Automated Forensic Techniques for Locating Zero-day Exploits

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

by

Stephen Kuhn

Thayer School of Engineering

Dartmouth College

Hanover, New Hampshire

December 2013

Examining Committee:

Chairman_____
Stephen Taylor, Ph.D.

Member_____
Eugene Santos, Ph.D.

Member_____
George Cybenko, Ph.D.

Member_____
Sarah Muccio, Ph.D.

_____
F. Jon Kull
Dean of Graduate Studies

Abstract:

This thesis approaches the problem of locating zero-day exploits used in network attacks. The hypothesis is that the advent of high-performance virtualization presents a unique opportunity to both discover these events and increase the attacker's workload. Much of the traffic in modern computer networks is conducted between clients and servers, rather than client-to-client. As a result, servers represent a high-value target for collection and analysis of network traffic. The observe, orient, decide, and act (OODA) loop for computer network attack involves *surveillance*, to determine if a vulnerability is present, selection of an appropriate exploit, use of the exploit to gain access, and *persistence* for a time sufficient enough to carry out some effect. The time spent in surveillance and persistence may range from seconds to months depending upon the intent of the attacker. In contrast, exploitation of the system can occur in milliseconds. The difficulty in generating a suitable exploit and its potential for reuse, dictates that an attacker's first on-host action is likely to be the removal of any on-host trace associated with the exploit. Therefore, the first notice that an intrusion has occurred may well be several months later when an effect is eventually perpetrated. The intervening period is populated by terabytes of network traffic, reboots, upgrades, and changes of operating system state obfuscating the analysis after the fact.

This thesis approaches the problem of locating the initial exploit through a novel coarse-grained forensics technique facilitated by virtualization. This new capability simultaneously increases the attacker's workload associated with conducting surveillance and maintaining persistence on host systems. In addition, it provides mechanisms to identify network traffic corresponding with the initial intrusion and any subsequent

communication within the operating system and within the network. These goals are accomplished by enhancing non-determinism in operating systems: utilizing a *hypervisor* to refresh micro kernel's and introspection techniques that enable the hypervisor to peer into a running microkernel observing its state and recording actions of interest for later analysis.

Acknowledgments:

I would like to thank the people who have supported my journey through graduate school at Dartmouth. Special thanks to my adviser Dr. Stephen Taylor for his guidance and encouragement in this research. I am grateful to have a mentor with such vast experience and insight to support me through this process. I am especially thankful for his endless patience and support while I recovered from my back injuries. I would also like to thank Dr. Mark Borsuk and Dr. Sergey Bratus for their insights through teaching statistics, computer science, and many lunch discussions that opened my mind to new ways of thinking and furthered the quality of my research. Thanks also to the members of my dissertation committee, Dr. George Cybenko, Dr. Eugene Santos, and Dr. Sarah Muccio. Their contributions to this project have enhanced the quality and significance of my research in the field.

This work was not completed in a vacuum. I would like to thank my research group at Dartmouth: Kathleen McGill, Colin Nichols, Morgon Kanter, Mike Henson, Rob Denz, and Jason Dahlstrom. I worked with many brilliant individuals who broadened the value of my work, Raphael Mudge, Dan Schofield, and to those I failed to mention here I am eternally grateful. Through the various discussions, presentations, insights, and late night calls or chats, these colleagues and friends have increased my awareness and understanding of computer science and engineering as a whole. Their friendship and camaraderie provided support throughout the setbacks, challenges, and achievements of graduate education. Additional thanks to the staff at the Thayer School of Engineering,

especially Karen Thurston and Daryl Laware, for their continuous help in navigating the administrative and logistic details of pursuing a Ph.D.

I would like to acknowledge the Air Force Research Lab (AFRL) and Defense Advanced Research Projects Agency (DARPA) for their sponsorship of this project. This research was partly sponsored under agreement numbers FA8750-09-1-0213 and FA8750-11-2-0213.

I would not have contemplated this journey if not for my parents, Karen and Daniel who instilled within me a love of technology, the curiosity to ask why, and fortitude to continue in spite of challenges. Words cannot express my gratitude for the sacrifices they made for my education. My brother Patrick, and my Aunt Betty for their support throughout this process and the years, without your love and kindness I would not be the person I am today.

To Morgan thank you for being there with me and for all your support throughout these years. I simply could not have completed this endeavor without your love.

Contents

List of tables:

List of figures:

Chapter 1: Overview

1.1 Problem Statement:

How can software locate zero-day exploits when an attacker covers their tracks by deleting on-host evidence associated with the initial intrusion, in an attempt to hide their presence while gathering intelligence?

1.2 Hypothesis:

Forensic techniques based on virtual machine introspection, will enable process actions to be correlated with recorded message traffic, allowing rapid identification of exploits by substantively reducing the volume of traffic to be analyzed. These techniques will also serve to increase the incidence of attacks required to maintain a persistent presence during the intelligence gathering.

1.3 Threat Model:

The threat model used in this thesis involves intrusions employing remote control as outlined in Figure 1. It is comprised of several steps including *surveillance* to determine if a vulnerability exists (1), use of an appropriate ***exploit*** or other access method (1), privilege escalation (2), removing exploit artifacts, and hiding behavior (3,4). Surveillance may involve obtaining a copy of the binary code and using reverse engineering (5,6) or fuzzing (7,8) to facilitate a broad range of attack vectors including return oriented programming (9,10). The implant then *persists* for a time sufficient to carry out some malicious effect, obtain useful information, or propagate intrusion to other systems.

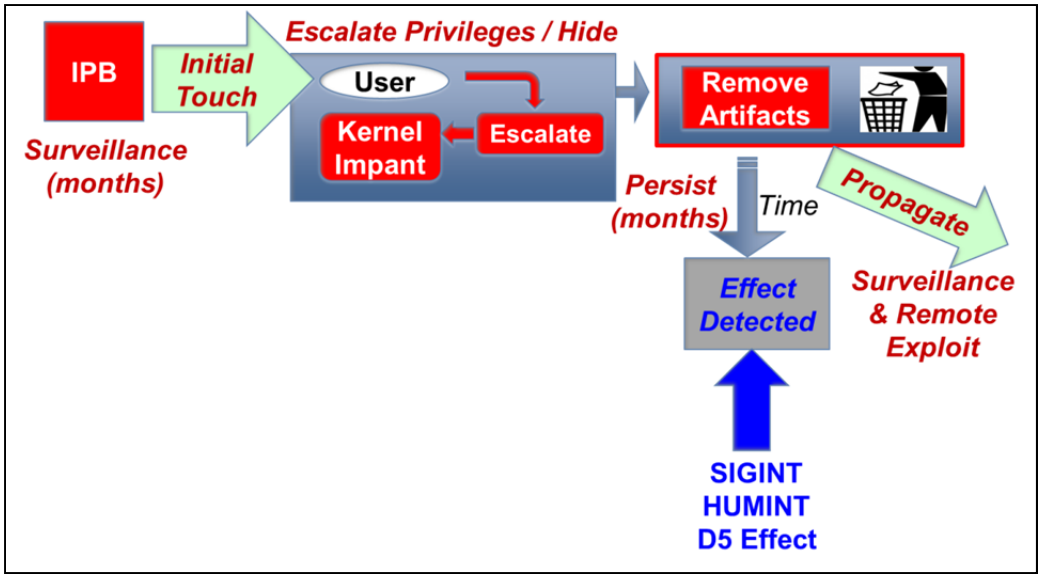


Figure 1: Threat Model for Intrusions with Remote Control

There are many methods and locations in which it is possible to embed code and maintain persistence. A well-know technique is to use a kernel-level root-kit which has the ability *hide its own presence* (3). Removing all trace of the original exploit, used to gain access, is well within the capabilities of such techniques and can be expected since exploits are high-value, perishable assets: their discovery allows a defender the opportunity to pursue countermeasures and analysis of actions taken by the malicious code. Unlike the time to execute an exploit, the time spent in surveillance and persistence may range from minutes to *months or even years* depending upon the intended effect. Moreover, the presence of an intrusion may *never* be detected by network defenses but instead may be recognized indirectly either due to a deviation from expected behavior, or may be derived from intelligence sources. Exploits therefore represent a difficult to discover key to securing a system from future attack.

This intrusion process might be employed in a combat strategy similar to the Observe Orient Decide Act (OODA)-loop model, illustrated in Figure 2, and originally developed to increase success in aerial combat (1). The key tenant is that he who completes the loop faster, gains the best situational awareness, enabling them to make higher-quality decisions based upon better intelligence, thereby increasing the likelihood of success. Drawing on parallels with cyberspace, it translates into collecting information about a target system or network (Observe), analyzing traffic to identify system components and vulnerabilities (Orient), employing an appropriate *exploit* to gain access, embedding code within the target system to maintain and obscure a persistent presence (Decide), and after some period of time (*possibly months*) performing an action or effect (Act), such as extracting or destroying data.

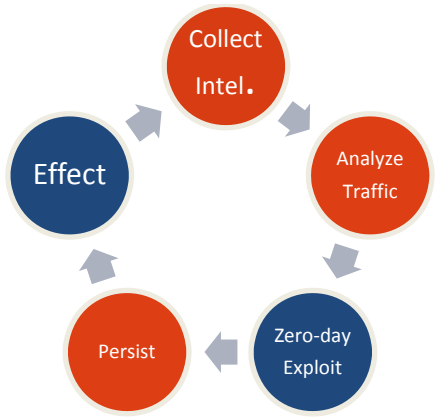


Figure 2: Cyber OODA Loop

The focus of this thesis is on ***servers***, *which* present high-value targets in modern client-server network architectures and play a key role in employing the OODA-loop. Servers typically remain active on networks for years, potentially decades; typically residing at a single static network location (i.e. IP/MAC address). Consequently, they present a stable

platform to observe and attack, form a natural peering point for network traffic analysis used in surveillance of other systems, and provide a stable pivot point to pursue alternate attack paths. As they age, modifying them to add additional security protections is an untenable task (11); even determining the risk that such systems pose can be a challenge (12). Regrettably, despite their key role, in recent years many within the security community have shifted attention to protecting clients (13).

Unfortunately, today's defense mechanisms, focused on detecting intrusions, have proven inadequate to confronting this threat model. At the network level, they include intrusion detection systems (14), firewalls (15), and virus scanners (16). In corporate and military environments additional host base systems are commonly deployed such as virus scanners, root-kit detectors (17)(18,19) and more recently website application firewall software (20)(21). These technologies share a common approach using *signature based detection*. Unfortunately, this method has limited capability to prevent infection from a previously unobserved or "zero-day" exploit. Although anomaly detection technologies (22) exist to fill the gap, *not all anomalous events are malicious, and not all malicious events are anomalous*. The former observation results in high-false alarm rates, the latter results in low detection rates. As a result, persistent malicious code may never be detected, and if it may be difficult to disambiguate.

1.4 Approach:

The approach to forensics explored in this thesis is based upon recent advances in virtualization technology, now available in commodity processors used in the design of high-performance servers. This technology provides the ability to run one system inside another through sophisticated memory address translation, protection, and isolation capabilities. The inner system, often called a *hypervisor*, has the ability to introspect into a virtual machine, running on top of it, and observe its state. Further, a hypervisor is able to set control conditions associated with a virtual machine, such that when protected resources are accessed, the hypervisor is notified. Unfortunately, to maintain an account of malicious process actions, current hypervisors track every byte in memory transactions, significantly reducing system performance. To alleviate this burden, this thesis explores novel course-grain, low overhead, tracking technology that can be incorporated within message-passing microkernels. The technology lowers the recording burden by storing only those actions designating the *trail of progress* that can potentially originate from an exploit. This trail enables a new course-grained forensics technique for exploit discovery.

Hypervisors also afford the ability to non-deterministically restart virtual machines to a *known gold-standard* state, thereby denying kernel-level persistence over long periods. All attacks, under the threat model, seek the highest level of privilege for the purpose of hiding; therefore owning the base of the software stack is of critical importance. Gold-standard hypervisors *implemented through a read-only software*, particularly those with small code foot-prints and low incidence of vulnerabilities, offer a mechanism to reliably extend trust from the hardware into trust in system software. Trust can then be extended

to a micro-kernel though refresh: by restarting the kernels virtual machine. This facility continually removes an attackers presence on a system, even if it is undetected, forcing the attacker to continually renter their OODA-loop and increasing their workload. One attribute of this approach is that the new server may be subtly different from the old, this *diversity* is intended to disrupt any re-attack. Unfortunately, servers are particularly difficult to refresh, as they often support real-time applications typified by *streaming-video*. The ability to refresh a server presents a further opportunity – to disrupt surveillance – by changing the network properties of a replacement virtual machine such that it appears in a completely different location on the network, behind what are potentially different defensive postures. This offers the opportunity to elevate protection in response to risk or mission objectives.

This thesis achieves these goals through the following core technologies:

- **Memory Introspection**: A memory translation technology that makes it possible to observe arbitrary memory locations within a running microkernel from the hypervisor and decode the resulting information. This represents an enabling technology for new tools that are able to decode the content of memory, monitor all running processes, and/or unwind the function call stack. All of these higher-level tools can operate from the safety of an outside observer, the hypervisor, to prevent tampering with the results by any malicious code present in the kernel or higher layers of the system.

- **Event Tracking**: A low-overhead tracking technology that provides detection and recording for events of interest for later analysis. This technology is not provided by

hardware on Intel 64-bit architectures. Events that are of particular interest in exploit discovery are: process creation and inter-process communication.

- **Network Tracking**: A low-overhead tracking technology that allows course-grain recording of network events. The record of these events provides the ability to correlate process creation history with network traffic/messages.

- **Automated Exploit Discovery**: An automated, high-speed correlation technology that combines the history of process creation with the history of network traffic. This technology rests upon a unique database addressing scheme that facilitates identification of all packets that may have been associated with an intrusion, assuming an anomalous event has been identified. It also illuminates what actions were taken by the attacker on the system after the intrusion.

- **Network-Hiding Hypervisor.** A novel hypervisor design that hides its location in the network to deny surveillance while non-deterministically refreshing kernel-level trust by discarding microkernels. A unique attribute of this design is that it allows servers to be *relocated* and *refreshed* in the presence of real-time access. The concepts have been demonstrated for the particularly difficult case of a web-server that maintains connections to static, streaming, and dynamic content while being continuously refreshed.

1.5 Contributions:

The primary contributions of this thesis are:

- A novel forensic hypervisor design that increases attacker workload by denying surveillance using network hiding, and denying persistence by continuously refreshing kernels. This work was published in: Kuhn, Stephen, and Stephen Taylor. "Increasing attacker workload with virtual machines." *MILITARY COMMUNICATIONS CONFERENCE, 2011-MILCOM 2011*. IEEE, 2011.

- Operating system mechanisms to associate messages with *process genealogy*. This work involved novel techniques for observing virtual machine state, recording and correlating recorded network traffic. Kuhn, Stephen, and Stephen Taylor. "A forensic hypervisor for process tracking and exploit discovery." *MILITARY COMMUNICATIONS CONFERENCE, 2012-MILCOM 2012*. IEEE, 2012.

- Exploit discovery techniques and algorithms based on indexed, course-grain *tracking* of processes and *elimination* of messages. The latter significantly reduces the search space associated with exploit discovery.

- An experimental study of forensic exploit discovery that locates test exploits against a known ground truth and quantifies the overheads associated with introspection, tracking, and traffic elimination.

1.6 Analysis and Metrics:

These contributions have been demonstrated through exemplars and proof of concept implementations:

- The forensic introspection techniques have been incorporated into a from-scratch hypervisor design developed within the research group at Dartmouth College. These capabilities include the ability to successfully decode memory locations in the kernel, inspect both active and suspended processes, and present a history of the function call stack. Benchmarking the performance impacts associated with forensic introspection is a tractable analysis and the results section of each chapter discusses the impact of each techniques.
- The exploit discovery capability has been demonstrated through practical scenario's using large-scale experiments that employ the LARIAT network traffic generator. These experiments served to quantify the reduction in the amount of traffic, which must be examined during forensic analysis. The experiments also verified the ability to view process history and track inter process communication.
- The network hiding hypervisor design was implemented in Linux leveraging the KVM virtual machine package for virtualization. It was verified using the market dominant Apache web server and across multiple Linux distributions to provide diversity. The network hiding concepts have been incorporated into a novel from-scratch hypervisor design within the research group.

Unfortunately, no practical technique has yet emerged to assess the impact on attacker workload induced by network hiding and refresh. Clearly it is dependent on a large

number of, sometimes subjective, factors: the skill of the attacker, the presence of unknown vulnerabilities, the available attack vectors (insiders, RF, local access, remote exploits), and the amount of manpower or other resources that the attacker may expend.

1.7 Thesis Organization:

Chapter 2 lays a foundation discussing the primary ideas, and the related research in the area of forensics and process tracking for exploit discovery. It summarizes the current state of practice and the presents the work summarizing the performance impacts of the existing approaches.

Chapter 3 introduces the hypervisor and kernel developed, as part of this thesis. It discusses the core design philosophy and the introspection techniques. This chapter discusses the details of the hypervisor mechanisms necessary to understand modifications for performing analysis and correlation of messages.

Chapter 4 describes the research, background, and challenges of incorporating process recording into the hypervisor. In addition, this chapter details the research and background associated with network tracking.

Chapter 5 presents an experimental study of forensics based on common scenarios.

Chapter 6 describes the network hiding and trust-refresh techniques and associated results.

Chapter 7 concludes the thesis, discusses directions for future research, and lessons learned.

Chapter 2: Related Research in Virtualized Forensic Acquisition

2.1 Primer on Virtualization

The forensic challenge of collecting and understanding memory use in analyzing computer network attacks has been the focus of numerous studies (23-27). The ability for multiple virtual machines to execute on a single physical machine, with one virtual machine observing the other, has made virtualization an attractive avenue for performing forensic analysis in real-time. This new field of forensics is known as Virtual Machine Introspection (VMI) (28,29). The primary challenge is concerned with tracking and accounting for memory use: The content of memory is obscured by the introduction of an additional layer in the memory hierarchy, needed to implement the virtual machine abstraction and secure virtual machines from one another. This additional layer is illustrated in Figure 3. At the base of the hierarchy, physical memory is allocated by the hypervisor. Virtual machines execute inside independent contiguous virtual memory spaces. Applications then execute on top of virtual machines, in their own contiguous spaces, mapped to those of the underlying virtual machine memory. At each layer virtual memory structures are typically implemented through paging in the traditional manner, making the contiguous memory spaces appear continuous to each layer (30)(31).

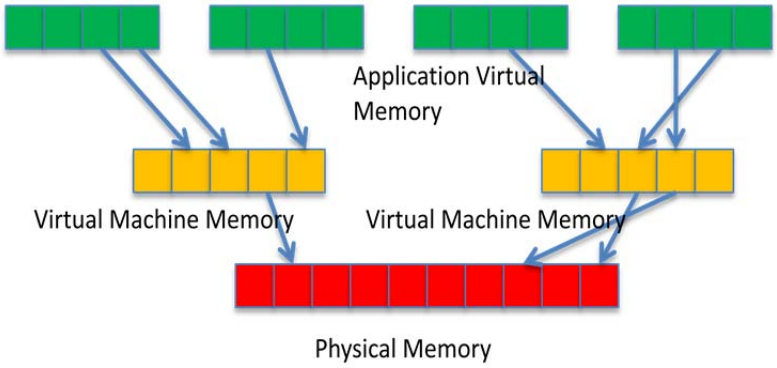


Figure 3: Memory Virtualization support for virtual machines

To understand the contents of memory the virtual memory layers must be directly accounted for in forensic analysis. If a forensic tool were to collect the memory content without first applying the appropriate translations, it would have no frame of context and the information retrieved would appear as random data. This *semantic gap* defines the difference in view from inside and outside an executing virtual machine (32).

A second challenge in forensic analysis is assuring the accuracy of the information obtained. Forensically collecting the content of memory from inside an untrusted kernel decreases the reliability of the evidence by virtue of the presence of malicious code (33). Root-kits and other malware seek to obscure their presence by manipulating the results from standard collection tools though providing false information (34). Typical examples include, modification of the active process queue to ensure that malicious processes are hidden, alteration of directory listings to hide files, modification of program execution flow to inject malicious code paths, and modifying standard memory locations to hide code, transposing memory types such as instruction with data, to advert detection by memory scanners.

The prevailing thought in hardware approaches to forensics (33), (35) is to use Direct Memory Access (DMA) to retrieve the contents of memory. DMA allows peripheral hardware to directly access main memory and bypass the processor, saving time and improving performance (36). In theory, this prevents any malicious code from modifying the information reported, as it never passes through the processor. Sadly, malicious software approaches have been demonstrated that circumvent DMA collection by manipulating the configuration of a memory controller (37)(38).

The creativity of attackers, the constant evolution of hardware and software, and the use of legitimate access by insiders, have all served to inject new vulnerabilities that continually invalidate the assumptions of forensic tools. Virtualization provides a unique opportunity to observe memory from the isolated and protected environment of the hypervisor, provided that its physical memory is not accessible from the virtual machines that execute upon it. Hypervisors also allow forensic analysis of additional information, such as the content of registers, which are unavailable to external hardware observation methods. In general, hypervisors provide a smaller, more stable code base, presenting a smaller attack surface, and opening the potential for formal verification (39)(40). Finally, hypervisors do not require the deployment of new hardware as most systems today include support for virtualization.

2.2 Combating and Analyzing Malicious Code with Virtualization

Traditionally the forensic process is defined by discrete steps, including the acquisition of data, extraction of sought after information and analysis to synthesize the results. However, to achieve the required synthesis in time, as tools become more mature they

incorporate solutions of each phase into a single application. Conversely, nascent techniques tend to focus exclusively on emerging forensic applications or research. This results in a convoluted picture of the problem space in regards to forensic tools. Consider the challenge of detecting root-kits, which are programs that seek to conceal themselves from detection. Detecting new root-kits requires novel analysis techniques, but underneath nearly all the detection methods is some form of forensic technology that is analyzing the system to detect them. Further complicating the issue, the term forensics is used interchangeably, as a description of acquisition techniques and analysis methods. In addition, because root-kit detection requires novel parts of both aspects of forensics, literature on root-kit detection must be considered when surveying the current state of forensics. The goal of this chapter is to present a survey existing forensic tools, restricted in scope to those that rely on virtualization.

In an effort to disrupt the continual arms race between exploitation, detection, and mitigation, virtualization has emerged as the dominant tool. Virtualization research and techniques have emerged for analyzing new malicious code and detecting as well as defending against novel attacks. Traditional tools aimed at analysis of operating systems are unable to resolve the correspondence between processes executing on virtual machines and their allocated memory. The introduction of rootkit technologies, providing the ability for malicious code to hide its appearance and actions further complicates memory analysis.

2.3 Hardware-based Acquisition Techniques

The problems associated with collection in the presence of malware, have led to the conviction that independent hardware, free from tampering, is required to observe the true content of memory. These approaches are rooted in early work to provide file system integrity through the use of peripheral hardware cards (41). This work introduced the idea of an *independent auditor* trusted to oversee actions in the system being monitored. The card is assumed to be installed prior to any malicious event and has three primary modes: *management*, *running*, and *alarm*. The management mode could conceptually be accessed only at boot time from a secure interface to prevent API attacks (42). The running mode is principally used for monitoring. Finally, the alarm mode is triggered by predefined policy violation events that occur during the running mode. The card logs events over an out-of-band channel to a secondary computer for later inspection. Since the auditor stores all logs on an out-of-band system, they are assured to be free from tampering and can be analyzed without affecting the performance of the system under observation. Audits may be conducted without a specific alarm to serve as a reference state for future comparisons. Logs provide an entry point for post-attack forensic analysis providing valuable information to help reconstruct damaged files. Since advanced threats are likely to delete actions taken by the initial exploit, having a snapshot of the unmodified file system state allows an analyst to reconstruct the events, which occurred just after infection. Unfortunately, this approach does not capture changes to memory that do not involve file access.

Copilot (35) extended this early research into memory forensics by using DMA to retrieve the contents of the memory subsystem. Hashes of critical kernel structures and

memory regions are compared against critical structures in memory to detect changes. This general approach is not tied to the signature of any specific attack, but rather to unexpected changes to kernel structures allowing copilot to detect modification. Although DMA returns physical memory addresses, operating system kernels use virtual memory addresses. Copilot provides a reverse translation based on static offsets to obtain information concerning the statically mapped kernel structures in Linux. To obtain information associated with dynamically loaded kernel modules, Copilot monitors page translation tables. Operating systems that mix code and data within a single page prevent Copilot from computing hashes of all kernel structures. Instead, Copilot monitors locations where jump instructions are likely to be added by taking a hash of the system call table, a popular first target for root-kits. These problems may be ameliorated by enforcing the separation of code and data, with the appropriate execute-only and read/write protections (43). Support for this separation is now available in the paging structures associated with current processor designs (44).

In (33), Carrier and Grand have presented an alternative memory acquisition device. Their solution utilizes a secondary microprocessor on a PCI card that connects to external storage via an out-of-band channel and uses DMA to retrieve the contents of memory. The program code of this card resides in Read Only Memory (ROM) to prevent tampering. Although the card is able to successfully capture system memory, it is unable to access those sections of memory reserved for the Video card and BIOS; this would violate memory protections and cause the system to crash. The card has no facility to detect root-kits, but may provide the basis for out-of-band forensics.

