areas of interest in code by intelligently filtering out specific data during the process of dynamic debugging. Kernel debugging can be extremely time consuming and tedious, especially when little is known about the module being debugged. This is partially a result of the amount of time that can be wasted searching through code to find a problem or piece of functionality. This issue is especially problematic in advanced operating systems, where a great deal of execution almost constantly takes place due to the large amount of code being executed.

Through the application of the research performed under this thesis, specific sets of virtual breakpoints can be tailored to target particular functionality within large sets of code. Previously, the memory or I/O location would already need to be known, or could require a person to manually step through and analyze large amounts of undesired code and data. The potential now exists to greatly decrease the amount of time it takes to debug through large amounts of code by applying more advanced virtual breakpoints targeting specific functionality.

Environments exist where it would be extremely difficult to perform debug analysis with traditional debugging facilities. The enhanced debugging capabilities that can be provided through the use of a VMM can greatly ease dynamic debugging analysis on many of these environments. This may include the ability to dynamically analyze system crashes, error states, and kernel debugger programs.

Errors that would typically crash or reboot a program or system generally leave behind some static debug information or nothing at all. With the capabilities provided by this research, the debugger has the potential to dynamically analyze the state of the system all the way into an error state. This will dramatically simplify the process of determining the cause of a crash. For example, a VM-Exit condition can be set to trap on exceptions. Exceptions are often generated due to error conditions or undetermined behavior, such as a divide by zero error. With the ability to trap to a handler within a VMM immediately upon generation of an exception, the code location that caused the exception can be located, as well as a full potential analysis of the system state. Another VM-Exit reason that can be handled through the use of Intel VT is a triple fault. A triple fault is generated when an x86-based CPU attempts to invoke an exception handler that invokes a fault itself. An x86-based system forces a reset or shutdown to occur when a triple fault occurs, making this extremely difficult to debug with traditional means. With Intel VT extensions, this error state can now be trapped on, demonstrating exactly where the system crashed, instantly providing the debugger with dynamic analysis opportunities. It may also be possible to recover from a crash without fully rebooting the entire system.

1.3.3 Additional Applications and Experiments

The technology presented within this thesis is not limited to the specific experiments and examples that are overviewed here. I am currently, and will continue to, extend the research performed under this thesis by completing additional experiments and attempting to discover new applications of this technology. Some of the experiments and applications that I'm currently working through, and planning to work through in the future, will be listed as well, in section 5.2 - Future Work.

1.4 Thesis Layout

This section provides an overview regarding the layout of this thesis. Each section will be listed below with a short description of the information contained in that section.

1.4.1 Chapter 1 – Introduction

Chapter 1 provides a high-level overview of the thesis topic, touches on the contributions that have been discovered through this research, and identifies related research. The overview gives a high-level description of the thesis topic and important related technologies. The contributions section explains how the research performed under this thesis will contribute to the current state-of-the-art within the community. The related research section will overview similar research that is publically available and explain how the research is related to this thesis topic.

1.4.2 Chapter 2 – Technology Background

Chapter 2 gives an overview of both kernel debugging and virtualization technologies, as these two technologies construct the foundation for this thesis. A section is devoted to each technology, giving both an analysis of the technology and an overview of how this technology is currently being used in the public.

1.4.3 Chapter 3 – Enhanced Debug Capabilities

Chapter 3 details the core contributions presented by this thesis, concentrating on the idea of providing an enhanced set of virtual breakpoints to a kernel debugger through the use of a hardware-assisted VMM. Some of the capabilities provided by this concept are overviewed, and demonstrated through a series of experiments. The experiments each identify a specific functional enhancement that can be provided to a kernel debugger, giving an overview of the experiment, its implementation details, and an explanation of the experiment's results.

1.4.4 Chapter 4 – Applications of Enhanced Debug Capabilities

Chapter 4 concentrates on some of the applications of the technology presented in this thesis that can be obtained through the use of the aforementioned enhanced debugging capabilities. The first broad set of applications is focused on filtering out large sets of unwanted data during dynamic analysis. The second broad category concentrates on debugging through challenging environments such as program or system error states.

1.4.5 Chapter 5 – Conclusion

Chapter 5 will review the key research contributions that were presented within this thesis. This will help to summarize the most relevant information that was detailed throughout this paper. In addition, a section will be given in the thesis that outlines the future work that will follow the completion of this thesis.

2. Chapter 2 – Technology Background

2.1 Virtualization Technology

In the area of computer science and information technology, virtualization is the ability to abstract a system from its underlying resources. This essentially means to create a virtual version, rather than an actual version of an entity or resource. Often, a host control program called a hypervisor, or virtual machine monitor, provides the virtualization layer for guest software to run within.

According to the Popek and Goldberg virtualization requirements [2], a true hypervisor must provide:

- Equivalence and Fidelity
- Resource Control and Safety
- Efficiency and Performance

Equivalence and fidelity ensure that software running under the control of a hypervisor experiences identical behavior to that of running on physical hardware. Resource control and safety requires that a hypervisor remain in absolute control of virtualized resources. To provide efficiency and performance, the majority of machine instructions must execute without hypervisor interruption.

Some well-defined types of virtualization include[3]:

- Hardware Virtualization
- Software Virtualization
- Memory Virtualization

- Storage Virtualization

Applications and methods of virtualization are used under a great deal of circumstances, some of which will be discussed in Chapter 2 – Technology Background. Every possible use is not covered, but many common applications are listed below.

2.1.1 Types of Virtualization

2.1.1.1 Hardware Virtualization

Also referred to as platform virtualization, this type of virtualization technology utilizes a host software program to simulate a computer, or hardware environment, for guest software to run on. The guest software can be an entire operating system, as the hypervisor is essentially providing a simulated version of a set of system hardware. The guest software will most likely not detect that it is not running on true hardware. The host software program responsible for simulating a hardware environment is commonly referred to as a hypervisor or virtual machine monitor.

Sub categories of hardware virtualization include hardware-assisted virtualization, full virtualization, partial virtualization, and paravirtualization.

2.1.1.1.1 Hardware-Assisted Virtualization

It is important to note that hardware virtualization and hardware-assisted virtualization are not synonymous. Hardware-assisted virtualization requires that a physical hardware component provide architectural support to aid in the development of a hypervisor. This support is commonly provided through CPU extensions on modern processors. In 2005 and 2006 both Intel and AMD independently provided virtualization processor extensions in some of their product lines. Intel’s hardware-assisted virtualization technology was given the name Intel VT-x, while AMD’s was given the name AMD-V. Both are examples of modern hardware-assisted virtualization technologies.

Hardware-assisted virtualization reduces resources required for development and maintenance of virtualization technologies as it practically eliminates the requirement for changes to be made to guest operating systems. This characteristic also provides for a more robust and extensible technology since little integration is required for a specific set of software. Performance increases can also be enjoyed because the virtualization extensions exist in hardware, reducing much of the software execution overhead. The main disadvantage of hardware-assisted virtualization is that the hardware extensions do not exist within every processor line, requiring a user to obtain specific hardware to utilize this functionality. Also, different vendors, such as AMD and Intel, use different specifications for their extensions, meaning the different hypervisors would need to be implemented for variant hardware.

2.1.1.1.2 Use of Intel's Hardware Assisted VT by this Thesis

The research performed under this thesis specifically makes use of many of the processor extensions provided by Intel's VT-x [1]. Intel VT-x is an umbrella term for Intel's suite of virtualization technologies. It is important to note that only a sub-set of these extensions are utilized for debugging purposes. For example, only a portion of the Intel VT-x exit reasons and architectural features are leveraged. Also, Intel's VT-d (Intel Virtualization Technology for directed I/O) [26] is not yet leveraged by this thesis, although this could provide additional useful future work for further enhancing debugging capabilities related to analysis of hardware and I/O. This is because Intel VT-d adds extra virtualization capabilities to the standard Intel VT based capabilities by allowing for I/O device assignment, DMA remapping, interrupt remapping, and the ability to catch DMA and interrupt errors that could have corrupted system memory or impacted VM isolation.

2.1.1.1.3 Full Virtualization

Full virtualization[4] simulates an entire hardware environment for guest software, such as an operating system, to run on. All system elements that are required by software and provided by a physical machine must be simulated. With full virtualization, software that is capable of running on system hardware, such as an operating system, should be able to run identically within the virtual machine. Although many close attempts have been made by programs, such as VMware, true full virtualization was not easily obtainable on x86 systems until the recent introduction of both Intel’s and AMD’s hardware-assisted virtualization extensions. The reason is that every action performed within a virtual machine must not have an effect on anything outside of the virtual machine, including other virtual machines, the hypervisor, or the physical hardware, but must still maintain identical functionality. To account for instructions that could alter overall machine state, such as privileged instructions, a form of translation often occurs to replace the privileged instructions with a safe set of instructions that emulated the same task. With instruction translation, the exact same output cannot feasibly be guaranteed for any set of input. For example, virtualization of hardware I/O would be very difficult to translate identical output for a complex device.

2.1.1.1.4 Partial Virtualization

Partial virtualization[6] simulates part of a hardware environment, allowing for some software to be run without modification. As opposed to full virtualization, an operating system may not be able to run in a partial virtualization environment because the full hardware environment is not simulated. Partial virtualization can be tailored to run specific software, by simulating all of the resources required by that specific set of software. A common application of a partially virtualized environment is to provide address space virtualization for the purpose of separating memory resources between separate users, or virtual machines. For this case, each virtual machine would receive its own set of memory, all of which are contained within the system's overall physical memory resource.

2.1.1.1.5 Paravirtualization

Paravirtualization[7][8] allows for multiple software instances, such as operating systems, to run within the same set of hardware by placing each set of guest software in its own domain, rather than simulating the system's hardware. Guest software will need to be modified to run within a paravirtualization environment. This modification requires that guest software, such as an operating system, be hooked so that certain events can be redirected to the paravirtualization environment to make use of its custom API.

Paravirtualization often provides better performance than many full virtualization environments because every element in hardware is not emulated. The drawbacks associated with this approach include a loss of flexibility and security. Flexibility is lost because the guest operating system requires specific modifications for compatibility with a paravirtualization environment. As a result, certain operating systems may not support a paravirtualized environment. Security can be negatively impacted by paravirtualization by allowing the guest operating system to run closer to hardware. If the software modifications made to a guest operating system did not successfully isolate all of the hardware, the guest operating system could have an effect on other guests through modification to the system hardware. For example, if a guest operating system was able to write to the host system's entire physical memory range, it could overwrite memory state of other guest operating systems.

2.1.1.2 Software Virtualization

Software virtualization typically refers to using an operating system kernel to isolate or simulate resources for application level guest software components. Multiple guest instances can be simultaneously running inside of the same operating system and its corresponding kernel.

2.1.1.2.1 Operating System Level Virtualization

For operating system level virtualization[9], an operating system kernel will host multiple user-mode guest software instances. Each guest software instance will be isolated from other guest instances so that they cannot modify each other's resources, or those of the host operating system kernel. Operating System level virtualization is often used by servers to host multiple distrusted guests. Security is provided through the isolation of each guest's set of resources.

This type of virtualization suffers very little performance impact because it does not require simulation or emulation of underlying resources. A major disadvantage associated with operating system level virtualization is the fact that it is inflexible since guest software must be built to run under the same operating system as the host kernel.

2.1.1.2.2 Application Virtualization

Application virtualization[10] makes use of encapsulation to separate applications from operating system resources. An operating system resource is often replaced with a virtual resource for a specific application to interface. All attempts from an application to access a specified system resource will be caught by the virtual environment and directed to the virtual resource that has been reserved for that application. For example, newer versions of the Microsoft Windows operating system have virtualized unprivileged calls to the system registry to enable the use of legacy programs that required registry access with user privileges. This was performed to provide security protection to the Windows registry without disabling the use of legacy programs.

2.1.1.3 Memory Virtualization

Memory virtualization[11][12] is used to combine memory as a shared resource between multiple individual systems into a distributed memory pool available for use as a shared resource by one or more systems. Memory virtualization can also be used to extend memory beyond a physical system's capacity and to allow for shared memory between multiple systems. By extending a system's memory capacity through a combined use model, a higher degree of performance can be achieved. For example, with a larger pool of memory, large-scale memory intensive operations can be performed without writing data out to disk, which is a common performance bottleneck. By allowing for the memory of one physical system or server to be shared between multiple systems, less hardware can be required to perform numerous tasks.

2.1.1.3.1 Virtual Memory

Virtual memory[1] is commonly used to give software the impression that it has a set of contiguous usable memory, rather than the physical set of fragmented memory. This can be accomplished by using a mechanism called paging. Paging is a process that maps a system's physical address space to virtual addresses. More virtual addresses can be available than physical addresses, allowing memory to be allocated more flexibly.

2.1.1.4 Storage Virtualization

Storage virtualization[13][14] provides the ability to abstract virtual storage from physical storage so that data can be accessed without regard for physical storage space on a hardware level. This type of virtualization provides the opportunity for both a distributed file system and to pool storage space from multiple devices into a common pool. A distributed file system allows multiple users access to a shared set of files located within one storage media. Pooled storage can provide one large set of storage space distributed over multiple physical devices, without requiring the user to face the complexities of manually distributing large amounts of data across these devices since the storage pool is virtualized into one common resource.

2.1.1.5 Emulation

Emulation[15] allows for one device to imitate a separate program or another device so that the resulting behavior appears identical to the original device being imitated. Both hardware and software environments can be emulated. Using an emulator, a set of hardware or an operating system can run software that it was not designed to run. Emulation can allow hardware and software to run programs that would otherwise be incompatible with a hardware or software-based environment, save the cost of purchasing hardware when it can be emulated, and allow for the use of legacy devices or applications.

2.2 Kernel Debugging

Applications and methods of kernel debugging are used for a variety of purposes. A list of many common kernel debugging applications is provided below, as well as a list of some publically available kernel debuggers, and also, some implementation details explaining how a developer would go about creating a kernel debugging application.

Kernel debugger's are commonly used to provide the ability to dynamically step through a low-level representation of an OS kernel, as a high-level representation may not be available. This is especially true at runtime of a program or device driver module where source code is not available. If symbols are available, a higher-level representation of the kernel code can be debugged. Microprocessors, such as those adhering to the x86 architecture, provide low-level dynamic debugging support through CPU design. More specifically, Intel x86-based processors offer a set of debug registers that present a user with up to four hardware breakpoints to aid in trapping on I/O or memory addresses. In an x86-based system, hardware breakpoints trap execution flow to interrupt vector 0x01 for handling by a debugger program, while software breakpoints (triggered by the 0xCC op-code) trap execution flow to interrupt vector 0x03.

A very common application of a kernel debugger is to aid in the development and bug fixing process of operating systems and device drivers. Dynamic software analysis of code that directly controls primary system resources can be extremely difficult to approach. This is especially true through static analysis, with a nearly unlimited set of potential system states in a complex system. This can become even more complicated where the software meets the hardware and all interactions do not provide predictable outcomes. A kernel debugger can also be used to perform analysis of an unknown program where source code is not available.

2.2.1 Hardware Breakpoints

The following six steps provide an overview of the functionality required to implement a hardware breakpoint based debugging capability. A kernel debugging capability was developed under an independent study preceding this thesis. Portions of the code from this independent study have been leveraged in this section[16] to help describe the process of developing a hardware breakpoint capability using Intel debug registers on an x86-based processor.

Step 1 – Set Breakpoint Conditions

Breakpoint conditions determine which type of operation will cause the debugger to trap to its breakpoint handler. The options exist to breakpoint on Instruction Execution, Data Writes (only), Port I/O reads or writes (x86 IN and OUT instructions), and both data reads and writes. The following code example demonstrates these four options and displays the values that each option is associated with.

```
#define DE_INST_EXEC 0x00 //- Break on Instruction Execution Only
#define DE_DATA_WRITES 0x01 //- Break on Data Writes Only
#define DE_PORT_IO_RW 0x02 //- Break on I/O reads or writes
#define DE_RW_NO_IF 0x03 //- Break on Data Reads or Writes but
not Instruction
```

The values representing each breakpoint condition must be set by masking and unmasking corresponding bits in DR7 (Debug Register 7).

Step 2 – Set Breakpoint addresses in debug registers

The address that a breakpoint is used to monitor must be set in DR0-DR3 (Intel Debug Registers 0, 1, 2, and 3). The address can be either a virtual memory address, or a port I/O address (This is distinguished by the condition that was set in Step 1). Each of the four debug registers can correspond to a different breakpoint, allowing up to four hardware breakpoints.

Step 3 – Enable / Disable Debugging Extensions

The DE (Debugging Extensions) flag is controlled by a bit in CR4 (Intel Control Register 4) and is used to control how the CPU interprets breakpoint condition configurations. The different breakpoint condition configuration interpretations are listed below based on whether or not the DE flag is set in CR4.

<u>Breakpoint Condition Configurations</u>

WHEN DE FLAG SET

00_2 - Break on Instruction Execution Only

01_2 - Break on Data Writes Only

10_2 - Break on I/O reads or writes (Meaning Port IO - INs and OUTs)

11_2 - Break on Data Reads or Writes but not Instruction Fetches

WHEN DE FLAG NOT SET

00_2 - Break on Instruction Execution Only

01_2 - Break on Data Writes Only

10_2 - Undefined

11_2 - Break on Data Reads or Writes but not Instruction Fetches

Step 4 – Set Breakpoint Length

The length that is monitored starting from the breakpoint address that was set in one of the debug register (DR0-DR3) is specified by setting a breakpoint length condition field within DR7 (debug register 7). The four possible breakpoint lengths are 1 byte, 2 bytes, 4 bytes, and potentially 8 bytes in some processor families. These lengths are represented by the following data combinations, which are written into DR7:

Breakpoint Length Configurations

00_2 - 1 Byte Length

01_2 - 2 Byte Length

10_2 - Undefined (or 8 byte length for some families)

11_2 - 4 Byte Length

Step 5 – Enable / Disable a Breakpoint

Once a breakpoint condition has been set, the breakpoint address has been indicated, the DE flag is enabled, and the length of the breakpoint is chosen, the breakpoint can be either enabled or disabled. A breakpoint will only trap to its handler if it is set to “enabled”. Also, a total of four breakpoints can independently be enabled or disabled. The breakpoint is enabled or disabled by toggling a bit that is located in DR7 (debug register 7).

Step 6 – Hook Interrupt 0x01 and use for breakpoint handler

Hardware breakpoints, such as those mentioned in the above sections, will transfer execution to interrupt handler 0x01 when triggered. Interrupt 0x01 must be hooked in order to provide control over a hardware breakpoint. This can be accomplished by determining the virtual address of the IDT (interrupt descriptor table) using the x86 SIDT command, and modifying the pointer to the location of the interrupt 0x01 handler in the IDT structure. Once modified, interrupt 0x01 must point to code that was placed in memory to handle a breakpoint.

2.2.2 Software Breakpoints

A brief overview detailing the generation of a software breakpoint will be provided in this section. Software breakpoints are much simpler to develop than hardware breakpoints, and therefore, will not require as much detail to describe, compared with a hardware breakpoint.