In (45), Baliga, Ifode, and Chen have developed another hardware proof-of-concept card for rootkit detection using *system invariance*. The authors have adapted the Diakon software package to guess likely program invariants, characteristics of the program that do not change, from multiple runs (46). In contrast to Copilot that utilizes hashing to detect changes in *known* kernel structures, this system detects violations of the observed invariants. The associated hardware card allows their software package to analyze kernel memory for root-kits or malicious code on a separate machine. Offloading the workload to a separate machine minimizes performance impact to the host system and provides anti tampering capability. This presents a potential solution to the challenge of pages that mix code and data. The software is able to detect fourteen publically available rootkits as well as two proposed in the literature**.**

Unfortunately, it has been demonstrated that DMA based hardware forensics can be deceived by manipulating the configuration of the memory controller (37). In addition, the prohibitive costs and delays associated with the approach makes it difficult to maintain as a viable solution. The costs to add new hardware to every system quickly scale out of budget and requiring a second system per instance for analysis further drives up costs in fiscally constrained times. Further, hardware approaches have not been designed as general-purpose forensic packages allowing an analyst to inspect memory for other purposes, such as evidence collection by law enforcement or counter-terrorism. As a result, these tools are not sufficient to discover the chain of events associated with an exploit, evidence may be deleted before detection occurs.

2.4 Virtual Memory Introspection

Three general approaches to using virtualization technology for forensic memory analysis have appeared in the literature. One method is to provide plug-in modules for generic hypervisors, or alternatively custom-built hypervisors, to provide a general-purpose *forensic toolkit.* Another approach is to use a hypervisor as simply a *secure platform* to execute traditional security tools, such as network intrusion detectors or virus scanners, but observe a virtual machine instead of the application code running on it. Finally, other projects use a *customized hypervisor* for detecting root-kits and malicious code.

2.4.1 Introspection Toolkits

General solutions for resolving the semantic gap between physical memory, virtual machine memory, and application memory have already been developed and are available. Implementations include *XenAccess* (47), MAVMM (48), VMware's vprobes tools, and Virtuoso' (49).

XenAccess provides a programmatic interface that can extract virtual machine information to an underlying hypervisor. The information may include the installed kernel modules, which processes are running, symbol table addresses, the location of a specific symbol, and virtual memory content. This information is accessed directly by the hypervisor rather than requesting the virtual machine to report the information preventing the virtual machine from either having knowledge of the actions of the hypervisor or attempting to deceive it. The XenAccess framework has been integrated with the Volatility memory forensics toolkit to provide a generic solution for hypervisor

memory forensics. Unfortunately, the XenAccess API only operates with the Xen hypervisor.

VMware has released an interface similar to XenAccess called vprobes. The interface uses a C like programming language called Emmett for programmatic access. Alternately, a scripting language is provided to access the information about the guest machines. The interface divides access into static and dynamic probes. The static probes are triggered at architecturally significant points, such as when a system is powered on or disk access occurs. Dynamic probes occur when the guest executes an instruction at a predetermined address. The address used is the guest linear address; the translation from hypervisor to guest addressing is built into the vprobes software. A third type of probe the Data probe is used to specify when a specific address is read from or written to allowing finer granularity of monitoring control. The vprobes scripting language further supports conditional expressions and includes several predefined functions as well as user definable functions (50).

Earlier work by VMware was seen in the development of modules. Subsequent iterations of the VMware hypervisor have been modified to allow inclusion of security tools encapsulated through the notion of these *modules* (28). The challenges associated with this approach are to develop an operating system independent API that bridges the semantic gap, translates virtual machine addresses into operating system specific addresses, and to define a policy engine which handles alerts from each module. Each security tool has a policy framework maintained by the module. In general, the hypervisor administrator is responsible for defining the actions that trigger an alert, and

specifying how the system reacts, by writing rules for the policy engine. For example, an IDS alert may trigger closing a network port.

Six modules were implemented in an initial proof-of-concept. They operate by executing periodically to check for signs of malicious activity. The *lie detector* module executes a comparison between the results provided by common system administration tools and those obtained directly through the API. For example, it may list running processes using the standard *ps* command, executing it internally on the virtual machine and externally using introspection to extract the same information through the API. If the results differ, the module alerts due the likely presence of malicious modification. The *integrity detector* module for user programs compares cryptographic hash values of immutable program segments, such as the code and text segments, with the hashes of programs loaded into the policy engine. This approach is particularly suited to long running memory resident applications such as sshd, inetd, and syslogd. The *signature detector* scans the file system for the signatures of known malicious software. The *raw socket detector* looks for the use of raw network sockets, a typical indicator of attempts to conceal network traffic. The last two modules trigger by continuously monitoring for predefined changes to a virtual machine. The *memory access enforcer* monitors critical sections of the kernel, such as the system call table, for unexpected modifications. Finally, the *NIC access enforcer* prevents an Ethernet device from entering promiscuous mode or changing its hardware address. The flexibility of the policy engine allows administrators to tailor the response to different alerts, the alert can be logged for later reference or the virtual machine execution can be halted for further analysis.

MAVMM is a custom-built hypervisor specifically designed to monitor a single virtual machine with as little system interaction as possible. The authors designed MAVMM to interface with a virtual machine transparently to prevent malicious software from detecting the presence of virtualization. This allows unfettered analysis of malicious code that is often expressly designed to change its functionality when executing in a virtual environment to avoid detection (51)(52).

The software tool 'Virtuoso' addresses the dynamic nature of the semantic gap problem (49). The ability to introspect information from a kernel is highly dependent upon the location and format of the structures in memory. Operating systems are frequently patched and updated, often changing these locations and breaking existing tools that rely on these locations to extract information. Virtouso solves this problem by dynamically detecting new memory locations or structure modifications to these memory regions and reconfigures the introspection tools, adapting to the changes. The tools are built by running a training phase in the guest that queries the guest API functions, such as GetPID, and recording the trace. The in-guest program contains marks, which annotate the start and end of tracing for functions. These marks are necessary because each recording contains extraneous information related to memory management and hardware interrupts. These traces are analyzed to develop an out of guest tool for introspection. To verify their results they implemented the introspection tools to generate programs for six common functions. The test cases were built and tested on a modified version of the QEMU emulator running on the custom built Haiku OS, Windows, and Linux.

2.4.2 Secure Platforms

Unfortunately, many traditional computer security tools are vulnerable to deception through a wide variety of techniques (53,54). At the heart of the problem is the ability of malware to *race-to-the-bottom of the execution stack*, by gaining a higher level of privilege or a lower point of access in the network stack, allowing it to deceive detection tools.

An alternative use of virtualization relies on the independence of the hypervisor to provide a secure platform from which to execute security tools. (55) have developed a customized KVM hypervisor that is capable of hosting numerous tools including intrusion detection systems, security accounting systems, and Quality of Service (QoS) monitoring applications. In this approach, as the hypervisor receives network packets, a virtual switch inside the hypervisor transmits a copy of each packet to an independent virtual machines, whose sole purpose is to execute standalone versions of the security tools. Hosting the tools through virtualization prevents the race-to-the-bottom since the hypervisor sees network packets first and provides isolation between virtual machines. The implementation is limited to passive security applications that do not require communication or interaction with the underlying virtual machine. An attractive property of the approach is its ability to dynamically share resources including multi-core processors. If a particular virtual machine requires extra memory for a short period, it can be granted and revoked as needed. VMwall is an alternative packet-based approach that goes beyond the traditional IDS deployment (56). VMWALL contains a firewall in the hypervisor that checks network traffic against a white list of applications that are allowed to access the network. Illegal accesses are simply blocked.

The secure platform approach has also been applied in the context of a honey pot, a system specifically designed to lure attackers by appearing as a vulnerable target [Jiang and Wang 2007]. The approach relocates forensic instrumentation out of the virtual machine and into a hypervisor. This allows system instructions on the honey pot to be monitored from the hypervisor, for example, the *sysenter* instruction is trapped to monitor transitions from user to kernel space. The system was tested against a typical worm and was able to successfully monitor its actions. In a continuation of this work, the semantic gap is bridged and the operating system state is reconstructed to run rootkit detection on the resulting structures (57). The hypervisor is used as a secure platform to host commercially available software, such as tripwire, for additional detection. This process reduces development time and allows use of continually supported commercial software.

2.4.3 Forensic and Run-time Analysis Tools for Rookkit Detection

In the race-to-the-bottom of the execution stack, there is a continuous arms race between malware and detection technology. Accurate detection of root-kits requires an understanding of the methods by which they are hidden, a taxonomy of these methods has been presented by (58). Category 1 root-kits modify user level binaries to avoid detection and are easily detectable by modern virus checking software. Category 2 root-kits hook the system call table to redirect function calls to malicious code. Category 3 root-kits modify the kernel text to include malicious code. Category 4 root-kits hook the interrupt descriptor table in a similar fashion to Category 2. In addition, a recent method, direct kernel object manipulation (DKOM), directly modifies kernel structures in active memory and could reasonably be added to the taxonomy as a fifth category. The DKOM rootkit is particularly difficult to mitigate because of the dynamic nature of kernel data.

There are three prevalent approaches in the literature to detecting root-kits using virtual machine introspection: *cross-view detection*, *shadowing*, and *control flow verification*.

Cross-view Detection: Static sections of application software and kernel code, contained in code and text segments, should not change during normal program execution. Detecting changes to these areas can be accomplished by computing a hash value from code loaded into the virtual machine and comparing it with a pre-computed hash value. (59) demonstrate this concept on application code while (60) embodies a similar approach focused on kernel code. The challenge in monitoring the virtual machine is providing enough coverage to catch all instances of change without adversely affecting system performance. Unfortunately, the view of static code, as contained in an executable file and its format in memory, may be altered to obscure malicious code (38) (61). Another limitation of this approach is the reliance on the operating system to follow predefined formats for processes structures and code locations. If the operating system does not enforce these formats, information could be moved to a non-standard location and thereby avoid detection. This deception can be mitigated by dynamically detecting new structures at execution time. Unfortunately, this approach could be subverted by un-loading and reloading code inside the detection cycle of the auditor (62,63). The Argos proof of concept extends these ideas beyond verification of executable code, and supports tracking of network data from reception at the network interface through execution. Argos relies on the QEMU emulator, to intercept and identify invalid use of network data in several locations such as jump targets, function addresses, and instructions (62). Further, certain system functions are blocked from consuming data that originated from the network. This low level of tracking allows Argos to identify the specific target

process of malicious code, inspect changes made during the initial steps of an attack, and, quickly develop generic signatures for novel malicious code.

Direct Kernel Object Manipulation (DKOM)root-kits rely on dynamic data manipulation and avoids hooking the system call table. An approach targeting this class of root-kits has been demonstrated based on the notion of *access invariant* properties of kernel structures (64). The key finding is that only designated functions should be used to access a particular kernel data structure. Accesses to these structures from an unauthorized function are deemed *invariant*, denied and logged. To explore the concept, the QEMU emulator was modified to monitor accesses to kernel structures and functions. Unfortunately, since the technique monitors memory accesses, there is a performance penalty associated with it. The cost of monitoring was measured on five typical applications in both a modified and unmodified system and resulting in slowdowns ranging from 13% to 34%. The approach could be combined with traditional static analysis for a more robust solution. However, it would not be effective against the return-to-libc class of attacks, which utilize existing, expected, code.

Another cross-view approach compares information reported by virtual machine utilities, such as *ps* and *netstat*, to the results obtained by directly retrieving the information from the memory of a virtual machine [Litty et al. 2008]. This allows trusted versions of generic operating system tools to be employed in monitoring processes, files, and network connections. The approach encompasses the ability to dynamically detect processes executing in memory, without relying on pre-defined layouts. The approach prevents covert execution of code, such as the use of a java virtual machine, or other processes hidden in the run-able process queue. Unfortunately, it cannot detect code

injected into a legitimate execution environment. The approach is similar to the lie detector module used in VMware (c.f. Section: Secure Platforms) but is specifically oriented to root-kit detection.

Shadowing: Shadowing techniques operate by moving or copying a portion of the virtual machine's kernel execution into the hypervisor. This provides an extra layer of security by ensuring that rootkits do not have equal privilege with the virtual machine kernel even if the virtual machine is compromised. The effect of this *shadow* copy is that the virtual machine appears to be executing instructions, but the instructions are actually executed in the hypervisor. There are two methods for utilizing the concept: Either the results are provided directly back to the virtual machine, or the hypervisor computes an independent copy and compares the result to that of the virtual machine for consistency.

Recall that the Intel read/write/execute protections available to modern operating systems operate at the granularity of pages. Unfortunately, this protection method is often unused in modern kernels (e.g. Linux and Windows) which use mixed memory pages containing both data and code (65). Every page must be marked as executable, even if only a tiny fragment of the page is executable code. (66) have implemented the shadowing technique that provides protections at a finer granularity. This is achieved by loading an authenticated copy of kernel modules into the hypervisor at boot time. When a virtual machine attempts to access memory, the hypervisor resolves the semantic gap to provide access to physical memory. This fact is employed in shadowing by trapping accesses to the loaded kernel modules and verifying only executable code is activated. Later work extended this technique for the analysis of more complex rootkits that obfuscate their actions (67). Memory allocations are tracked and shadowed memory is stored separately.

This allows comparison between the memory segments of the virtual machine and the protected *shadow* copy in the hypervisor, for the purpose of forensic analysis of malware. (45) have proposed a similar scheme to that of (67), but adds the ability to monitor file system accesses. This implementation maintains a tree of authorized access dependencies that designates which processes are allowed to access specific files. Both implementations were able to successfully detect twenty-seven rootkits that were in use at the time. Unfortunately, the performance impact of both methods cannot be compared, as the hardware specifications were not presented. Both methods assessed performance impact by measuring kernel compile time, the approach by (45) reduced compile time by 0.87% in QEMU, and Riley's approach had a 6.37% impact on compile time in VMware.

An enhanced version of shadowing was demonstrated in Secvisor (68). This custom hypervisor shadows memory pages, as in prior efforts, but overcomes the problems associated with mixed kernel pages, by maintaining a copy of the page translation table in the hypervisor and marking only memory pages directly related to the current processor mode as executable. This causes unapproved code that attempts to execute with kernel privilege to initiate a trap to the hypervisor. Secvisor incorporates well-defined properties to assure only approved kernel code is executed. The core protection properties are restated here:

P1: Every entry into kernel mode should set the Instruction Pointer (IP) to an instruction within approved kernel code.

P2: The IP should continue to point to approved kernel until exit from kernel mode.

P3: Any exit from kernel mode, which exit to user mode, should set the privilege level to user level.

P4: Memory containing approved code should not be modified by any code executing in the CPU or peripheral device, unless expressly approved by the hypervisor code.

Properties P2 and P4 are satisfied by inherent capabilities of the processor, which provides page table memory protections. The remaining two properties require that the hypervisor traps all entrances to kernel code and verifies the integrity of kernel code by the hashing technique. For example, the sysenter interrupt call is replaced with a software interrupt to ensure that the hypervisor is able to interrogate the process attempting to enter the kernel. The performance impact on processor benchmark tests was typically 3%, except in the case of a gcc benchmark, which ran 57% slower than native performance. Application level performance was strongly impacted: two benchmarks, kernel build and Postmark, reported 66% and 51% performance reductions respectively.

Integrity Monitoring. The embedding of an arbitrary code segment into the normal control flow a program (i.e. an application or the kernel) is difficult to distinguish without apriori knowledge of the legitimate control flow. Both cross-flow and shadowing approaches are susceptible to attacks that compromise a programs control flow (69). Control flow integrity verification is an extension of integrity monitors discussed in (70,71). The primary idea is to verify that the target of any branch in a program's execution is to a legitimate and expected address. This involves apriori mapping of every possible transition that is allowed by a given program. Control flow methods are resilient to changes in root-kit code commonly used to escape detection by signature methods.

Unfortunately, developing an accurate map of all kernel execution flows is a nontrivial problem owing to the sheer number of control structures, multiple layers of interrupt handling, and concurrency. In addition, the method requires that the apriori control flow is available and protected at run-time, and that a mechanism is interposed during program execution to check transitions against the expected flow. This interposition is naturally suited to implementation through an integrity monitor (70) implemented within a hypervisor (72). Validation of static kernel segments (i.e. code and text) is accomplished by the familiar hash method. At each branch, the hypervisor verifies the code has not been modified, comparing it to the previously computed hash. It then verifies that branches jump to targets within the kernel's approved control flow graph. To monitor dynamic kernel components, such as the heap, stack, and registers, the approach, where appropriate, follows pointers to locate potential changes in control flow. The control flow monitor only executes at predefined intervals, revalidating the current state of the kernel for each run and allowing performance to be traded for security. All 18 of the test rootkits studied were detected by this method. The performance impact was undetectable for five and ten second detection intervals. However, the performance impact increased to 83% if the monitoring frequency is increased to every second. An attacker could constantly load and unload their code inside the predefined monitoring window in order to avoid detection.

Although not using hypervisors, (73) modified a user application to utilize control flow integrity checking by adding a tag for the target of each branch. Every branch was verified at run time against a database of the tags in accordance with the control flow graph. For this method to be successful, three requirements must be met: there can be no

duplication of tags elsewhere in the code, the code must be read only, and data must be non-executable. These requirements could be satisfied through compiler techniques and appropriate runtime protection, and may inspire future hypervisor based approaches.

2.5 Summary

The research challenges associated with the forensic analysis of memory in virtualized environments presents numerous obstacles. Forensics and introspection has benefited from a long history associated with hardware root-kit detection methods. Unfortunately, the high cost of hardware development and evolution of virtualization has prevented the emergence of an enduring solution. Virtual machine introspection offers many of the same opportunities as with hardware introspection methods, capabilities such as providing a protected environment to view the operation of a virtual machine. However, it remains to be seen if this environment can be confined to a smaller attack surface than conventional operating systems with less vulnerabilities and similar performance.

A true synthesis of each effort's capabilities was not practical. Each approach seeks to mitigate malicious software by overcoming and detecting the creative means authors of the malicious software have devised to circumvented traditional application control and detection. Instead, Table 1 compares the core forensics capabilities of each tool necessary to achieve its goal. The primary virtualization platform used by each author is contrasted with its specific support for memory analysis on either shadowed page tables or hardware extended page tables. Currently, no tools exist that support Intel's hardware based extended page tables. The granularity of resolution inside the memory pages is compared

as well. Recall the granularity of memory support is important in systems, such as Linux, that have mixed pages of code and data and require analysis at the subpage level.

The tools ability to inspect the systems primary components including processes, files and network was ascertained and compared for each application. Finally, the performance impact of each application as reported in the literature is described. Unfortunately there was no single benchmark used across the applications, and nearly all authors report their findings as a relative performance impact over an unmodified hypervisor on a different application with the exception of Paladin and Argos which reported clock time increases in specific functions and overall respectively.

					Ability to inspect			
	Name	Platform	Memory Inspection Support	Granularity	Processes	Files	Network	Overhead
Control Flow	SBCFI	Xen	Shadow Tables	Unknown	x			0-20%
	Abadai	Custom	Not described	Unknown	x			30-74%
Cross View	Patagonix	Xen	Shadow Tables	Sub page	x			1-14%
	Monitou	Xen	Shadow Tables	Page	x			No Report
	XenKimoko	Xen	Shadow Tables	Page	x			80%
	Argos	Qemu	Not described	Dynamic	x		x	52% time
Shadowing	Secvisor	Custom	Shadow & NTP (AMD)	Sub page	x			8%
	Paladin	Vmware	Not described	Unknown	x	x		1.5-3.5us
	Gibraltar	Custom	Not described	Sub page	x			0.49%

Table 1: Forensic introspection capabilities matrix

A considerable number of the techniques and technologies result directly from or are complicated by, the failure of modern operating systems to adopt available page protection mechanisms that enforce the separation of code and data. Moreover, while much of the community is focused on root-kit detection, there appears to be a notable absence of techniques to *capture exploits*. The now common procedure of collecting network traffic at enclave boundaries appears to open an opportunity for hypervisors that tie virtual machine actions to network traffic through introspection.

Chapter 3: Enabling Forensics and Process Correlation

3.1: The Layered Security Model

The prevailing method of operating systems security design has focused on a static base of trust. The system then layers this trust through the notion of rings as seen in Figure 4 where the most protected resources are in the center and each outward ring is extended trust from this center core or kernel through various interfaces.

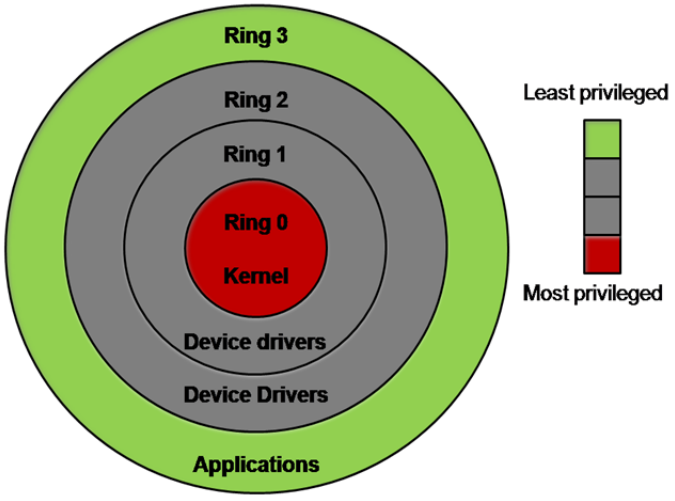


Figure 4: OS protection rings

These rings are based upon the research of MULTICS style protections (43). In this layered ring approach, the kernel is responsible for the most sensitive operations of the system, outward there are several additional rings to segregate device drivers from the kernel and protect them from abuse by user space, finally there is the “user land” domain where everyday system user’s applications execution occurs.

In addition to security, the MULTICS approach affords inherent benefits to system stability. Code executing in a lower privilege layer is prevented from crashing the layers

with a higher privilege beneath it. The ring model enables the system to gracefully handle errors allowing the layer with higher privilege to catch the error. The system code executing at the higher privilege level can then recover the system by either correcting or terminating the offending process. For example, if a user process executes a divide by zero, without such protections, the instruction would cause the processor to fault. The ring model enables the processor to catch the offending process and reports the error to the next highest privilege layer usually the kernel for remediation.

Early hardware was especially intolerant of performance impacts and led to the adoption of the monolithic kernel by the major operating system developers such as windows and Linux (74). To enable faster graphics performance on early hardware operating systems included the drivers into the kernel, eliminating the middle two rings in Figure 4. Paradoxically drivers have been shown to contain some of the highest error rates per lines of code (75). A system based upon such a design became known as a monolithic kernel based operating system.

The alternate approach in contrast is the micro-kernels that maintains separation between kernel, driver and user access to resources. Unfortunately, a micro-kernel design is more complex to implement than one without the separating layers. In addition, there is a performance penalty to implementing the aforementioned architecture, which negatively affected graphics performance on early hardware.

3.2: A Clean-slate Approach

To explore the concepts of network hiding, refresh of trust, and forensics warrants a renewed analysis of the current security practices, and consider how these could be

enhanced through a clean slate approach. In spite of the current mechanisms, continual streams of new vulnerabilities appear that undermine the security of the kernel allowing undetected infection and persistence by malicious code. In addition, research has shown that the number of vulnerabilities is directly correlated with the size of the code base, supporting the viability of reducing the attack surface as a means of improving the overall stability and security of the operating system (76,77) (78-80). Because the number of vulnerabilities associated with an operating system is generally related to the size of the code base. Micro-kernel designs, such as MINIX (81), have recently become popular again because they involve a small trusted core, opening the door to formal verification and presenting a minimal attack surface(40) (82).

As these concepts are rooted in the lowest layers of the operating system core functionality, a simple patch to existing operating systems is not possible. Exploring a revisit of these concepts required a complete rewrite of the entire architecture from the ground up, as part of this thesis the from-scratch operating system **Bear** was developed. The system includes native support for 64-bit address spaces, multicore processors, virtualization technology, and MULTICS style separation of the kernel, users, and driver code. The full architecture is depicted in Figure 5.

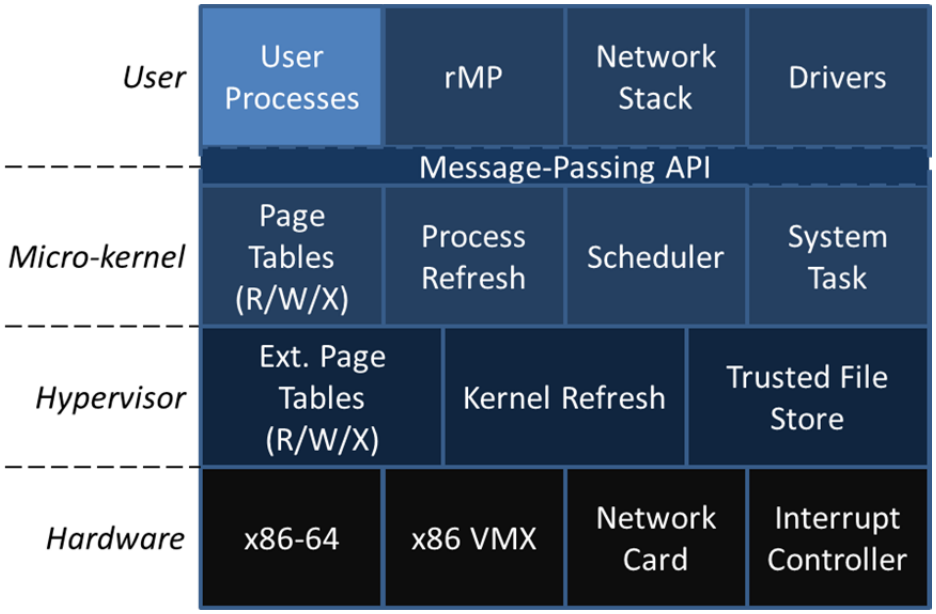


Figure 5: The Bear Operating System Architecture

The system includes a hypervisor, micro-kernel, user space applications, and drivers. While not directly related to the thesis efforts numerous hardware sub-systems were developed as part of the clean slate approach enabling Bear to function. These contributions are detailed in Appendix B. The full system operates on Intel x86 64-bit multi-core blade servers, dell workstations (x86-64bit). The microkernel component also supports the ARM M3, M4, A8, and A9 processor architectures. However next generation ARM processor will include currently lacking support for virtualization enabling future expansion in this product space. This thesis contributed to the architecture through implementation of the hardware device subsystems, hypervisor development specifically the exit handling system and memory control, embedding the introspection techniques into the hypervisor code, and verification of the kernel refresh techniques described in chapter 6.

The micro-kernel and hypervisor extensively share among their code base, following the principles of reducing the attack surface of the overall system through a smaller code base. The interface between each layer is hardened by the enforcement of the MULTICS read, write, and execute protections. Unfortunately, because so few operating systems used all four levels of protection afforded by the layered approach, Intel and AMD deprecated support for the inner two rings seen in Figure 4. Bear solves this issue in a novel manner. Drivers are loaded as user space applications with an extremely small kernel interface that reserves its resources and notifies the kernel of its presence. Each application and driver are loaded into separate memory spaces with strict seclusion from the kernel, using a message passing inspired by the MINIX operating system (81,83). The message passing interface enables the system task to mediate between the processes for resources. Using a message passing interface also enables Bear to standardized across messages and provides a uniform interface for system calls, inter-process and inter-processor communication. The interface is extended outside the local processor cluster through and MPI-like programming interface that maps processes to processors through a user level application, rMP(84).

3.3 Memory

System memory operates by presenting each application with what appears to be unrestricted access to the entire physical memory space. The operating system maintains control of the true physical memory and creates a virtual layer upon the physical memory as seen in Figure 6. The memory manager subsystem is responsible for maintaining this virtual layer, managing the disparate chunks of free physical memory, called pages, and allocating the available pages to an application. Modern processors support multiple sizes

of pages, Bear uses the common 4096 page size. This virtual layer enables each application to function under the assumption that it has access to the full memory address space. A virtual memory model allows application developers to ignore the details of system memory management. The rings model is enforced through bit settings on each individual page indicating to the system which level is allowed to access that particular page. In addition, the individual page setting includes the bits for controlling read, write, and execute permissions.

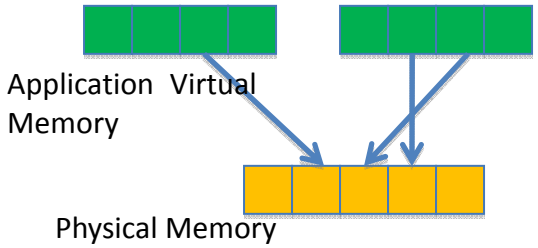


Figure 6: System Virtual memory

To understand how the memory system operates requires treating the management of free memory and the access of memory as two separate yet interdependent functions: The management of available physical memory and the process of accessing data stored at a physical location. Recall, the memory manager divides the memory into pages. Each page is typically 4kB in size on x86-64 processors, though recent advances allow for larger pages on newer processors. To index a 64-bit address space with 4kB pages requires four levels of paging hierarchy where each level of the structures can effectively reference 12 bits of address space or 512 pages Figure 7. Even though modern processors support 64-bit addresses, the current limit of the addressable memory space is

48 bits. The upper bits are reserved by the processor manufacturer for sign extension and future capability.

During the system boot, initialization the hardware provides information concerning the amount and state of physical memory to the operating system. The operating system then creates and initializes these structures known as the page tables, which are also then stored in memory. During this initialization, the system creates the virtual memory layer described above. The operating system then must manage the allocation and tracking of which pages have been allocated. Once initialized the hardware manages the translation between the virtual to physical location. Figure 7 decomposes the virtual address into the steps the hardware walks to locate the physical memory location. The root of the memory structures is contained in the hardware register, known as CR3. This register is used to locate the top most paging structure. The corresponding 12 bits of the virtual address in question is used to index into each level of the tables and extract the location of the next level until it arrives at the desired page.

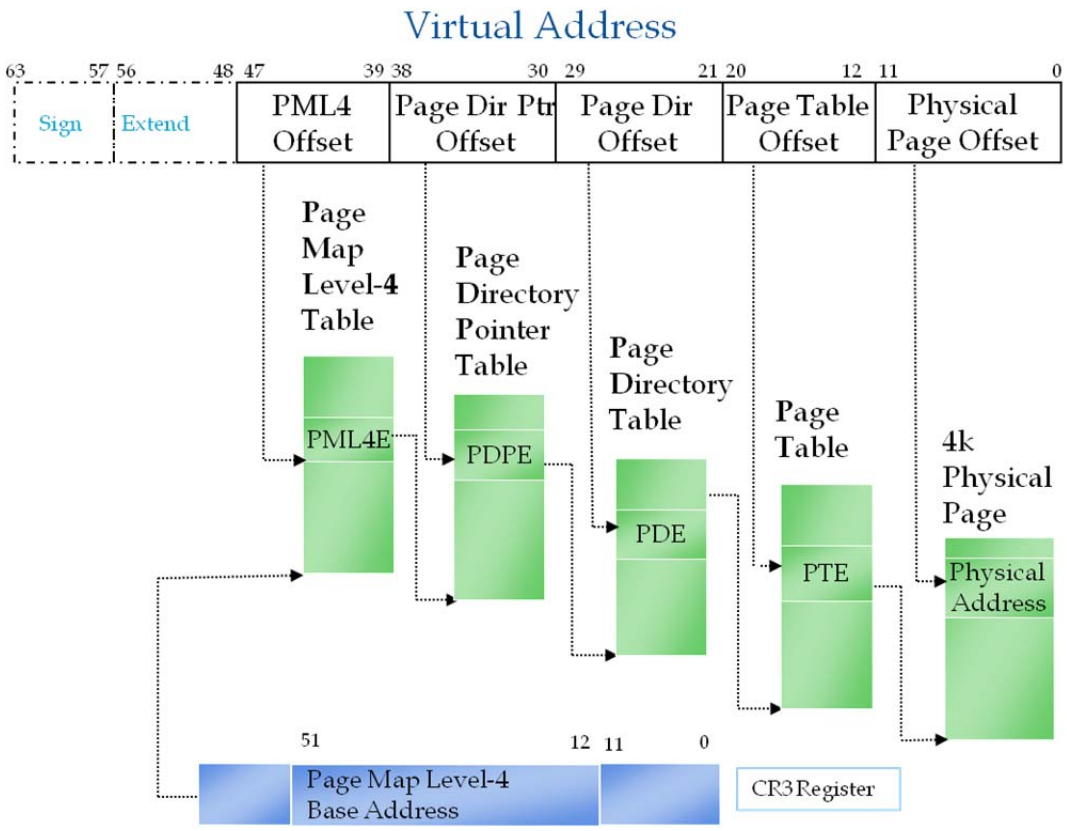


Figure 7: Intel paging hierarchy

Each level of the page table contains 512 entries for 12 bits. However, too physically address the memory the memory manager requires 64 bits. As the first 15 are reserved and the next 12 are defined by the top, 12 bits from the CR3 register pointing to the PML4T. The reaming 40 bits of address are actually stored in the page table structures. However, each entry in the structure is also 64 bits. The remaining bits of each page entry structure are used to control permissions, control cache and 11 are reserved by the specification. However, this was a brief explanation of the memory management functionality, for further details please consult the Intel developer manuals (85).

3.4 Interrupts

An understanding of interrupts is critical to both achieving a clean slate design and necessary for introspection capabilities. Interrupts enable asynchronous operation of the processor and operating system by allowing hardware to *interrupt* the flow of execution. Additionally interrupts enable user programs to request access to resources protected within higher privilege execution rings, serving as a boundary between them. There are three basic types of events, which share the term interrupt: traps, exceptions and interrupts.

The terms exceptions and traps are used interchangeably there is however a key distinction between them. Exceptions are generated automatically by the processor in response to a condition that would cause illegal or impossible execution. The classic example is if an instruction attempts to divide by zero. When such a condition occurs, the processor immediately halts the current execution flow, saves the state of the current execution by pushing the information onto a special stack used by the interrupt handler. Finally, the processor transfers control to the exception handler routine loaded for the particular event. In the Intel x86/x86_64 bit architectures the first 32 interrupts are reserved for exceptions. The full list is presented in Table 2.

Vector/No.	Mnemonic	Description Source
0	Divide Error	DIV and IDIV instructions.
1	Debug	Any code or data reference.
2	Interrupt	Non-maskable external interrupt.
3	Breakpoint	INT 3 instruction.
4	Overflow	INTO instruction.
5	BOUND Range Exceeded	BOUND instruction.
6	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.
7	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	CoProcessor Segment Overrun (reserved)	Floating-point instruction. Not used after Intel 386 processors
10	Invalid TSS	Task switch or TSS access.
11	Segment Not Present	Loading segment registers or accessing system segments.
12	Stack Segment Fault	Stack operations and SS register loads.
13	General Protection	Any memory reference and other protection checks.
14	Page Fault	Any memory reference.
15	Reserved	
16	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	Alignment Check	Any data reference in memory.
18	Machine Check	Error codes (if any) and source are model dependent.
19	SIMD Floating-Point Exception	SIMD Floating-Point Instruction
20 - 31	Reserved	
32 - 255	Maskable Interrupts	External interrupt from INTR pin or INT n instruction.

Table 2 Intel x86/x86_64 Interrupts and Exceptions.

The last row describes the maskable interrupts, these are the trap or user definable software interrupts. The numbers 32-255 are configurable by software for the operating system to use. The difference between traps and exception is that traps are programmatically initiated. The name software interrupts arises from their instantiation in software. In reality, both exceptions and software interrupts are simply a specialized function call. They are known as vectored interrupts because they point or vector to a defined function unconditionally.

These software interrupts are used as points of entry into the kernel or a higher privilege level. The arguments from the calling function are pushed onto the stack and control is transferred to the prescribed function handler. Using a separate stack to pass the

arguments between the user space call and the kernel allows it to inspect the data and prevent unauthorized access or control flow.

The third types of interrupts are hardware based ones. These are the *true* interrupts because they occur asynchronously from an external event such as a keyboard press, network packet arrival, and hard drive access. Interrupts free the processor from costly and constant polling of the devices to check for status changes. The processor is free to submit a job to the hard drive, which may take several milliseconds to complete a task, continue execution and wait for the interrupt to signal the requested data is ready.

These vectors or function pointers are stored in the Interrupt Descriptor Table (IDT). The IDT is enabled through the load IDT (LIDT) instruction. The LIDT instruction notifies the processor the location in memory the table was loaded into. Each entry in the table contains the function pointer to call associated with the interrupt vector, the segment or permission level to use (kernel or user) and various control bits. Each entry is 256 bytes, resulting in a 4096 byte structure for the entire IDT. Using fixed entry sizes enables quick math of 256 * Vector to index to the correct entry in the table. The overall size of the table fits within a natural page boundary, enabling faster memory access to the structure because it can be accessed on a single read of memory. The exceptions to this architecture are the hardware interrupts, because they are controlled by a separate chip that uses overlapping numbers with the processor's reserved exception numbers. The hardware interrupts must be remapped to higher number software interrupts to prevent conflict with the reserved exception interrupt numbers.

In Bear, the interrupts serve multiple purposes. They support the hardware devices on the system: the keyboard, network driver, and system timer. They importantly act as the gateway to the micro-kernel. The standard user functions are implemented through these interfaces such a fork and exec to create new processes, malloc to request new memory. They are used to communicate to the driver processes. Finally, they serve as a standardized interface to pass messages between processes. A particular benefit of using messages over interrupts is the interface is similar to the one used for parallel computing. Bear was constructed to leverage the similar message formats, allowing messages communication to both other processes and to parallel computing nodes (local and remote) using the same interface.

3.5 Virtualization

Virtualization typically refers to an abstraction that hides the implementation details of a particular computer system forming a *virtual machine*. Typically, this abstraction serves to allow sharing of the underlying hardware by multiple operating systems concurrently. Though virtualization has existed since the late 1950's (86-88), it has typically operated only on server class mainframe machines, such as the IBM 360 (89)**.** Virtualization was restricted to the mainframe market due to implementation challenges associated with commodity x86 systems (90,91). Effective virtualization of a computer requires that each of the sub-systems must provide a mechanism to arbitrate and restrict access to, and utilization of resources. Sub-systems such as the CPU, memory, devices, and communication are the first necessary targets as they enable the rest of the system. In addition, another layer must be added to memory, as there would also be no way to prevent one virtual machine from access the hypervisor's memory. Finally, hardware

must also be segregated preventing one device from insecure direct memory accesses across virtual machines.

Virtualization technology has experienced a resurgence due to two recent innovations: Software solutions have emerged that trap and handle instructions, which violate virtualization requirements through binary translation (92,93). Intel and AMD processors have been augmented with new instructions, termed VTX extensions (94) that save, restore, and manipulate an entire operating system context without the overheads associated with emulating instructions through software means. Both of these implementation strategies provide secure and efficient mechanisms previously restricted to business class machines that provide a mechanism to arbitrate access and allocate resources. However, the hardware based mechanisms have a natural speed advantage. Following the growth of the market for virtualization support for full processor, context switches were added to the processor. These features enable the processor to switch between running virtual machines and the hypervisor's operating code.

Commercial virtual machine implementations serve two markets. On personal computers and workstations, the market is dominated by *hosted* approaches, such as VMWARE workstation. These systems operate upon and utilize the features of an installed operating system, such as Linux and MAC OS X. These implementations are sometimes referred to as Type II hypervisors. In the server arena, an alternative standalone implementation, termed a *hypervisor*, replaces the generic operating system with a small, optimized runtime environment that executes on *bearmetal*. This environment provides only the ability to bootstrap and control virtual machines and is typified by the Xen (95) and KVM (96) solutions. Hypervisors of this variant are often referred to as Type I, the

closer proximity to the CPU typically results in faster performance. The key distinction between KVM and Xen is the design choice of managing hardware. Xen creates a specially modified virtual machine to arbitrate access to hardware where as KVM relies on the existing Linux driver framework and security model effectively turning the OS into a hypervisor. In the end, both methods require a near complete Linux kernel. The performance differences have been analyzed in multiple studies (97-99), there is no clear best choice as each model has its inherent benefits and disadvantages, depending on the type of work being performed.

3.5.1 Instruction and CPU support for Virtualization

Unfortunately, early x86 processor designs did not include secure and efficient capabilities for switching the context of an entire operating system. Virtual machines can be compared to a process running inside an operating system. It follows then as the kernel protects and enables access to resources through specific instructions, that the hypervisor must have a similar mechanism to protect access to resources. These instructions are the processor instruction set used to control the flow of execution in the CPU, the classic instructions such as MOV, SUB, and LOAD. However, eleven of the instructions are termed sensitive because they would enable one virtual machine to effect the operation of another. In this scenario, there is no mechanism to differentiate between the authority of the hypervisor and the running virtual machine. Effectively solving this problem requires a ring beneath the kernel sometimes called Ring -1.

This added layer of protection was realized though the inclusion of a Virtual Machine Control Structure (VMCS) and Virtual Machine Extension VMX processor instructions.

The VMCS contains two primary sets of information. The first is settings for controlling the permissions of the virtual machine. Settings such as, which registers the virtual machine is allowed to access, or weather interrupts are passed through to it or handled by the hypervisor. The other set of information contained in the VMCS is the processor context. The context referees to every register in the CPU that contains the current state of execution.

The VMCS is accessed with new instructions for virtualization control called VMX instructions. These enable the processor to load the VMCS, start/stop virtual machine execution, and most importantly *VMExit* or switch control from a virtual machine to the hypervisor control. During the first VMon instruction, the processor state, which reflects the current hypervisor conditions, is saved to a special protected area in the processor. The chosen VMCS is loaded into all the appropriate fields and execution beings at the location of the instruction pointer indicated by the VMCS. The VMCS itself is limited to 4k, a single page to speed up memory swap transactions. There is one VMCS for each virtual machine on the system. The structure contains a large number of options, variables and settings. Of particular interest are the settings that control execution flow and locations of memory structures, which must be obtained for forensic introspection.

3.5.2 Memory-support for Virtualization.

To support virtualization an additional layer in the memory hierarchy has been inserted, virtualized memory is not to be confused with virtual memory described above. This additional layer, illustrated in Figure 8, is necessary to implement the virtual machine abstraction and secure virtual machines from one another. At the base of the hierarchy,

physical memory is allocated by the hypervisor. Virtual machines execute inside independent discontinuous virtual memory spaces. Applications then execute on top of virtual machines, in their own spaces, mapped to those of the underlying virtual machine memory. At each layer virtual memory structures are typically implemented through paging in the traditional manner (30)(31). The figure presents more substantive view of the memory hierarchy used in virtualized memory systems. In the figure, the physical memory of the virtual machine is in reality pages of virtual memory from the hypervisor, represented by the blue dashed line indicating the entire hypervisor virtual memory space. The blue lines in Figure 8 represent traditional page table translation that is supported by the memory manager. The black dashed lines are Extended Page Table translation mechanisms discussed later. The magenta extension is the memory space reserved for the hypervisor itself to function and store its programmatic code.

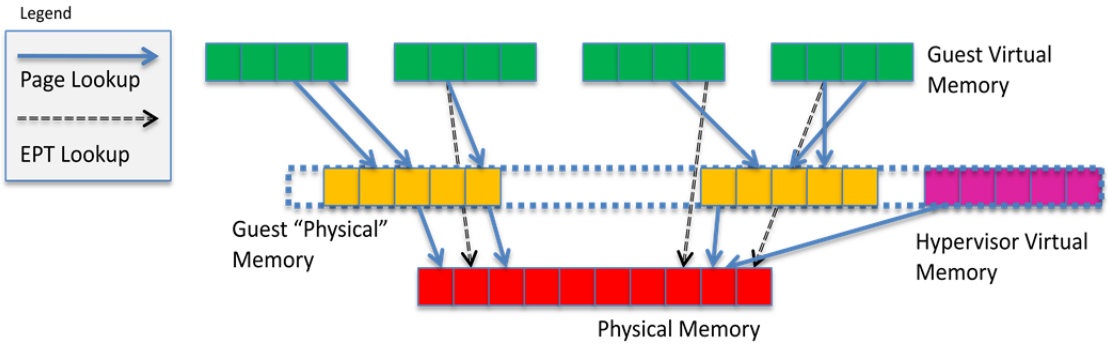


Figure 8: Memory Virtualization support for virtual machines

However, incorporating an additional layer to memory presents a challenge because the virtual machines kernel and the hypervisor attempt to operate at the same privilege level. Recall there are only two states available to protect memory. This leaves no mechanism to prevent one virtual machine from accessing another's memory or that of the hypervisor

itself. To protect memory spaces in virtualized environments several methods have been developed that secure the memory of one system from another. Understanding these is crucial to fulfilling the task of developing forensic methods to observe running virtual machines.

The first method is to manage the guest's memory inside the hypervisor by maintaining a *shadow* copy of the virtual machine's pages inside the hypervisor. The hypervisor is then responsible for maintaining and tracking which pages have previously been allocated to each virtual machine. Each memory access by a virtual machine is then trapped by the hypervisor, decoded and appropriate action taken to allow or deny access. This results in complex schemes and large code bases necessary to maintain independent spaces for each virtual machine.

The alternative method for implementing virtualized memory is through hardware support in the latest generation of Intel and AMD processors. These processors now support extended or nested page tables that provide isolation between the hypervisor and virtual machine memory spaces. These extended page tables enable the virtual machine to independently manage the memory spaces assigned to it. The hardware abstraction allows the virtual machine to run without interruption from the hypervisor. In short, the EPT page tables translate assigned memory pages from a virtual machine's physical to a true physical memory location bypassing the hypervisor's own page tables. The additional tables are shown in Figure 9. An in depth comparison of the two methods is presented by (100).

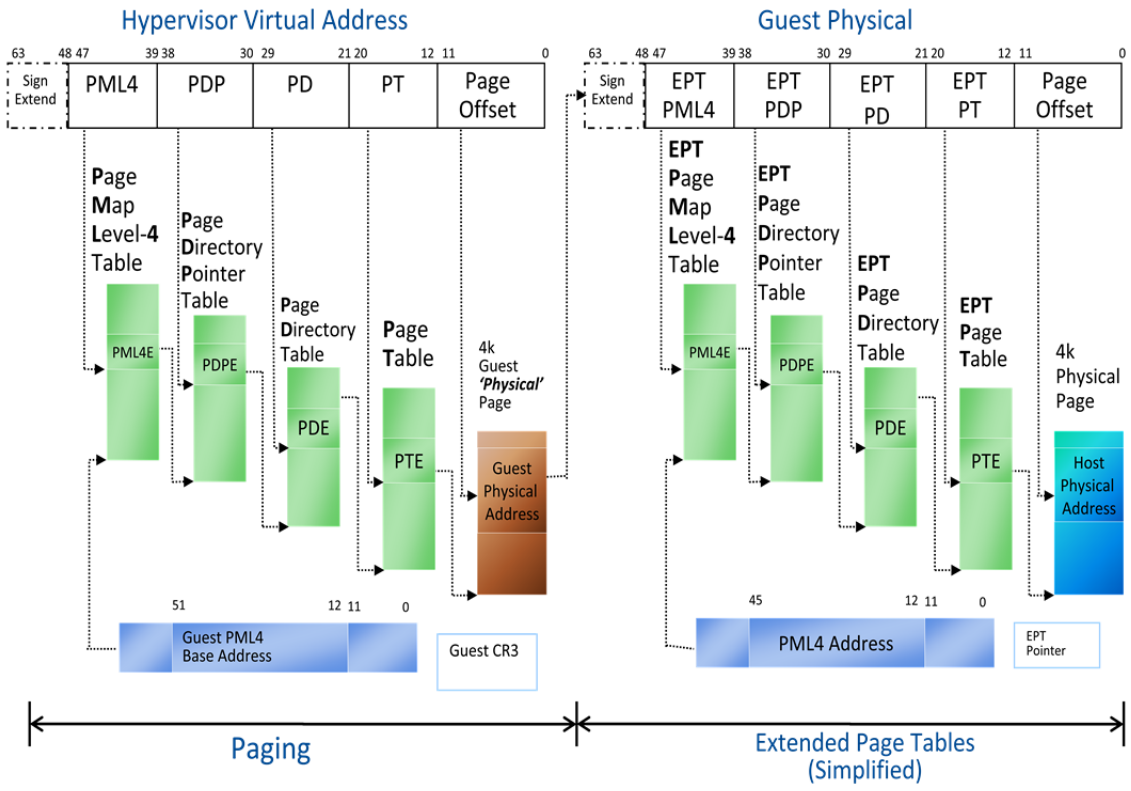


Figure 9: Intel Extended Page Tables Diagram

3.5.3 Virtualization support for devices

Virtualization software has struggled with the problem of Direct Memory Access (DMA) for several years. DMA allows devices to directly access memory and bypass the operating system and memory manager of the processor. The first generation of virtualization technologies was primarily focused on implementing the core concepts of virtualization to allow sharing of the processor itself. Following the success of these technologies to alleviate the workload done by the software programmer, Intel and AMD released hardware solutions to the problem of memory virtualization. The hardware solution supports dynamic assignment of memory pages to allow the virtual machine direct access without intervention by the hypervisor. These additional hardware features

allows the virtual machine to manage its own memory space, without intervention by the hypervisor.

However, these did not solve the problem of DMA access, which could still bypass the hypervisor and allow a malicious virtual machine to overwrite memory inside another virtual machine or potentially the hypervisor. This would completely compromise the security of the system. In short, VT-D technology accomplishes two things: it allows a hypervisor to protect itself from devices, and it allows guests to transparently and safely control peripheral devices.

Figure 10 illustrates the core differences between the approaches to hardware virtualization. The traditional approach was to load the actual drivers into the hypervisor, or a protected domain, and a minimal virtualized interface is presented to virtual machines to make calls against to transfer or receive information. This presents two problems; it increases the code base significantly thereby increasing the attack surface of the system. Second, the use of a generalized virtual driver is a clear indication to malicious software that it is executing inside a virtualized environment and provides a trivial check that does not require any special privileges.

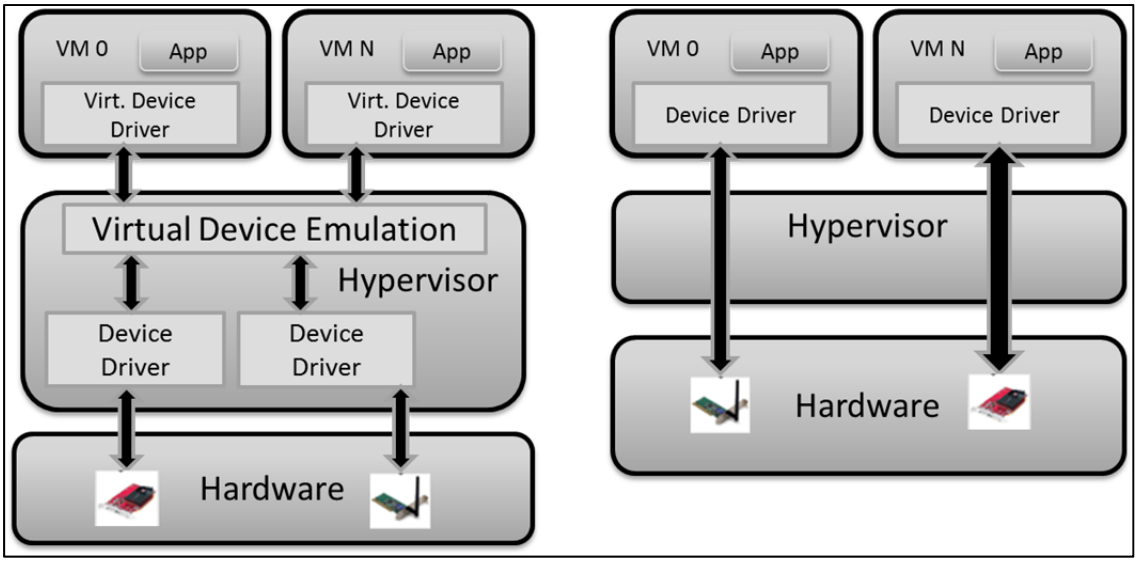


Figure 10: Comparison of hardware virtualization approaches

The latter approach in Figure 10 passes the devices through directly to the virtual machine. This enables direct access of the device by the virtual machine, bypassing the hypervisor. This is enabled by the processor and memory manager by remapping of the memory spaces to the individual virtual machines. Intel has enabled this through new VT-D processor instructions. The advantage is less code is required in the hypervisor and there is no need for a specialized device driver.