The x86 architecture provides a specific opcode, 0xCC, which will generate a software breakpoint when executed. To set a software breakpoint, the first byte of an instruction is replaced with a 0xCC, as opposed to a hardware breakpoint which does not require modifications to the the code being debugged since it is using the hardware support. When a software breakpoint is triggered through the execution of the 0xCC opcode, control flow is trapped to interrupt 0x03 to be handled by debugging software. The debugger is responsible for saving the value of the byte of the instruction that was replaced by 0xCC, and for subsequently restoring this value once the 0xCC causes a breakpoint to trigger. Unlike a hardware breakpoint, which is limited to four simultaneous breakpoints, an unlimited number of software breakpoints can be placed in code at any time. Another major difference between a software breakpoint and a hardware breakpoint is that a software breakpoint can only be set on a memory address in executable code, whereas a hardware breakpoint can be used to monitor code execution, read/write access to a memory address, and execution of a port I/O command. A software breakpoint can only be used on executable code because it needs to be executed itself to cause the instruction pointer to trap to interrupt 0x03.

2.2.3 Examples of Debugger Applications

A famous, but obsolete, x86-based kernel debugger was named SoftICE. Soft-ICE is believed to be named as a reference to a “Software In-Circuit Emulator”. SoftICE was a kernel debugger that was designed for the Microsoft Windows operating system, although older versions exist for DOS. Developed by NuMega, and later acquired by Compuware, SoftICE was a very prominent kernel debugger from the late 80’s through the early 2000’s. A major advantage that SoftICE had over other Microsoft Windows kernel debuggers such as WinDBG was the ability to halt the entire operating system and provide debugging services without the use of a second machine, with an included DOS style graphical user interface. Other typical kernel debuggers required the use of two machines dedicated to the debugging process. The more modern versions of SoftICE have been documented to make patches in the Microsoft Windows Kernel to acquire control of the operating system when in use, and appear to make use of Intel processor debug registers to obtain its core functionality.

A modern example of a kernel debugger, that provides a similar interface to SoftICE, is Syser Kernel Debugger. Syser Debugger is developed by Sysersoft for the Windows NT family of operating systems. Syser Debugger appears to make use of Intel debug registers for its core functionality, as well as wrapping itself into the Windows operating system (most likely with a series of hooks). Like SoftICE, Syser Debugger is also capable of debugging with only a single machine available. The graphical user interface has been developed to look and function very similar to that of SoftICE, but appears to be more robust and stable. One of the major problems that SoftICE encountered was hooking into graphics drivers for its GUI. This was a problem because graphics drivers can vary widely depending on the vendor and model of the video card that is present, and very rarely release a developer's specification – leading to proprietary device drivers. Syser Debugger has fixed this problem by utilizing a technique where it hooks into Microsoft Windows Direct X API calls to generate its graphical user interface. Multi-core support and advanced memory searching features are also provided by Syser Debugger.

Finally, an example of another very widely used kernel debugger is Microsoft's WinDbg. WinDbg provides the ability to debug both user mode and kernel mode applications in Microsoft Windows NT-based operating systems. Although WinDbg provides a powerful Windows-based graphical user interface, it typically requires the use of two computers to harness its full set if capabilities. Despite this restriction, newer versions of WinDbg allow for single machine kernel debugging capabilities, but with a subset of its capabilities. A somewhat unique and very helpful feature of WinDbg is its ability to debug kernel mode memory dumps that are created when a bug check is issued. Another unique feature of WinDbg is its ability to load sets of windows debugging symbols from a debug server to tailor a debugging session to a more specific set of circumstances such as a private source code library.

3. Chapter 3 – Enhanced Debugging Capabilities

As mentioned in chapter 1, a VMM can be used to provide an enhanced set of debugging features, in addition to those already provided by standard software and hardware based debug breakpoint functionality. This is accomplished by using a VMM's ability to trap on specific system-level events to implement precise breakpoints. These breakpoints can be tailored to more specific conditions than traditional hardware and software breakpoints that must either specify a memory or port I/O address. With the addition of these enhanced capabilities, a debugger can be implemented to provide more fine-grained control over the software being debugged.

To help explain and demonstrate these concepts, a map of chapter 3 is as follows:

- 3.1 - Traditional Debugging Capabilities: Basic information about standard debug capabilities
- 3.2 - Enhanced Debugging Capabilities: Debugging enhancements introduced in this section
- 3.3 - Exit Reasons / Breakpoint Functionality: Brief overview of some relevant exit reasons that could aid in the development of virtual breakpoints
- 3.4 - Experiments: Detailed analysis of the enhanced debugging capability experiments
- 3.5 - Conclusions of Enhanced Debugging Capabilities: A summary of the key concepts and results associated with this chapter

3.1 Traditional Debugging Capabilities

Traditional debuggers provide support to set breakpoints on either a memory address or a port I/O address. Up to four hardware breakpoints can be set within an x86-based CPU without making any modifications to the target software. A practically unlimited number of software breakpoints can be placed, on memory addresses only, by replacing the first byte of an instruction with a 0xCC opcode. This special opcode causes execution flow to transfer to the system's interrupt 0x03 handler. The interrupt 0x03 handler, most likely set up by the debugger, is responsible for restoring the correct data in memory for the instruction that was at least partially replaced by the 0xCC opcode so that the instruction can still be executed correctly.

These standard debugger capabilities are useful, but do not provide a means to configure the conditions that will cause a breakpoint to a degree of control beyond memory or Port I/O addresses. In addition, hardware breakpoints are very limited, only allowing for four simultaneous breakpoints to be set.

An Independent study, completed prior to this thesis, has been referenced[16] and provides a detailed explanation of the implementation of a hardware-based kernel debugger. A portion of this independent study details the steps required to create a debugger backend, and steps through this process with functional source code. Enough detail is included to allow developers to implement and configure their own kernel debugger breakpoints by directly programming the CPU.

3.2 Enhanced Debugging Capabilities

By using a VMM's exit reason functionality to implement a set of virtual break points, very fine-grained control can be specified over conditions that cause a debugger to trap to its breakpoint handler. For the purpose of this thesis, this control can be as specific as the architectural conditions that trigger exit reasons, as defined by Intel's VT. Once an Intel VT-based VMM has been set up, configured, and launched, any of the architectural events, listed in Appendix A –Basic Exit Reasons, can trigger the guest operating system to exit and pass control to the host VMM.

Virtual breakpoints can be set and used through the implementation of an exit reason handler placed within the host VMM. When the host VMM gains execution control due to a guest VM-Exit, it is essentially allowing the software controlling the host VMM to trap on a specified condition and execute a handler. This concept is analogous to that of a debug breakpoint; the difference being the exit handler is controlled and executed from a VMM rather than a standard Ring 0 program.

3.2.1 Enhancement 1 - Setting Virtual Breakpoints

The first major enhancement that this provides to our VMM-based debugging capability is the ability to set a breakpoint and trap to our debug handler for a multitude of specific events, rather than just the memory and port I/O events that a standard debugger traps to. This allows for more specific system-level configuration of breakpoints. Each of these events has been explained in further detail below in Section 3.3 - Exit Reasons / Breakpoint Functionality. It is also important to note that standard memory and port I/O breakpoints can still be set in addition to the virtual breakpoints detailed in this thesis. This enhancement is detailed below in Section 3.4.1 - Experiment 1 – Demonstration of virtual breakpoints.

3.2.2 Enhancement 2 - Unlimited Virtual Breakpoints

The next enhancement provided by this debugging capability is the ability to set nearly unlimited breakpoints rather than being limited to four hardware-based memory and port I/O breakpoints or using software breakpoints that modify the target code and can only trap on specified memory addresses. A nearly unlimited number of breakpoints can be set based on any event that can trigger an exit condition. Multiple breakpoints can even be set for any specific type of exit condition by using a routing algorithm that specifies for certain conditions in addition to a specific exit condition to trigger a breakpoint. For example, an exit handler can specify any number of port I/O reads or writes that will trigger a virtual breakpoint when an I/O instruction exit occurs, spanning far beyond the limit of four I/O breakpoints allowed by standard processor debug registers. An overview of this enhancement can be found below in Section 3.4.2 - Experiment 2 – Implementation of more than four hardware breakpoints.

3.2.3 Enhancement 3 - Full Control over Guest State

Another enhancement that can be provided to a debugger using a VMM-based debugging facility is the full control that the debugger will gain over the target OS. Intel's hardware-assisted virtualization technology is inherently built to maintain full control over the system state of the guest OS (target OS for our debugger). In many cases this control can supersede that of a standard kernel debugger, allowing for easier viewing and modification of the target's system state. For example, a VMM can easily read and modify the instruction pointer (EIP register) of a guest OS, whereas this poses a complicated task when running in the kernel. This concept has been demonstrated below in Section 3.4.3 - Experiment 3 – Full Control over Guest OS System State.

3.3 Exit Reasons / Breakpoint Functionality

This section will detail the functionality associated with some of the VMX exit reasons that can be used as a condition to trigger a virtual breakpoint. Intel's VMX-based VMM technology currently supports fifty-five exit reasons, each of which can be used as a condition to cause a virtual breakpoint. Some of these exit reasons are overviewed below along with a description of what will cause this exit reason to occur and how it can be used as a virtual breakpoint. These exit reasons have been provided so that the reader can better understand the functionality associated with various exit reasons and the similarity that these exits can have to a breakpoint.

Exception:

Exceptions are often generated when an instruction causes an error, or a special event such as a hardware breakpoint occurs. An example of an error that causes an exception is a divide by zero error. An exit condition is generated when guest software causes an exception to occur. With the ability to trap on exceptions, a virtual breakpoint can be configured to trap on types of system errors and special events.

Triple Fault:

A triple fault is experienced when an exception is generated while attempting to call a double fault handler. A double fault occurs if the CPU experiences a problem while handling an interrupt or exception. This type of exit reason can be used to configure a virtual breakpoint to trap on the occurrence of a triple fault. This has the potential to be a very powerful capability since triple faults often force the machine to reboot, making debugging through a triple fault a difficult challenge.

CPUID Command:

An exit reason can be configured to occur on the execution of the CPUID instruction. The CPUID instruction can be used to obtain processor specific information, such as to determine if various CPU features are supported by the current processor. This type of exit condition has the potential to create a virtual breakpoint within a program that will trap when software is attempting to profile the CPU before making critical decisions based on CPU capabilities.

Control Register Access:

Control registers are used to configure specific processor settings such as a processor's mode of execution, the availability of physical address extensions, enabling paging, or setting the base address of the system page table. Any access to a control register can be configured to cause an exit reason. This provides the potential for a virtual breakpoint to be tailored to specific changes in system state, such as a transition from real to protected mode.

MOV DR (move data into a debug register):

Debug registers are used to configure and enable hardware breakpoints that will cause an exception when a specified memory or port I/O address is accessed. By writing data to debug registers, a VM-Exit can occur. This will enable a virtual breakpoint to trap on software that is configuring or enabling/disabling a hardware breakpoint, which provides the opportunity to debug a debugger, or to debug software that interacts with a standard debugger's hardware breakpoint capability.

I/O Instruction (i.e. IN or OUT Commands):

I/O instructions are used to pass data between software and a specified hardware component. Any I/O address can be configured to cause a VM-Exit by either setting all I/O access to cause a VM-Exit, or by specifying a set of ports that will trigger VM-Exits in the hypervisors IO Bitmap VMCS field. A standard hardware breakpoint can set up to four simultaneous I/O breakpoints. This capability can be used to set virtual breakpoints to simultaneously monitor more than four I/O ports, in combination with other types of breakpoints.

RDMSR/WRMSR:

The RDMSR/WRMSR commands are used to read or write data to an MSR (Model Specific Register). MSRs are used to control processor features that are available to specific lines of processors. Any read or write access to an MSR can be handled by a VM-Exit. This capability will help to configure virtual breakpoints that can be tailored to break on any specific functionality associated with an MSR.

IDTR:

The IDTR is used to store the base physical address and length of the IDT. Any access to the IDTR can be configured to cause a VM-Exit. With the ability to monitor access to the IDTR, a virtual breakpoint can be configured to help identify when any software attempts to place a hook in the IDT.

3.4 Experiments

This section will describe the experiments that were conducted in order to demonstrate the concepts overviewed throughout Chapter 3 – Enhanced Debugging Capabilities. Three categories of experiments were performed as follows:

- Demonstration of Virtual Breakpoints
- Implementation of more than four hardware breakpoints
- Fast and easy retrieval of processor system state

A summary of each experiment will be provided in order to overview the experiment's intent. This will be followed by the details required to set up and run the experiment, including implementation details. Finally, the results of each experiment will be provided to demonstrate the feasibility of the enhanced debugger capabilities presented in this paper.

3.4.1 Experiment 1 – Demonstration of virtual breakpoints

Experiment 1 Overview:

This experiment will demonstrate the ability to set and trap on virtual breakpoints. This capability is the foundation of this thesis, as it provides a technique to trap to a software handler when specified conditions are met, similar to a debugger trapping to an interrupt handler when a specified memory or port I/O address is read, written, or executed. Any of the fifty-five exit reasons listed in Appendix A –Basic Exit Reasons can be used as a virtual breakpoint, but because the concept is the same, only a few of these virtual breakpoints will be demonstrated as part of this experiment. For this experiment, the following virtual breakpoint conditions will be set and trapped on by our VMM's virtual breakpoint handler function:

- Port I/O address 0x0CF8 Read or Write - A virtual breakpoint will be set to trap on reads or writes to port I/O address 0x0CF8. Port 0x0CF8 is a standard port used to index into PCI configuration space, where information can be retrieved and set within various I/O devices. This virtual breakpoint demonstrates the ability to trap on both reads or writes to a specified port I/O address.

- Read/Write MSR (Model Specific Register) Execution – One more virtual breakpoint will be demonstrated that traps on accesses to specified Model Specific Registers. MSRs allow for interactions with various processor capabilities. For example, MSR 0x00000010 is used to read or reset the processor's time stamp counter. This virtual breakpoint will be set to trap on reads of the time stamp counter MSR.

- VMX Instruction Execution – Another virtual breakpoint will be set to break on any VMX instructions that are executed. When Intel's VMX instruction set is used to bring up and make use of a VMM, various VMX instructions must be executed. The debugging capabilities presented in this thesis will be capable of monitoring VMX activity on a target system. Without this capability, debugging a target could experience complications if a Intel VT-based VMM were loaded. This is because a VMM can maintain control over system resources such as processor capabilities, physical memory, interrupt management, and various I/O interactions, effectively trumping the control that a standard debugger could maintain over the processor. It is important to note that if a VMM was loaded before the debugger, it could prevent the debugger from correctly loading its VMM component.

- Control Register access – The virtual breakpoint handler will also be set to trap on Control Register 4 access. The processor's Control Registers are responsible for controlling various features of the CPU. More specifically, control registers manage options such as paging, processor modes, some cache settings, and enabling/disabling certain instruction sets, such as VMX instructions. This virtual breakpoint demonstrates the ability to trap on control registers, and also to specify which control registers will trigger the breakpoint. It will also be demonstrated that specific settings within the control register can be used to trigger a virtual breakpoint, rather than just general read/write access.

Experiment 1 Implementation:

Experiment 1 loaded a VMM and performed configurations to trap, or set virtual breakpoints, on access to I/O port 0x0CF8, the RDMSR instruction, the VMXON command, and access to the CPU's Control Register 4. The source code shown below in Figure 4 - Chapter 3 - Experiment 1 Source Code demonstrates the implementation of the handler that was included within the VMM to handle the aforementioned exit conditions that were configured within the VMM. These handlers can also include code to pause execution, communicate with a user interface, and allow a user to take actions against the target machine's execution, such as stepping through the code or analyzing the system state.

The port I/O virtual breakpoint has been specially configured within the VMM to set up and enable an I/O bitmap that is responsible for determining exactly what port I/O accesses cause VM-Exits to trap to the VMM. An I/O bitmap was used rather than setting all port I/O accesses to cause a VM-Exit, and implementing a filter, because the latter induces large amounts of processing overhead since port I/O instructions execute frequently within a standard system.

This was accomplished by enabling the “Use I/O bitmaps” bit within the IA32_VMX_PROCBASED_CTLS MSR that must be modified from within a VMM. The I/O bitmap consists of two 4KB aligned memory, and is filled with data representing which port accesses will cause a VM-Exit. Each bit within the memory range represents a port, where a value of 1 indicates that the port will cause a VM-Exit when used, and a value of 0 indicates that port accesses will execute as normal. Once the I/O bitmaps are allocated and configured, the VMCS must be provided with the 64-bit physical address of each I/O bitmap.

The port I/O handler, shown below in Figure 4 - Chapter 3 - Experiment 1 Source Code, prints to a debug interface when a specified I/O instruction is encountered and proceeds to execute the instruction. For this experiment, I/O port 0x0CF8 is set as a virtual breakpoint. Once a specified I/O instruction triggers our handler, the VMM’s Exit Qualification can be parsed to determine specific information about the port I/O instruction that was executed. The exit qualification information is retrieved from the VMCS at offset 0x6400 using the VMREAD command.

The RDMSR Virtual Breakpoint has been configured within the VMM to trap on RDMSR instructions. The handler can be used to ignore certain MSR accesses, and to trap on

MSR accesses specified by a virtual breakpoint. The handler used in our experiment, shown below in Figure 4 - Chapter 3 - Experiment 1 Source Code, currently prints out the MSR that was accessed by the RDMSR instruction. The purpose of this was to demonstrate that the code can easily determine which MSR was read, and filter this data within the handler based on the MSR address specified by the virtual breakpoint. The handler next proceeds to execute the RDMSR instruction as intended on the correct MSR address so that the output can be returned to the guest VM. An MSR bitmap could also be used to determine which MSRs cause exit conditions, but was not necessary for this experiment because the frequency of MSR accesses in a standard system is much lower. An MSR bitmap can be enabled by enabling the “Use MSR Bitmaps” bit within the IA32_VMX_PROCBASED_CTLS MSR. Once this option is enabled, a bitmap structure in memory is used the same way that the aforementioned I/O bitmap was used to specify which port I/O addresses caused VM-Exits.