The original hardware funded under the CRASH effort does not have full support for VT-D, the solution is to implement a workaround that gives similar functionality. Bear uses a novel technique to boot strap the devices as described in a thesis by Colin Nichols (84). Implementation of VT-D itself will be discussed as future work.

As part of this body of work, the drivers had to be written from scratch. Specifically before virtualizing any devices, the driver code must function as a standalone driver in the kernel. The first step is initializing the peripheral component interface PCI bus,

which controls access to all the devices on the system. Detecting devices on the PCI bus involves querying specific control ports in memory at pre-determined offsets to access the information. The implementation of the PCI bus proved itself a vexing imposition. However, once the specification was fully implemented it enabled access to the hard drive and network card(s) supporting the base concepts of forensically storing information about the network later.

3.6 Forensics in Virtualized Systems

Using the secure separation of the hypervisor as a platform for observing other virtual machines makes it not only, an attractive avenue for performing forensic analysis, but also an assured way to collect data, free form malicious influence. However, to understand memory contents the additional layer of memory must be accounted for in forensic analysis and acquisition. If a forensic tool were to collect the memory content without first applying the appropriate translations, it would have no frame of context and appear as random data. This *semantic gap* reflects the difference in view from inside and outside an executing virtual machine (32).

Since shadow page tables are managed in the hypervisor they can be accessed with knowledge of the specific implementation and applying a translation from guest virtual to hypervisor virtual addresses in the hypervisor. Typically, this translation is a simple linear shift in the address space, subtracting a known reference value of where the guest memory was placed relative to the hypervisor arrives at the desired location.

Acquiring memory contents from systems utilizing hardware extended page tables is a non-trivial task. Neither the processor nor the memory manager provides an interface to

drive the memory manager's EPT translation mechanism. The hypervisor cannot tell the memory manager to walk the extended tables for a particular guest address and retrieve the contents. The application must locate each level of the extended page tables inside the hypervisor's page tables, by walking all four levels of the page tables to find the address of the topmost EPT structure. The application then must repeat this process locating each level of the extended paging structure inside the four levels of the hypervisor page tables. This process requires from 29 to 34 separate memory accesses depending on the depth of access required.

A second challenge in forensic analysis is assuring the accuracy of the information obtained. Forensically collecting the content of memory from inside an untrusted kernel decreases the reliability of the evidence by virtue of the presence of malicious code (33). Root-kits and other malware seek to obscure their presence by manipulating the results from standard collection tools by providing false information (34). Typical examples include modification of the active process queue to ensure that malicious processes are hidden, alteration of directory listings to hide files, modifying program jumps to inject malware, and modifying standard memory locations to hide code.

Recall DMA was the dominant method for securely extracting the contents of arbitrary memory contents through secondary hardware means by using peripheral hardware components. Unfortunately, the creativity of attackers, the constant evolution of hardware and software, and the use of legitimate access by insiders, have all served to inject new vulnerabilities that continually invalidate the assumptions of forensic tools. Virtualization provides a unique opportunity to observe memory from the isolated and protected environment of the hypervisor, provided that its physical memory is not accessible from

the virtual machines that execute upon it. Hypervisors also allow forensic analysis of additional information, such as the content of registers, which are unavailable to other methods. In general, hypervisors provide a smaller, more stable code base, presenting a smaller attack surface, and opening the potential for formal verification (39)(40).

3.7 Chapter summary

The introduction of virtualization while a great benefit to information technology though its ability to enable cloud computing and desktop virtual environments for safer browsing has created a new landscape for analysis. The concepts presented in this chapter presented many of the new benefits afforded by virtualization technology. However, the new technologies have created new and unforeseen gaps in the ability to detect malicious code. Virtualization has also created new opportunities for observation that were previously not possible. The security community has endeavored to address these challenges by utilizing the new opportunities presented by virtualization technology as seen in chapter 2. In addition, such capabilities as those presented in this thesis, which virtualization has enabled, allow for new research and exploration in the search for more secure computing methods.

Chapter 4: Process and Network Correlation through Introspection

Real-time forensic reconstruction of a processes memory and interaction history is impractical in modern computing environments because the volume of data processed by a typical server is immense. However, access to this information would speed the search for zero-day exploits and designate precisely which system components could have been affected by an intrusion. Unfortunately, it may be several months after the infection before any latent effect is observed and there is no way to attest which, if any, of the affected processes are related to the original intrusion. In addition, the system under observation cannot be trusted to record the necessary forensic information as the infection may deliberately hide its presence. These problems subsequently hamper system recovery and data verification efforts. This chapter describes the efforts to enhance the Bear hypervisor with coarse-grained process tracking and in doing so utilizes next generation Intel virtualization technology, leveraging *extended page tables* and enforcing *MULTICS style protection techniques* described in chapter 3. Custom forensic introspection techniques are used to walk the extended page tables, to inspect a virtual machines state and track the associated processes and network traffic. A description of the steps necessary to perform tracking is presented; the real-time performance impact is quantified at less than 5.9µs for each system call.

4.1 Background

The forensic challenge associated with computer network attacks has been the focus of numerous studies (23-27). Detailed reconstruction of the system state requires the recording of all memory transactions. The foundation of this reconstruction stems from

taint tracking techniques, which attempt to continually store a complete picture of all system memory at each moment in time(101-104). The sheer amount of memory bandwidth in modern processors makes recording all interactions nearly impossible and prohibitive in a long term basis.

Additionally, collecting the content of memory from inside an untrusted operating system kernel decreases the reliability of the evidence by virtue of the presence of malicious code (33). Root-kits and other malware seek to obscure their presence by manipulating the results from standard collection tools by providing false information (34). Discussed examples include modification of the active process queue to ensure that malicious processes are hidden, alteration of directory listings to hide files, modifying program jumps to inject malware, and modifying standard memory locations to hide code.

Unfortunately, when faced with an advanced persistent threat, the time span between infection and observed effect could be *months*; the ability to collect and store a comprehensive memory interaction is thus impractical. Moreover, the attack may not necessarily be detectable through an intrusion detection system or direct observation of unusual behavior: detection may occur through an *out-of-band source*, such as signals or human intelligence assets. Recovery from the intrusion is time consuming because it is difficult to ascertain the extent to which data stored on the system has been corrupted. Further, the time lapse between intrusion and effect leaves little trace of the exploit if a sophisticated adversary covers their tracks. The lack of information about what has been modified typically forces an administrator to perform a complete system re-installation since no system components can be trusted. This approach yields no insight into the adversary's access methods or the extent of data corruption.

Recall that in the introduction to virtualization technology chapter two discussed how the advent of virtualization provides a unique opportunity to observe a running micro-kernel. This opportunity stems from the isolated and protected environment of a hypervisor, because the hypervisor's execution and memory spaces are not accessible from the micro-kernel. This independence affords the ability to observe the micro-kernel during execution, ensuring that forensic data is free of contamination by malicious code in the micro-kernel or user processes.

One forensic hypervisor has already been developed for AMD processors (105). Unfortunately, this approach depends upon customizable control of the interrupts that trigger virtual machine exit events. The latest Intel processors support this customizable control only for hardware interrupts. These interrupts are reserved for use by the processor for specific events, and are not user configurable. The result is that hypervisors based on Intel hardware cannot easily capture kernel transitions. In (105) Krishnan, Snow, and Monrose focus on fine-grain of memory analysis and tracking, imposing considerable overhead on system performance. Their implementation is based upon a previous generation of virtualization hardware that does not support extended page tables and their associated protection capabilities. Another approach is Backtracking Intrusions work by (106). Although their goal is the same, the approach is distinctly different. Backtrack relies on the User Mode Linux (UML), which creates virtual machines as user processes inside of the host Linux kernel. The goal of Bear is to reduce the attack surface by constructing a minimal hypervisor and micro-kernel not a full Linux kernel. Additionally UML is restricted to running Linux operating systems. The techniques and code necessary to monitor the running virtual machine are different. The latest Intel

hardware does contain support for monitoring system calls made by the virtual machine that the approach had to overcome. In addition, access to memory structures inside the virtual machine are obscured by modern virtualization hardware making such observations a non-trivial effort. The memory of the virtual machine is no longer directly mapped into a portion of the hypervisor's memory space and is not accessible via a simple linear offset.

4.2 Process Event Tracking and Exploit Discovery

The approach developed as to achieve the goals of this thesis records only *process interactions*, rather than all memory accesses. This information provides a coarse-grained yet reproducible forensic history. Following system initialization, when any process forks or communicates with another process, the association between processes is recorded by the hypervisor. Figure 11 shows the resulting hierarchy of interactions: When process A forks process C, the parent-child relationship is stored in a database; If Process A communicates with process W, this communication is also recorded. After the presence of untrusted code has been determined, *irrespective of the source of this information*, the process history can be used to quickly identify which processes are potentially impacted.

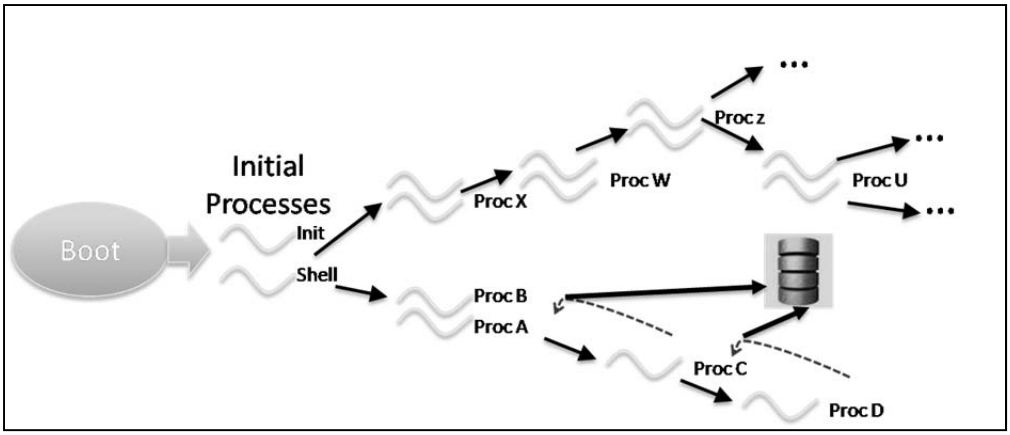


Figure 11: Process Hierarchy

4.3 Naïve Process Interception

An alternative to enable process tracking is configuring the micro kernel to notify the hypervisor directly when it is creating processes or detects processes communicating. The processor has a built in mechanism that enables a virtual machine to *call* the hypervisor, the *vmcall* instruction. It signals the processor to transition control to the hypervisor by forcing a vmexit. Unfortunately, implementing the logic to support the *vmcall* function is the purview of developer; the instruction must detect the exiting instruction length and add this back to the current instruction pointer of the virtual machine to resume execution successfully. Extracting the appropriate information from the VMCS to determine the correct state at which to resume the microkernel's execution was implemented as part of the exit handler routines. The logic for the *vmcall* instruction itself was not difficult to implement, however it was complicated by the complete lack of salient documentation in the Intel manuals. After overcoming these challenges, the code was inserted into the Bear micro-kernel to execute this instruction whenever a process attempted to fork or pass a message to another process. This process

served as an initial proof of concept for testing the ability to record process information and later served to verify the correctness of enhanced interception technique, which does not require the kernel's cooperation. The validation test was inspired by the cross view detection methods described in chapter 2. The technique could later be used as a verification check on the running Bear kernel as well, comparing the reported results with those extracted through introspection to detect malicious modification.

4.4 A Novel Approach to Capturing Interrupts

Recall interrupts are the central mechanism by which hardware devices and software asynchronously signal the micro-kernel for resource access and hardware service. For example, any time a process forks a child, or communicates with another process it triggers a software interrupt. Unfortunately, when processes are communicating and exchanging data in shared memory, the potential exists for the destination process to be affected by actions of the source. This problem is mitigated by the design philosophy of the Bear micro-kernel, which forces all process interactions to occur through the *message-passing interface*. This prevents processes from directly modifying the memory spaces of other processes. The use of a shared interrupt for both message-passing and process creation provides a natural observation point in the micro-kernel to monitor these interactions and record the process history.

The overall method for intercepting processes related events is described in Figure 12. To catch micro-kernel interrupts in the hypervisor, the software initially disable the interrupt at system boot. This forces a hardware exception when the interrupt occurs. Through configuration in the VMCS, exit control fields, these exceptions are handled by the

hypervisor, which terminates the micro-kernel and exits the associated virtual machine, transferring control to the hypervisor. The hypervisor then determines and records the process numbers of the parent and child. The forensic code in the hypervisor then enables the interrupt through memory introspection into the microkernel's IDT. At the same time, the forensic code uses the memory introspection code to mark the page associated with the interrupt handler function as non-readable. Finally, the hypervisor resumes execution of the micro-kernel.

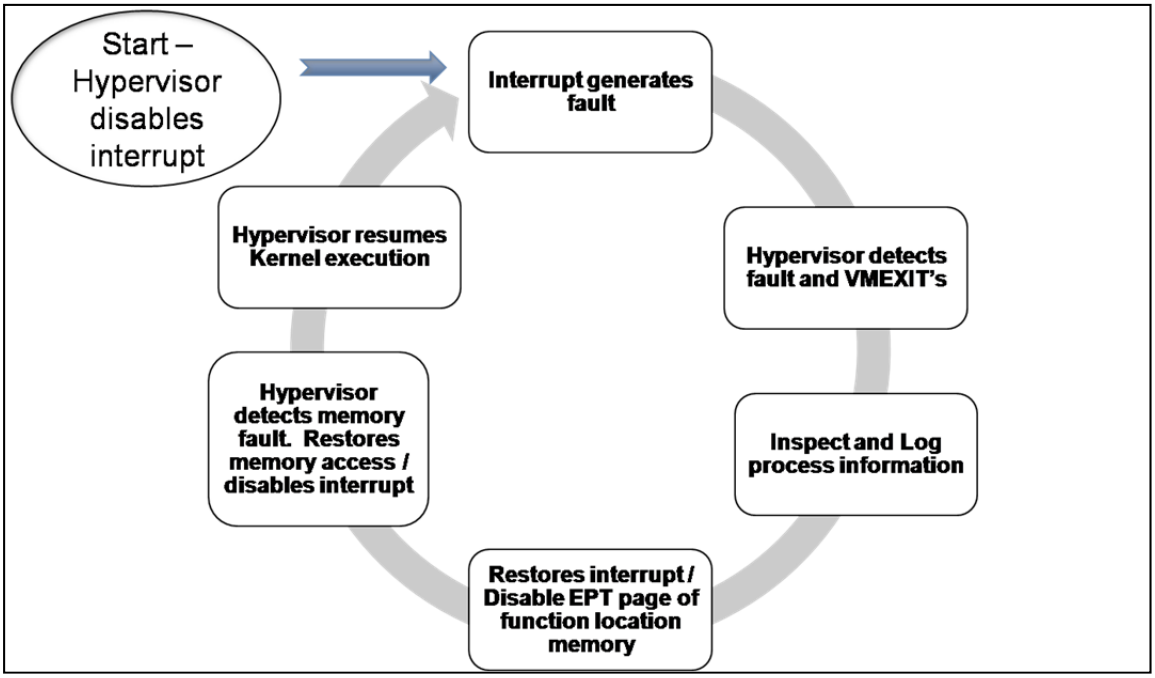


Figure 12: Overall cycle to capture processes.

The micro-kernel handles the interrupt. This precludes the hypervisor from performing the computationally intensive task of clearing the error from the user and interrupt stack spaces. As the microkernel must perform this task inherently, kernel execution resumes in less overhead and less code is added to minimize the attack surface. After cleaning up the processor state from the interrupt, the microkernel then calls the associated function with

the interrupt vector. As access to the memory page the function resides on was restricted, this induces an extended page table (EPT) memory violation. Subsequently resulting in a second *vmexit* and control returned to the hypervisor a second time. During this *vmexit* they hypervisor re-disables the interrupt and enables access to the memory location of the function as readable. Finally, the micro-kernel resumes execution and executes the interrupt handler function successfully. In summary, a double entry into the hypervisor is required for each interrupt: the first catches and services the interrupt; the second re-disables it so that subsequent interrupts can be caught. As both process creation and communication use a shared interrupt, the method for capturing their occurrence only uses one technique to support detecting process creation and inter process communication by inspecting and decoding the type of the message invoking the interrupt.

Recall in the background discussion that Intel microprocessors are not designed to intercept interrupts. Initial investigations of the functionality of the processor identified the fields in the VMCS, which contain the information about the software interrupt vector. After finishing the hardware support for Bear these fields turned out to be informational only and did not affect the run time environment of the virtual machine.

Closer inspection of the processor features identified a potential solution, the virtual machine controls could be set to exit if an exception occurs. Studying the functionality of interrupts revealed that if an interrupt was call while in a disabled state, the processor would generate a segment not present exception. Using the memory introspection techniques detailed in section 4.6 could then be used to observe the relevant information.

Once an exception occurred, the fault had to be cleared from the kernel by the hypervisor. If the kernel was allowed to repair the error, it would be aware of the observation and malicious code could then possible work to escape detection. Further, the microkernel's state had to be fixed to then continue normal functionality and call the function vectored in the interrupt call. The latest Intel processors maintain multiple stacks requiring complex code to 'fix-up' each memory area and reset the virtual machine's processor state and successfully continue execution in the virtual machine. Repairing and then manipulating the virtual machine's processor state externally is a non-trivial task given the complexity of the interrupt, error handling, and execution control systems.

To this point, the technique could capture the initial fault, but how to continue execution required a better approach. The unique approach stemmed from the realization that the processor had not actually changed state up to this point. The virtual machine exit from the exception occurs before any changes have been made to its state. Once the relevant information was extracted, the interrupt could then be activated and the virtual machine would continue execution completely unaware of the stoppage and observation.

Readily apparent is the drawback of the approach, once interrupts are turned on, they will no longer induce exceptions that can be observed. However, the processor functionality presents an opportunity. Once in the interrupt routine the next immediate function called has to be the one loaded into the interrupt vector's address pointer. As this location is a pointer to somewhere else in memory, the forensics code could simply disable the kernels access to this region through the extended page table control settings. This is done while the state is extracted and the interrupt is enabled. By disabling access to the memory location containing the function during the interrupt routine the kernel is prevented from

calling any other function and having code, which could escape observation, by the forensic routines.

The kernel would handle the interrupt, and the only extra processor overhead is levied by the virtual machine exit and forensic code. The kernel would fault again, attempting to access a region now restricted to it causing a memory protection exception. Allowing the hypervisor to quickly disable the interrupt again, so that subsequent process actions might be observed and re-enable the kernel's access to the memory page.

4.5 Challenges

An issue with forensic tracking became known through testing on physical hardware. The test implementation of Bear in the BOCHS emulation environment does not demonstrate the same characteristics as real hardware. In detail, when run on physical hardware the forensic code was being called non-deterministically multiple times. This would result in erroneous process correlation pairs being recorded and contaminating the database of process and network socket history. Stemming from the apparent nondeterministic nature of the execution, the initial supposition was that the memory cache was not being cleared or a stale value was referenced. Believing caching was causing the previous instruction in memory to be referenced and the forensics code to loop back to the same location and execute twice. If correct this would cause out of date information to remain in memory causing the processor to call the forensic tracking code in the hypervisor and increment the counters.

To identify the source of the looping, each type of caching was investigated to determine if it was the source of the loop in the forensics code. The first step is clearing the current

instruction pipeline by issuing a CPUID instruction. To flush the translation look aside buffer requires a memory move to and from the CR3 register. The EPT pages can be specifically flushed from the TLB by the *invep* function, which was developed for bear.

None of these resulted in resolving the problem. However, flushing the TLB via the Control Register 3 (CR3) register will not flush pages that are marked as global. Memory pages, which are frequently referenced, can be marked as global, typically, this is done for the kernel. This prevents them from being flushed from the TLB during a CR3 read/store. A TLB flush is required during a process context switch to invalidate old mappings in the buffer this also flushes the kernel pages and affects performance negatively. To alleviate this, the kernel pages are set as global. In addition, it was discovered that flushing the TLB when pages are added to a process is not required, offering a further performance enhancement. It appeared that clearing the IDT from the TLB might be causing the issue, however tests revealed that this was not the case. The next steps served to isolate the issue to either the memory setting in the kernel or the hypervisor.

To investigate the memory cache settings in the kernel I developed and tested multiple versions of the cache configurations on the Intel hardware. To provide background, the memory pages may be un-cached, stored as write back, write through, or write combined. In addition to, translations between virtual and physical addresses are stored in the translation look aside buffer described above. The MTTR's, which are the legacy memory type registers set in the bios, combine with the selected memory caching type in the page tables to form the *effective* memory type. The result is that the true cache type is a combination of the settings in the MTTR registers and the type selected in the

individual page configuration bits seen and each type seen in Table 3: Intel x86 Cache settings taken from the Intel manuals was tested.

MTRR Memory Type	PAT Entry Value	Effective Memory Type
UC	UC	UC
	WC	WC
	WT	UC
	WB	UC
	WP	UC
WC	UC	UC
	UC-	WC
	WC	WC
	WT	UC
	WB	WC
	WP	UC
WT	UC	UC
	UC-	UC
	WC	WC
	WT	WT
	WB	WT
	WP	WP
WB	UC	UC
	UC-	UC
	WC	WC
	WT	WT
	WB	WB
	WP	WP
WP	UC	UC
	UC-	WC
	WC	WC
	WT	WT
	WB	WP
	WP	WP

Table 3: Intel x86 Cache settings

Continuing with the caching research, the hypervisor EPT pages and the kernel memory pages were configured as un-cacheable, write back, and write through. Each of these configurations had no effect on resolving the loop issue. However, during this analysis it was discovered that the kernel pages cache type selection could influence the hypervisor EPT cache type in a cascading fashion similar to the MTTR registers effect on paging. In instances where EPT tables are in use, the same table is used. However, the MTTR type is replaced by the kernel paging type and then combines with the EPT selected

caching type arriving at the *effective EPT* cache type used by the memory manager for each EPT page. The influence of kernel's memory page cache settings on EPT effective cache type can be resolved by modifying the memory structures of the EPT pages to include a specific configuration bit. This bit causes the EPT pages effective cache type to ignore any configuration selected by the kernel and implement solely the cache typed selected in the EPT pages. Unfortunately, this has had no effect on the looping issue, but does offer a greater understanding of how caching can be configured for greater performance. The *invept* function is being reused as part of the virtual machine rotation implementation because it must be called to invalidate any mappings from the buffer when creating new virtual machines.

The looping problem was caused by a combination of how interrupts function and how their functionality differs on a real processor from an emulated environment, and what is a possible error in the way the Intel Xeon processor handles legacy interrupts. After exhaustively testing all possible combinations of cache, control on the processor the problem was ruled to exist elsewhere. The only other nondeterministic function currently running on Bear was interrupts. After modifying the interrupt routines to print out when they fired, interrupts were observed occurring inside other interrupts.

The STI/CLI assembly commands, in theory, suppress interrupts, but they do not stop all interrupts. Although it was believed, they did. If another interrupt occurs inside a hardware interrupt while the processor is servicing another interrupt the functionality is handled by the nested interrupt processor logic. Upon closer inspection of the Intel manuals, this was determined to be a feature. Recall the forensics introspection works by disabling the interrupt under observation. This would force the kernel to fault and vmexit

to the hypervisor. Inside the introspection code, even for just a few cycles, the next registered interrupt would fire. This behavior is correct however, what then occurred is this registered interrupt would bypass all code checks and fault back to the hypervisor appearing as the outer interrupt vector. The processor failed to update the vector in the VMCS field that contains information about the reason for the exit. As a result, there was no way to differentiate between the system interrupt and any other on the system such as the network card that fired while the forensics code was running. The solution is to disable all unwanted interrupts directly on the interrupt controller chip during the first *vmexit* and later re-enable them to resume execution.

4.6 Memory Introspection

Unfortunately, the underlying hardware used by the hypervisor does not distinguish micro-kernel data-structures, such as the list of active processes, as distinct entities; they hypervisor simply controls memory access in general. Moreover, the complexities associated with this hardware make the reconstruction of this information difficult: the content is obscured through an additional layer of memory translation that implements protection between multiple virtual machines, each executing an instance of the micro-kernel, and the hypervisor. This additional layer is illustrated in Figure 8. User processes are mapped to virtual memory, implemented by page tables in the micro-kernel. The physical memory is partitioned between virtual machines, implemented by *extended page tables* inside the hypervisors virtual memory. This *semantic gap* reflects the difference in view from inside and outside an executing virtual machine (32).

The hypervisor notifies the memory manager a block of physical memory has been allocated to a virtual machine (i.e. micro-kernel). The hypervisor itself has no knowledge of how the memory is used. However, by virtue of the Intel architecture specification, the location in memory and format of the interrupt jump table is known; moreover, the structure of system calls used by fork and process communication is also known. Unfortunately, the *Intel memory manager does not allow itself to be controlled directly*: There is no method to input a virtual machine address and request the memory manager to return the corresponding data located at the physical address (i.e. in the hypervisors virtual memory).

In addition, because the base of the micro-kernels page tables (contained in register CR3) is the only portion of the tables that is stored in an accessible location; the other three tables used in paging (PDPT, PDT, and PT) are located somewhere in the memory of the extended page tables inside the hypervisor's page tables. Figure 13: Code Driven Forensic EPT Walk shows the steps required to perform the translation from the virtual machine virtual address to the hypervisor virtual address.

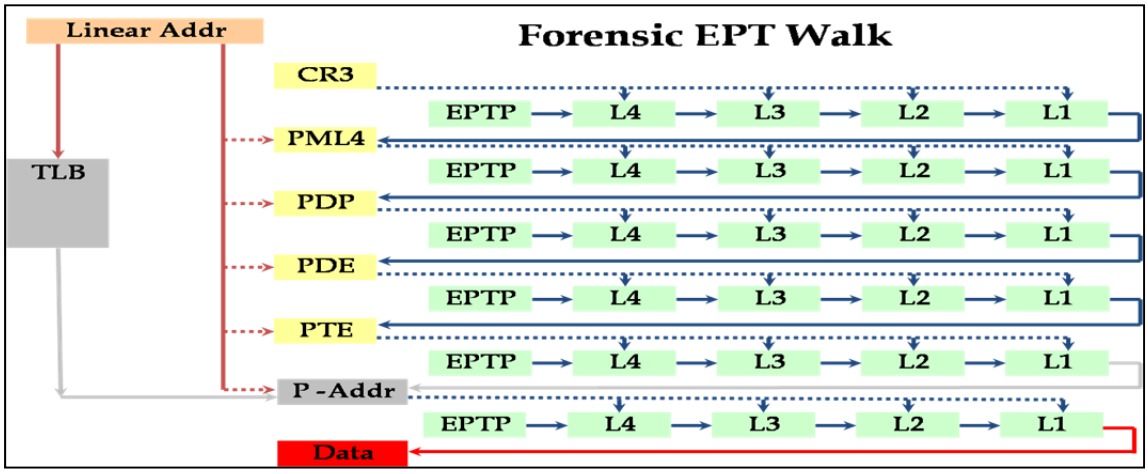


Figure 13: Code Driven Forensic EPT Walk

The forensic introspection code must extract the location of the CR3 register from the hypervisor control structure. Next, it must walk the hypervisor extended page tables to extract the location of the top-level micro-kernel page structure (PML4T). The code then uses the correct portion of the micro-kernel virtual address as an index into PML4T to locate the next level of the micro-kernel page tables, again stored in the hypervisor extended page tables. The code subsequently iterates over the steps in Figure 3 to locate each level of the virtual machines page table inside the hypervisor's tables. Finally, it arrives at the hypervisor virtual address, and returns this value as a pointer, or performs one final walk of the extended page tables to retrieve the contents of the memory. In total, this requires 25 to 29 steps, performed in software, for every memory access that must be inspected. This memory introspection is required by several steps of the process for intercepting interrupts and recording the content of micro-kernel structures.

4.7 Extending Forensics tools in Bear

As part of the thesis contribution, a set of general-purpose utilities were created in the development of the process correlation capabilities. These tools are geared towards enabling overall forensic analysis of the running micro-kernel and could be used as part of later investigations. Specifically these tools allow the hypervisor to unwind the program stack, inspect all running processes in the system, decode sockets, and access the ELF program information to correlate instruction addresses with function names.

Recall each forensic introspection into a running kernel's memory requires a walk of the hypervisor's entire EPT paging structures, seen in Figure 13. Further, each process is itself contained an entire set of page tables into which the kernel is mapped, Figure 14.

Access to information contained in a different process which is not currently running, requires an additional walk of the EPT tables to find the start of the particular process. For example to introspect into the network process while the webserver is currently running would require this method. The cross process introspection tool provides a generic capability to introspect into any process regardless of state, active, waiting, or sleeping.

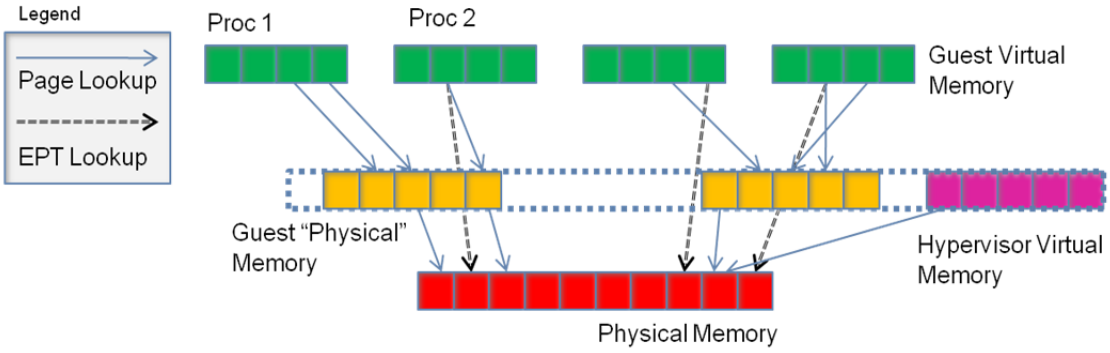


Figure 14: Enhanced virtualized memory diagram

The ELF executable program format, used by Bear, includes the program symbols information necessary for the loader to run the program. These symbols store the names of the functions and variables in a human readable format. Access to these symbols allows for easier debugging and testing of the system. However, more importantly the symbols can be used in the forensic code to provide greater detail when observing the guest kernel and programs. The symbols can be used to programmatically intercept access based on function name, versus its cumbersome 64-bit numerical address. For example, the process tracking code can observe the function called by interrupts by name instead of address. This allows greater situational awareness into the current state of the guest kernel or program.

The difficulty in implanting the import of the symbol information is the ELF format was designed to be an extensible schema. There are potentially tens of fields stored in the program format, only a select few of these must be included, predominantly the code, data and text sections. The first task is to locate the offset from the fixed file header that points to the location of the string list of the present sections. Then iterate over each section to determine its type, an especially frustrating design choice with ELF standard is there can be multiple sections with the same name. In parallel, the main program header contains an offset to the string table, which stores the names of the human readable symbol strings. This string table must be located and read into memory for quick access.

To locate a string name for a function requires taking the address offset iterating over the list of symbols. Once the correct symbol is located, the structure contains and offset into the string table with the corresponding human readable name. Inverting the process to convert a name to and address is more computationally expensive because each symbol must be located as above and checked to see if its name matches the one searched for.

The utility of the code is demonstrated by the forensic introspection code. Each time the kernel is compiled the locations of the various functions may change. If the point at which the kernel resumes execution after a virtual machine exit is statically programmed it would require changing the address and recompiling the hypervisor after each minor change in the kernel. This method is used to dynamically determine the location to resume execution so that the kernel continues after each process interception without the need for manual code changes.

The program stack contains a history of all the recently called functions and the data in variables they were called with. Access to the history of functions called is useful for both debugging, when errors occur, and to back track functions called on a system. For example to trace the functions, a malicious process called. The pseudo code to unwind the stack is outlined below.

```
Print current Instruction pointer(IP);
Get next IP = introspect(current stack base addr +8);
Get next stack base = introspect(current stack base);
Get next IP = introspect(next stack base +8);
do{
        Get next stack base = introspect(current stack base);
        Get next IP = introspect(next stack base +8);
} while(rip > 0);
```

Not shown is that the instruction pointer is printed each time it is updated. Further, the ELF forensics code above can be used to decode the numerical function address to the human readable name simplifying analysis.

Finally, the forensics code can extract the contents of the Interrupt descriptor table IDT and display all 256 entries. It uses the ELF code to extract the names of each function an interrupt calls. This tool can be used to verify or detect changes to the IDT, which is a popular target for attackers. The code also identifies which interrupts are currently in use and how they are configured.

4.8 Performance Evaluation

The approach to performance evaluation was to construct an application that performs process forking a large number of times, and then measure the wall-clock time consumed by the process with and without forensics. Unfortunately, this approach fails to take into account the change in memory structure induced by the fork system call, which is unrelated to forensics. To remove this bias, the interrupt handling process was measured directly with and without forensics. Each measurement involves a generic interrupt handler that models the system call interrupt, but does not create a process. This interrupt hander was executed 1200 times and the instruction counts and runtime for each cycle were collected and averaged. The results, shown in Table 4: Impact of Event Capture on Systems Calls, quantifies these averages with the associated standard deviation. The results indicate that forensics increases the cost of system calls by 33k cycles corresponding to 11µs penalty for each system call. This was observed on a 2.8 GHz processor. Further testing once the memory caching settings were configured revealed a 16k processor cycle or 5.9 µs penalty when caching was enabled. Although in the presence of caching, introspection represents a greater penalty percentage wise. The net performance impact is a smaller penalty compared with overall process runtime. Further, the time is relative to the speed of the processor, modern processors are over 4GHz now and would impart a smaller time impact on system calls. The impact of virtual machine exits is lessened with each processor as Intel seeks ways to reduce this with each generation of processor. As the Bear operating system is enhanced with greater capability, more in-depth benchmarking will be performed to assess the performance impact across the overall process and system runtime on future processors.

	No Forensics	With Forensics	No Forensics w/ memory caching	Forensics w/ memory caching
Average Cycles	164,792	197,758	18,139	34,888
Std. Dev.	76,234	104,724	206	1803
Performance delta		+33k cycles +11.8µs		+16k cycles +5.9 µs

Table 4: Impact of Event Capture on Systems Calls.

The benchmarks utilize Intel's recommended benchmarking methods, specifically the RDTSC and RDTSCP instructions to obtain the cycle counts for each measurement. These methods alleviate concerns with concurrency, and out-of-order execution, speed fluctuations. The forensic results were not communicated off host or recoded to remove any bias from the disk and network drivers.

These micro benchmarks provide allow analysis of the impacts at the function level. However, as this only tests one function. To validate the ability to capture multiple differing processes, which can occur out of order, a comprehensive test executing multiple nested fork, and message passing calls was devised. Two processes, one server, one client, were constructed to fork multiple processed and send messages between them. The 'B' process would simply reflect the message back to the source. The 'A' process would fork two children processes, one of which would also fork a sub-process. Each child process would send a message to the 'B' server, wait for a reply and then exit. The schema is depicted in Figure 15.

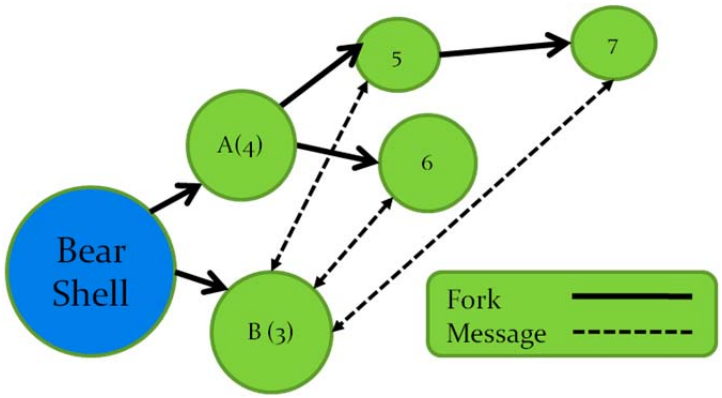


Figure 15: Bear process capture test design.

To confirm the results ground truth was collect by instrumenting both processes to display when they were communicating on the screen. These results were compared with the recorded history in the hypervisor observed through the introspection techniques. The results are presented in Figure 16. The left half of the image is the view from the hypervisor, the right is the kernel command line and the process output. Type 0 is a process forking, type 1 is a process passing a message, and type 2 is a process binding to a port. The image demonstrates that the introspection technique developed as part of the thesis is able to capture multiple events with deferring wait times between the processes. On the left half messages are seen passing from all the child process to the server process 3. The replies are seen being sent to each process. On the right half, the results are confirmed in order originating from the child processes and server exactly mirroring the ground truth confirming the capability.

Figure 16: Process Capture test.

4.9 Network Recording

The ability to record the process history is only half of the picture. To correlate the processes history effectively with the network message history. There must also be a recorded history of the network traffic. In addition, there must be a mechanism to associate these packets with the aforementioned processes. There are several methods to record the network packets, tools such as wireshark (107), tcpdump (108), and snort (109). However as no existing methods have looked at recording the process history, nor do they have access to said information. The question is how to associate the packets with the processes?

4.9.1 Network Event Detection

The separation of drivers into user space requires all network traffic to transit the micro kernel, using the Bear message passing interface used by interrupts. The hypervisor now

has the ability to capture and record any data passing through the interrupt kernel interface.

The appeal of this approach is to leverage the existing code base developed to record processes. The process recording interface could be modified to detect any packets going to the network daemon, which controls the network card. However, this approach would add multiple extra introspection passes to each web request. Relative to the size of the website it would incur additional passes through the introspection code determined by Equation 1. Each web request would require a send and receive; both necessitating two additional vmexit's and passes through the introspection code. The size of the packets is bound by the Ethernet standard to a maximum of 1492 bytes for data.

Equation 1: Number of exits vs. size of website

$$4 * ((\text{sizeof website}) / 1492$$

Thus, the performance penalty would grow with the size of the website and number of transactions to transfer requested data.

Attempting such an approach identified a severe shortcoming of the method. Common information about socket behavior is incorrect. When the system accepts a new packet, it does not create a new high number port to service it. The high number port comes from the client and it maintains this port number for the duration of the connection with the server. The question then is how are sockets delineated, they are defined by the union of five pieces of information the source and destination IP address, port and the protocol being used, e.g. tcp, udp, or raw. This would identify a particular *flow* of information between the server and client, but not enable unique identification of specific packets.

4.9.2 Correlation based approach

Inspired by the previous work implementing the coarse grained process tracking and the desire to record packets with a lower computational complexity. The concept emerged to record the network traffic based on flows inside the hypervisor and use open source tools to record the individual packets. The flow based packet record could then be used to extract the associated packets in the network repository. In addition, unlike existing monolithic kernel based operating systems Bear enforces that only one process may communicate with a socket at a time. Singular socket use is enforced by the kernel that verifies the process is the current owner of the socket before permitting any communication through it. Consequently, if the time that the socket is bound and closed is recorded along with the process ID; then all packets between the open and close of a particular socket are associated with that process, as seen in Figure 17 below.

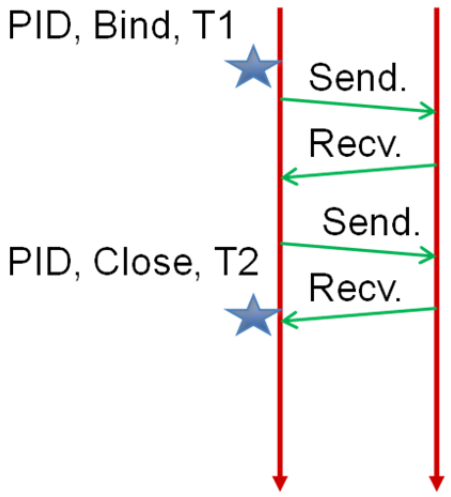


Figure 17: Network Correlation

There are several benefits to tracking processes and network communication in this fashion. The first is that socket capture may utilize the existing code base developed for

process tracking. Bear encodes the information about the socket during the system call interrupts. The forensics code was extended to track this information. An additional benefit is the reduction in forensics overhead. A web server requires the exchange of nine packets, at a minimum, to establish an http connection. The overhead of the forensics code is now reduced because the interception only has to occur during specified socket operations, open and close, instead of introspecting every single send and receive command. A sampling of over 60,000 websites revealed the average size to be 24kB per webpage, requiring on average 16 packets to transfer the website. This confirms the approach as more computationally efficient, by observing the flows versus individual packets.

4.10 Summary

This chapter has presented a novel forensic hypervisor that operates on Intel platforms using extended page tables and a full complement of MULTICS style protections. The ability to track processes and network connections in real-time from the safety of the hypervisor will enable new methods to understand computer network attacks. The ability to quickly ascertain which processes and data are impacted will reduce the time to perform damage assessments, allowing quicker and more selective restoration of services. Further, the ability to extract related network packets will enable new avenues of traffic analysis for locating exploits. The traditional barrier to real time forensics has been the performance cost and storage requirements. The approach stores only course-grain information and system calls incur only a 5.9µs overhead. Since system calls are typically a small component of overall program execution time, this overhead is likely insignificant.

Chapter 5: Results and Analytics

Real-world tests that involve a large number of systems and exploits tends to invoke stress in any information technology department and incur extreme administrative overhead from management or just a succinct ‘no’. However, a realistic test of the components built as part of the thesis involving multiple hosts is necessary to generate significant amounts of traffic, enough to create a large enough packet ‘haystack’ in which an exploit(s) is hidden. Central to the test is the framework built to capture the process information and network traffic seen in Figure 18. The left half of the figure is the Bear software stack equipped with introspection code. The blue circles are the guest virtual machine micro-kernels and the dots inside represent user processes. The right half of the image is the haystack of information represented by the repositories of the collected network traffic and process introspection data.

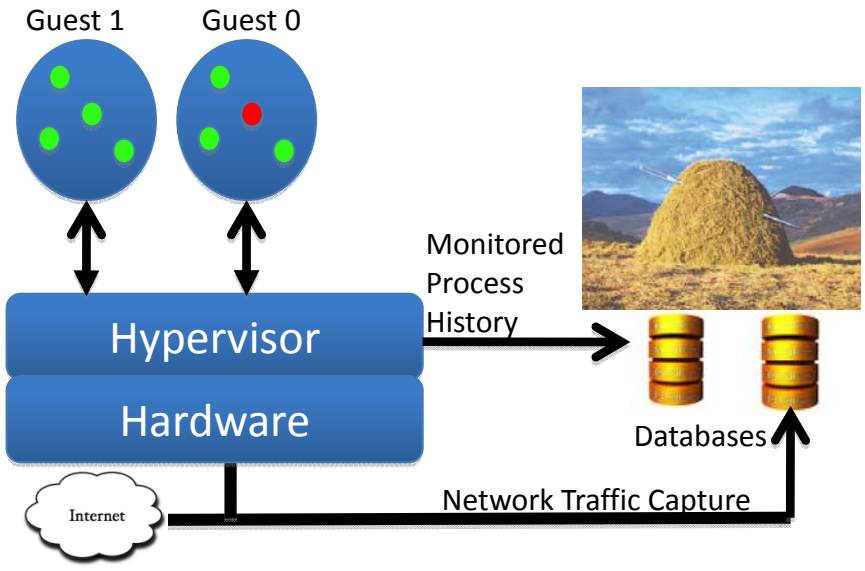


Figure 18: Exploit Capture Test Framework

5.1 Bear Validation Framework

Recall the goal is to develop methods that protect servers. Web servers are the public face of many organizations providing an always-open point of access to their network. In addition, web servers often connect to backend services such as databases, or run nested execution environments, python, JavaScript, or adobe flash. As these additional services parse and execute their own programming code in real time, looking for signatures of malicious code is impossible as it devolves to the classic halting problem (110,111).

The assumption of eventual compromise and validating the introspection methods requires a server that creates a unique process for every web connection. A unique process per connection will facilitate the correlation between processes and packets in the analysis later. A survey of the existing webserver implementations left no viable candidates suitable for testing. The leading web server, assessed by market use, Apache relies on numerous Linux libraries that are not present in Bear. The alternative lighthttpd does not support creating a new process per connection. The solution was to write a webserver from scratch that forks a new process in Bear per each incoming connection.

The design goal was to develop a web server that created a new process to service each incoming request. To simplify the design process a non-forking web server was first created. This highlighted the significant differences in the latest generation of the HTML protocol. The initial implementation idea for the web server was to listen on the receive socket then on each accept, fork a connection and send the html code. However, the HTML protocol changed significantly, from the 1.0 protocol. Modern HTML protocol implementations as defined by the W3C consortium are continuous connections until

terminated by the browser or by the server in the case of an error. The HTML protocol defines the entire specification, there is no subset for enabling only responses to page requests, parsing of the specification is up to the developer to find the components necessary for a test implementation.

A stateful connection supports the forensics work, since it is known that each connection will use one socket and process. Using a stateful connection the server first listens to connections, once the connection has been established and accepted by the client. The server then parses the socket for the next request. It is usually this next request in which the body of the webpage itself is transferred to the client. Then the server must wait for the client to close the connection or send responses to follow on requests. Particularly troublesome is the chrome browser: after transfer of the main web page is complete, it sends requests for various favorites icons and address bar icons to the server. As a prototype demonstration these requests are ignored, however the server must gracefully handle them to not crash. Knowing these requests are coming was a key detail in the first implementation.

Having created a successful prototype web server, the next enhancement was to introduce the fork command so that unique processes would service incoming requests. Under the standard Linux model, this is trivially handled in pseudo code by:

```
Bind() //to the socket
        Listen() //on the socket/port
        while(1)
                //Waiting to accept new connections
                        Fork() //on new connections
                                if(parent)
```

Close (listening port) //to free memory

if(child)

Close (parent listening port) //continue listening

Service_html_request()

The bear microkernel design is inherently different form Linux: as sockets are created they are bound to the creating process. Attempting the code as structured above results in curious and undefined behavior of the system because a child process cannot access the socket on which to communicate. This is not only a security feature, but also a requirement because the network driver will try to wake up the owning process when packets are received. If the process that is woken up is not in a state to receive the packet, the process may crash, or enter an undefined state.

To work around this issue, new functionality was developed that allows the process, which owns a socket to transfer control to another process. This was added through an additional interrupt, which tells the kernel to assign future control to the requested process. The current owner of the socket calls the interrupt function, passing the information through the interrupt message and the kernel implements the control switch. The scheduler also had to be modified, ensuring that the parent process is the first scheduled after a fork command, allowing it to transfer control before packets arrive.

With this experimental server implemented, the Bear framework consists of the hypervisor its associated forensic introspection code and the storage class for recording the information, the micro-kernel to run the system, user level drivers to support the network, and a functioning web server to serve web pages.

5.1.1 Recording Traffic

To record the traffic received and sent from the server all traffic was mirrored to another stand-alone system running Linux. This method follows normal security practice: the traffic is unadulterated and the system is not reachable from any other machine, assuring the ability to record clean data. The packets were recorded using the popular open source tool tcpdump. Tcpdump records packets in the open standard *pcap* format. The open capture format facilitates alter analysis through either custom developed tools or using the open source wireshark packet analysis framework.

5.2 Experimental Network Topology

Simply browsing to the Bear webserver would only generate traffic from a small subset of machines. One or two unique machines do not represent even a small local area network. Conversely, the LARIAT network simulator can simultaneously instrument thousands of machines to probabilistically browse the Internet. LARIAT contains a library of over 60,011 websites taken from the open Internet. Assessing the average size of the website database used in LARIAT revealed the average website size to be 24KB.

The LARIAT client machines contain these website addresses, and the model driven ‘ LARIAT user’ browses to them given the assigned probability at the start of the experiment, based upon the selected user profile. The websites use real world IP addresses, located on routed networks. A functioning DNS and routing table along with associated routers are maintained in LARIAT, so that not only are the websites characteristic of real world traffic, the topology of the network and traffic requests are also indicative of real world interactions across the open Internet.

In the experiments described here, 35 client machines were deployed with the LARIAT client to simulate a reasonably sized network subnet. In addition, LARIAT allows for time manipulation of experiment runs. The speed with which the clients operate can be increased by multiples. This speed setting allows compressing 6 days of experiment collection into a single day.

The experimental network topology is seen below in Figure 19. The LARIAT Internet is encompassed in the world icon, containing the websites, DNS, Microsoft domain controller. The bear server resides on a switched local network added to the global 'Internet'. To add the Bear website to the mix of websites Lariat users browse, DNS records were updated to point at the Bear address. Using DNS redirection eliminates the need to recompile or attempt to modify the complex Lariat code base and achieves the same outcome, as the redirection is transparent to the host.

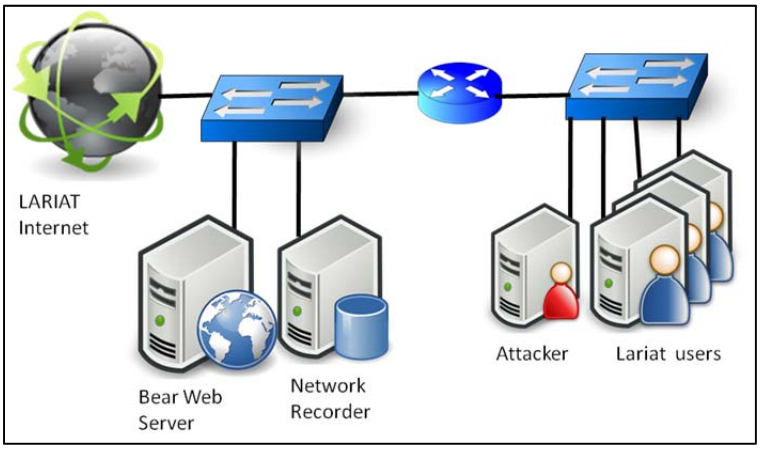


Figure 19: Test Network Topology

The local network to the right of the router is the LARIAT user network, this contains the automated hosts controlled by LARIAT. In addition, this network contains the workstation used to launch exploits and communicate with a rootkit.