The VMX Instruction Virtual Breakpoint was configured to specifically trap on execution of the VMXON command. The handler in our experiment was used to provide debug output indicating that the VMXON instruction was called, and to prevent the instruction from executing since a VMM was already loaded. A virtual breakpoint can easily be sent for each VMX-based instruction as each VMX instruction has its own exit reason.

```
//////////////////////////////////////////////////////////////
//Chapter 3 - Experiment 1                                  //
//Set virtual breakpoints on:                               //
//     -Port I/O address 0x0CF8 read or write    -0x0000001E //
//     -RDMSR Execution                          -0x00000031 //
//     -VMX Instruction (VMXON Command)          -0x00000027 //
//////////////////////////////////////////////////////////////
```

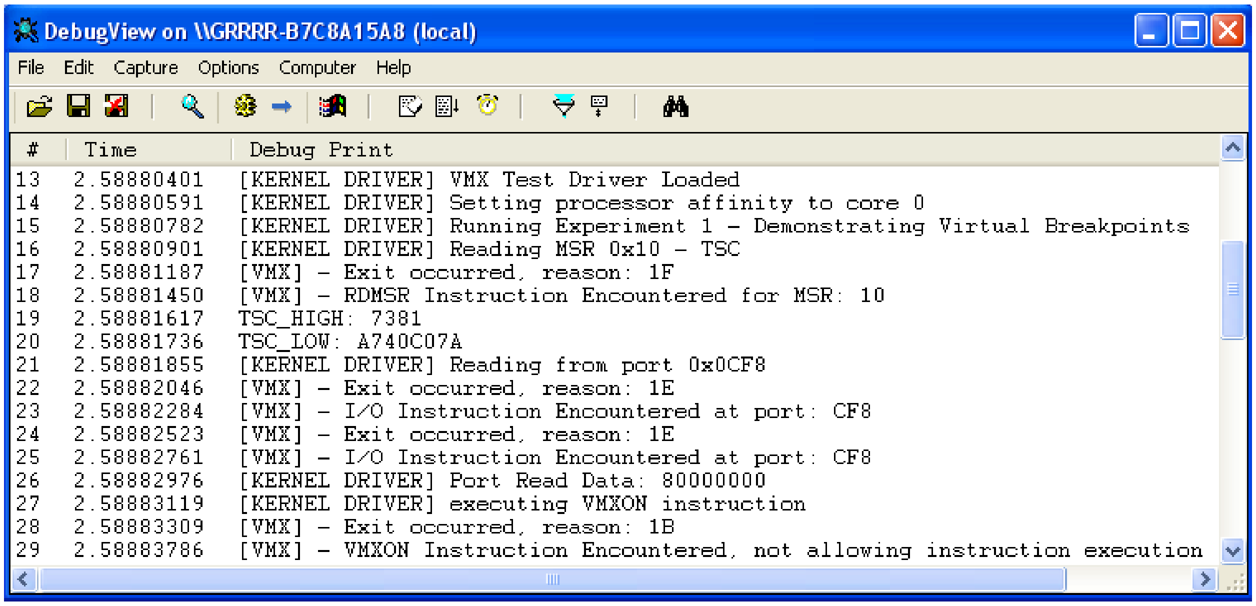
```
//Exit Reason Handler if an I/O instruction that has been set in
the I/O bitmap is executed
if( ExitReason == 0x0000001E )
{
      DbgPrint("[VMX] - I/O Instruction Encountered at port: %X",
      ExitQualification >> 16);

      //Execute the Port I/O instruction
      __asm
      {
            popad

            mov edx, ExitQualification
            shr edx, 16 //Determine port number that caused exit

                  //Determine the direction of the I/O access
                  mov eax, ExitQualification
                  and eax, 0x00000008 //Bit 3 determines
                  direction - 1 for IN, 0 for OUT

                  cmp eax, 0
                  jz OUT_INS

                  //Determine the size of the I/O access
                  mov eax, ExitQualification
                  and eax, 0x00000003

                  //Execute the appropriate instruction based on
                  size
                  cmp eax, 0
                  JE IN8 //8-bit IN
                  cmp eax, 1
                  JE IN16 //16-bit IN
                  JMP IN32 //32-bit IN

                  IN8:
                  in al, dx

                  IN16:
                  in ax, dx

                  IN32:
                  in eax, dx

                  jmp Resume

                  //Branch here if the I/O instruction direction
                  is OUT
                  OUT_INS:

                  //Determine the size of the I/O access
                  mov eax, ExitQualification
                  and eax, 0x00000003

                  //Execute the appropriate instruction based on
size
                  cmp eax, 0
            je OUT8 //8-bit OUT
            cmp eax, 1
            je OUT16 //16-bit OUT
            jmp OUT32 //32-bit OUT

            OUT8:
            mov eax, GuestEAX
            out dx, al

            OUT16:
            mov eax, GuestEAX
            out dx, ax

            OUT32:
            mov eax, GuestEAX
            out dx, eax

            jmp Resume
      }
}



//Exit reason handler for an RSMSR instruction execution
if( ExitReason == 0x0000001F )
{
      DbgPrint("[VMX] - RDMSR Instruction Encountered for
      MSR: %X", GuestECX);
      __asm
      {
            popad
            mov         ecx, GuestECX
            rdmsr
            jmp Resume
      }
}

//Exit reason handler for a VMXON instruction execution
if( ExitReason == 0x0000001B )
{
      DbgPrint("[VMX] - VMXON Instruction Encountered, not
      allowing instruction execution");
      __asm
      {
            popad
            jmp Resume
      }
}
```

Figure 4 - Chapter 3 - Experiment 1 Source Code

<u>Experiment 1 Results</u>:

The output from our experiment 1 virtual breakpoint handler has been recorded and shown below in Figure 5 – Chapter 3 - Experiment 1 Results (Debug Output), for each virtual breakpoint experiment. A Windows device driver was used to execute code that triggers the aforementioned virtual breakpoints. This kernel code is shown in Appendix C – Kernel Code used in Experiments, Figure 18 - Kernel code to trigger virtual breakpoint experiments.

The debug output shown directly below displays the output that was printed when the exit reason handler code from Figure 4 - Chapter 3 - Experiment 1 Source Code was triggered by our kernel module code shown in Appendix C – Kernel Code used in Experiments, Figure 18 - Kernel code to trigger virtual breakpoint experiments. These results demonstrate that a VMM was effectively used to create breakpoints on specific system-level functionality including port I/O access, MSR access, VMX instruction execution, and control register access. As previously mentioned, a virtual breakpoint can be set for any of the 55 exit reasons listed in Appendix A –Basic Exit Reasons.


DebugView on \\GRRRR-B7C8A15A8 (local)
File Edit Capture Options Computer Help

```
#   Time         Debug Print
13  2.58880401   [KERNEL DRIVER] VMX Test Driver Loaded
14  2.58880591   [KERNEL DRIVER] Setting processor affinity to core 0
15  2.58880782   [KERNEL DRIVER] Running Experiment 1 - Demonstrating Virtual Breakpoints
16  2.58880901   [KERNEL DRIVER] Reading MSR 0x10 - TSC
17  2.58881187   [VMX] - Exit occurred, reason: 1F
18  2.58881450   [VMX] - RDMSR Instruction Encountered for MSR: 10
19  2.58881617   TSC_HIGH: 7381
20  2.58881736   TSC_LOW: A740C07A
21  2.58881855   [KERNEL DRIVER] Reading from port 0x0CF8
22  2.58882046   [VMX] - Exit occurred, reason: 1E
23  2.58882284   [VMX] - I/O Instruction Encountered at port: CF8
24  2.58882523   [VMX] - Exit occurred, reason: 1E
25  2.58882761   [VMX] - I/O Instruction Encountered at port: CF8
26  2.58882976   [KERNEL DRIVER] Port Read Data: 80000000
27  2.58883119   [KERNEL DRIVER] executing VMXON instruction
28  2.58883309   [VMX] - Exit occurred, reason: 1B
29  2.58883786   [VMX] - VMXON Instruction Encountered, not allowing instruction execution
```

Figure 5 – Chapter 3 - Experiment 1 Results (Debug Output)

The code shown in Appendix C – Kernel Code used in Experiments, Figure 18 - Kernel code to trigger virtual breakpoint experiments, was used to cause our virtual breakpoints to trigger and trap to our VMM handler code, as previously mentioned. The first section of code reads data in from port 0x0CF8 and triggers our virtual breakpoint that is monitoring access to port 0x0CF8. The second section of code uses MSR 0x00000010 to read the current value of the system's time stamp counter, and triggers our virtual breakpoint that is monitoring read access to MSRs. The third section of code executes a VMXON instruction to trigger the virtual breakpoint that is specifically monitoring the VMXON x86 instruction.

3.4.2 Experiment 2 – Implementation of more than four hardware breakpoints

Experiment 2 Overview:

Experiment two will demonstrate the ability to concurrently set well over four virtual breakpoints. Any number of virtual breakpoints, of both the same and different types, can be set concurrently. To compare this capability directly to breakpoints that can be set by a standard debugger, port I/O breakpoints will be used. Intel processors currently support four hardware breakpoints, which are required to break on port I/O access. This experiment will simultaneously demonstrate ten virtual port I/O breakpoints and demonstrate the ability to trap to a handler for each of these breakpoints. It is important to note that the ten virtual breakpoints set in this experiment does not represent a limit, but is being used to avoid needlessly repeating the same experiment. A virtual breakpoint can feasibly be set for every single port I/O address, as well as concurrently setting additional virtual breakpoints for other purposes. This experiment will also concurrently set a breakpoint to trap on access to MSR 0x00000010 to demonstrate this concept.

Experiment 2 Implementation:

Experiment 2 loaded and configured a VMM to trap on access to I/O ports 0x00-0x60, as well as on reads or writes to MSRs. MSR 0x10, which provides access to the system's Time Stamp Counter, was specifically chosen to be filtered to represent a breakpoint applied to MSR 0x10. I/O ports 0x00-0x60 were chosen to demonstrate the ability to set a great deal of simultaneous port I/O VBPs.

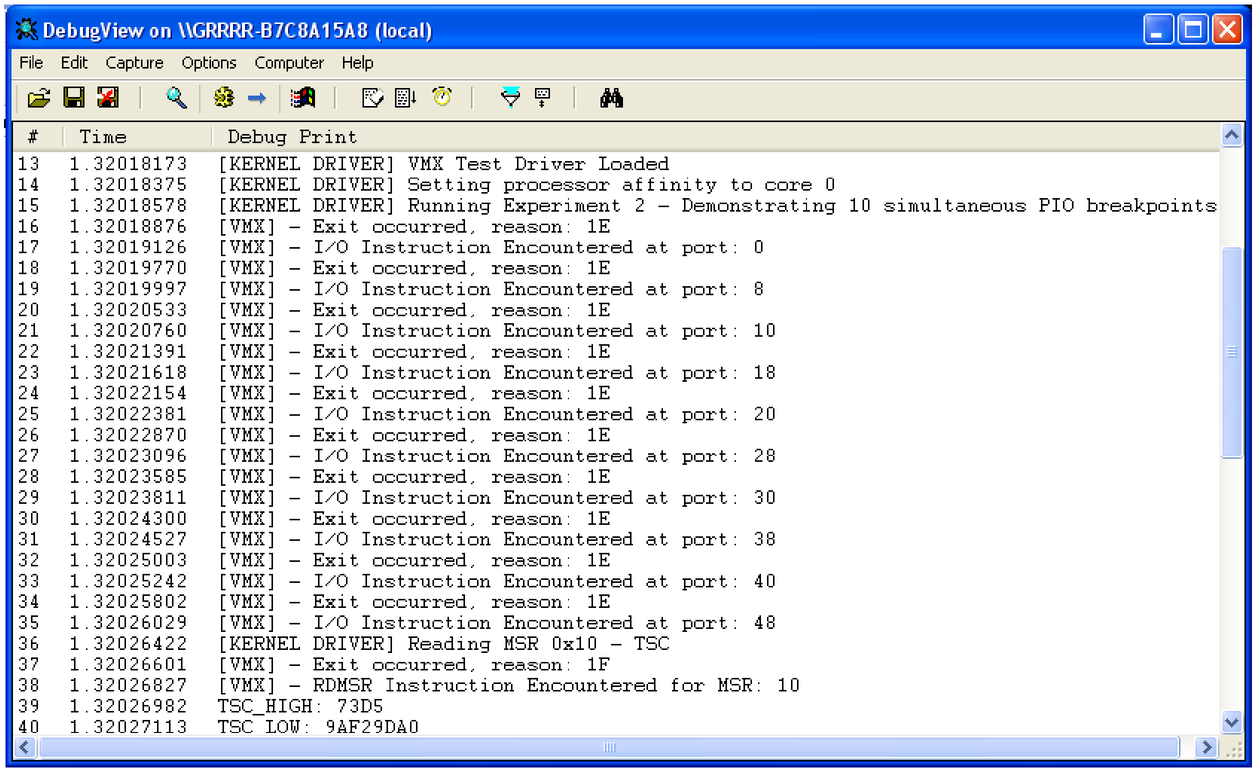
The I/O Port VBPs were enabled by setting the corresponding bits within the I/O bitmap that was described in detail under experiment one. The MSR 0x10 VBP was enabled by disabling the "Use MSR Bitmap" bit within the IA32_VMX_PROCBASED_CTLS MSR, and then filtering MSR exit conditions to trap on accesses to MSR 0x10. Through the use of basic filter functionality in software, accesses to all other MSRs can be ignored. If different MSRs were called frequently on a specific system, producing significant execution overhead, a debugger implementation could enable the aforementioned "Use MSR Bitmap" to specify exactly which MSRs cause exit conditions.

This experiment chose to use a software filter rather than an MSR bitmap to demonstrate an alternative method of specifying VBPs as opposed to that used by the port I/O VBP implementation. It was also feasible to use this simpler technique for MSR breakpoints because the amount of MSR calls being made will unlikely cause significant execution overhead due to an abundance of VM-Exits.

<u>Experiment 2 Results</u>:

The output from the experiment 2 virtual breakpoint handler has been recorded and shown in Figure 6 - Chapter 3 - Experiment 2 Results (Debug Output). This code demonstrates the large number of breakpoints that were concurrently set and trapped to. Similar to experiment 1, a Windows device driver was used to execute code that triggered the virtual breakpoints set in experiment 2. This kernel code is shown in Appendix C – Kernel Code used in Experiments, Figure 19 - Kernel code to trigger multiple simultaneous virtual breakpoints experiment.

The debug output shown directly below displays the output that was printed when the VMM's exit reason handler was triggered for the virtual break points set for experiment 2, by the kernel code shown in Appendix C – Kernel Code used in Experiments, Figure 19 - Kernel code to trigger multiple simultaneous virtual breakpoints experiment. These results demonstrate that a great deal of port I/O breakpoints can simultaneously be set, providing many more than a standard debugger. This experiment also demonstrated that we are not limited to port I/O based virtual breakpoints, as an MSR breakpoint was also concurrently set and trapped to with this experiment. There is no specific limit on the number or type of virtual breakpoints that can simultaneously be set.


DebugView on \\GRRRR-B7C8A15A8 (local)
File Edit Capture Options Computer Help

```
#   Time        Debug Print
13  1.32018173  [KERNEL DRIVER] VMX Test Driver Loaded
14  1.32018375  [KERNEL DRIVER] Setting processor affinity to core 0
15  1.32018578  [KERNEL DRIVER] Running Experiment 2 - Demonstrating 10 simultaneous PIO breakpoints
16  1.32018876  [VMX] - Exit occurred, reason: 1E
17  1.32019126  [VMX] - I/O Instruction Encountered at port: 0
18  1.32019770  [VMX] - Exit occurred, reason: 1E
19  1.32019997  [VMX] - I/O Instruction Encountered at port: 8
20  1.32020533  [VMX] - Exit occurred, reason: 1E
21  1.32020760  [VMX] - I/O Instruction Encountered at port: 10
22  1.32021391  [VMX] - Exit occurred, reason: 1E
23  1.32021618  [VMX] - I/O Instruction Encountered at port: 18
24  1.32022154  [VMX] - Exit occurred, reason: 1E
25  1.32022381  [VMX] - I/O Instruction Encountered at port: 20
26  1.32022870  [VMX] - Exit occurred, reason: 1E
27  1.32023096  [VMX] - I/O Instruction Encountered at port: 28
28  1.32023585  [VMX] - Exit occurred, reason: 1E
29  1.32023811  [VMX] - I/O Instruction Encountered at port: 30
30  1.32024300  [VMX] - Exit occurred, reason: 1E
31  1.32024527  [VMX] - I/O Instruction Encountered at port: 38
32  1.32025003  [VMX] - Exit occurred, reason: 1E
33  1.32025242  [VMX] - I/O Instruction Encountered at port: 40
34  1.32025802  [VMX] - Exit occurred, reason: 1E
35  1.32026029  [VMX] - I/O Instruction Encountered at port: 48
36  1.32026422  [KERNEL DRIVER] Reading MSR 0x10 - TSC
37  1.32026601  [VMX] - Exit occurred, reason: 1F
38  1.32026827  [VMX] - RDMSR Instruction Encountered for MSR: 10
39  1.32026982  TSC_HIGH: 73D5
40  1.32027113  TSC_LOW: 9AF29DA0
```

Figure 6 - Chapter 3 - Experiment 2 Results (Debug Output)

The kernel code shown in Appendix C – Kernel Code used in Experiments, Figure 19 - Kernel code to trigger multiple simultaneous virtual breakpoints experiment was used to demonstrate that well over 4 virtual breakpoints are simultaneously set on port I/O addresses, as well as additional virtual breakpoints of other types. The first section of code reads from ten different port I/O addresses within the range of 0x60 port I/O addresses that are currently enabled as virtual breakpoints. This code starts at port 0x0000 and increments by 8 bytes for each consecutive port I/O read. This was done to demonstrate that each port read would have represented a completely different breakpoint for a standard kernel debugger since each port I/O breakpoint can only span

4 bytes. The second section of code read from MSR 0x10 to demonstrate the virtual breakpoint set on MSR 0x10.

3.4.3 Experiment 3 – Full Control over Guest OS System State

Experiment 3 Overview:

Even though the debugging capabilities detailed in this thesis are running within a VMM, in a separate mode than the Guest OS, the software running within the VMM still has full control over the Guest OS. In many cases running software from within a VMM provides better control over the OS state that if the code were run from within the OS's own kernel. As a result, the debugging capabilities presented within this thesis can still read and modify the system state, as does a standard debugger, but often with less effort required. For example the instruction pointer (EIP register) cannot normally be directly modified, even when the code is executing with Ring 0 privileges within the kernel. An indirect trick must be used to modify the value of the instruction pointer from Ring 0, for example, by altering the return address of a function call on the stack and calling a ret instruction to pop the modified return address into the EIP register. Code running in the VMM, such as the debugging software presented in this thesis, has full control over the system state. As a result, the debugging capabilities presented within this thesis can easily read and modify guest OS state. Standard x86 processor registers such as the general purpose registers EAX, EBX, ECX, and EDX can be modified within the VMM using a MOV or POP instruction, and will retain their value when returning to the guest. Other processor registers can be modified within the VMM by writing their desired value to the corresponding guest state fields within the VMCS using the x86 VMWRITE command. The non-standard registers include control registers, debug registers, the stack pointer register, the instruction pointer register, the flags register, the task register, the GDT register, the LDT register, and the IDT register, some specific MSRs, and the SMBASE register[17].

Experiment 3 will demonstrate the ability to easily modify the value of the instruction pointer (the EIP register) in the guest OS, from code executing within the VMM. The purpose of this experiment is to show that the debugging capabilities presented in this thesis can easily modify the state of the target OS, even for registers that would normally be very complicated to alter. This is an important feature for any kernel debugger as the user may desire to view or alter various aspects of the system during testing or analysis. To demonstrate the modification of the EIP register, when a specified case within the VMM is reached, the VMM will increment the value of the EIP register within the guest OS by altering its value using the VMCS.

Experiment 3 Implementation:

Experiment 3 was performed by adding code to our VMM that examines each exit condition for a specific set of circumstances to hold true, and then increments the instruction pointer by 12. It would be trivial for a debugger running in the virtual machine monitor to alter the instruction pointer based on user input, but a full interface is not the main focus of this thesis, so code is used to demonstrate this capability as with the other experiments. The source code shown in Figure 7 - Instruction Pointer Modification in VMM demonstrates how we altered the instruction pointer before returning to the guest OS. The GuestResumeEIP variable contains the address that the guest OS instruction pointer will receive when the VMM returns control to the guest OS after it is trapped to as the result of an exit condition being triggered. Normally the VMM is responsible for incrementing the instruction pointer by the length of the instruction that triggered an exit from the guest OS into the VMM. This ensures that the same instruction in the guest OS in not infinitely repeated. If the instruction pointer was not incremented before returning to the guest OS, the same instruction would get executed, causing the same exit reason to occur repeatedly. Once the GuestResumeEIP variable is calculated and updated, it is written to the VMCS location responsible for restoring the instruction pointer upon re-entry into the guest OS. To alter the instruction pointer, the code shown below simply adds an addition 12 bytes before writing the value to the VMCS. The conditions used to trigger this experiment include a port I/O access (exit reason 0x1E) to I/O port 0x01.

It is important to note that for a debugger implementation, when the debugger has reached a virtual breakpoint and has halted the system to the point that the user can observe and modify system state, it will have passed control to the VMM, making the aforementioned concept feasible in implementation.

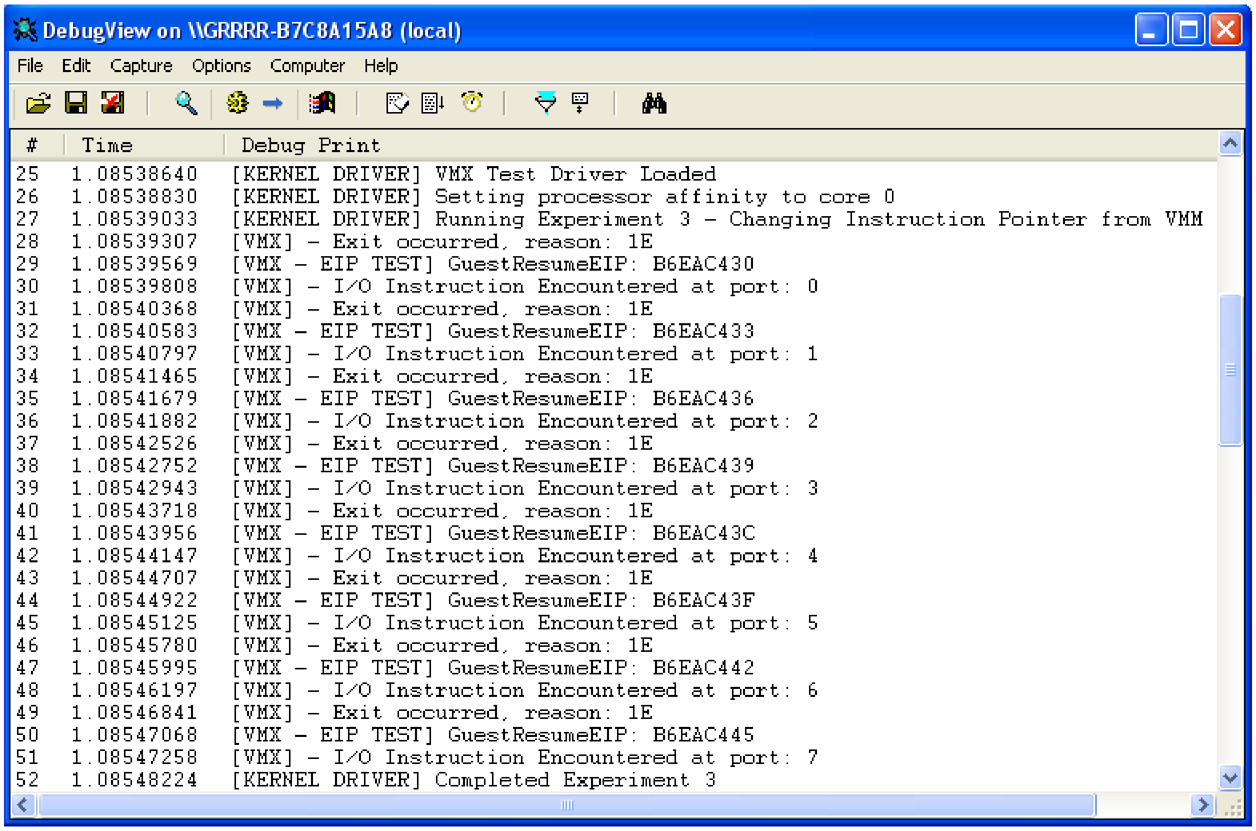
```
GuestResumeEIP = GuestEIP + ExitInstructionLength;

//Chapter 3 Experiment 3 - Modify instruction pointer. Add 12 to
instruction pointer to skip instructions under specified
condition
if( (ExitReason == 0x1E) && ( (ExitQualification >> 16) == 1) )
{
	DbgPrint("[VMX - EIP TEST] Specified EIP Condition
	detected, incrementing EIP by 12");

	GuestResumeEIP = GuestResumeEIP + 12;
}

WriteVMCS( 0x0000681E, (ULONG)GuestResumeEIP );
```

Figure 7 - Instruction Pointer Modification in VMM

Figure 20 - Kernel Code Used to Observe EIP Modification, found in Appendix C – Kernel Code used in Experiments, consists of the kernel code that was used to help demonstrate the modification of the instruction pointer from within the VMM. The code includes 8 sets of port read instructions that increment the port being read after each instruction completes. Each set of instructions consists of three bytes of machine code, 0xEC, 0x66, and 0x42. Therefore, after the execution of each corresponding IN and INC instruction, the instruction pointer will have incremented by three bytes. This code starts out by reading port 0, and finished by reading port 7. A virtual breakpoint is set for each of the 7 ports when this code is executed for this experiment. This code is run twice, once to trigger the virtual breakpoints for each I/O instruction, and again to trigger the virtual breakpoints as well as to modify the instruction pointer when port 0x01 triggers a breakpoint. The instruction pointer is incremented by 12 bytes, to skip over 4 sets of port I/O instructions. The results of this experiment are detailed below in the results section.

<u>**Experiment 3 Results**</u>**:**

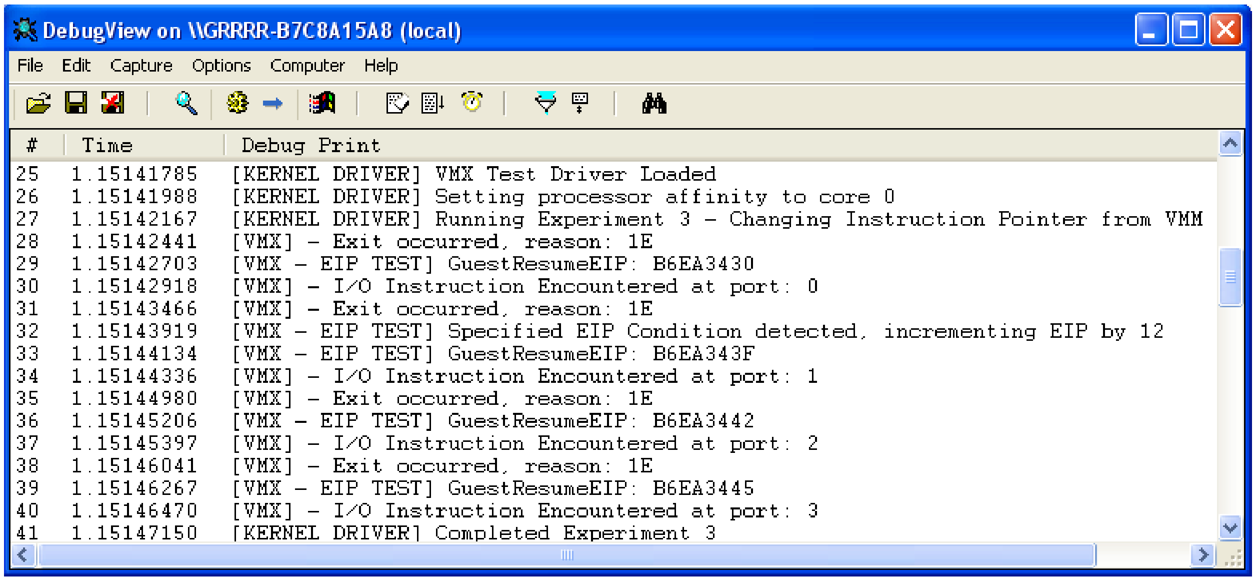
Two screen shots are provided below to demonstrate the results of this experiment; Figure 8 - Sequence of I/O Instructions Run in Kernel without EIP Modification and Figure 9 – Sequence of I/O Instructions Run in Kernel with EIP Modification from VMM. The first of the two figures shows the resulting output from the execution of our aforementioned kernel code that executed 8 port I/O instructions, incrementing the port number after the execution of each instruction. It is important to remember that virtual breakpoints were set on each I/O instruction. The virtual memory address representing the instruction pointer after each port I/O instruction is printed out after the completion of each of the I/O instructions. It is clearly demonstrated that the execution of each IN and INC instruction increments the instruction pointer by 3 bytes, since the machine code for these two instructions adds up to a length of 3 bytes. Each I/O instruction is executed as expected, triggering a virtual breakpoint that prints out some EIP and exit reason data.


DebugView on \\GRRRR-B7C8A15A8 (local)
File Edit Capture Options Computer Help

```
#   Time        Debug Print
25  1.08538640  [KERNEL DRIVER] VMX Test Driver Loaded
26  1.08538830  [KERNEL DRIVER] Setting processor affinity to core 0
27  1.08539033  [KERNEL DRIVER] Running Experiment 3 - Changing Instruction Pointer from VMM
28  1.08539307  [VMX] - Exit occurred, reason: 1E
29  1.08539569  [VMX - EIP TEST] GuestResumeEIP: B6EAC430
30  1.08539808  [VMX] - I/O Instruction Encountered at port: 0
31  1.08540368  [VMX] - Exit occurred, reason: 1E
32  1.08540583  [VMX - EIP TEST] GuestResumeEIP: B6EAC433
33  1.08540797  [VMX] - I/O Instruction Encountered at port: 1
34  1.08541465  [VMX] - Exit occurred, reason: 1E
35  1.08541679  [VMX - EIP TEST] GuestResumeEIP: B6EAC436
36  1.08541882  [VMX] - I/O Instruction Encountered at port: 2
37  1.08542526  [VMX] - Exit occurred, reason: 1E
38  1.08542752  [VMX - EIP TEST] GuestResumeEIP: B6EAC439
39  1.08542943  [VMX] - I/O Instruction Encountered at port: 3
40  1.08543718  [VMX] - Exit occurred, reason: 1E
41  1.08543956  [VMX - EIP TEST] GuestResumeEIP: B6EAC43C
42  1.08544147  [VMX] - I/O Instruction Encountered at port: 4
43  1.08544707  [VMX] - Exit occurred, reason: 1E
44  1.08544922  [VMX - EIP TEST] GuestResumeEIP: B6EAC43F
45  1.08545125  [VMX] - I/O Instruction Encountered at port: 5
46  1.08545780  [VMX] - Exit occurred, reason: 1E
47  1.08545995  [VMX - EIP TEST] GuestResumeEIP: B6EAC442
48  1.08546197  [VMX] - I/O Instruction Encountered at port: 6
49  1.08546841  [VMX] - Exit occurred, reason: 1E
50  1.08547068  [VMX - EIP TEST] GuestResumeEIP: B6EAC445
51  1.08547258  [VMX] - I/O Instruction Encountered at port: 7
52  1.08548224  [KERNEL DRIVER] Completed Experiment 3
```

Figure 8 - Sequence of I/O Instructions Run in Kernel without EIP Modification

The next figure, Figure 9 – Sequence of I/O Instructions Run in Kernel with EIP Modification from VMM, provides output from the same kernel code execution, but after we have included the instruction pointer modification code into the VMM as shown above in Figure 7 - Instruction Pointer Modification in VMM. The resulting output clearly shows that the virtual breakpoint triggered by the I/O port 0x01 access caused the instruction pointer to increment by 12, skipping four of the port I/O instructions and demonstrating code execution at an instruction pointer 12 bytes past where the kernel code would expect.


DebugView on \\GRRRR-B7C8A15A8 (local)
File Edit Capture Options Computer Help

```
#   Time         Debug Print
25  1.15141785   [KERNEL DRIVER] VMX Test Driver Loaded
26  1.15141988   [KERNEL DRIVER] Setting processor affinity to core 0
27  1.15142167   [KERNEL DRIVER] Running Experiment 3 - Changing Instruction Pointer from VMM
28  1.15142441   [VMX] - Exit occurred, reason: 1E
29  1.15142703   [VMX - EIP TEST] GuestResumeEIP: B6EA3430
30  1.15142918   [VMX] - I/O Instruction Encountered at port: 0
31  1.15143466   [VMX] - Exit occurred, reason: 1E
32  1.15143919   [VMX - EIP TEST] Specified EIP Condition detected, incrementing EIP by 12
33  1.15144134   [VMX - EIP TEST] GuestResumeEIP: B6EA343F
34  1.15144336   [VMX] - I/O Instruction Encountered at port: 1
35  1.15144980   [VMX] - Exit occurred, reason: 1E
36  1.15145206   [VMX - EIP TEST] GuestResumeEIP: B6EA3442
37  1.15145397   [VMX] - I/O Instruction Encountered at port: 2
38  1.15146041   [VMX] - Exit occurred, reason: 1E
39  1.15146267   [VMX - EIP TEST] GuestResumeEIP: B6EA3445
40  1.15146470   [VMX] - I/O Instruction Encountered at port: 3
41  1.15147150   [KERNEL DRIVER] Completed Experiment 3
```

Figure 9 – Sequence of I/O Instructions Run in Kernel with EIP Modification from VMM

This demonstration has proven the ability of our debugging capabilities to easily view and modify the system state of a guest OS being debugged. This access and modification to the guest OS provides a greater degree of control than a standard kernel debugger running within the OS's kernel due to its privileged control of the target and ease of access/control over the processor's registers.

3.5 Conclusions of Enhanced Debugging Capabilities

This section has demonstrated the fundamental ideas of this thesis; that analogies between a VMM's exit conditions and a kernel debugger's breakpoint capabilities can be used to greatly enhance the current state of the art in kernel debugging on x86-based systems. This similarity has been used to enhance the capabilities that can be provided by a kernel debugger through the introduction of a plethora of additional breakpoint conditions, a greater number of hardware-based breakpoints, and a debug environment that has extremely granular control over the system being debugged. The experiments within this section clearly demonstrate these concepts and provide examples in order to prove the feasibility of each claim, as well as to help increase the reader's understanding of each concept. The first experiment, overviewed in section 3.4.1 - Experiment 1 – Demonstration of virtual breakpoints, created and triggered various types of virtual breakpoints and triggered these breakpoints with code executing in the kernel, which trapped to our virtual breakpoint handler. The second experiment, overviewed in section 3.4.2 - Experiment 2 – Implementation of more than four hardware breakpoints, demonstrated the ability to set a nearly unlimited number of virtual hardware breakpoints, as opposed to a standard x86-based processor's four hardware breakpoint limit. The third experiment, overviewed in section 3.4.3 - Experiment 3 – Full Control over Guest OS System State, demonstrated that our virtual debugger, operating from within a hardware-assisted VMM, has powerful and granular control over the target system being debugged. The newly provided enhancements presented within this chapter open up the opportunity for numerous applications of enhanced kernel debugging, which will be covered in the following chapter, Chapter 4 – Applications of Enhanced Debugging Capabilities.

4. Chapter 4 – Applications of Enhanced Debugging Capabilities

The enhanced kernel debugging capabilities that have been presented in chapter three provide the opportunity for numerous enhanced debugging applications. To demonstrate this concept, a few of the applications of this technology will be presented and overviewed throughout this chapter, along with step by step demonstration and analysis of each application that is presented.

To help explain and demonstrate these concepts, a map of chapter 4 is as follows:

- 4.1 - Applications of Enhanced Debugging Capabilities: How enhanced debugging capabilities can be applied in practice
- 4.2 - Background Information for Applications of Enhanced Debugging Experiments: Background information on the concepts required to understand the chapter 4 experiments
- 4.3 - Experiments: Detailed analysis of the applications of enhanced debugging capabilities experiments

4.1 Applications of Enhanced Debugging Capabilities

As indicated in section 1.3.2 - Applications of Enhanced Debugging Capabilities, the enhanced debugging capabilities presented in chapter 3 can be tailored and applied to aid a debugger under more specific circumstances. Two sets of circumstances will be overviewed and demonstrated within this chapter. In addition, extra examples will be discussed that I am currently implementing, or that I plan to implement in the future as time permits.

4.1.1 Application 1 - Filter Unwanted Data During Dynamic Analysis

A powerful application of the virtual breakpoint concept presented throughout this thesis is the ability to tailor these virtual breakpoints to trap under extremely specific conditions in order to filter out large sets of unwanted data during dynamic analysis of a system. It is often difficult to isolate a desired section of code during dynamic analysis of a complex system, especially if source code is not available. Often, a bug is being sought, but the location is unknown. Standard memory and port I/O breakpoints may not provide the functionality necessary to quickly isolate the functionality being sought after. The experiments presented in section 4.3.1 - Smart Filtering of Data During Dynamic Analysis will help demonstrate a few scenarios where this concept proves useful.

4.1.2 Application 2 - Debugging Exceptions and Error States

Another great application of the concepts presented in this thesis is the ability to easily debug through exceptions and error states. Bugs or errors that would have previously hindered the capability of a debugger, operating system, program being debugged, or required extremely complex algorithms to handle, can now be trapped on and dynamically analyzed with greater ease. In addition, the exact location of a bug can be determined and analyzed through a virtual breakpoint, with a minimal amount of time and effort. This can also be accomplished without the requirement to obliviously step through large amounts of code in search of the bug. The experiments presented in section 4.3.2 Analysis of Complex Debug Environments will help demonstrate a couple scenarios associated with these concepts.

4.2 Background Information for Applications of Enhanced Debugging Experiments

This section will provide background information on various architectural concepts that help to understand the experiments that are overviewed in section 4.3. The architectural concepts that will be overviewed include the following:

- Processor mode switch using Control Register 0
- Model Specific Registers: IA32_THERM_INTERRUPT and IA32_PAT
- A hardware breakpoint generated exception
- A divide by zero error

Control register 0 is responsible for setting various operating modes and states of the processor. Specific to this thesis, bits 0 and 31 of control register 0 are of interest, because these bits are used to configure paging and to set the processor's mode of execution to either real mode or protected mode[1]. By enabling paging, the processor translates physical addresses to virtual addresses. For each translation, a virtual address's cache type and access rights are also determined. Paging is enabled by setting bit 31 of control register 0 and disabled by clearing this bit. Paging can only be enabled when executing in protected mode, otherwise a general protection exception will occur. A processor's mode of execution is set to real mode if bit 0 of control register 0 is cleared, and set to protected mode if this bit is set to 1. When in real mode, the processor can address a 20-bit memory space and has direct access to physical memory. When in protected mode, a processor can address an extended memory range of 32 bits, paging is used for address translation, and other various capabilities such as ring-based privilege levels. Traditionally an x86-based system begins booting in real mode for compatibility purposes, and eventually switches into protected mode when an operating system such as Microsoft Windows is loaded. Addition modes of operation, such as, but not limited to, IA_32e mode can also be enabled with various processor configurations.

Model specific registers are provided to allow for specific functionality and various processor implementations. Many MSRs on Intel x86-based processors are supported across many lines of processors. MSRs that will not change on future processor generations are considered architectural MSRs, and are given the prefix "IA32_". The RDMSR x86 assembly command can be used to read from an MSR, while the WRMSR x86 assembly command can be used to write to an MSR. Two MSRs that will be overviewed this section, which are relevant to this section's experiments, include the IA32_PAT MSR and the IA32_THERM_INTERRUPT MSR. The IA32_PAT MSR contains eight fields, each of which is responsible for specifying a memory cache type. The cache types include uncacheable, write combining, write-through, write-protected, and write-back. Page table entries configured their corresponding memory pages cache type by selecting one of the fields within the PAT. The IA32_THERM_INTERRUPT contains bit settings that enable interrupts to be generated under specific temperature conditions. Multiple interrupts can be enabled to trip when specified processor temperature values are detected. These values are also programmed into and stored within the IA32_THERM_INTERRUPT.