5.3 Experiment Design

Two different sets of experiments were conducted using the above network. In the first set, after the LARIAT network simulator had started running, the Linux system non-deterministically launched a simulated exploit at the webserver. The design is depicted in Figure 20. The simulated exploit was an *http post* command versus the typical *get* command to download a webpage. A post command is used to send data to the webserver, typically destined for the database with a standard query language (SQL) command embedded in the post. These SQL commands are a common target for exploitation as described in the threat model, enabling a realistic scenario without using an actual exploit, to alleviate administrative concerns. The configuration enabled a measurement of the size and reduction in network traffic, the ability to quantify the size of the database storage in a 'real world' use, and the time to identify packets associated with the exploit.

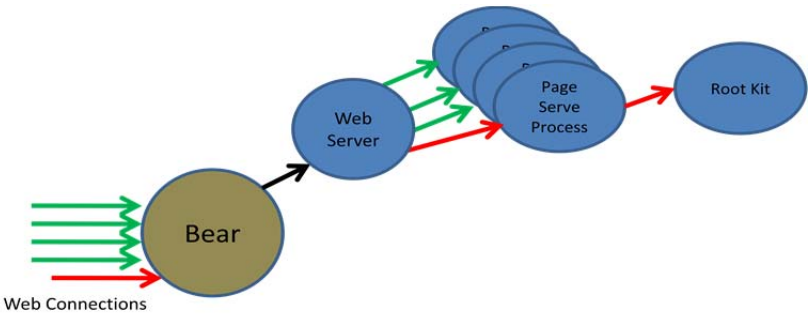


Figure 20: Experiment design for bear

A second set of experiments was carried out to enhance the realism of the scenario. In the second set, a percentage of the web server user processes would communicate with the database shown in Figure 25. Into this traffic haystack, the exploit was launched to start the rootkit process. Randomly after a period of time, the Linux attacker station would communicate with the rootkit instructing the compromised process to communicate with the database, leaving no direct connection to the initial exploit. This scenario represents a more realistic use model and attack path. The database communication was left ambiguous, but it enabled quantification of the ability to isolate the rootkit process. The design also allows for quantification of the point at which communication with the database was potentially compromised. Having this information the analysis tool can then identify and measure the percentage of traffic related to the attack, from which processes they came, which IP addressed received potentially modified requests, and which processes on the server received data from the malicious code.

5.4 Impact Assessment

In addition to the performance impacts caused by introspecting the running process, a method needed to be devised to store the information. The data needed to be recorded in such a manner as to not further negatively impact performance. In addition, the method has to record the data in as compact a fashion as possible. Recording the information in the hypervisor for later analysis would require storing the results on disk. Physical media is the slowest instrument in which to write each transaction where as memory has the fastest recording capability. However, Bear lacked a push down storage mechanism such as a vector. Adding a vector class was straightforward. Expanding the memory allocator

was also needed to develop the *realloc* function to copy existing memory and expand it as the vector grew.

Once this was complete, the second task was to compact storage. To minimize storage requirements the information is stored in a union of structures in the C programming language. The trade off is ease of access and creates a uniform size for each object in the vector and simplified later extraction for analysis. The uniformity meant that each object was the size of the largest structure even if the data did not take up as much space. The layout of the union and structures can be seen in appendix A. The result is that utilizing packed structures; each entry requires 16 bytes of storage space.

Evaluating the combined impact of both the network and process introspection capture techniques was performed on the test web server. The *httperf* application was used to generate request traffic and recorded the response time of the Bear stack with and without forensic introspection enabled. In addition, the forensic introspection was evaluated both with and without the recording to disk enabled, as the intent is to later remove the disk and work only in memory over the network. To sufficient stress the server and generate a quantity of traffic, *httperf* was configured to request 4000 page serves, utilizing not more than 20 open sockets. The number of connection attempts was chosen to ensure enough samples were collected to be statistically significant. *Httperf* samples the standard deviation every 5 seconds, thus to ensure >30 samples the test must run for at least 150 seconds. The number of open connections was limited to 20 as this is the current number of sockets supported by Bear before it begins to queue connections and would inject artifacts of the memory system into the test. Each test run was conducted after a clean

restart of the system, to remove artifacts present from additional memory usage in the kernel.

5.4.1 Impact Assessment Results

The results in Figure 21 show the impact of the network recording techniques on the response time of the server to web connections. These values represent the TCP lifetime connections from when the connection was first opened until it was closed. The results indicate that overall impact of the correlation process improved the server performance. The addition of writing the results to the local hard disk slightly decreased performance by only 3.7ms. The performance increase with no disk could stem from caching effects of the hypervisor.

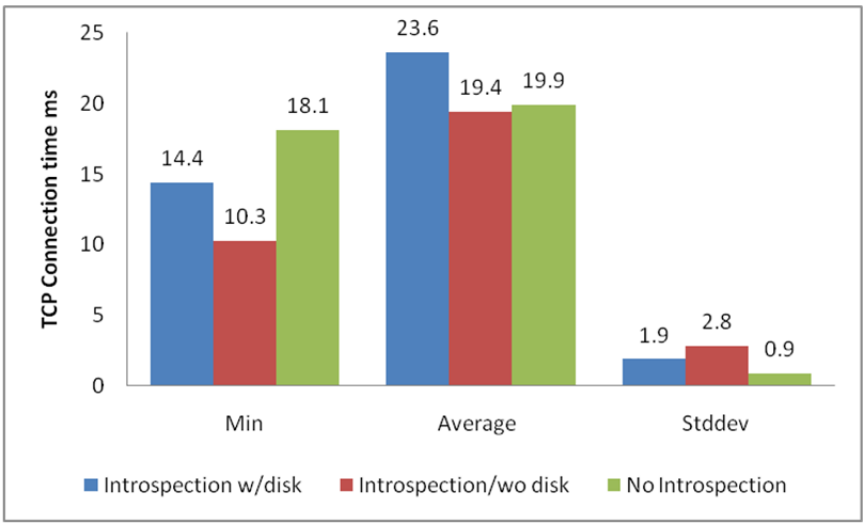
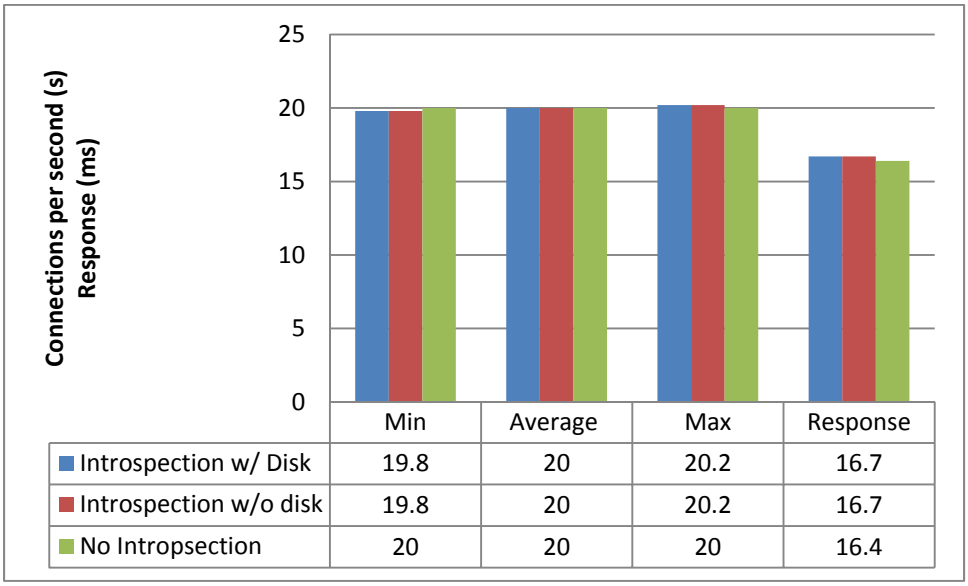


Figure 21: Network Impacts Analysis

The data above serves as a micro-benchmark to the network artifacts imposed by the correlation process. The overall connection time for serving each web page is time that a user would likely observe impacts in. These values are reported in Figure 22. The first

three columns are the mix, average, and maximum rates at which the server was able to service requests. The standard deviations (not shown) were 0.0 – 0.1s across the trials.



	Min	Average	Max	Response
Introspection w/ Disk	19.8	20	20.2	16.7
Introspection w/o disk	19.8	20	20.2	16.7
No Intropsection	20	20	20	16.4

Figure 22: Server Reply and Response times.

In addition, the response times were negligibly impacted and the introspection imparted only a .3ms delay in the server's response to the initial request. The net effect of recording the process and network information, both with and without disk access was, on average, imperceptible to the user's web page experience.

5.5 Exploit Capture without Database.

In the first experiment, the collected results from the data store on the hypervisor were approximately 140k entries only requiring 2.1MB. The data represented the equivalent of 18 days of event recording on the hypervisor, enhanced by LARIAT's time multiplication capabilities. The packet database for the duration was approximately 37MB of traffic directed at the server representing over 286,000 packets to search.

To isolate the relevant packets then first the malicious process must be identified. Any web user process that forks another process is immediately suspicious. In Figure 23 below an algorithm was developed to parse the dataset in the hypervisor to identify any processes that forked more than one child. The webserver is process number 13, quickly identifying processes 40/41 as suspicious. This analysis tool takes less than a second to run on the entire dataset. The remaining processes identified are the test processes *a* and *b.* These were used to verify both message passing and the ability to introspect and record multiple messages that use the fork and waitpid commands.

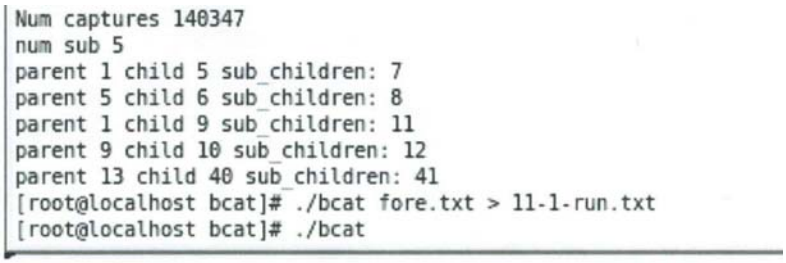


Figure 23: Experiment process log

Having identified the process of interest, number 41, the next task is to extract any network information related to this process. A search of the log reveals which network flow corresponds to this particular trace. In this case, process 41 originated from IP address 43.85.14.19 on port 57227. With this information, a filter is constructed to parse the network database. Filtering the network log in Figure 24 highlights there are only six related packets that originated from the IP address and port combination in question.

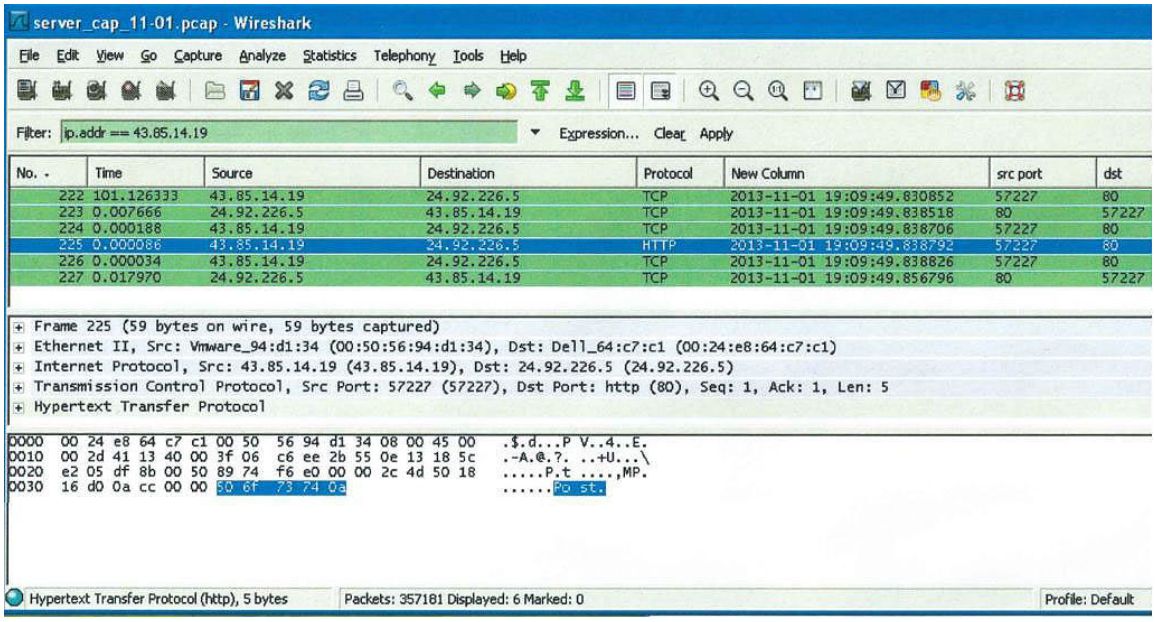


Figure 24: Network Capture experiment 1

Analyzing the packets, only one contains http web traffic. Observing the contents of this particular packet, the test exploit is clearly visible in the highlighted text: *Post*. Overall, this analysis takes less than 5 minutes depending on the capacity of the machine to process the network capture log. In a real world scenario where the particular exploit is not known this would represent a 99.998% reduction in the amount of packets that would needed to be manually inspected, searching for the root cause of the intrusion.

5.6 Exploit capture with Database.

In the second experiment, the system was run for 6 days, collecting the equivalent of over 5 weeks of traffic. The hypervisor process data-store contained over 193,000 entries representing over 47,000 successful web pages served by the test web server. The network traffic database encompassed just short of 400,000 packets. The size of the

repositories containing the information was a tiny 2.9MB file size on the hypervisor data-store and 57MB of recorded full network traffic.

Recall in this experiment the scenario was modified to include the more realistic use of a notional database process. This process would receive communication from a percentage of the webpage requests to generate a sizeable amount of messages in which the malicious communication from the exploit could be embedded. The enhanced scenario is depicted in Figure 25. The lower circle represents the database, which communicates with the webpage service processes and then is injected with communication from the notional exploit rootkit.

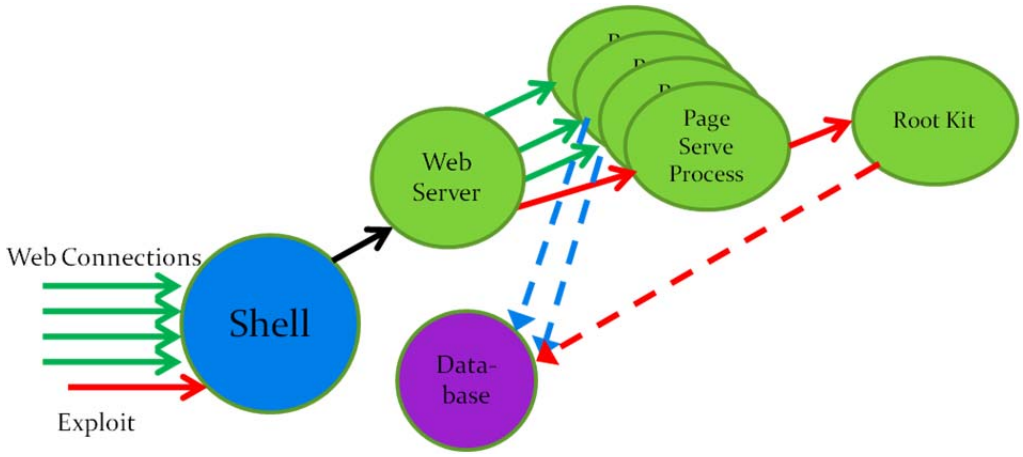


Figure 25: Enhanced experiment scenario

The webserver chose 10% of the page processes to communicate with the database to generate the ground truth traffic.

As with the previous experiment, the test was automated. Once running the Linux system was configured to randomly launch the fake test exploit after a minimum of several hours up to a day. It would then wait another predetermined length of time and then randomly

communicate with the now present ‘exploit’ process. The analysis tool from the first experiment was used to process the data on the hypervisor. The output from this analysis is show in Figure 26, the first series of process relationships are an artifact of the test programs execution trace. These entries increase the size of the *haystack* and validate the results collected are expected. The last entry indicates one of the web servers, process number 13, page service processes started an extra process, number 221. As with before, this is anomalous behavior and warrants further investigation. In addition, the location and number of the process was unknown until the analysis began. Extracting these results takes less than a second to run on the database.

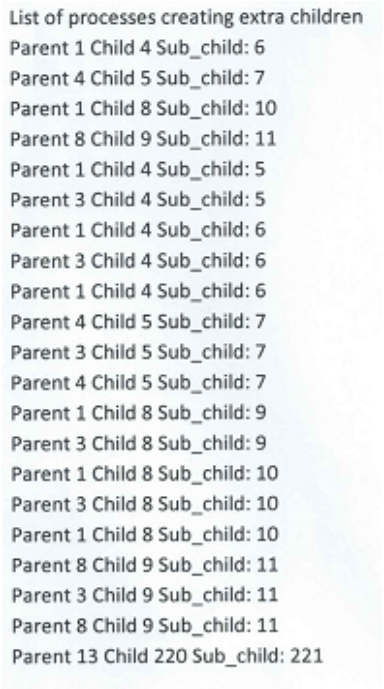


Figure 26: Process analysis results.

The next step in was to investigate what this process did and look for further anomalous behavior. Figure 27 shows the results of the analysis tool written to extract the communication patterns from process 221. The first few lines reveal that several

processes opened receive network sockets. The first two sockets are expected, port 1770 is the MPI network receive socket, and port 80 is the standard web service port. The last line shows the malicious process number 221 binding to port 400. Looking further into the data, the tools extracts any time this process accepted a new network connection indicating the port and IP address. Finally, the tool reveals the number of times the process communicated with the database process.

```
[root@localhost bcat]# ./bcat fore.txt
Num captures 193253
PID 2 Binding port 1770
PID 13 Binding port 80
PID 221 Binding port 400
Comm from 221 to 4
PID 221  accepting on 53364 43.85.14.19 Comm from 221 to 4
PID 221  accepting on 54940 43.85.14.19 Comm from 221 to 4
PID 221  accepting on 43136 43.85.14.19 Comm from 221 to 4
PID 221  accepting on 52088 43.85.14.19 Comm from 221 to 4
PID 221  accepting on 51081 43.85.14.19 Comm from 221 to 4
PID 221  accepting on 40982 43.85.14.19 Comm from 221 to 4
PID 221  accepting on 35047 43.85.14.19 Comm from 221 to 4
PID 221  accepting on 34882 43.85.14.19 Comm from 221 to 4
PID 221  accepting on 34885 43.85.14.19 Comm from 221 to 4
PID 221  accepting on 48698 43.85.14.19 Comm from 221 to 4
```

Figure 27: Communication analysis

Summarizing the dataset shows that in 193,253 recorded events, 21 processes created sub-children. Of these processes, one opened an unexpected network socket. In addition, the database received 4716 messages from the web page processes, of these messages there were 11 messages originating from the malicious process. The 11 messages match the number the test script was configured to communicate. Demonstrating and validating the ability of the forensics tools to intercept all communication with the malicious process.

The results of the analysis on the hypervisor log can be used to extract the relevant packets from the network repository seen in Figure 28. The figure depicts the packets for one communication session with the malicious process. Correlating the IP addresses and port numbers, in total 66 packets of 398,901 communicated with the malicious process.

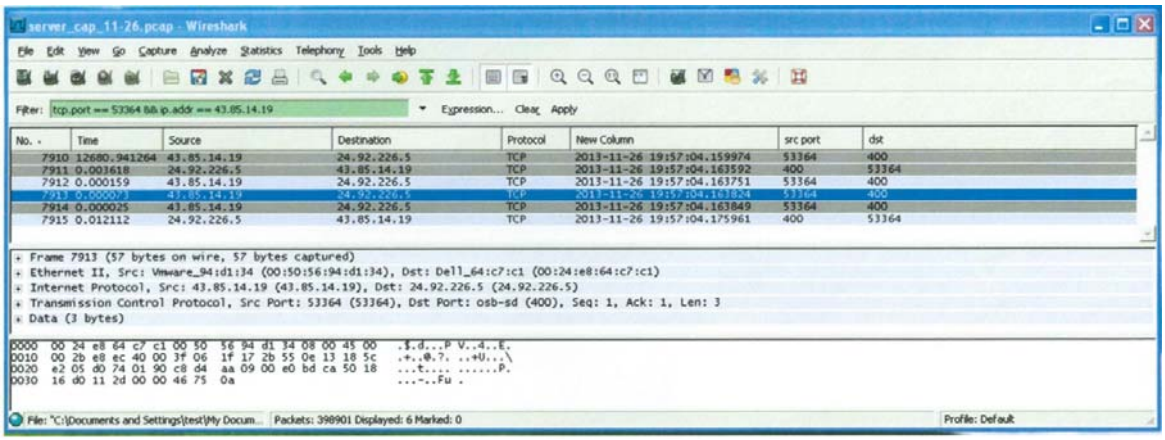


Figure 28: Experiment 2 network capture

These tools enable a 99.8 reduction in the number of packets, which must be analyzed to observe the actions of the malicious process. In addition, analyzing the impacts of the tools for quantifying internal communication can be viewed in two ways. The first is to assume only messages directly communicating with the malicious process are corrupted. In that scenario the 11 messages the process sent and the ability to correlate those represents only 0.23% messages must be analyzed further. However, in this scenario 10% of the web pages communicated with the database. The worst-case outcome is if the offending process cannot be identified, the total packet count requiring manual inspection is 29,000 or 7% of the database that would require manual analysis. The second approach is to look at the network database and assume all traffic the exploit process was implanted is assumed corrupted. The result is naturally relative to the point at which the

exploit was launched compared to the start of recording. In this scenario, the ability to exclude packets prior to communication with the exploit process means 7915 packets can be assured free from contamination.

5.7 Summary

For typical use cases, the automated forensic techniques developed in this thesis decrease the amount of traffic that must be manually inspected to locate a zero-day exploit by more than 90%. Processes incur imperceptible overhead to record and present no distinguishable impact on web-user experience. The experiments described show that they allow exploits to be discovered. Even when there is *no direct link* between the process that receives the original exploit and that, which is impacted, i.e. a rootkit is the cause of malicious activities to the database not the process that received the exploit.

Chapter 6: Network Hiding Hypervisor

The challenges faced by network defenders in protecting servers from malicious code increase daily. The recent SANS vulnerability trends report showed that application vulnerabilities surpassed operating system vulnerabilities, specifically noting that attacks against web services account for over 60% of observed attacks (112). The proliferation of advanced web programming languages, php, javascript, and ruby has increased the complexity of traffic analysis. This increased complexity is compounded by a massive increase in traffic volume from streaming media and peer-to-peer traffic. These factors complicate the task of distinguishing malicious traffic from benign traffic.

Offering assured and available internet services is a key challenge faced by, service providers, the military, and corporations. These services include web hosting, file storage, remote software access, and central database access. As a result of information becoming concentrated in servers, they are a primary target for adversaries to exploit, forcing providers to expend considerable resources protecting them. Typical defense mechanisms, similar to those described for host or user systems include the combination of, intrusion detection systems (14), firewalls (15), and virus scanners (17) at the network level. In corporate and military environments additional host base systems are commonly deployed such as virus scanners, root-kit detector(18,19) and more recently website application firewall software (20)(21). These technologies share a common approach using signature based detection methods. Unfortunately, these methods have limited capability to prevent infection from previously undetected malware. As a result, persistent malicious code may never be detected. Although anomaly detection technologies exist to fill the gap, not all anomalous events are malicious, and not all malicious events are anomalous (22).

The challenge of internet addressing and administrating Domain Name Services or DNS, leads most systems administrators to assign static addresses to their servers. Static address assignment allows DNS servers to maintain long update cycles, preventing services from appearing offline and unnecessary traffic. Since a DNS server caches previous address lookups, there is a significant performance advantage from a server remaining in place. Unfortunately, this presents a static observable target for adversaries to analyze using network scanners such as NMAP and NESSUS. This surveillance process provides a roadmap to the available vulnerabilities and allows appropriate exploits to be isolated or developed. After access is gained, persistence on the static target allows stable reentry point to carry out effects.

The combination of stationary targets, undetectable malicious software, and the small timescales over which a host is exploited allows the attacker to operate inside the defenders OODA loop. This chapter discusses a novel network hiding technology that moves servers around a network to deny surveillance and periodically reconstitutes them to deny persistence. The goal is to increase attacker workload to the point where the timeliness of attacks is significantly longer than the timescale of day-to-day operations, making attacks irrelevant even if they are successful and never detected.

6.1 Related Work

While the existing replication technologies provide acceptable solutions to the reliability challenge, they are not without problems. One example is the use of a round robin DNS approach where several servers host the same content (113). As new connections arrive,

they are recycled through the list of available servers. If one of the servers in the queue fails then the clients attempting to connect with it will fail to reach the intended website.

Virtualization research came into existence during the late 1950's (114,115) and has been widely used in the business class server market ever since. Virtualization presents an abstract interface to the underlying hardware for operating systems as illustrated in Figure 29. This allows the hardware to be shared among several guest virtual machines.

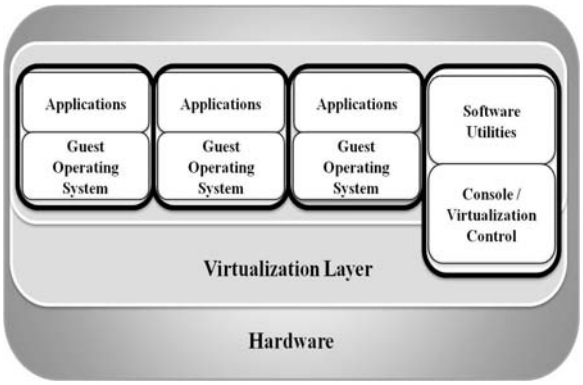


Figure 29: Hypervisor Architecture for Net Hiding.