Intel x86-based processors provide low-level dynamic debugging support through CPU design. More specifically, these processors supply a set of debug registers that present a user with up to four hardware breakpoints. These breakpoints can be set to monitor either a memory address or a port I/O address. When a specified breakpoint condition is encountered an exception is generated that traps execution flow to interrupt vector 0x01 for handling by a debugger program. The use of hardware breakpoints can be advantageous to software breakpoints, which modify part of the target instruction by replacing its first byte with the 0xCC opcode, because they do not modify the target software and can also be placed directly on port I/O addresses rather than just memory addresses.

For the x86 architecture certain error conditions cannot be handled, and will cause exceptions. When the DIV or IDIV assembly instructions are issued with a divisor of zero, a divide by zero error occurs. This causes a hardware-generated exception on the processor, exception zero, which passes control to an interrupt for handling.

4.3 Experiments

This section will describe the experiments that were conducted in order to demonstrate the concepts overviewed throughout Chapter 4 – Applications of Enhanced Debugging Capabilities. Two main categories of experiments included:

- Smart filtering of data during dynamic analysis
- Analysis of complex debug environments

A summary of each experiment will be given in order to overview the experiment’s intent. This will be followed by the details required to set up and run the experiment, including implementation details. Finally, the results of each experiment will be provided to demonstrate the specific application of the enhanced debugging capabilities presented in this thesis.

4.3.1 Smart Filtering of Data During Dynamic Analysis

This set of experiments demonstrates a couple cases where virtual breakpoints can be used to filter out large sets of unwanted data during dynamic analysis, and very quickly locate and analyze the specific section of code being sought after. These two experiments by no means represent the full scope of smart filtering capabilities that can be provided through the use of the virtual breakpoints overviewed in this thesis, but aim to help demonstrate how these virtual breakpoints can be used to isolate specific areas in code during dynamic analysis.

4.3.1.1 Experiment 1 - Processor Mode Switch

Experiment 1 Overview:

This experiment demonstrates the ability to trap on processor mode switches. More specifically, this experiment monitors for changes that would toggle the processor between protected mode and real mode. This experiment also monitors for changes to processor state that would enable or disable paging. Switching between real and protected mode, as well as enabling or disabling paging, will have a major effect on the system and the code running within the system. These changes could certainly cause side effects that produce errors in running code. The code used in this experiment specifically monitors for paging to be disabled from an enabled state, or for the protected mode enable bit in CR0 to be cleared to a setting representing real mode. Other settings provided by CR0 could also be monitored similarly via a virtual breakpoint.

These types of changes pose an enormous challenge to a debugger as it must anticipate the change in system state and adapt its own software to these changes so it does not crash. When enabling paging, code no longer has direct access to physical addresses including memory, and memory-mapped I/O. When switching from real mode to protected mode, 32 bits of memory is addressed instead of 20 bits, paging is enabled, segmentation occurs where available memory can be broken into separate segments of various configurations and attributes, as well as other settings such as the ring based security scheme to help enforce privileges of running code. With many challenges and issues associated with a mode switch, a virtual breakpoint that monitors for this behavior can isolate the event and provide the ability to analyze the system state immediately before the switch occurs, through the mode switch itself.

Experiment 1 Implementation:

Experiment 1 has set a virtual breakpoint to trap on write accesses to control register 0. This has been determined by using the exitQualification field to determine which control register was accessed, if it was a read or a write, and where the data is being written from or read to (so we can determine the data being applied to the register). Additional configurations have been set to monitor bits 0 and 31 of control register 0. The current state of the processor has these bits set to 1, as the guest OS is presently running Windows XP, which is running in protected mode with paging enabled. If either of these bits are cleared in the guest OS, the virtual breakpoint handler will print a debug message stating that these bits were accessed and cleared. The virtual breakpoint handler currently does not allow these bits to be cleared, and simply returns to the guest without allowing the modification to control register 0. The code used in this handler is shown below in Figure 10 - Chapter 4.3.1 - Processor Mode Switch Hypervisor Handler Code.

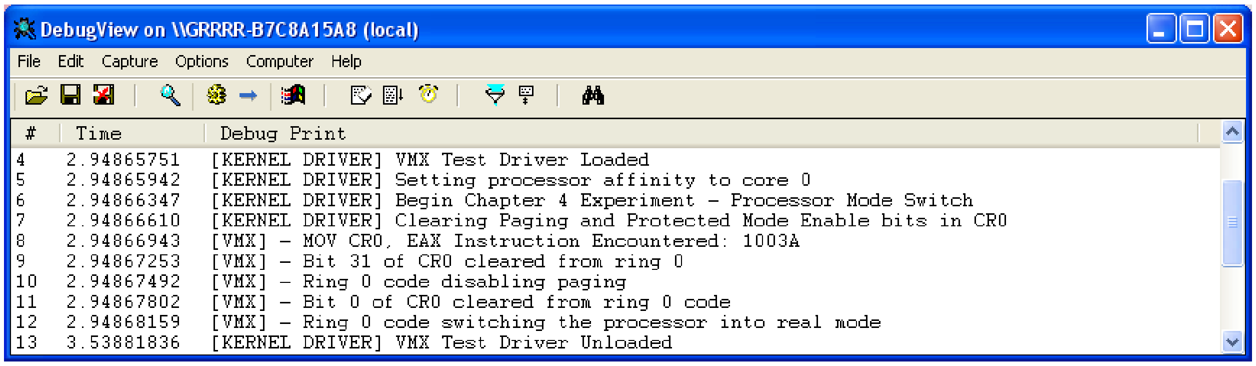
```
//Chapter 4 - Mode Switch Experiment
if( ExitReason == 0x0000001C) //Control Register Access
{
	//Which CR number was accessed
	movcrControlRegister = ( ExitQualification & 0x0000000F );

	//Access type - i.e. read or write
	movcrAccessType = ( ( ExitQualification & 0x00000030 ) >> 4
	);

	//Register or memory based access
	movcrOperandType = ( ( ExitQualification & 0x00000040 ) >>
	6 );

	//Which general purpose register involved
	movcrGeneralPurposeRegister = ( ( ExitQualification &
	0x00000F00  ) >> 8 );

//Check for Write Access to Control Register 0 from an AX based
register
//CR0 - Write - Register - EAX
if( (movcrControlRegister == 0) && (movcrAccessType == 0) &&
(movcrOperandType == 0) && (movcrGeneralPurposeRegister == 0) )
{
	DbgPrint("[VMX] - MOV CR0, EAX Instruction Encountered:
	%X", GuestEAX);

	if( (GuestEAX & 0x80000000) == 0)
	{
		DbgPrint("[VMX] - Bit 31 of CR0 cleared from ring
		0");

		DbgPrint("[VMX] - Ring 0 code disabling paging");
	}
	if( (GuestEAX & 0x00000001) == 0)
	{
		DbgPrint("[VMX] - Bit 0 of CR0 cleared from ring 0
		code");

		DbgPrint("[VMX] - Ring 0 code switching the processor
		into real mode");
	}

	__asm
	{
		popad
		jmp Resume
	}
}
```

Figure 10 - Chapter 4.3.1 - Processor Mode Switch Hypervisor Handler Code

The code shown in Appendix C – Kernel Code used in Experiments, Figure 21 - Kernel Code to Trigger Processor Mode Switch Breakpoint, was used to trigger the breakpoint that was set to trap on changes to the processor mode. This was accomplished by writing a value to CR0 where bits 0 and 31 are both cleared to 0. This would effectively put the processor into 16-bit real mode. Because this experiment is running in a 32-bit protected mode Microsoft Windows environment, the hypervisor does not allow for this change to take place, but acknowledges the attempted change through debug output.

<u>Experiment 1 Results</u>:

The handler used in this experiment successfully trapped on write accesses to the CR0 register. The virtual breakpoint handler in the hypervisor detected that bit 0 of CR0 was being cleared to disable paging and that bit 31 was being cleared to switch the processor into real mode. The debug output generated from the virtual breakpoint handler is shown below in Figure 11 - Output from Processor Mode Switch Breakpoint Experiment. As previously mentioned, the hypervisor was configured to disallow these changes to avoid crashing the guest operating system during this experiment. As with any virtual breakpoint handler, the exact code location that triggered this virtual breakpoint can always be determined since the hypervisor has direct access to the guest OS's instruction pointer value from when it caused an exit condition to occur. This experiment has demonstrated one application of a virtual breakpoint placed on a control register. Any specific bit within CR0 or another control register can just as easily be configured to trigger a virtual breakpoint. As with most virtual breakpoints, the applications can be tailored to a large set of scenarios.


DebugView on \\GRRRR-B7C8A15A8 (local)
File Edit Capture Options Computer Help

```
#   Time        Debug Print
4   2.94865751  [KERNEL DRIVER] VMX Test Driver Loaded
5   2.94865942  [KERNEL DRIVER] Setting processor affinity to core 0
6   2.94866347  [KERNEL DRIVER] Begin Chapter 4 Experiment - Processor Mode Switch
7   2.94866610  [KERNEL DRIVER] Clearing Paging and Protected Mode Enable bits in CR0
8   2.94866943  [VMX] - MOV CR0, EAX Instruction Encountered: 1003A
9   2.94867253  [VMX] - Bit 31 of CR0 cleared from ring 0
10  2.94867492  [VMX] - Ring 0 code disabling paging
11  2.94867802  [VMX] - Bit 0 of CR0 cleared from ring 0 code
12  2.94868159  [VMX] - Ring 0 code switching the processor into real mode
13  3.53881836  [KERNEL DRIVER] VMX Test Driver Unloaded
```

Figure 11 - Output from Processor Mode Switch Breakpoint Experiment

4.3.1.2 Experiment 2 - Specific MSR Functionality

Experiment 2 Overview:

This experiment has demonstrated the overarching ability to tailor virtual breakpoints to trap on functionality associated with accesses to specific MSRs. More specifically, the IA32_PAT MSR and the IA32_THERM_INTERRUPT MSR registers are monitored for write access during this experiment. As mentioned in Section 4.2 - Background Information for Applications of Enhanced Debugging Experiments, The IA32_PAT MSR is used to control cache types of memory ranges specified through entries in the page table and the IA32_THERM_INTERRUPT MSR is used to specify when an interrupt will be triggered under specific thermal conditions. Two scenarios have been overviewed where a code bug associated with either of these MSRs has occurred, and where it is extremely difficult to track down the specific code module that is responsible for this bug.

The first scenario relates to the ability of the IA32_PAT MSR to determine the cache type of various pages of memory. During the development of an operating system, this MSR may be configured to control the cache type associated with different memory ranges. Once set up by the OS, this register is not expected to experience modifications. If a code module in an OS boot loader or kernel were to contain a small bug where the ecx register incorrectly calculated an offset that ended up writing to this register, the effects may not be noticed immediately, but the OS would most likely become unstable or crash later on, when random pages of memory were configured to a different cache type than expected. The bug would be difficult to track down since the instability or crash might not occur until long after the misconfiguration has occurred since the affected memory pages may not be used immediately after. Even with the knowledge that the specific MSR was being misconfigured, dynamically tracking the erroneous code down could prove to be very time-consuming in a large operating system environment. Static analysis would be difficult since many MSRs could be written to, and the value of the ecx register would not be known for each MSR interaction without a dynamic view of the system state. With the ability to set a virtual breakpoint on the specific MSR in question, any attempts to access this register could be set to cause breakpoints. Upon triggering the breakpoint, the exact location of the code as well as the system state, including the ecx register, could be determined. Inspection would quickly demonstrate that the ecx register was incorrectly writing to the wrong MSR from a specified code module.

The second scenario covered in this experiment is similar to the first, where an MSR is misconfigured from the boot loader or kernel of an OS, but for this scenario the MSR in question is the IA32_THERM_INTERRUPT MSR. A second scenario was used to demonstrate the wide array of functionality that a MSR-based virtual breakpoint can address. In this case, the IA32_THERM_INTERRUPT MSR was configured to trigger an interrupt upon the processor reaching a critical temperature. This functionality can be used to prevent thermal damage to the processor. If the data in this MSR were corrupted similarly to the way the IA32_PAT MSR was corrupted in the first scenario, critical CPU temperatures could be reached with no software warning, potentially causing a hard reboot or even CPU damage. Once it has been discovered that this interrupt setting is being cleared at some point in the execution of an OS boot loader or kernel, it would be very difficult to track down the exact point that the MSR was modified. By setting a virtual breakpoint on this MSR, the offending error code could be easily tracked down without the need for stepping through large amounts of code during dynamic analysis, in search of the error.

Experiment 2 Implementation:

Experiment 2 sets a virtual breakpoint that will trap on write access to the IA32_PAT MSR as well as the IA32_THERM_INTERRUPT MSR. The exit condition will trap on the x86 assembly WRMSR command, the technique used to write to an MSR. When MSR write access is detected by the hypervisor, it is configured to filter and ignore all MSR access except for the aforementioned MSRs, effectively creating a virtual breakpoint on specific MSRs. For the purpose of this experiment, writes to both of these MSRs are prevented from taking effect in the hypervisor so that we do not destabilize the guest OS. The virtual breakpoint handler code is shown in Figure 12 - MSR Functionality Virtual Breakpoint Handler Code.

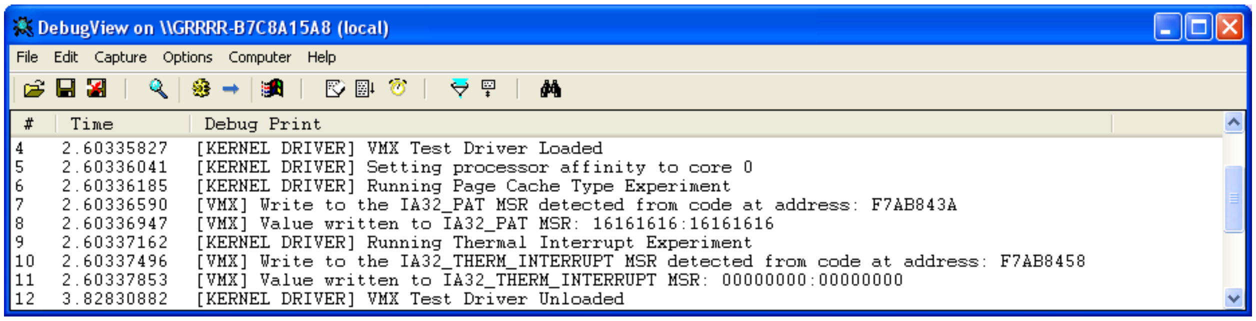
```
//Hypervisor code to trap on write access to IA32_PAT and
IA32_THERM_INTERRUPT MSR's
if( ExitReason == 0x00000020 ) //WRMSR exit reason
{
	if( GuestECX == 0x00000277 ) //IA32_PAT
	{
		DbgPrint("[VMX] Write to the IA32_PAT MSR detected
		from code at address: %X", GuestEIP);
		DbgPrint("[VMX] Value written to IA32_PAT MSR:
		%08X:%08X", GuestEDX, GuestEAX);

		__asm
		{
			popad
			jmp Resume
		}
	}

	if( GuestECX == 0x0000019B ) //IA32_THERM_INTERRUPT
	{
		DbgPrint("[VMX] Write to the IA32_THERM_INTERRUPT MSR
		detected from code at address: %X", GuestEIP);

		DbgPrint("[VMX] Value written to IA32_THERM_INTERRUPT
		MSR: %08X:%08X", GuestEDX, GuestEAX);
		__asm
		{
			popad
			jmp Resume
		}
	}
}
```

Figure 12 - MSR Functionality Virtual Breakpoint Handler Code

The code shown in Appendix C – Kernel Code used in Experiments, Figure 22 - Kernel Code to Trigger MSR Based Virtual Breakpoints, was used to trigger the breakpoints that were configured to trap on manipulation of either the IA32_PAT MSR or the IA32_THERM_INTERRUPT MSR, in order to monitor changes to the system's memory page cache types and the system's thermal interrupt configurations. This was accomplished by forcing a write to each of these MSR registers to simulate the corruption of the functionality previously detailed in this section's overview.

<u>Experiment 2 Results</u>:

The VMM handler in this experiment was able to trap on any write accesses to the specified MSRs. The virtual breakpoint handler successfully detected when either of the specified MSRs were being modified. The breakpoint handler's debug output prints out the value that was being written to one of the MSRs along with the memory address (the current instruction pointer) of the code that was performing the MSR write to either MSR. It is important to remember that the memory address of the code that triggered the breakpoint can be printed for any virtual breakpoint. This debug output is shown below in Figure 13 - Output from MSR Based Virtual Breakpoints. This experiment has successfully demonstrated using a virtual breakpoint to trap on functionality associated with an MSR. MSRs control a wide array of system functionality, and would otherwise be difficult to trap on without already knowing exactly where in the code the MSR was being modified from. In many cases this is a difficult problem to solve, especially if source code level debugging is not an option, or if the MSR was unexpectedly being written to, as mentioned in the scenario provided earlier in this section.


DebugView on \\GRRRR-B7C8A15A8 (local)
File Edit Capture Options Computer Help

```
#   Time         Debug Print
4   2.60335827   [KERNEL DRIVER] VMX Test Driver Loaded
5   2.60336041   [KERNEL DRIVER] Setting processor affinity to core 0
6   2.60336185   [KERNEL DRIVER] Running Page Cache Type Experiment
7   2.60336590   [VMX] Write to the IA32_PAT MSR detected from code at address: F7AB843A
8   2.60336947   [VMX] Value written to IA32_PAT MSR: 16161616:16161616
9   2.60337162   [KERNEL DRIVER] Running Thermal Interrupt Experiment
10  2.60337496   [VMX] Write to the IA32_THERM_INTERRUPT MSR detected from code at address: F7AB8458
11  2.60337853   [VMX] Value written to IA32_THERM_INTERRUPT MSR: 00000000:00000000
12  3.82830882   [KERNEL DRIVER] VMX Test Driver Unloaded
```

Figure 13 - Output from MSR Based Virtual Breakpoints

4.3.2 Analysis of Complex Debug Environments

The experiments overviewed in this section represent a set of cases where virtual breakpoints can help to dynamically debug through traditionally complex environments. The experiments in this section are only two of many situations where virtual breakpoints can decrease the complexity of debugging over traditional debugging capabilities. These cases are centered on debugging of exceptions and error states generated from both Ring 0 and Ring 3.

4.3.2.1 Experiment 1 - Hardware Breakpoint Exception

Experiment 1 Overview:

The experiment presented in this section demonstrates the ability to set a virtual breakpoint to easily trap on an exception. The exception chosen for this experiment was generated from the kernel and was caused by a traditional hardware breakpoint. Traditional x86-based hardware breakpoints are configured through a processor’s debug registers to monitor a specified memory or port I/O address, and trigger an exception to interrupt 0x01 when triggered. This exception was selected to demonstrate the ability to trap on Ring 0 based exceptions, as well as to point out the fact that our debugger can even be used to debug a traditional kernel debugger that makes use of hardware debug breakpoints. Using the virtual breakpoint concept provided throughout this thesis, these exception states, such as the exception overviewed in this section, can be debugged without making any modifications to the guest operating system that would potentially alter or contaminate the system state, such as hooking the interrupt 0x01 handler code. The virtual breakpoint used in this section also allows for analysis of the system before the exception handler is reached.