Virtualization has recently gained entrance into the consumer and medium sized business market with VMware's breakthrough research in binary translation (116). Intel and AMD have released hardware support and acceleration that obviates the need for binary translation. These hardware advancements have allowed the open source community to develop competing technologies such as Xen (117) and KVM (118). The open source nature of these systems has made virtualization a popular platform for security research. As virtual machines are separated from the underlying hardware, researchers are able to analyze malware from the relative safety of the hypervisor (119). Others have sought to use the virtualization to provide a clean slate approach that prevents persistent infection of

client machines. This approach generates a new virtual machine each time an application is executed preventing persistence in the application code base (120,121).

6.2 Network Hiding

The proof-of-concept network hiding provides web services through Apache and is based on server *relocation* and *reconstitution* operations. The server's location is periodically migrated within the local enclave IP space and into alternative enclaves using multiple network cards. This has the effect of presenting a moving target to adversaries, while maintaining connectivity to local clients. It has the effect of increasing attacker workload associated with *surveillance*. Leveraging KVM hypervisor technology, the server is repeatedly reconstituted to a fresh service state, potentially using a *different* operating system, akin to a full system reinstall. This has the effect of changing the attack surface while ensuring that malicious code is removed without attempting to detect its presence. This reconstitution denies persistence over long time-scales while falsifying any existing surveillance information that the attacker may somehow have garnered. These operations can be carried out at random intervals and times. This presents a completely non-deterministic view of the network structure from outside an organizations local area network, while maintaining availability within it.

The approach presents several technical challenges, in particular how to:

- control server relocation and reconstitution,
- preserve connectivity with existing clients, and
- advertise service so that currently unconnected, but authorized, clients are able to locate and connect to the server.

Recall that each server can be assigned one out of several network interface cards (NIC's), and each NIC is connected to a separate logical enclave assumed to be behind a unique firewall/proxy server. Each logical enclave has a DHCP server that serves IP addresses randomly from a large non-routable IP space, orders of magnitude larger than the number of hosts located in the enclave.

Figure 30 outlines the non-deterministic process used to provide network hiding. The process is implemented through KVM hypervisor commands to *define, start*, *undefine* (or remove), and *destroy* (or terminate execution) virtual machines. A programming interface to these commands is provided through the Linux virtual machine library *Libvirt* (122). The library also provides configuration and control of network interfaces. The key feature of this design is that even though the server's presence moves around the physical network by switching network interfaces, all of the transitions occur on top of a single hypervisor. The process begins when the hardware is restarted. The hypervisor enters the ***Start*** state and a *primary* virtual machine ***P*** is created using a random operating system, chosen from a set of pre-configured base-images. An appropriate web server is then bootstrapped on top of the operating system. The proof-of-concept uses Ubuntu and Fedora with Apache. A random NIC card and MAC address (enclave) is chosen to operate on, and a random IP address is obtained from the associated DHCP server. The network services for the primary virtual machine are then initialized with these properties.

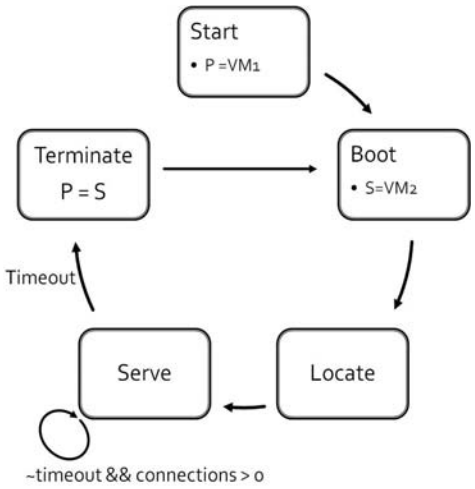


Figure 30: Regeneration Process

The technology subsequently cycles through four primary states:

- ***Boot***: A *secondary* virtual machine ***S*** is created in the background with a different operating system, web server, IP-address, and MAC address (NIC card). On multi-core systems, this operation has negligible impact on the performance of the primary virtual machine.
- ***Locate***: A private DNS server is notified of the network address of the primary server. This DNS server can be used only by authenticated clients to determine the location of the server.
- ***Serve***: The primary server responds to incoming connection requests and serves web content. During this activity, a random timeout is set and a running count is kept of the number of active connections. The active server continues serve as long as the timeout has not expired and there are active connections.
- ***Terminate***: If the timeout expires, the primary virtual machine terminates but only when all existing connections to it close. The secondary virtual machine becomes the

primary virtual machine, all new connections are forwarded to the new primary, and the technology cycles to the BOOT state, where a new secondary client is created.

Figure 31 shows the resulting software stack sharing two NIC cards. There may be two active virtual machines at any one time, each serving a portion of the open connections. The restricted DNS service allows only trusted clients to access the new location of the server when it relocates. This adds an additional layer of indirection that attackers must penetrate, *within a finite time*(121)(121)(118)(108), in order to locate, identify, access, and affect an active server. In Figure 31, the first RedHat server is a previous primary VM whose timeout has already expired but whose connections have not all closed. The first Ubuntu VM is the current primary, its timeout has not yet expired, it serves new connections, and it has been infected as indicated by the dot. The second RedHat VM is the secondary; it will not become active until the next timeout. Notice that each virtual machine uses a unique network connection.

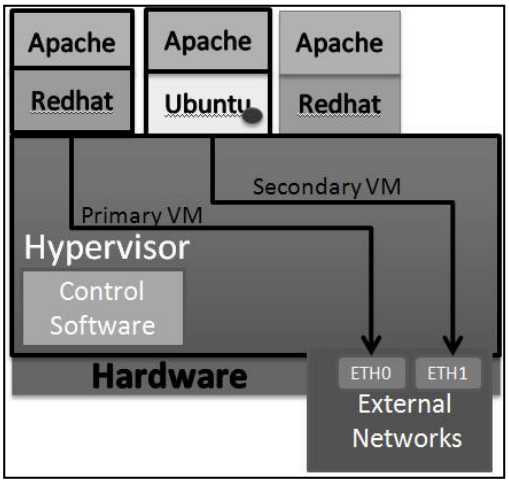


Figure 31: Denying Persistence.

By completely reconstituting the virtual machine hosting the web server, any possibility that malicious code could remain in the system is removed. This method is preferable as opposed to a socket migration technique, because it precludes reintroduction of malicious code from a transferred state.

Unfortunately, dynamically creating network connections inside the hypervisor is not as straightforward as indicated in Figure 31. In order to ensure that any new virtual machine can use any NIC card in the pool and prevent an attacker from observing the server location in one network from another, it is necessary to completely separate the LAN segments. LibVirt provides the ability to generate a virtual bridge inside the hypervisor to effect this separation. Figure 32 refines Figure 31 to illustrate this additional functionality.

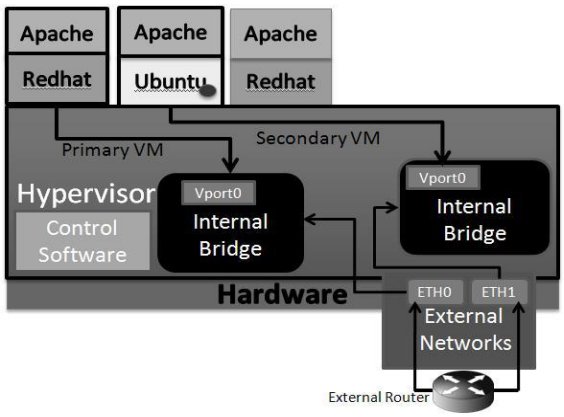


Figure 32: Dynamic Network Isolation.

The virtual bridges mimic the classical local area networks formed by connecting several computers to a physical bridge. One bridge is created and assigned to each physical NIC card. Any virtual machine may be dynamically connected to any physical NIC; this is achieved by assigning the virtual machine a virtual port on the bridge associated with the

NIC. The external networks are assumed to be connected via a router hosted outside the hypervisor allowing remote clients to communicate with any active virtual machine.

Observing the enclave structure externally, as an attacker would, the topology appears as multiple distinct Local Area Networks (LAN's) connected via a central router as shown in Figure 33. In reality, the physical implementation is a single server with multiple connections as shown in Figure 31.

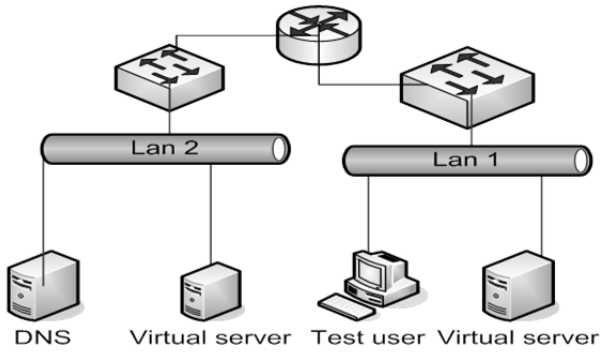


Figure 33: Logical Network Topology

Consider an attacker had previously targeted the present server based upon its IP and MAC address, server type and version. After each reconstitution, this information is no longer valid. The attacker must now re-enter his OODA loop and reorient to the new target address because previous surveillance is now incorrect. Even if the server had previously been infected, this implant is no longer present and cannot be contacted or tasked.

6.3 Private DNS.

Recall that the regeneration process requires a private DNS server to allow clients to locate a server after it has migrated around the network. Beyond the simple lookup of IP-

addresses, the latest DNS implementations, such as BIND (123) provide enhanced functionality for dynamically configuring name references. This involves the update of two components of the DNS record: the CNAME, describing translations between names, and the 'A' records, translating names to IP addresses. Each virtual server must be uniquely named, however, from the viewpoint of external clients; all servers must be accessed through a single consistent name. The CNAME reference system allows this aliasing by creating a name that references another name. The result is a two-step process for resolving generic service names, such as www.myserver.org. The DNS server resolves the alias from www to the unique server name and then resolves the unique server name and returns the numeric IP address.

Dynamic DNS or DynDNS provides the control software to dynamically push updates to a DNS server when a new address is selected. DynDNS was originally invented to support home users on modems whose addresses frequently change (124). Previously the only modality for effecting updates required modifying the static configurations of the BIND/DNS server and fully restarting the system, causing a loss of service. Applying updates through the DynDNS mechanism allows us to maintain a consistent server presence, while virtual machines and their associated IP addresses may change.

6.4 Lessons Learned:

Connection Migration. Hot swapping network connections between virtual machines presents a significant challenge. Connections to an existing web server cannot be migrated to another web server without causing a connection reset. To understand the problem the

types of network connections are categorized, with respect to web servers, into three primary classes:

- Static pages of plain html.
- Streaming pages, containing video or file transfers.
- Stateful pages, such as server side applications that maintain state on the server. For example, shopping carts.

Static web pages are inherently handled by the client's web browser. If the client accesses a page during a VM reconstitution, the client automatically re-requests the page. In the experiments, no perceived impact was observed to the user experience. If the web server is streaming content, as is the case with a file download, the hypervisor cannot simply break the connection; the client would have to resume downloading from the beginning. The same is true with stateful content. The new server would not have the same state and this would cause consistency errors.

One solution is connection state migration in which the active connection is moved between virtual machines in real-time. This presents many challenges, as it requires modification of both the TCP subsystem in the Linux kernel and the web server itself. The MIT CSAIL lab conducted early research in this area (125). The crux of the problem is recreating the state of both the TCP stack, and the web server. To fully migrate the connection and state of the web server would require close coordination between the network layer and application layer, effectively bridging the isolation principles of the layered network design. In addition, copying the state to the new VM would provide the opportunity to carry implants from one VM to another.

Recall that the solution presented here redirects all new incoming requests to the next web server and maintains the previous server until all connections are closed. As a result, streaming connections are not broken and stateful connections are retained for a designed time, limited by the migration timeout.

Dynamic Port Assignment. The initial attempts at denying surveillance exposed a severe limitation of the KVM/Libvirt hypervisor instantiations. The configuration of the hypervisor, the virtual bridges/switches, and the DHCP servers under the hypervisor's control cannot be dynamically configured. Attempts to migrate the server around the network IP space produced erratic behavior, occasionally clients could not connect, other times the migration was seamless. The behavior was caused by the internal configuration of the network infrastructure restarting, occasionally a new client would connect at just the right moment rebuilding the connection, other times not.

The solution to this problem involved dynamic generation of bridges inside the hypervisor to implement the concept illustrated in figure 4. Each time a new virtual machine must be attached to a NIC card, instead of simply connecting to a bridge port, the old bridge associated with the NIC is removed and replaced with a completely new one. This achieves the concept but is more difficult to implement.

Detecting Connection State. Recall that network hiding requires the ability to detect when all connections to a virtual machine have closed. With few exceptions, such as stateful firewalls, network equipment does not store information about the traffic, which passes through it. Similarly, the hypervisor does not maintain state for every aspect of the virtual machines it hosts. Connections are passed through the software bridge in a hypervisor just as in the physical world. Consequently, the bridge does not maintain state. One option

would be to modify the implementation of the bridge to count open connections. However, a Linux virtual server already maintains connection state, including the number of active connections, in the TCP directory associated the system directory */proc*. Unfortunately, this is not generally readable by the hypervisor since the file system of a virtual server is not mounted.

The virtual machine disk is a raw binary file when viewed from the hypervisor. In consequence, it can be mounted by the hypervisor as simple another disk. Although it is possible to read the blocks from this disk directly, the encoding format of the disk requires accessing the disk from its root node. Without rebuilding this file system structure there is no simple way to identify the /proc directory the process needs to access. To solve this problem there is an open source library *fuse* (126) that allows virtual machine file systems to be mounted and accessed inside a hypervisor.

Unfortunately, the 'files' in the /proc directory are not traditional files; they exist only at the moment they are read in order to ensure that they reflect the most recent system state. When the /proc/tcp6 file containing connection information is accessed, Linux reads the network state and generates the file dynamically, returning the most up-to-date information to whoever accessed the file. The *fuse* library mounting the file system receives the state of the files the moment it accesses them, subsequent reads of the configuration files then have no change. Relying on outdated connection state would lead us to disconnect clients.

In order to solve this problem a module was developed that monitors the state of the connections inside the virtual server and replicates the information in a traditional file accessible from the hypervisor through the fuse library. Once the connection state is

accessible, it is possible to monitor the number of active connections as needed by the network hiding process.

System Administration. An added benefit of the reconstitution process is that it accommodates testing and deployment of server upgrades independent of the active virtual machine. Upgrades can be tested independently and rolled into the reconstitution queue at any time. This provides a seamless upgrade path for security patches, bug fixes, and configuration changes.

6.5 Network hiding in Bear

Building upon the lessons learned implementing the network hiding techniques in KVM. As part of the Bear clean slate development, the interfaces to implement the same technology were incorporated into Bear. However, Bear is still an experimental framework and for the moment lacks the necessary interfaces to communicate with a private DNS server, precluding a full working implementation. The ability to create new virtual machine from scratch was demonstrated. Although, the current hardware lacks support to share the network card among multiple virtual machines running concurrently. Working around the issue the hardware can maintain connectivity to both machines for a short length of time.

The Broadcom network cards require a keep-alive function be activated. If the function is not activated the card shuts down and can only be woken up through a complete re-initialization. Swapping between running virtual machines causes one card to eventually power down yet was enough to test the harness code for destroying and creating a new micro-kernel. To demonstrate the network hiding implementation in Bear as part of this

thesis, a novel solution to timing the switches between running kernels was developed. The hypervisor currently maintains no interrupt control. Previous work by implemented the hypervisor pre-emption timer to exit running kernels at fixed intervals. This method lacks the ability to set a non-deterministic timer beyond several seconds. To overcome this challenge in the interim I added a special *vmcall* and ported the timer functionality to the hypervisor. When the system timer interrupt fires, in addition to increasing the clock on the running kernel, through the *vmcall* interface it updates the clock in the hypervisor. This inventive use of *vmcall* instructions enabled a working timer in the hypervisor was used to control switches between virtual machines with greater fidelity. The lack of hypervisor interrupts will be solved in future work. Relying on the running kernel to increase the timer is a known vulnerability, but served to demonstrate the concept and verify the timer code until the Advanced Programmable Interrupt Controller has been implemented with virtualization technology.

6.6 Chapter Summary:

This chapter describes a proof-of-concept KVM-based network hiding technology for servers. This architecture has been implemented in the context of web-servers, but equally applies, and can be adapted to, many other services. It increases attacker workload by denying the ability to persist for any meaningful length of time on an infected server. It also denies surveillance by non-deterministically moving a server around the physical network while changing operating system and network properties. The primary challenges involved maintaining control, providing connectivity for active users, and directing new clients to the existing server location. The solution provides uninterrupted service while effectively increasing attacker workload. Further, the solution provides an

enhanced capability to upgrade offline machines with patches and upgrades without disrupting the current state.

Implementations of the concepts presented here were significantly complicated by the overall complexity of KVM and Linux. A specialized hypervisor that supports only the operations required to deny surveillance and persistence might provide more direct and natural access to core system parameters making several of the libraries mentioned here irrelevant. Much of the complexity was derived from inconsistent support for hot swapping of network cards across multiple operating systems. In addition, difficulties arising from lack of dynamic configuration support for networking in the existing LibVirt implementation. As a result, it is expected that over time the network hiding process described here will become easier to implement and support in KVM.

7. Conclusions

7.1 Future Work

Recall that in chapter 3.5, the virtualization of devices was discussed and hardware to implement the latest generation techniques for virtualized hardware support VT-D was unavailable. The lack of properly virtualized devices presents a serious security concern for Bear. A malicious user could disabuse the Direct Memory Access system to subvert the protection rings between the processes and the kernel itself. Knowledge of this weakness could even enable an adversary to escape from the kernel boundary and maliciously tamper with the hypervisor. In addition, virtualization of devices presents a relatively unexplored area for the effects virtualization of devices has on forensics and what the optimal implementation is in the presence of micro-kernel.

In addition, the techniques presented here rely on the presence of a micro-kernel to enforce the boundaries necessary to accurately track the process genealogy. However, given the market penetration of the Linux kernel, it would be worthwhile to investigate approached to disable inter-process communication and port the introspection techniques to the Linux kernel for more wide spread use.

7.2. Lessons Learned

For typical use cases, the automated forensic techniques developed in this thesis decrease the amount of traffic that must be manually inspected to locate a zero-day exploit by more than 90%. They incur imperceptible overhead on active processes to record and present no perceptible impact on web-user experience. The experiments described show

that they allow exploits to be automatically discovered even when there is no direct link between the processes that receives the original exploit and that which is impacted i.e. a rootkit is the cause of malicious activities to the database not the process that received the exploit.

This thesis has examined the issue of locating exploits in ever-growing databases of network traffic. The thesis presents novel methods to identify on host actions for better granularity of system recovery after a system has been compromised. In addition, the thesis presented the idea of increasing attacker workload through non-deterministic refreshing of the entire system from a gold standard, while simultaneously migrating to new locations in the network. Invalidating the previous intelligence gathered about a target, denying the ability to persist for a length of time, and forcing new methods of attacking the system to be used thus potentially exposing more access methods. These efforts were accomplished through novel techniques for hypervisor introspection of a running kernel and a coarse-grained forensic recording. They were accomplished as part of a team that developed the from-scratch Bear hypervisor and micro-kernel. Together they serve to remove avenues of attack from the adversary's toolbox and provide better mechanisms for recovery and post incident analysis.

The forensic introspection methods developed allow the hypervisor to arbitrarily observe the running kernel. The tools enable the hypervisor to extract the running list of processes, unwind the previous instructions called on the program stack space. Observe when new processes are created and when they communicate with the network. These tools provide the ability to reconstruct the specific events taken by processes, specifically with the aim to replaying the actions taken on-host by an adversary.

Introspection was facilitated through the development of a novel technique to track system calls on the Intel x86_64 architecture. Intel processors do not natively support the capability to track system calls from the hypervisor. This involved a deep understanding of the system interrupt interface and an innovative approach to tracking virtual machine exits from a running kernel through selective memory modification of the running micro-kernel. The result was the ability to record the interactions of the micro-kernel at the event boundary of interrupts to record and reconstruct events later.

The network hiding and refresh techniques demonstrated through a proof-of-concept implementation on open source software, successfully demonstrates the ability to disrupt the attackers OODA loop. Network hiding and refresh tools invalidate the information gained by and attacker up to that point. By varying the operating system settings, deny the attacker the same ease of access through a previously used path. Wiping away any advanced persistent threat potentially loaded on the system.

Combined with existing methods for recording network traffic, the ability to reconstruct the process history and correlate it with the related network packets is a new capability. This combines the domains of network analysis and system forensics. Further, this thesis presents results demonstrating the technique in a tractable manner with almost negligible impact to performance observed by the user. Unlike existing approaches, presented in chapter two, which can affect performance up to orders of magnitude the approach presented here, has a near imperceptible effect on user interactions with the server.

Appendix A Forensic Introspection Storage structures

The most efficient way to store the information about events captured by the hypervisor is highlighted in the structures below. The Bear system was developed in the C language, the best way to minimize storage and facilitate later analysis is a packed union of the structures. As mentioned, the tradeoff is that each entry's memory footprint will be the size of the largest in the union. In this case, 16 bytes are required for each entry.

```
enum Type { FORK, COMM, BIND, ACCEPT, XFER, CLOSE, EXEC };
typedef struct{
        uint32_t parent;
        uint32_t child;

}__attribute__ ((packed)) Process_t ;

typedef struct{
        uint16_t port;
        uint32_t pid;
        uint32_t ip;
        int32_t time;
}__attribute__ ((packed)) Bind_t;

typedef struct{
        uint32_t pid;
        uint16_t port;
        uint32_t ip;
        int32_t time;
}__attribute__ ((packed)) Accept_t ;

typedef struct{
        uint16_t sock;
        int32_t time;
}__attribute__ ((packed)) Close_t;

typedef struct{
        uint32_t src;
        uint32_t dst;
}__attribute__ ((packed)) Comm_t;

typedef struct{
        uint16_t sock;
```

```
        uint32_t old_pid;
        uint32_t new_pid;
}__attribute__ ((packed)) Xfer_t;

typedef struct{
        uint32_t parent;
        uint32_t child;
}__attribute__ ((packed)) Exec_t ;


typedef struct{
        enum Type type:8;
        union  {
                Comm_t comm;
                Process_t fork;
                Accept_t accept;
                Close_t close;
                Bind_t bind;
                Xfer_t xfer;
                Exec_t exec;
        }__attribute__ ((packed));
}__attribute__ ((packed)) Forensic_info_t ;
```

Appendix B: Hardware support in Bear:

File System. Although not central to the research goals, as with all operating systems, a working file system is required to access persistent storage. The BSD operating system uses the HammerFS file system, which has several desirable features (e.g. efficient file storage, performance and file history). With these features in mind, I investigated exporting HammerFS for use in the system. Unfortunately, current BSD/Linux File systems are tightly coupled to numerous internal operating system functions. This makes segregating the file system into a standalone entity extremely difficult and probably not tractable in the context of this project.

During the course of the research, an open source version of the FAT file system, called FATFS, was found available for use with no license restrictions. The FATFS file system implements the Microsoft FAT file system, still widely used as the basis of almost all portable media, such as USB sticks and portable device flash media. FATFS is designed for embedded systems and to be easily ported to different platforms. The advantage of an embedded design is that the file system was constructed in a self contained manner and segregates out the interface into a minimum set of functions. FATFS was also designed to use minimal system resources and has an extremely small code base.

To instantiate, the file system only six functions must be implemented: *read, write, initialize, status, control,* and *time*. These functions encompass the job of the disk driver, which is responsible for coordinating the lowest level access to the physical disk. There are three primary types of physical disk interfaces in use, IDE, SATA, and SCSI -- of these, IDE is the least complex. Initially believing the Dell Blade hardware was

backwards compatible with the IDE protocol, work began on developing an IDE disk driver in a virtualized development environment for faster testing, with the intent of porting it to the physical hardware after it was complete.

To verify the backwards compatibility of the blade hardware I investigated implementing the driver on the physical hardware. Unfortunately, Dell chose to use a proprietary RAID chipset and does not use the Intel controller. The chipset implements the Serial Attached SCSI or SAS protocol, which is a variation of the SATA protocol, and is significantly more complex than traditional IDE disk access. The disk driver registers itself with the SAS protocol handlers, requiring an implementation of the infrastructure support for generating SAS protocol packets. The protocol also deprecates support for PIO mode, all transfers must be done through DMA. To determine the DMA addresses requires support for PCI devices. All SATA devices are registered in the PCI device bus. To detect these addresses requires the implementation of a PCI subsystem to initialize the devices and detect their memory locations. The final stack of systems required to simply implement a file system is shown in Figure 34. To build a simple file system because of the hardware restrictions requires a PCI subsystem to effect DMA transfers, implementation of an advanced disk driver and protocol handling of the SAS interface layer.

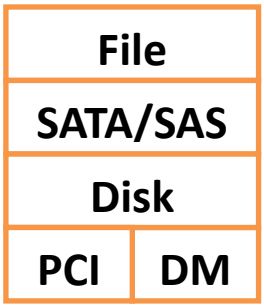


Figure 34: File system required subsystems

PCI System

The Peripheral Component Interconnect PCI bus is the home to all accessory hardware, which is connected to a modern computer, with some exceptions for specialized graphics. This bus allows the system to detect and access the resources of hardware plugged into the motherboard of the system. The PCI bus uses I/O ports similar to IDE. The familiarity gained in working with these ports through the disk driver development eased the development curve of the PCI bus. After researching, the protocol the first task was developing code to 'walk' the PCI bus and enumerate the devices present. Once all the devices were detected, determining the specific configuration is accomplished by reading the configuration registers of each device. The registers store the information in a series of little endian bytes. After locating the byte that determines the specific type of header the device uses, the information can then be loaded into memory structures representing each device based on the layout in the specification XX. The most difficult part of detecting the device configuration is locating the memory regions assigned to it.