<u>Experiment 1 Implementation</u>:

This experiment has configured and set a virtual breakpoint that will trap when a hardware breakpoint generates an exception in the guest OS. The exit reason code for an exception is 0. A hardware breakpoint will trigger exception number 1. To ensure that a specific exception causes a VM-Exit, the bit corresponding to the exception number must be set in the exception bitmap field of the VMCS. As shown below in Figure 14 - Portion of Hardware Breakpoint Exception Hypervisor Handler, our virtual breakpoint handler is configured to detect exception number 1. It is important to note that the ExitQualification field receives the exception that caused the VM-Exit, which allows for us to easily distinguish between different exceptions when setting a virtual breakpoint to trap on a specific exception. When the hardware debug breakpoint causes the exception in this experiment our virtual breakpoint handler code passes the exception along and allows for interrupt 0x01 to handle the exception in the guest OS.

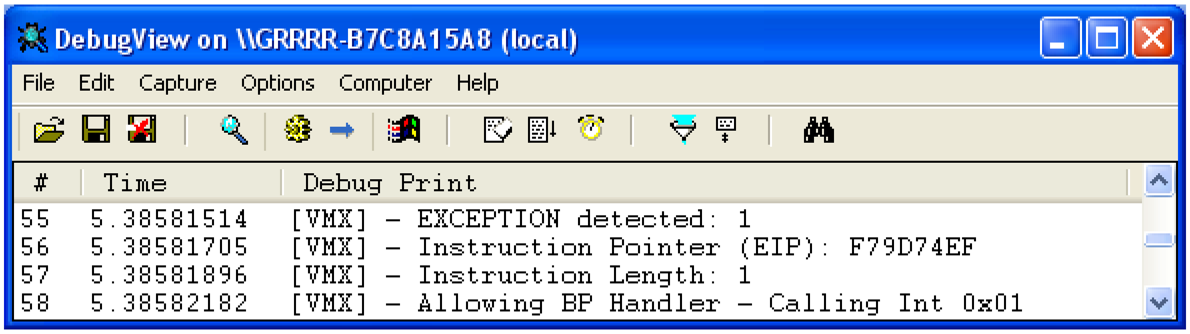
```
if( ExitReason == 0x00000000 )
{
        DbgPrint("[VMX] - EXCEPTION detected: %X",
        ExitQualification);

        DbgPrint("[VMX] - Instruction Pointer (EIP): %X",
        GuestEIP);

        DbgPrint("[VMX] - Instruction Length: %X",
        ExitInstructionLength);

        if(ExitQualification == 1)
        {
                DbgPrint("[VMX] - Allowing BP Handler - Calling Int
                0x01");
                __asm
                {
                        int 0x01
                }
        }
        ...
}
```

Figure 14 - Portion of Hardware Breakpoint Exception Hypervisor Handler

To trigger the virtual breakpoint set for this experiment, a kernel hardware breakpoint was set to trap on access to I/O port 0x0CF8. A kernel driver was then loaded and executed, which wrote data to I/O port 0x0CF8 in order to trigger the kernel hardware breakpoint that was previously enabled. These steps caused an exception to be generated in order to handle the kernel-based hardware breakpoint. Interrupt 0x01 was hooked when setting up the kernel hardware breakpoint so that the kernel hardware breakpoint could be handled in the guest, to avoid a crash.

<u>Experiment 1 Results</u>:

The virtual breakpoint handler that was written for this experiment successfully trapped on the exception that was generated when the guest OS's hardware breakpoint was triggered by the port I/O access that it was configured to break on. When this exception was generated, the hypervisor's virtual breakpoint handler detected the exception, printed out that exception 1 was generated by the guest, printed the memory address of the code that caused this exception, and allowed the exception to pass control to interrupt handler 0x01 in the guest OS so that is could handle the hardware breakpoint that caused the exception. This process is shown below in the debug output screenshot, in Figure 15 - Output from Hardware Breakpoint Exception Virtual Breakpoint.


DebugView on \\GRRRR-B7C8A15A8 (local)
File Edit Capture Options Computer Help

```
#    Time          Debug Print
55   5.38581514    [VMX] - EXCEPTION detected: 1
56   5.38581705    [VMX] - Instruction Pointer (EIP): F79D74EF
57   5.38581896    [VMX] - Instruction Length: 1
58   5.38582182    [VMX] - Allowing BP Handler - Calling Int 0x01
```

Figure 15 - Output from Hardware Breakpoint Exception Virtual Breakpoint

4.3.2.2 Experiment 2 - Divide by Zero Error

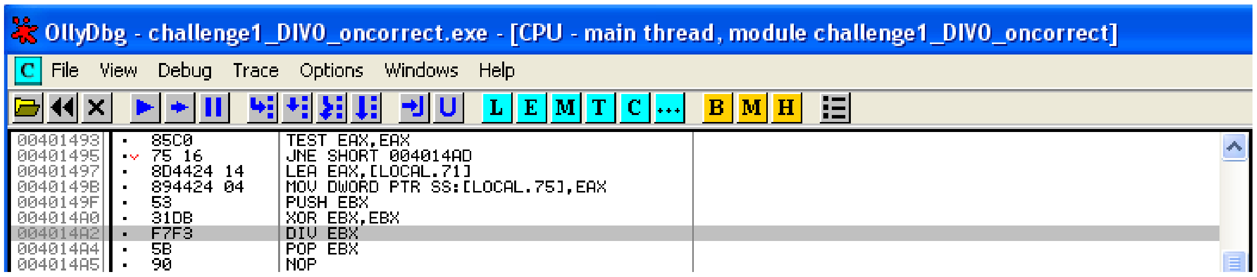
Experiment 2 Overview:

The experiment overviewed in this section demonstrates that a virtual breakpoint can be used to monitor additional exceptions, beyond the hardware breakpoint exception that was previously covered. These exceptions can even originate in the guest OS from Ring 3, and do not have to be executing with full privileges like the hardware breakpoint exception. With this capability, a virtual breakpoint can be used to trap on any error that causes an exception. This type of breakpoint can be set to a specific exception, or simply trap on any exception to allow for further analysis. This capability can significantly simplify the process of locating errors in code during dynamic analysis, since the breakpoint actually traps on the error itself.

This approach avoids interference with the system being debugged since it does not require hooking of the interrupt handlers that the exception may trap to, and also allows for analysis of the system before the interrupt handler is triggered in the guest. For this experiment, code that will cause a divide by zero error has been placed in a Ring 3 user mode application, rather than in the kernel. The user mode application with the divide by zero error will be executed to cause an exception for our virtual breakpoint to handle.

Experiment 2 Implementation:

For this experiment, a virtual breakpoint handler is used that is extremely similar to the hardware breakpoint exception experiment. For this experiment, the virtual breakpoint handler is monitoring for any exception, so that any errors that cause exceptions can be detected. To cause a divide by zero exception, a Ring 3 application was modified to purposely execute a DIV instruction with a divisor of zero. This causes a divide by zero error to occur, generating exception 0. For this experiment, the application that was used to generate this exception was disassembled using OllyDbg to show the error generating code, in an assembly level representation. The relevant portion of this application is shown below in Figure 16 - Assembly Level Representation of Divide by Zero Ring 3 Application. The XOR EBX, EBX instruction at memory address 0x004014A0 causes the EBX register to clear, ending up with the value 0x00000000. The DIV EBX instruction at memory address 0x004014A2 attempts to execute a DIV instruction with a divisor of zero.

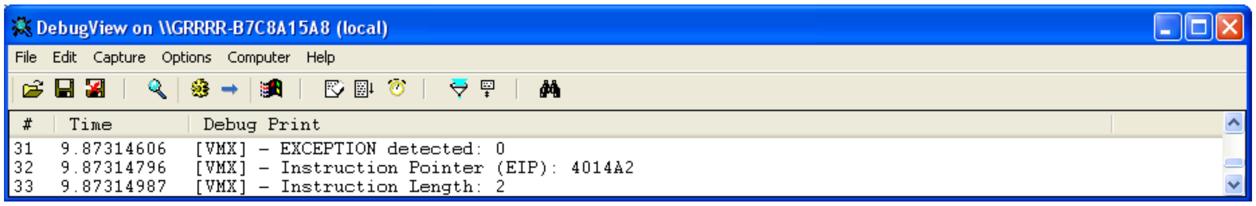

OllyDbg - challenge1_DIV0_oncorrect.exe - [CPU - main thread, module challenge1_DIV0_oncorrect]

```
00401493  .  85C0          TEST EAX,EAX
00401495  .v 75 16         JNE SHORT 004014AD
00401497  .  8D4424 14     LEA EAX,[LOCAL.71]
0040149B  .  894424 04     MOV DWORD PTR SS:[LOCAL.75],EAX
0040149F  .  53            PUSH EBX
004014A0  .  31DB          XOR EBX,EBX
004014A2  .  F7F3          DIV EBX
004014A4  .  5B            POP EBX
004014A5  .  90            NOP
```

Figure 16 - Assembly Level Representation of Divide by Zero Ring 3 Application

<u>Experiment 2 Results</u>:

The virtual breakpoint handler that was used for this experiment was able to trap on the exception that was generated when the divide by zero error occurred in the Ring 3 applications. When this error occurred, exception zero was generated within the guest. The virtual breakpoint handler printed output to demonstrate that the exception was trapped on by our virtual breakpoint. This output included the exception number that was generated, the memory address of the code that generated the exception, and the length of the instruction that generated the exception. A screenshot of this debug output is shown below in Figure 17 - Output from Divide by Zero Exception Virtual Breakpoint. It is important to notice that the instruction pointer displayed in Figure 17 - Output from Divide by Zero Exception Virtual Breakpoint matches the memory address displayed in the application disassembly shown in Figure 16. This verifies that the virtual breakpoint handler's expectation of the memory address of the code that caused the virtual breakpoint is accurate.


DebugView on \\GRRRR-B7C8A15A8 (local)
File Edit Capture Options Computer Help

```
#   Time          Debug Print
31  9.87314606    [VMX] - EXCEPTION detected: 0
32  9.87314796    [VMX] - Instruction Pointer (EIP): 4014A2
33  9.87314987    [VMX] - Instruction Length: 2
```

Figure 17 - Output from Divide by Zero Exception Virtual Breakpoint

With the virtual breakpoint capabilities demonstrated in this section, it becomes feasible to set a virtual breakpoint that will trigger on a variety of error states. These error states can even be detected before the system attempts to handle them, or in some cases, blue screens or completely crashes. Now the state of the system can be dynamically analyzed as the bad instruction is executed. This application of enhanced debugging capabilities can help to greatly decrease the time it takes to find code errors during dynamic analysis.

5. Chapter 5 - Conclusion

5.1 Summary

The research performed under this thesis has successfully demonstrated that enhancements can be made to a kernel debugger through the use of a hardware assisted virtual machine monitor. This was accomplished by utilizing the correlation between an Intel VT based VMM's exit condition functionality and that of a kernel debugger hardware breakpoint. Under each circumstance, a specific condition is used to halt processor execution, and pass control to a software handler. The VMM's exit condition's have been used as "virtual breakpoints" to trap to a breakpoint handler location in the VMM when a specified condition has been reached in the guest VM. This new set of capabilities has provided a plethora of functional enhancements to a kernel debugger, which can enhance a debugging capability under a large number of different debugging scenarios. The experiments presented in this thesis, within chapters 3 and 4, have adequately demonstrated the enhancements that an Intel VT based VMM's exit conditions can provide to a debugger. A brief overview of this concepts and experiments will be summarized in this chapter, along with a list detailing future areas of research related to this thesis.

5.1.1 Traditional vs. Enhanced Debugging Capabilities

To reiterate the enhancements that have been provided to a debugger through the use of virtual breakpoints, or tailored VMM exit conditions, a comparison of tradition and enhanced debugging capabilities is given below.

Traditional debug breakpoints on an x86-based system use facilities provided by the processor to trap on specified memory or port I/O addresses. Two categories of these breakpoints exist; hardware breakpoints and software breakpoints. Hardware breakpoints are configured using internal processor registers, and are able to trap on both memory and port I/O addresses. Up to four hardware breakpoints can be set at one time. When a hardware breakpoint condition is met, an exception is generated, calling interrupt 0x01 to handle the breakpoint. Also, hardware breakpoints do not alter the target software. Software breakpoints are placed only on memory addresses and are configured by replacing the first byte of the instruction at the specified memory address with a 0xCC opcode. When the 0xCC opcode is executed by the processor, an exception is generated that calls interupt 0x03 to handle the breakpoint. The interrupt handler for software breakpoints is responsible for replacing the portion of the instruction that was replaced with 0xCC. An unlimited number of software breakpoints can be placed at a given time.

The virtual breakpoints detailed within this thesis are configured by loading an Intel VT-x based VMM, and tailoring VM-exit conditions to trap on specified functionality. VM-exits can be configured to trap on a vast number of exit conditions, each providing a specific condition that can be used as a virtual breakpoint. When a VM-exit occurs, execution control is passed to a software handler within the VMM. This functionality is analogous to the ability of a standard breakpoint to trap on a specified condition and then pass execution control to an interrupt handler.

5.1.2 Enhanced Debugging Capability Experiments

As mentioned in the previous section, 5.1.1, this thesis makes use of a VMM's ability to set and configure VM-exits to trap on vast number of specific conditions to create virtual breakpoints. Virtual breakpoints provide the ability to trap on a great deal of system level events, providing a debugger with much more granular control over target code than a standard memory or port I/O breakpoint can provide. In addition, an unlimited number of virtual breakpoints can be set without the need to modify target software. The VMM used to set, configure, and handle these virtual breakpoints also provided a great degree of control over target software through the use of the VMCS.

The following experiments were provided to help demonstrate the enhanced debugging capabilities that can be obtained through the use of virtual breakpoints:

- Demonstration of virtual breakpoint
- Implementation of more than four hardware breakpoints
- Full control over guest OS system state

A demonstration of virtual breakpoints was provided by setting virtual breakpoints that will trap on specified port I/O addresses, reads or writes to specified MSR's, and the execution of certain VMX instruction. This experiment exhibited 3 types of the many virtual breakpoints that can be configured using exit reasons. One each virtual breakpoint was set, a kernel driver was loaded to execute instructions that caused the virtual breakpoint to trap to its software handler within the VMM. The VMM's virtual breakpoint handler was configured to print out a debug message indicating that the breakpoint condition was met, and that execution flow was trapped and passed to the VMM's breakpoint handler.

The next demonstrate was used to show that more than four hardware breakpoints can be set simultaneously. Since port I/O breakpoints were previously configurable through hardware breakpoints, not configurable through software breakpoints, and limited to a total of four simultaneous breakpoints, port I/O-based virtual breakpoints were chosen for this experiment. To demonstrate that more than four of these breakpoints could be set, this experiment set virtual breakpoints to trap on any I/O port from 0 to 0x60. A kernel driver was loaded that executed I/O instructions to access port 0x00, port 0x08, port 0x10, port 0x18, port 0x20, port 0x28, port 0x30, port 0x38, port 0x40, and port 0x48. The VMM's software handler was set to print debug output indicating any I/O ports that were read or written which had been set to trigger a virtual breakpoint. The debug output demonstrated that at least ten port I/O-based virtual breakpoints had been simultaneously set.

The third enhanced debugging capability experiment was used to help demonstrate that a virtual machine monitor has full control over guest software, which is extremely important to debug software. This control is available through configurations to a VMM's VMCS, since it holds the state of the guest virtual machine. To demonstrate this, the VMM was used to directly modify the instruction pointer (EIP register value) with ease, which normally can't be done directly and would require a good amount of effort. This experiment set a series port I/O breakpoints that were triggered in series by a kernel driver. The VMM breakpoint handler printed debug output indicating each port that was accessed, and the current instruction pointer of the guest software that caused the virtual breakpoint to occur. The VMM breakpoint handler was then modified to increment the instruction pointer by 12 when a virtual breakpoint on I/O port 0x01 was triggered. The same kernel driver was then executed. The results demonstrated that the instruction pointer artificially increased by a value of 12 when the breakpoint on I/O port 0x01 was triggered. The 12 bytes of instructions following the I/O access that caused the breakpoint were passed over in the kernel driver, which caused fewer instructions to execute, triggering fewer virtual breakpoints from the guest.

5.1.3 Applications of Enhanced Debugging Capability Experiments

The enhanced debugging capabilities provided by these thesis can be utilized for a variety of debugging applications. Experiments within this thesis helped to demonstrate a few of the many potential applications of this technology. These experiments included the following:

- Smart Filtering of Data During Dynamic Analysis
  - Processor Mode Switch
  - Specific MSR-based Functionality
- Analysis of Complex Debug Environments
  - Hardware Breakpoint Exception
  - Divide by Zero Error

The first category of experiments, smart filtering of data during dynamic analysis, included two experiments that were used to use virtual breakpoints for the purpose of isolating specific functionality within code, where the location of the functionality could have previously been unknown. The idea was to show that increased granularity of virtual breakpoints could be tailored to isolate a desired section of a program without the need to step through large amounts of code in search of this functionality. The second category of experiments, analysis of complex debug environments, was used to demonstrate that situations that could be difficult to locate and analyze can easily be located and dynamically evaluated.

The processor mode switch experiment was used to demonstrate that a virtual breakpoint could be set to trap on major changes in system state such as a processor mode switch. Under this experiment a breakpoint was placed on CR0, which monitored the bits used to enable/disable paging and to switch between protected mode and real mode. If either bit was modified in the guest, that the VMM breakpoint handler detected the change and provided debug output to show how this breakpoint could successfully be configured.

The MSR-based functionality experiment was used to demonstrate that functionality associated with one of the processor's MSR's could be monitored by a virtual breakpoint. If this MSR functionality was altered, a breakpoint would trap and pass control to the VMM's breakpoint handler. Two scenarios were laid out for this experiment, each revolving around the concept that an MSR was accidently written to, which later has adverse side effects on an operating system. Since these MSR's were written to in error, the code mode that affected these MSR's is unknown. This would making debugging the problem difficult in a large complex system since the vicinity of the cause of the error is completely unknown, even if it has been identified that the error is a result of an MSR value being altered. The experiment sets a breakpoint on the MSR in question to demonstrate that the cause of the problem can be trapped on, allowing for dynamic analysis of the code that caused the problem. As a result, debugging time can be dramatically decreased.

The hardware breakpoint exception experiment was used to show that a virtual breakpoint could be used to trap on exceptions generated by the processor. Specifically for this experiment, exception 0x01 was monitored, which is generated when a hardware breakpoint occurs. An interesting point here is the fact that the debugging capabilities provided by this thesis can be inherently use do debug a standard kernel debugger, even through hardware breakpoints. The ability to trap on exceptions also provides the ability to set a breakpoint to occur on error states that generate exceptions, which was covered in more detail within the divide by zero error experiment. It is also important to note that the breakpoints set to trap on exceptions work without the need to make intrusive modifications to the system state of the software being debugged, such as hooking interrupts, modifying target code, or executing code within the same context as the software being debugged. The virtual breakpoint also gains control before the exception handler obtains execution control.

The divide by zero experiment was used to demonstrate that a virtual breakpoint can be configured to trap when a variety of errors occurs. This capability can be useful for tracking and analyzing error states. With the ability to set a breakpoint on one or more exceptions, when an error occurs that causes an exception, the exact location of the code that caused the error can be immediately located an analyzed, removing the need to search out the cause of the error. For this experiment a virtual breakpoint was configured to trap on software that caused exception zero to trigger, and a divide by zero error was generated from an application running in ring 3. This ring 3 application was chosen to point out the fact that ring 3 software can also be monitored using virtual breakpoints. When the code with the divide by zero error was executed, the virtual breakpoint trapped to its software handler, which provided debug output indicating that the divide by zero error occurred. The location of the error in memory was also provided to demonstrate that the code can easily be analyzed which the guest software is paused.

5.2 Future Work

Upon completion of this thesis document, I still plan on pursuing additional applications of this technology. Some of the areas I plan on pursuing in the near future include the following:

- Implementation of additional types of virtual breakpoints
- Developing a GUI interface to the virtual debugger
- Software debugger for multiple environments/operating system's
- Error recovery

5.2.1 Implementation of Additional Virtual Breakpoints

In addition to the virtual breakpoints that were developed throughout the experiments presented in this thesis, I plan on continuing to identify and implement several more. Some additional virtual breakpoints that I plan on implementing include, but are not limited to, the following:

- Triple Fault
- Interrupt Hooking
- Debug Register Access

A specific exit condition is reserved for handling triple faults. A triple fault occurs when an exception is generated while the double fault handler is handling an exception that was generated when attempting to call a regular exception handler. When a triple fault occurs on an x86-based system, the entire system is typically forced to reboot, making this error extremely difficult to recover from, if possible at all. This would prevent a debugger from performing dynamic analysis through the error since the system reboot would clear memory and the system state that would have been visible when the error occurred. By creating a virtual breakpoint to trap on and handle triple faults, it will be possible to analyze the system as the triple fault occurred in order to help determine the cause of the triple fault, and possible the events leading up to it.

Another exit condition is reserved for access to the Global Descriptor Table Register (GDTR) and Interrupt Descriptor Table register (IDTR). This exit condition is triggered when the SIDT, LIDT, SGDT, or LGDT x86 assembly instructions are executed. The IDTR instructions are required to locate and/or change the base address of the interrupt descriptor table (IDT), since the IDT can be located anywhere in memory when the processor is executing in protected mode. The IDT is used to help provide a response to hardware interrupts, software interrupts, and exceptions. As a result, the IDTR is essential in the process of placing a hook on an interrupt, since it is required to help find the IDT. With the ability to monitor the IDTR, it could be determined when software was trying to place a hook on an interrupt.

Debug register access can also configured within a VMM to cause a VM-exit. With the ability to monitor all modifications to debug registers, a hardware debugger could be debugged through the use of our virtual breakpoints. Although not an extremely likely case, I find it interesting to provide a capability to help debug the development of a hardware debugger. Adding this capability will not be too difficult since it really only involves one piece of additional functionality that is not likely to require a complex breakpoint handler. Also, I plan on adding as much functionality as possible so the VMM debugger that was initiated under this thesis can be built into a more robust capability with functionality capable of handling as many use cases as possible.

5.2.2 GUI Interface to Debugger

Once I have implemented additional virtual breakpoint functionality, I plan on providing a Graphical User Interface (GUI) for the VMM debugger. This will be essential to provide usable control over the debugging capabilities presented under this thesis, for example, viewing and analyzing the full system state on breakpoints, performing operations such as single stepping code after a breakpoint has been reached, configuring specific virtual breakpoint functionality. I would like to provide a complex GUI application using C# .NET since it provides a great deal of control over its GUI applications.

Since the VMM debugger backend will be running from within a VMM, it will be a challenge to provide a ring 3 GUI that can interact with the debugger's core functionality. A typical kernel debugger GUI program only needs to communicate between ring 3 (user-land) and ring 0 (kernel). The VMM debugger presented under this thesis will require communication from ring 3 to ring 0 to the VMM and back. If a text mode GUI is used, the communication channel may not require this channel between different privilege levels and processor modes, but the interface will be ancient compared to a GUI designed in C#.

To provide communication between the C# GUI application and the core VMM debugger functionality, I plan on providing a ring 3 GUI application, a kernel module, and the VMM debugger capabilities. I will use IOCTL's to communicate between the C# GUI application and the kernel module. The kernel module will use a pre-determined port I/O range to pass data to the VMM. This will be accomplished by configuring the VMM to trap on any accesses to the pre-determined port I/O range. Since the VMM can view the entire guest state, information can be transferred easily through registers or memory. The virtual machine monitor can also transfer data back to the GUI by modifying the guest's register's or memory. As a result, a full communication channel can be established between the C# GUI application and the VMM debugger, allowing for the GUI to pass commands to the debugger, and receive system information back from the debugger that is required to update the GUI based on user input and system state.

5.2.3 Software Debugger for Multiple Environments

The MAVMM publication[5] has provided the idea to load a hypervisor from a modified GRUB boot loader. In fact, the author of this publication indicated that GRUB was successfully used to bootstrap MAVMM's hypervisor. I would like extend this idea to the VMM debugging capabilities outlined under this thesis in order to allow for the debugger run on different operating systems. A challenge associated with this task will be to provide a GUI for the VMM debugger that works across different OS's. My first attempt towards solving the interface problem will be to implement a text mode library from the VMM, similar to the interface provided by the SoftICE kernel debugger.

Appendix A –Basic Exit Reasons

0 – Exception or NMI (Non Makeable Interrupt)

1 – External Interrupt

2 – Triple Fault

3 – INIT Signal

4– Startup Inter-Processor Interrupt

5 – I/O SMI (System Management Interrupt)

6 – Other SMI

7 – Interrupt Window

8 – NMI Window

9 – Task Switch

10 – CPUID Command

11 – GETSEC Command

12 – HLT Command

13 – INVD Command

14 – INVLPG Command

15 – RDPMC Command

16 – RDTSC Command

17 – RSM Command

18 – VMCALL Command

19 – VMCLEAR Command

20 – VMLAUNCH Command

21 – VMPTRLD Command

22 – VMPTRST Command

23 – VMREAD Command

24 – VMRESUME Command

25 – VMWRITE Command

26 – VMXOFF Command

27 – VMXON Command

28 – Control Register Access

29 – MOV DR (move data into a debug register)

30 – I/O Instruction (i.e. IN or OUT Commands)

31 – RDMSR

32 – WRMSR

33 – VM-Entry failure due to invalid guest state

34 – VM-Entry failure due to MSR loading

36 – MWAIT Command

37 – Monitor Trap Flag

39 – MONITOR Command *

40 – PAUSE command

41 – VM-Entry Failure due to machine check

43 – TPR Below Threshold

44 – APIC(Advanced programmable Interrupt Controller) Access

46 – Access to GDTR or IDTR *

47 – Access to LDTR or TR *

48 – EPT Violation *

49 – EPT Misconfiguration *

50 – INVEPT *

51 – RDTSCP *

52 – VMX-preemption timer expired *

53 – INVVPID *

54 – WBINVD Command

55 – XSETBV Command

Appendix B – Index of Acronyms

VMX	– Virtual Machine eXtensions
VMM	– Virtual Machine Monitor
VM	– Virtual Machine
VMCS	– Virtual Machine Control Structure
VT	– Virtualization Technology
VT-d	– Intel Virtualization Technology for Directed I/O
DR	– Debug Register
CR	– Control Register
BP	– Break Point
IDT	– Interrupt Descriptor Table
MSR	– Model Specific Register

Appendix C – Kernel Code used in Experiments

Kernel Code to Trigger Virtual Breakpoints

```
DbgPrint("[KERNEL DRIVER] Running Experiment 1 - Demonstrating
Virtual Breakpoints");

//Test rdmsr access on the TSC
DbgPrint("[KERNEL DRIVER] Reading MSR 0x10 - TSC");
__asm
{
      PUSHAD //Save general purpose register state

      mov ecx, 0x00000010 //Prepare to read MSR # 10

      rdmsr //Read the MSR specified by ECX

      mov TSC_HIGH, edx //Save the top 32 bits of data read
      mov TSC_LOW, eax //Save the lower 32 bits of data read

      POPAD //Restore general purpose register state
}
DbgPrint("TSC_HIGH: %X", TSC_HIGH);
DbgPrint("TSC_LOW: %X", TSC_LOW);

//Test PIO access at port 0x0CF8
DbgPrint("[KERNEL DRIVER] Reading from port 0x0CF8");
__asm
{
      pushad //Save general purpose register state

      mov dx, 0x0CF8 //Prepare to read port 0x0CF8

      mov eax, 0x80000000 //Prepare to write the value 0x80000000

      out dx, eax //Write the value in eax to the port in dx

      in eax, dx //Read from the port specified by dx into eax

      mov temp32, eax //Save the value read from the port

      popad //Restore general purpose register state
}
DbgPrint("[KERNEL DRIVER] Port Read Data: %X", temp32);

//Test a VMX instruction - VMXON
DbgPrint("[KERNEL DRIVER] executing VMXON instruction");
__asm
{
      pushad Save general purpose register state
```

```
        //Place bytecode into memory that will build the VMXON
        [ESP]

        //assembly instruction since our compiler does not
        recognize this

        //instruction
        _emit 0xF3
        _emit 0x0F
        _emit 0xC7 // VMXON [ESP]
        _emit 0x34
        _emit 0x24

        popad //Restore general purpose register state
}
```

Figure 18 - Kernel code to trigger virtual breakpoint experiments

<u>Kernel Code to Trigger Multiple Simultaneous Virtual Breakpoints</u>

```
DbgPrint("[KERNEL DRIVER] Running Experiment 2 - Demonstrating 10
simultaneous PIO breakpoints");

//Test ten PIO breakpoints
__asm
{
        pushad

        mov cx, 0x0A

        mov dx, 0x0000
        LOOP1:
                in al, dx

                add dx, 0x08
                dec cx
        JNZ LOOP1

        popad
}

//Test rdmsr access on the TSC
DbgPrint("[KERNEL DRIVER] Reading MSR 0x10 - TSC");
__asm
{
        PUSHAD

        mov ecx, 0x00000010

        rdmsr

        mov TSC_HIGH, edx
        mov TSC_LOW, eax
        POPAD
}
DbgPrint("TSC_HIGH: %X", TSC_HIGH);
DbgPrint("TSC_LOW: %X", TSC_LOW);
```

Figure 19 - Kernel code to trigger multiple simultaneous virtual breakpoints experiment

Kernel Code to Help Observer Powerful VMM Control Over Guest

```
//Test Modification of Instruction Pointer (EIP) from VMM
DbgPrint("[KERNEL DRIVER] Running Experiment 3 - Changing
Instruction Pointer from VMM");

//Modified EIP Experiment
__asm
{
        pushad

        mov dx, 0x0000

        //Port Read 0
        in al, dx //0xEC
        inc dx //0x66 0x42

        //Port Read 1
        in al, dx
        inc dx

        //Port Read 2
        in al, dx
        inc dx

        //Port Read 3
        in al, dx
        inc dx

        //Port Read 4
        in al, dx
        inc dx

        //Port Read 5
        in al, dx
        inc dx

        //Port Read 6
        in al, dx
        inc dx

        //Port Read 7
        in al, dx

        popad
}
```

```
DbgPrint("[KERNEL DRIVER] Completed Experiment 3");
```

Figure 20 - Kernel Code Used to Observe EIP Modification

Kernel Code to Trigger Processor Mode Switch Breakpoint

```
DbgPrint("[KERNEL DRIVER] Begin Chapter 4 Experiment - Processor
Mode Switch");
DbgPrint("[KERNEL DRIVER] Clearing Paging and Protected Mode
Enable bits in CR0");
__asm
{
      pushad

      //Write a value to CR0 that clears bits 0 and 31
      mov eax, 0x0001003a
      mov cr0, eax

      popad
}
```

Figure 21 - Kernel Code to Trigger Processor Mode Switch Breakpoint

Kernel Code to Trigger MSR Based Virtual Breakpoints

```
DbgPrint("[KERNEL DRIVER] Running Page Cache Type Experiment");
__asm
{
      pushad

      //-----IA32_PAT-----//
      mov ecx, 0x00000277 //PAT MSR
      mov edx, 0x16161616 //Cache Type of WC_WB - alternating (WC
                          is 0x01, WB is 0x06)
            mov eax, 0x16161616

            wrmsr

            popad
      }

      DbgPrint("[KERNEL DRIVER] Running Thermal Interrupt
      Experiment");
      __asm
      {
            pushad

            //-----IA32_THERM_INTERRUPT-----//
            mov ecx, 0x0000019B //PAT MSR
            mov edx, 0x00000000 //Clearing all register values
            mov eax, 0x00000000

            wrmsr
```

```
                popad
        }
}
```

Figure 22 - Kernel Code to Trigger MSR Based Virtual Breakpoints

Appendix D – Related Literature

This section provides an analysis of publications that contain information related to the topic of this thesis. For each related publication, an overview is provided, as well as information detailing the relationship between the publication and the topic of this thesis.

Anti-debugging Framework Based on Hardware Virtualization Technology [18]

Overview

Most modern anti-debugging products do not guarantee protection against code running with high privileges, for example, in Ring 0. This publication demonstrates an anti-debugging framework that utilizes a Virtual Machine Monitor (VMM) to provide protection against debug analysis.

Current hardware architecture has a flawed security model, and code running with a high enough privilege (i.e. Ring 0) can access the entire system. Code executing in user mode (Ring 3) can only control its own specific set of resources. Because of this security issue, once code is executing in Ring 0, it cannot be guaranteed that restrictions can successfully be put into place.

Chip vendors, such as Intel and AMD, have been working to enhance their processors with the addition of hardware virtualization extensions. With the introduction of these modern extensions, specific processor instructions provide additional capabilities to a VMM allowing for the VMM to achieve greater performance than a traditional software-only VMM. These extensions also allow for a VMM to run directly on the hardware.

A VMM has the ability to monitor system events of a guest virtual machine on the fly. The VMM can cause the virtual machine to relinquish control to the VMM through a VM-Exit condition. Once control is passed to the VMM, additional processing can be performed under the control of the VMM. The VMM also maintains data structures to keep track of the complete system status of both the VMM and Guest virtual machine. This paper uses a Windows device driver to load a VMM in Windows in order to add additional protection to the system without changing any existing hardware or part of the existing operating system.

Anti-Debugging - Software Debug Breakpoints

The Intel x86 architecture uses an INT 0x03 (software breakpoint) to place a software breakpoint into code. This is accomplished by substituting the first byte of an instruction with the 0xCC code. When a software breakpoint is executed, it transfers control to a debugger, which performs debug analysis and then proceeds to replace the 0xCC byte with the original byte that was replaced by the 0xCC. This paper suggests that a VMM can be used to hook an INT 0x03 instruction in order to prevent a debugger from using a software breakpoint to perform debug analysis.

Anti-Debugging - Hardware Debug Breakpoints

Hardware breakpoints can be set in Intel’s debug registers. When the Instruction Pointer (IP) shares the same value as one of Intel’s debug registers, an INT 0x01 exception will occur. This paper also suggests that the INT 0x01 instruction can be monitored to prevent hardware breakpoints from passing control to a debugger.

Anti-Debugging - Process Protection

A process control block (PCB) holds information describing a process and is used by the kernel to manipulate various aspects of its process. The Intel CR3 register stores the base physical address of the page directory, which helps point to the current process. When a process is scheduled by the kernel, the CR3 register is loaded with the page directory pointer from the process's PCB. A VMM can easily be configured to disallow access to a specific process from any other process by preventing an unallowable process from obtaining CR3 data from the other process's PCB.

Relationship to thesis

The concepts demonstrated in this paper are similar to those presented within this thesis, but for the opposite purpose. This thesis is utilizing the extra functionality provided by a VMX based VMM in order to provide a debugger with additional power and capabilities; whereas this publication is using the functionality provided by a VMX-based VMM in an attempt to prevent a debugger from successfully breaking on the code that it is attempting to debug. If the capabilities presented within this thesis and the publication that is being analyzed were simultaneously loaded into the system, the capability that was loaded first would succeed and the other would fail. This is because a VMX-based VMM provides the ability to virtualize VMX instructions. Therefore, whichever capability was loaded first would have complete control over the other capability since the first would be able to handle VMX events triggered by the second.

<u>MAVMM: Lightweight and Purpose Built VMM for Malware Analysis</u> [5]

Overview

Malware analysis is critical to gain insight into the intention of malicious software, as well as the risks posed by this software. Many current malware analysis techniques are flawed because they run alongside the malware, rather than below – or at a higher privilege. VMMs such as Xen or VMware are currently being used to improve malware analysis capabilities. Unfortunately, general purpose VMMs are designed for performance and functionality, not for malware analysis. Because of this, these general purpose VMMs are now being detected and evaded by modern malware. This paper details a VMM called MAVMM, which has been specifically built for the purpose of malware analysis and also to provide the research community with an easy to modify hardware virtualization framework. The MAVMM environment was built with the following characteristics:

- Hardware Virtualization Technology – Intel or AMD hardware virtualization support is available
    - This decision was intended to increase virtualization performance, and to simplify VMM implementations.
    - The specific implementation of MAVMM used AMD Secure Virtual Machine extensions.
- Special Purpose Hypervisor
    - The author believes that a thin, lean, and simple hypervisor will lead to increased transparency and security.
- Boot-strapped hypervisor
    - MAVMM must be loaded earlier than the software under analysis to ensure that it is running at a higher privilege level.
    - The GRUB boot loader was used to boot strap MAVMM.
- Protected Hypervisor Memory

- A nested paging technique is used to protect the VMM memory. This prevents guest and host physical addresses from being mapped to the same actual address.
- The IOMMU, a hardware virtualization feature, is also used to protect from DMA.

MAVMM provides specific analysis data from target applications including an execution trace, the application's memory page, a log of intercepted system calls, disk monitoring capabilities, and network monitoring capabilities. Information is exported from the guest system using either a USB drive or serial port. These choices were selected to minimize interactions with the guest to avoid detectability.

Relationship to Thesis

Both the MAVMM and the research provided under this thesis are centered on an implementation of a hardware-based VMM. Each technology used a VMM to provide specific control over system events. The MAVMM architecture was mainly intended to provide malware analysis capabilities, where this thesis aims to provide advanced debugging capabilities. Because MAVMM provides a simple implementation of a hardware based VMM, and is built for AMD processors, it could be leveraged as a starting point to port the debugging research provided under this thesis to take advantage of some of the different capabilities provided by an AMD hardware-based VMM, as opposed to the Intel-based VMM used for this thesis.

<u>Debugging operating systems with time-traveling virtual machines</u> [19]

Overview

Traditionally, the user of a debugger will detect an error in program execution and spend a great deal of time working backwards from where the error was detected until the cause of the error is identified. This common approach requires time and re-execution of the same code in order to find the erroneous code location. This approach can be especially problematic when debugging an operating system for the following reasons:

- Operating systems are non-deterministic
    - Their execution and state is effected by a great deal of events that cannot be re-created under the exact same circumstances.
- Operating systems can run for extended periods, making it difficult to re-create the same event under debugging circumstances.
- The act of debugging may have un-intended side effects on the state of the OS.
- Cyclic debugging may cause hardware to fail due to timing issues.

By running an OS inside of a virtual machine, a debugger can examine and control execution of the OS without having such an effect on its state. The Time Traveling Virtual Machines (TTVM) capability will take snapshots on the system during execution so that it can provide the ability to return to the exact state without requiring to re-run large amounts of code, which may have otherwise lead to the aforementioned problems with OS debugging.