PCI devices each contain from one to six Base Address Registers (BARS). Each register contains the addresses necessary for communicating with the device. The registers

indicate the type of access used, either IO port or memory, and the starting location. To determine the type the first bit is checked, IO ports are indicated by a 1 and memory registers by a 0. Then to determine the upper limit of memory or ports a series of 1's are written to the register, the resulting value's first bit is masked, and then bitwise inverted to determine the upper limit. At this point the software is successfully able to register all devices in the system and detect their port and memory spaces assigned by the system BIOS.

Disk Driver

The challenge with developing a driver is the lack of available documentation. Vendors typically develop drivers in-house with inherent knowledge of how their hardware works, while following the protocol defined in the specifications. The public specifications, such as the IDE protocol distributed by the T13 standards body, defines every possible function and functionality of the protocol, it does not provide a development roadmap or minimum subset that must be implemented. The IDE protocol has two overall modes of transferring data from the disk: Programmable I/O PIO and Direct Memory Access DMA. Disk initialization is accomplished via PIO mode transfers, which made it the natural development starting point. Each disk contains a number of data registers, a command register and an alternate status register. These registers are mapped to system ports by the BIOS at boot, typically to standardized addresses. However, the location must be confirmed by testing the suspected ports, as the manufacturer could map them to different locations. The key design concept behind understanding how to use the registers is they are bidirectional, and each has multiple meanings depending on the particular command issued. To issue a command the software loads the values into the

data registers and issue the instruction to the command register. After the instruction is issued, the command register then serves as the status register and values are written back into the same data registers. To determine when the disk has finished filling the registers with the requested information software must either poll the device status or register and handle the standard interrupt. For early development, the driver polls the device currently.

After spending copious amounts of time reverse engineering the proprietary LSI raid driver used in the Dell system, an impasse was reached as the memory systems are under development in parallel. This occurred because the protocol used to communicate with the drive required memory mapped I/O registers. These registers in previous generations were represented as hardware ports, mapped by the BIOS with no intervention required by the host operating system. As the name suggests the memory, mapped registers must be mapped into the system virtual memory, allowing access to the control registers of the device. This is in addition to the requirement for Direct Memory Access DMA transfers, which require the mapping of physical memory into the virtual address space and facilities for translating from physical to virtual and the reverse. Further the system does not yet map out to the physical memory range where the registers are located, preventing access to the control registers of the hard drive.

In the process of developing a driver, a solution was discovered. The Dell hardware does support modern SATA disks, they simply required is a different hardware card. After acquiring the card and new SATA drives the task then became interfacing with the SATA protocol. There are three modes of operation within the SATA protocol: sub-legacy, native PCI, and AHCI. In this particular server, Dell chooses to only implement the latter

two modes. AHCI again requires DMA and was ruled out. Native PCI mode is similar to the legacy protocol used on the current driver. The key distinction is that the ports are no longer located at fixed addresses and are reassigned by the BIOS at boot. The new location of the ports was enumerated with additions to the existing PCI subsystem code. However, development was not as straight forward as assumed. Working with a physical disk exposed flaws in the previous driver implementation because the virtualized driver was not subject to the strict timing requirements of the drive. Locating the errors required first confirming the protocol set used by the disk. Once it was confirmed to operate in native mode, with register access enabled, the task was to debug the driver code for the appropriate timings.

Once the appropriate references were located, the ATA reference manuals, careful study of the timing requirements for physical hardware and proper command sequence allowed the challenge to be overcome. The first task was to access the drive configuration space, to assign the size requirements and model variables to the file system. The overall process to initialize a disk is outlined in Figure 35. The device presence is detected by the BIOS, along with its access type that can only be modified by the BIOS developer. The operating system is then responsible for scanning the PCI bus to find the device and then issue commands to read its configuration. Then issue the commands to initialize the drive software, extract the information about the device such as size, number of sectors, vendor, and memory access mode. Finally, the information is passed to the file system so that it can format the disk in a recognizable configuration at the logical layer for future use.

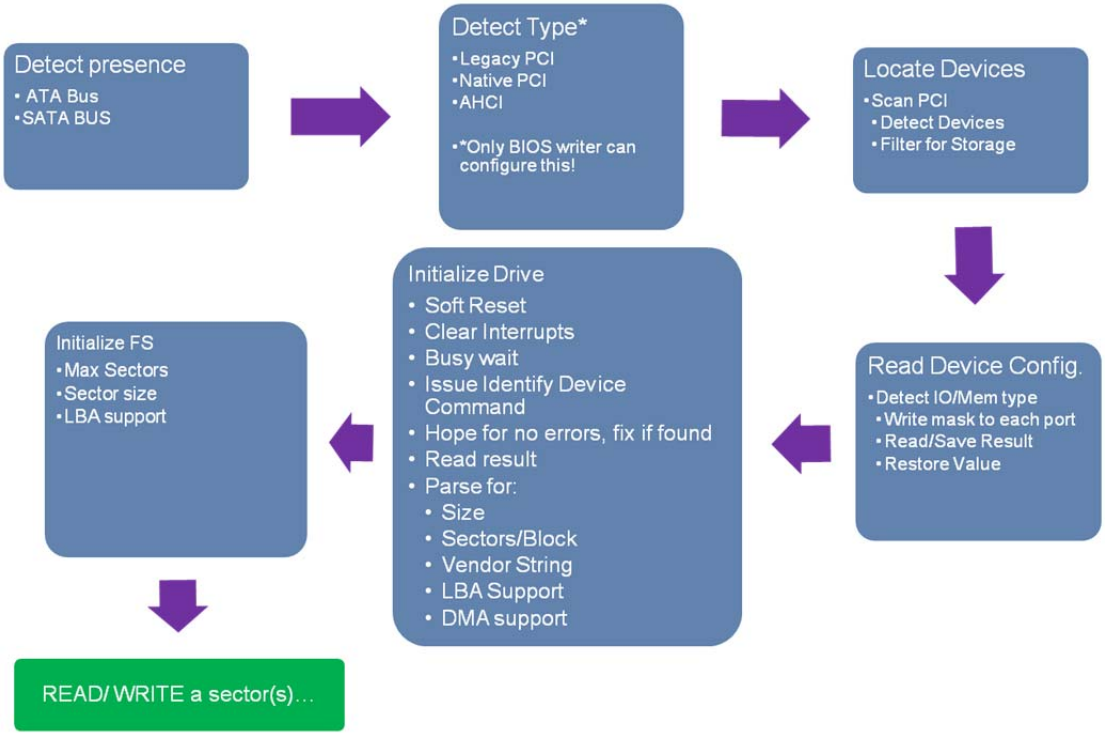


Figure 35: SATA disk initialization flow chart.

Reading from the disk follows a close protocol to drive identification and implemented next, followed by the ability to write to sectors of the disk, Figure 36. The green box is the polling functions that must be called before attempting to send a command to the disk or read from it. These register checks verifies that the disk is physically prepared to receive new commands and indicates the completion of read jobs and the error state if there is one. To then read or write a sector first the devices registers are loaded, then the count register with the number of blocks to read. Without the use of DMA the code must loop checking until the disk is ready. Once data is available, the driver may continue to read and write until the number of desired blocks has been transferred to or from the disk.

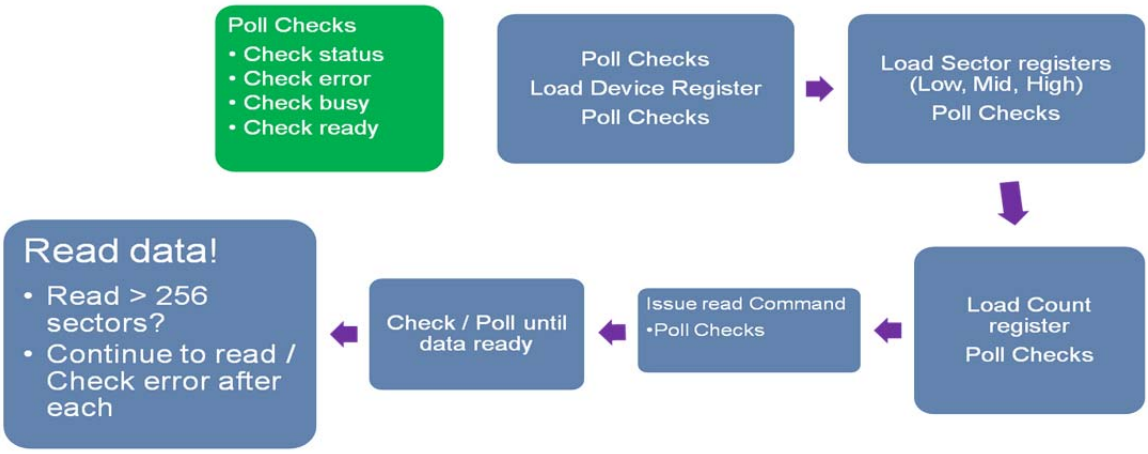


Figure 36: Hard Disk Access.

The disk driver is successfully able to read and write multiple sectors and accurately determine the size of the disk and initialize the requisite parameters. Having achieved this, the task was to bind the driver to the file system. The driver prototype has been re-written in the format required by the FATFS interface.

References:

1. O'Gorman J, Kearns D, and Aharoni M. Metasploit: The Penetration Tester's Guide. No Starch Press. 2011.

2. Davi L, Dmitrienko A, Sadeghi A, and Winandy M. Privilege escalation attacks on android. In: Anonymous Information Security. Springer, 2011: 346-360.

3. Hoglund G, Butler J. Rootkits: subverting the Windows kernel. Addison-Wesley Professional. 2006.

4. Levine J, Grizzard JB, and Owen HL. Detecting and categorizing kernel-level rootkits to aid future detection. Security & Privacy, IEEE 2006; 4: 24-32.

5. Eilam E. Reversing: secrets of reverse engineering. Wiley. com. 2005.

6. Eagle C. The IDA pro book: the unofficial guide to the world's most popular disassembler. No Starch Press. 2008.

7. Forrester JE, Miller BP. An empirical study of the robustness of windows NT applications using random testing. 2000; 59-68.

8. Sutton M, Greene A, and Amini P. Fuzzing: brute force vulnerability discovery. Pearson Education. 2007.

9. Checkoway S, Davi L, Dmitrienko A, Sadeghi A, Shacham H, and Winandy M. Return-oriented programming without returns. 2010; 559-572.

10. Prandini M, Ramilli M. Return-oriented programming. Security & Privacy, IEEE 2012; 10: 84-87.

11. Sullivan K, Knight JC, Du X, and Geist S. Information survivability control systems. 1999; 184-192.

12. Weber C. Assessing security risk in legacy systems. Cigital, Inc., Copyright 2006; .

13. Golomb G. Finding injection attacks by looking for injection attacks is a fail. 2011.

14. Scarfone K, Mell P. Guide to intrusion detection and prevention systems (IDPS). 2007; 800-94.

15. Garfinkel S, Spafford G. Web security commerce. Sebastopol, CA, USA: O'Reilly & Associates, Inc. 1997.

16. Norton security - antivirus software | norton store. 2010, from http://buy.norton.com/estore/mf/landingProductFeatures.

17. Quynh NA, Takefuji Y. Towards a tamper-resistant kernel rootkit detector. 2007; 276-283.

18. Anti-rootkit | free rootkit removal | rootkit detection - sophos. 2010, from http://www.sophos.com/products/free-tools/sophos-anti-rootkit.html.

19. RootkitRevealer. 2010, from http://technet.microsoft.com/en-us/sysinternals/bb897445.aspx .

20. Desmet L, Piessens F, Joosen W, and Verbaeten P. Bridging the gap between web application firewalls and web applications. 2006; 67-77.

21. Barracuda web application firewall - web application protection against hackers and vulnerabilities. 2010, from http://www.barracudanetworks.com/ns/products/web-site-firewall-overview.php.

22. Tan KMC, Killourhy KS, and Maxion RA. Undermining an anomaly-based intrusion detection system using common exploits. 2002; 54-73.

23. Case A, Cristina A, Marziale L, Richard GG, and Roussev V. FACE: Automated digital evidence discovery and correlation. Digital Investigation 2008; 5: S65.

24. Petroni J,Nick L., Fraser T, Walters A, and Arbaugh WA. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. 2006.

25. Petroni NL, Walters AA, Fraser T, and Arbaugh WA. FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. Digital Investigation 2006; 3: 197-210.

26. Schatz B. BodySnatcher: Towards reliable volatile memory acquisition by software. Digital Investigation 2007; 4: 126.

27. Simon M, Slay J. Enhancement of forensic computing investigations through memory forensic techniques. Availability, Reliability and Security, International Conference on 2009; 0: 995-1000.

28. Garfinkel T, Rosenblum M. A virtual machine introspection based architecture for intrusion detection. Network and Distributed System Security Symposium 2003.

29. Hay B, Nance K. Forensics examination of volatile system data using virtual introspection. SIGOPS Oper.Syst.Rev. 2008; 42: 74-82.

30. Glaser EL, Couleur JF, and Oliver GA. System design of a computer for time sharing applications. 1965; 197-202.

31. Denning PJ. Virtual memory. ACM Comput.Surv. 1970; 2: 153-189.

32. Chen PM, Noble BD. When virtual is better than real. 2001; 133.

33. Carrier BD, Grand J. A hardware-based memory acquisition procedure for digital investigations. Digital Investigation 2004; 1: 50.

34. Schreiber SB. Undocumented Windows 2000 secrets: a programmer's cookbook. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. 2001.

35. Petroni J,Nick L., Fraser T, Molina J, and Arbaugh WA. Copilot - a coprocessor-based kernel runtime integrity monitor. 2004; 179-194.

36. Harvey AF. DMA fundamentals on various PC platforms. April 1991; Application Note 011.

37. Rutkowska J. Beyond the CPU: Defeating hardware based RAM acquisition. BlackHat 2007.

38. Bahram S, Jiang X, Wang Z, Grace M, Li J, Srinivasan D, Rhee J, and Xu D. DKSM: Subverting virtual machine introspection for fun and profit. Reliable Distributed Systems, IEEE Symposium on 2010; 0: 82-91.

39. L4Ka Team. L4Ka pistachio kernel. from http://l4ka.org/projects/pistachio/.

40. Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, and Winwood S. seL4: Formal verification of an OS kernel. 2009; 207-220.

41. Molina J, Arbaugh WA. Using independent auditors as intrusion detection systems. 2002; 291-302.

42. Bond M, Anderson R. API-level attacks on embedded systems. Computer 2001; 34: 67-75.

43. Corbat'o F. J., Vyssotsky VA. Introduction and overview of the multics system. 1965; 185-196.

44. Intel Corp. Intel virtualization technology specification for the IA-32 architecture. from www.intel.com/technology/vt/.

45. Baliga A, Iftode L, and Chen X. Automated containment of rootkits attacks. Computer Security 2008; 27: 323.

46. Ernst MD, Perkins JH, Guo PJ, McCamant S, Pacheco C, Tschantz MS, and Xiao C. The daikon system for dynamic detection of likely invariants. Sci.Comput.Program. 2007; 69: 35-45.

47. Payne B. XenAccess. 2008.

48. Nguyen AM. MAVMM: Lightweight and purpose built VMM for malware analysis. Computer Security Applications Conference, Annual 2007; 441.

49. Dolan-Gavitt B, Leek T, Zhivich M, Giffin J, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. 2011; 297.

50. VMware I. Vmware vproves reference. from http://www.vmware.com/pdf/ws7_f3_vprobes_reference.pdf .

51. Moser A, Kruegel C, and Kirda E. Exploring multiple execution paths for malware analysis. Security and Privacy, IEEE Symposium on 2007; 0: 231-245.

52. Xu C, Andersen J, Mao ZM, Bailey M, and Nazario J. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. 2008; 177.

53. Apk. Interface promiscuity obscurity. Phrack 1998; 8.

54. Halflife. Bypassing integrity checking systems. Phrack 1997; 7.

55. Li Q, Hao Q, Xiao L, and Li Z. VM-based architecture for network monitoring and analysis. 2008; 1395-1400.

56. Srivastava A, Giffin J. Tamper-resistant, application-aware blocking of malicious network connections. 2008; 39-58.

57. Jiang X, Wang X, and Xu D. Stealthy malware detection and monitoring through VMM-based “out-of-the-box” semantic view reconstruction. ACM Trans.Inf.Syst.Secur. 2010; 13: 12:1-12:28.

58. Baliga A, Chen X, and Iftode L. Paladin: Automated detection and containment of rootkit attacks. 2006.

59. Litty L, Lie D. Manitou: A layer-below approach to fighting malware. 2006; 6-11.

60. Quynh NA, Takefuji Y. Towards a tamper-resistant kernel rootkit detector. 2007; 276-283.

61. Sparks S, Butler J. Shadow walker - raising the bar for rootkit detection. Phrack Jul 2005; 11(63): 8.

62. Portokalidis G, Slowinska A, and Bos H. Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. SIGOPS Oper.Syst.Rev. 2006; 40: 15-27.

63. Jones ST, Arpaci-Dusseau AC, and Arpaci-Dusseau RH. Antfarm: Tracking processes in a virtual machine environment. 2006; 1-1.

64. Rhee J, Riley R, Xu D, and Jiang X. Defeating dynamic data kernel rootkit attacks via VMM-based guest-transparent monitoring. 2009; 74-81.

65. Appel AW, Li K. Virtual memory primitives for user programs. SIGOPS Oper.Syst.Rev. 1991; 25: 96-107.

66. Riley R, Jiang X, and Xu D. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. 2008; 1-20.

67. Riley R, Jiang X, and Xu D. Multi-aspect profiling of kernel rootkit behavior. 2009; 47-60.

68. Seshadri A, Luk M, Qu N, and Perrig A. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. 2007; 335-350.

69. Shacham H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). 2007; 552-61.

70. Erlingsson U, Schneider FB. SASI enforcement of security policies: A retrospective. 2000; 87-95.

71. Grizzard JB. Towards self-healing systems: Re-establishing trust in compromised systems. 2006.

72. Petroni J,Nick L., Hicks M. Automated detection of persistent kernel control-flow attacks. 2007; 103-115.

73. Abadi M, Budiu M, Erlingsson U, and Ligatti J. Control-flow integrity principles, implementations, and applications. ACM Trans.Inf.Syst.Secur. 2009; 13: 4:1-4:40.

74. Sources O. Voices from the open source revolution, january 1999, ISBN: 1-56592-582-3. URL: http://www.oreilly.com/catalog/opensources/book/toc.html 2.

75. Chou A, Yang J, Chelf B, Hallem S, and Engler D. An empirical study of operating systems errors. ACM. 2001.

76. Khoshgoftaar TM, Munson JC. Predicting software development errors using software complexity metrics. Selected Areas in Communications, IEEE Journal on 1990; 8: 253-261.

77. Endres A. An analysis of errors and their causes in system programs. 1975; 10: 327-336.

78. Rosenberg J. Some misconceptions about lines of code. 1997; 137.

79. McConnell S. Code Complete, Second Edition. Redmond, WA, USA: Microsoft Press. 2004.

80. Gaffney JE. Software Engineering, IEEE Transactions on title={Estimating the Number of Faults in Code 1984; SE-10: 459.

81. Ramesh KS. Design and development of MINIX distributed operating system. 1988; 685.

82. Heiser G, Elphinstone K, Kuz I, Klein G, and Petters SM. Towards trustworthy computing systems: Taking microkernels to the next level. SIGOPS Oper.Syst.Rev. 2007; 41: 3-11.

83. Miller FP, Vandome AF, and McBrewster J. MINIX 3: Unix-like, Operating system, 0BSD licenses, Embedded system, OLPC XO-1, IBM PC compatible, Emulator, Virtual machine, Bochs, VMware Workstation, Windows Virtual PC, QEMU, XScale, MINIX. Alpha Press. 2009.

84. Nichols C. Bear–a Resilient Core for Distributed Systems A Thesis 2013.

85. Intel Corp. Intel 64 and IA-32 architectures software Developer's manuals. 2012. 2012, from http://download.intel.com/products/processor/manual/325462.pdf.

86. Strachey C. Time sharing in large fast computers. International Conference on Information Processing 1959; 336-341.

87. Hoernes GE, Hellerman L. An experimental 360/40 for time-sharing. Datamation 1958; 14: 39-42.

88. Belady LA, Parmelee RP, and Scalzi CA. The IBM history of memory management technology. IBM J.Res.Dev. 1981; 25: 491-504.

89. Meyer RA, Seawright LH. A virtual machine time-sharing system. IBM Systems Journal 1970; 9: 199-218.

90. Popek GJ, Goldberg RP. Formal requirements for virtualizable third generation architectures. Commun ACM 1974; 17: 412-421.

91. Adams K, Agesen O. A comparison of software and hardware techniques for x86 virtualization. 2006; 2-13.

92. Rosenblum M, Garfinkel T. Computer title={Virtual machine monitors: current technology and future trends 2005; 38: 39.

93. Agesen O. Method and system for implementing subroutine calls and returns in binary translation sub-systems of computers. 2000; 09/688,091.

94. Uhlig R, Neiger G, Rodgers D, Santoni AL, Martins FCM, Anderson AV, Bennett SM, Kagi A, Leung FH, and Smith L. Intel virtualization technology. Computer 2005; 38: 48-56.

95. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, and Warfield A. Xen and the art of virtualization. 2003; 164-177.

96. Kivity A, Kamay Y, Laor D, and Lublin U, Liguori. KVM: The linux virtual machine monitor. OLS 2007; 225-2250230.

97. Deshane T, Shepherd Z, Matthews J, Ben-Yehuda M, Shah A, and Rao B. Quantitative comparison of xen and KVM. Xen Summit, Boston, MA, USA 2008; 1-2.

98. Che J, Yu Y, Shi C, and Lin W. A synthetical performance evaluation of openvz, xen and kvm. 2010; 587-594.

99. Cerbelaud D, Garg S, and Huylebroeck J. Opening the clouds: Qualitative overview of the state-of-the-art open source VM-based cloud management platforms. 2009; 22.

100. Adams K, Agesen O. A comparison of software and hardware techniques for x86 virtualization. SIGOPS Oper.Syst.Rev. 2006; 40: 2-13.

101. Clause J, Doudalis I, Orso A, and Prvulovic M. Effective memory protection using dynamic tainting. 2007; 284-292.

102. Huang Y, Stavrou A, Ghosh AK, and Jajodia S. Efficiently tracking application interactions using lightweight virtualization. 2008; 19-28.

103. Schwartz EJ, Avgerinos T, and Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). 2010; 317-331.

104. Zhu D, Jung J, Song D, Kohno T, and Wetherall D. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. SIGOPS Oper.Syst.Rev. 2011; 45: 142-154.

105. Krishnan S, Snow KZ, and Monrose F. Trail of bytes: Efficient support for forensic analysis. 2010; 50-60.

106. King ST, Chen PM. Backtracking intrusions. SIGOPS Oper.Syst.Rev. 2003; 37: 223-236.

107. Orebaugh A, Ramirez G, and Beale J. Wireshark & Ethereal network protocol analyzer toolkit. Syngress. 2006: .

108. Jacobson V, Leres C, and McCanne S. The tcpdump manual page. Lawrence Berkeley Laboratory, Berkeley, CA 1989; .

109. Roesch M. Snort: Lightweight intrusion detection for networks. 1999; 99: 229-238.

110. Sassaman L, Patterson ML, Bratus S, Locasto ME, and Shubina A. Security applications of formal language theory. 2011.

111. Sassaman L, Bratus S. The halting problems of network stack insecurity. UseNix 2011.

112. Dhamankar R, Dausin M, Eisenbarth M, King J, Kandek W, Ullrich J, Skoudis E, and Lee R. Top cyber security risks. from http://www.sans.org/top-cyber-security-risks/.

113. Round robin DNS load balancing. May 20, 2004. from http://www.content.websitegear.com/article/load_balance_dns.htm.

114. Goldberg RP. Survey of virtual machine research. IEEE computer Magazine 1974; 7: 34-45.

115. Creasy RJ. The origin of the VM/370 time-sharing system. IBM Journal of Research and Development 1981.

116. Lo J. VMware and CPU virtualization. http://download3.vmware.com/vmworld/2005/pac346.pdf .

117. XEN. from http://www.xen.org .

118. KVM. from http://www.linux-kvm.org .

119. Nguyen AM, Schear N, HeeDong Jung, Godiyal A, King ST, and Nguyen HD. Computer Security Applications Conference, 2009.ACSAC '09.Annual title={MAVMM: Lightweight and Purpose Built VMM for Malware Analysis 2009; 441.

120. Wang J, Jajodia S, Huang Y, and Ghosh A. On-demand virtual work system. pending; .

121. Jiang W, Yih H, and Ghosh A. SafeFox: A safe lightweight virtual browsing environment. System Sciences (HICSS), 2010 43rd Hawaii International Conference on title={SafeFox: A Safe Lightweight Virtual Browsing Environment 2010; 1.

122. Libvirt: The virtualization API. 2010, from http://libvirt.org/ .

123. Liu C, Albitz P. DNS and BIND. O'Reilly Media, Inc. 2006.

124. P. Vixie E, S. Thomson, Y. Rekhter, and J. Bound. RFC 2136: Dynamic updates in the domain name system (DNS UPDATE). 1997.

125. Snoeren AC, Andersen DG, and Balakrishnan H. Fine-grained failover using connection migration. 2001; 19-19.

126. FUSE file system in user space. From http://fuse.sourceforge.net .