Also, using a virtual machine to aid in debugging provides improvements over a traditional debugger as follows:

- Replaying non-deterministic inputs
- Saving and restoring virtual machine state

- No dependency on OS
- More convenient that a remote debugger

The VMM used for TTVM is a modified para-virtualized environment called User-Mode Linux. This environment is similar to host hardware, but not identical. TTVM makes use of this VMM in order to provide the ability to log, replay, and checkpoint the system state so that the debugger can revert to any point in a debugged process. Checkpoints can be used to help isolate where a bug occurred and step back through the exact conditions that led to the bug. Many sources of non-determinism are logged in order to more accurately re-construct the machine state to replay a debug session from a checkpoint. Many aspects of the system being debugged are logged at checkpoints, including CPU register data, the virtual machine's physical memory, the virtual disk, and VMM state or host kernel state that affects the execution of the virtual machine. TTVM made use of the GNU project debugger (GDB) to apply its time traveling concepts to a debug environment. TTVM helps to handle system state changes introduced by GDB such as the injection of the 0xCC opcode for software breakpoints.

Relationship to Thesis

TTVM makes use of a specific para-virtualized VMM for the purpose of adding checkpoints to debugging so that debugged code can be conceptually stepped through in reverse or re-run and debugged under the conditions it was originally executed under. The concepts introduced by TTVM are related to the topic of this thesis because an application of a VMM is being used in order to improve debugging. A major difference between TTVM and the research performed under this thesis is that TTVM operates within a para-virtualized VMM, where as this thesis utilizes a hardware assisted VMM. Using a para-virtualized VMM special care must be taken when interacting with devices requiring a form of hardware I/O such as port I/O or memory mapped I/O. The TTVM paper stated that this could be addressed by either using a software emulator for each specific hardware device or by using a tool to log and replay the execution associated with a device driver.

Ether: Malware Analysis via Hardware Virtualization Extensions [20]

Overview

Ether is a technology that utilizes hardware virtualization to analyze malware, while avoiding detection by the malware itself. This was chosen as a topic of importance because of the magnitude of anti-debugging and anti-VM capabilities contained in modern malware in order to prevent analysis. Detection is avoided by preventing the introduction of side effects into the system that can be identified by malware running within the system. By utilizing hardware virtualization extensions, Ether is able to isolate itself from the target OS environment that malware will be analyzed in, therefore, remaining transparent to the malware.

Using hardware virtualization extensions, Ether achieves this transparency by:

- Maintaining a higher privilege than the malware being analyzed.
- Preventing any side effects from being introduced by Ether into the guest environment.
- Guaranteeing that every instruction executed during analysis by Ether exhibits expected behavior, as it is not being emulated.
- Exception handling does not exhibit any modified or abnormal behavior.

- Timing changes introduced by Ether are masked to the malware.

To create Ether, the developer uses Intel VT in conjunction with the Xen hypervisor. Intel VT was selected because of the abundance of available documentation and hardware. Windows XP was selected as a target operating system for Ether's experiments because it is one of the most widely used modern operating systems, and therefore, a viable target for malware.

Using Ether, a tool called EtherUnpack was created to help unpack executable code that had previously been obfuscated and encrypted to prevent static analysis. Many code packers are capable of preventing dynamic analysis through the use of anti-debugging and anti-VM techniques. By protecting against many anti-debugger and anti-VM techniques that rely on detection methods associated with in-memory presence, CPU register modification, memory protection, privileged instruction handling, Instruction Emulation, and some timing-based detection methods, EtherUnpack was able to successfully assist in the analysis of an abundance of malware sampled from the wild.

Relationship to Thesis

The tools developed under Ether specifically make use of Intel VT in order to isolate an analysis tool from malware so that it can be surreptitiously analyzed. Analyzing dynamic code execution of malware is one of the many tasks that can be accomplished through the use of a debugger. The techniques used by Ether are comparable to those introduced by this thesis because they each utilize strengths introduced through Intel VT for the purpose of dynamic code analysis. A key difference between the research performed under this thesis and the research put into the Ether toolset is that this thesis is centered on utilizing Intel VT to enhance a variety of debugging techniques, while Ether was specifically oriented towards analyzing malware while evading detection.

Evolution in Kernel Debugging using Hardware Virtualization With Xen [21]

Overview

This publication provided insight in how to use Xen to debug a guest operating system without performing any modifications on the guest operating system. Using Xen, it is possible to debug virtually any PC-based operating system during any stage of its execution life, including the boot loader. Breakpoints can also be set without performing modifications to the operating system and its underlying code. Using Xen and Intel's VT, a guest operating system can easily run at its intended privilege level, but still be under control of the host VMM as it is used to transparently handle low-level functionality within the guest. This publication was provided by Intel's Open Source Technology center, and specifically makes use of Intel VMX processor extensions for virtualization support. The gdb and gdbserver-xen tools were used to help provide a debugging environment against the operating system that is running as a guest within Xen. Limitations are introduced through the use of GDB, as it only provided INT 0x03 debug traps and gdbserver-xen does not provide hardware breakpoints, watch points, or single stepping capabilities. A limitation introduced through the use of Xen is the fact that Xen's guest operating systems cannot directly access platform devices, preventing the ability to debug some low-level system software, such as a device driver. If support of Intel VT-d were integrated into Xen for this experiment, it could increase hardware level debugging capabilities.

Relationship to Thesis

The Intel publication makes use of virtualization technology to enhance debugging, as does this thesis. The main difference is that the Intel publication makes use of Xen and GDB to create a specific debugging experiment that cannot fully debug all system-level software/hardware interactions, while this thesis utilizes Intel VT to provide a thin VMM capable of debugging through virtually any x86-based software. Also, the Intel publication introduces the high-level idea of debugger/virtual machine interactions for the purpose of debugging an unmodified guest, whereas this thesis details a large set of functional enhancements that can be introduced through the use of Intel VT. This thesis then proceeds to identify many capabilities that can be provided to a debugger through these enhancements.

Holistic debugging [22]

Overview

The Holistic debugging publication states that software development efforts are constantly growing in size and complexity, increasing the difficulty to test and debug modern software. Common difficulties associated with software debugging include the inability to perform repeatable experiments (non-determinism), simultaneously analyzing distributed systems, and the fact that probing software often induces side effects. Engineers involved in various fields, including software development, are often affected by indeterminism and the probe effect. A common solution for these problems involves the utilization of a simulator to observe repeatable experiments without introducing side effects due to intrusion of the observed system. An issue associated with software simulation is the fact that it only provides information at their abstraction level. Also, simulations allow for repeated deterministic experiments, but the results are not identical to those that would be generated from a real-world system. The holistic debugging concepts presented in this publication run a distributed software system in a simulator and map the low-level data to source-level application data. This provides a programmable and extensible environment that is able to repeatedly observe the system state of real-time distributed systems.

Relationship to Thesis

The Holistic Debugging publication addresses the problems associated with debugging a system that contains non-deterministic properties and where debugging the system can introduce problematic side effects that may lead to an altered system state. These problems are partially alleviated through the use of a software simulator. The issues targeted within the Holistic Debugging publication are also directly relevant to the topic of this thesis.

The approach taken by this thesis has the potential to alleviate some non-deterministic system properties since a VMM has the ability to save and restore system state from a given point in execution. Also, a VMM provides the ability to probe the state of a system without introducing side effects into the system as a result of the probing. When using a virtual machine monitor, simulation of software can also be avoided, providing more realistic debugging experiments. Therefore, the introduction of hardware-based virtualization extensions helps to alleviate the issues identified in this paper, possibly with a much lower level of effort required.

Virt-ICE: Next-generation Debugger for Malware Analysis [23]

Overview

Dynamic malware analysis is a crucial method used to analyze malware. A critical tool in performing dynamic malware analysis is a debugger. Traditional debuggers are easy for malware to detect and also run in the same security domain as the debugger, which makes these debuggers an easy target for malware to tamper with. Virt-ICE aims to provide a debug environment that is invisible to malware so that it cannot be detected, as well as providing a debug environment that resides in a separate security domain so that it cannot be tampered with. This is accomplished by using virtualization technology to run malware inside of a VM. In addition to the aforementioned benefits provided by Virt-ICE, running malware inside of a VM protects the physical system from infection and saves time by providing an environment that can easily be reverted to a clean state after malware analysis is complete. Since the Virt-ICE debugging environment makes use of a virtual environment, it is possible to provide event-based control to the debugger, including the following:

- Before and after breakpoints
- Physical memory access monitoring, as opposed to virtual memory monitoring
- Interrupt monitoring
- I/O event monitoring

Virt-ICE was implemented through modifications to the Qemu emulator. These modifications were made to improve the debugging facilities provided by a current VM environment.

Relationship to Thesis

The idea behind Virt-Ice, using a virtual machine for malware analysis to obtain an advantage during dynamic debugging, is similar to that of this thesis. The Virt-ICE author even identified that a virtual environment could be used to provide event-based control over the guest environment. As opposed to this thesis, Virt-ICE is mainly interested in the advantages that a VM can provide to a malware analyst. Also, the technology being researched under this thesis is using pure hardware-assisted virtualization as opposed to an emulator.

Dynamic and Transparent Analysis of Commodity Production Systems [24]

Overview

This paper proposed a framework to aid in the dynamic analysis of production systems by leveraging hardware virtualization technology. This was accomplished by installing a VMM and migrating the system into a virtual machine. This framework will provide transparency to, and isolation from, the system being analyzed. The framework built by the authors of this paper was called HyperDbg. HyperDbg is a kernel debugger used to debug any critical kernel component.

The author stated that dynamic operating system analysis could be performed through the use of both kernel-based and VMM-based tools. Kernel analysis tools are described as modifications to the kernel to intercept events of interest through the installation of hooks into the kernel. These hooks are used to monitor run-time events, and may be difficult to place into operating systems that do not provide native support for dynamic analysis. VMM-based dynamic analysis attempts will typically run the target software in a virtual machine and respond to events of interest from the VMM. Although the standard VMM-based analysis approach provides transparency, it requires that the system be run as a guest virtual machine, or it cannot be analyzed. This can be even more problematic when components need to be analyzed at the hardware level, since a great deal of these interactions will be virtualized.

A framework was provided by this research that combines a VMM and analysis tools to aid in the analysis of a production system. The main advantages identified and provided through this approach include a debugging capability that is:

- Transparent to the guest system
- Does not require a user to recompile or reboot the target system
- Does not require the guest OS to be pre-installed as a VM
- Is nearly OS independent
- Prevents errors in the analysis tool from affecting the guest system

Relationship to Thesis

This publication contains concepts similar to this thesis because it makes use of hardware-assisted virtualization technology to aid in debugging, although the capabilities and advantages sought by this publication diverge from those identified in this thesis. This publication mainly seeks to create a framework that provides inherent advantages to a guest OS being debugged by running the analysis tools in an isolated VMM. This thesis seeks specifically to enhance debug capabilities by using exit conditions analogously to breakpoints in order to provide a much more powerful and extensive set of breakpoint capabilities. The difference between this thesis and the publication being reviewed is especially demonstrated by the technique that the publication uses to implement multiple simultaneous breakpoints. The publication makes use of software breakpoints for this concept, whereas one of the advantages of the virtual breakpoints presented within this thesis is the ability to set nearly unlimited virtual breakpoints without modification to the target software, even for hardware-based events.

Although it was not the main focus of this thesis, it is important to point out that the debugging capabilities provided by this thesis also share the inherent advantages of running a debugger from a VMM, as was identified within the Dynamic and Transparent Analysis of Commodity Production Systems publication.

Virtdbg - Using virtualization features for debugging the Windows 7 kernel [25]

Overview

VirtDbg is a code project that shares similarities with the topic of this thesis. At the time that this thesis was written, VirtDbg was an incomplete open source project, but I wanted to make note of its existence due to the significant underlying similarity that VirtDbg makes use of Intel VT for debugging purposes. Recently, VirtDbg was also presented at the Recon 2011 conference.

VirtDbg's purpose is to implement a kernel debugger with the assistance of the hardware virtualization technology provided by Intel's VT-x. Its main purpose is stated to provide a debugging environment for 64-bit versions of Windows 7 where available kernel debuggers are extremely limited. Using VirtDbg will allow a user to debug protected parts of the Windows OS, such as PatchGuard, that would otherwise be difficult to debug through using available debuggers. A slide show presentation about VirtDbg was given at the Recon 2011 conference, which outlines the difficulties performing dynamic analysis on the Windows 7 operating system. The author notes that this can be especially difficult with features such as patchguard enabled, since these features will prevent direct manipulation of critical kernel areas, which are required for use by a kernel debugger. Use Intel VT-x, a debugger can still debug parts of the operating system that would otherwise be prevented.

The VirtDbg code base is currently in a "very alpha state" and does not yet have a great deal of documentation or functionality provided beyond the main concept of its implementation; a debugger using the assistance of Intel VT-x based technology. The current VirtDbg code base can be located at http://code.google.com/p/hyperdbg/.

Relationship to Thesis

The VirtDbg project is related to the thesis because it is harnessing Intel's VT-x capabilities for the purpose of creating an advanced debugging capability. The main difference is that VirtDbg aims to debug a specific part of the MS Windows 7 operating system that is difficult to analyze due to strong security features, whereas the capabilities presented within this thesis aim to enhance the overall usefulness and power of a kernel debugging capability by providing more fine grained control over the debugging process. Although not the primary intention, the debugging features presented by the VirtDbg author can be realized by the capabilities presented under this thesis because the isolation and high privilege level of a debugger running within an Intel VT VMM will still be realized.

Hardware Assisted Virtualization - Intel Virtualization Technology [17]

Overview

This paper provides a well-organized, condensed version of the Intel VT-x information that can be found in the Intel Developer's Manuals. Some of the challenges required to provide virtualization on the Intel architecture are addressed, leading into the need for Intel's VT-x technology. A thorough explanation of the various aspects of VT-x is provided, organizing the data in a format that appears helpful to a developer creating or extending an Intel VT-x based hypervisor. The information in this paper can be obtained from the Intel Developer's Manuals, but the paper is organized so that various VT-x concepts can be quickly located and interpreted in a clear and understandable format.

Relationship to Thesis

This publication provides a great overview of the various capabilities that can be obtained through the use of Intel VT-x. The development required to prove the ideas behind this thesis was centered on Intel's VT-x using Intel VMX capabilities. This paper has helpfully summarized many of the development concepts required to build virtual breakpoints through the use of VM-Exit Reasons.

Appendix E – Bibliography

[1] Intel Corporation (2011). *Intel® 64 and IA-32 Architectures Software Developer's Manuals Volume 1 - 3B*. Retrieved from http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[2] Popek and Goldberg virtualization requirements. (n.d.). Retrieved July, 2011 from Wikipedia: http://en.wikipedia.org/wiki/Popek_and_Goldberg_virtualization_requirements

[3] Virtualization. (n.d.). Retrieved July, 2011 from Wikipedia: http://en.wikipedia.org/wiki/Virtualization

[4] Full Virtualization. (n.d.). Retrieved July, 2011 from Wikipedia: http://en.wikipedia.org/wiki/Full_virtualization

[5] Nguyen, A., Schear, N., Jung, H., Godiyal, A., King, S., and Nguyen, H. (2009). *MAVMM: Lightweight and Purpose Built VMM for Malware Analysis*. ACSAC 2009.

[6] Hardware Virtualization. (n.d.). Retreived July, 2011 from Wikipedia: http://en.wikipedia.org/wiki/Partial_virtualization#Partial_virtualization

[7] McCarty, R. (2007). Paravirtualization explained. Retrieved from: http://searchservervirtualization.techtarget.com/tip/Paravirtualization-explained

[8] VMware Incorporated (2007). Understanding Full Virtualization, Paravirtualization, and Hardware Assist. Retrieved from http://www.vmware.com/resources/techresources/1008

[9] Operating system-level virtualization. (n.d.). Retrieved July, 2011 from Wikipedia: http://en.wikipedia.org/wiki/Operating_system-level_virtualization

[10] Application virtualization. (n.d.). Retrieved July, 2011 from Wikipedia: http://en.wikipedia.org/wiki/Application_virtualization

[11] Memory Virtualization. (n.d.). Retrieved July, 2011 from Wikipedia: http://en.wikipedia.org/wiki/Memory_virtualization

[12] Cook, C. (2008). Memory Virtualization, the Third Wave of Virtualization. Retrieved from: http://vmblog.com/archive/2008/12/14/memory-virtualization-the-third-wave-of-virtualization.aspx

[13] Distributed file system. (n.d.). Retrieved July, 2011 from Wikipedia: http://en.wikipedia.org/wiki/Distributed_file_system

[14] IBM Corporation (2003). Storage Virtualization Technology. Retrieved from http://www-03.ibm.com/systems/resources/systems_storage_software_virtualization_tutorial_booklet1.pdf

[15] Emulator. (n.d.). Retrieved July, 2011 from Wikipedia: http://en.wikipedia.org/wiki/Emulation_%28computing%29

[16] Wilson, R. (2009). *Virtual Machine Extensions' Contributions and Interactions with Kernel Debugging Technology*. (Unpublished independent study). Syracuse University, Syracuse New York.

[17] Zabaljauregui, M. (2008). *Hardware Assisted Virtualization. Intel Virtualization Technology*. University of La Plata, Buenos Ares, Argentina.

Can be retrieved at: http://linux.linti.unlp.edu.ar/images/f/f1/Vtx.pdf

[18] Tengfei Yi, Aijun Zong, Miao Yu, Shang Gao, Qian Lin, Peijie Yu, Zhong Ren, and Zhengwei Qi (2009). *Anti-debugging Framework Based on Hardware Virtualization Technology*. 2009 International Conference on Research Challenges in Computer Science.

[19] King, S., Dunlap, G., and Chen, P. (2005). *Debugging operating systems with time-traveling virtual machines*. USENIX 2005.

[20] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. (2008). *Ether: Malware Analysis via Hardware Virtualization Extensions*. ACM CCS 2008.

[21] Kamble, N., Nakajima, J., and Mallick, A. (2006). Evolution *in Kernel Debugging using Hardware Virtualization With Xen*. 2006 Linux Symposium.

[22] Albertsson, L. (2006). *Holistic debugging*. MASCOTS 2006.

[23] Nguyen A. and Suzaki, K. (2010). *Virt-ICE: Next-generation Debugger for Malware Analysis*. Blackhat USA 2010.

[24] Fattori, A., Paleari, R., Martignoni, L., and Monga, M. (2010). *Dynamic and Transparent Analysis of Commodity Production Systems*. ASE 2010.

[25] Aumaitre, D. (2011). *Virtdbg - Using virtualization features for debugging the Windows 7 kernel*. Recon 2011.

[26] Intel Corporation (2011*). Intel® Virtualization Technology for Directed I/O - Architecture Specification*. Retrieved from: http://download.intel.com/technology/computing/vptech/Intel%28r%29_VT_for_Direct_IO.pdf

NAME OF AUTHOR: Ryan M. Wilson

PLACE OF BIRTH: Keene New Hampshire

DATE OF BIRTH: April 24, 1983

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

Clarkson University, Potsdam, New York

DEGREES AWARDED:

Bachelor of Science in Computer Engineering, 2005, Clarkson University

PROFESSIONAL EXPERIENCE:

Senior Computer Engineer, Assured Information Security, Inc., 2006 - Present

Special Thanks to:

My thesis advisor Dr. Jim Fawcett, mentor Dr. Joseph Sharkey, colleague/friend Paul Petzke, and my family - Nicole, Kayden, and Ashton.