A Survey of Forensic Analysis in Virtualized Environments

STEPHEN KUHN and STEPHEN TAYLOR, Dartmouth College, Hanover, New Hampshire

tr11-007

This article presents a survey of current approaches to memory forensics in virtualized environments. 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. Almost absent from the literature are capabilities to incorporate network traffic into the forensic process making remote exploits difficult to discover. A considerable number of the techniques are complicated by the failure of modern operating systems to adopt available protection schemes 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*.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey K.6.5 [**Information Systems**]: Management of Computing and Information Systems –Security and Protection.

General Terms: Forensics, Security, Malware Analysis

Additional Key Words and Phrases: Virtualization, Forensics, Security, Root-Kit Detection, Virtual Machine Introspection

1. Introduction

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 [Strachey 1959], [Hoernes and Hellerman 1958], [Belady et al. 1981], it has typically operated on server class machines, such as the IBM 360 [Meyer and Seawright 1970], due to implementation challenges associated with commodity x86 systems [Popek and Goldberg 1974], [Adams and Agesen 2006]. Unfortunately, early x86 processor designs did not include secure and efficient capabilities for switching the context of an entire operating system. The 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 [Rosenblum and Garfinkel 2005], [Agesen 2000]. More recently, Intel and AMD processors have been augmented with new instructions, termed VTX extensions [Uhlig et al. 2005], that save, restore, and manipulate an entire operating system context without the overheads associated with trapping instructions. Both these implementation strategies provide secure and efficient mechanisms previously restricted to business class machines.

[1] This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-09-1-0213.

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. In the server arena, an alternative standalone implementation, termed a *hypervisor*, replaces the generic operating system with a small, optimized runtime environment. This environment provides only the ability to bootstrap and control virtual machines and is typified by the Xen [Barham et al. 2003] and KVM [Kivity et al. 2007] solutions.

The forensic challenge of collecting and understanding memory use to analyze computer network attacks has been the focus of numerous studies [Case et al. 2008], [Petroni et al. 2006], [Petroni et al. 2006], [Schatz 2007], [Simon and Slay 2009]. 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) [Garfinkel and Rosenblum 2003], [Hay and Nance 2008]. 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 1. 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 [Glaser et al. 1965][Denning 1970].

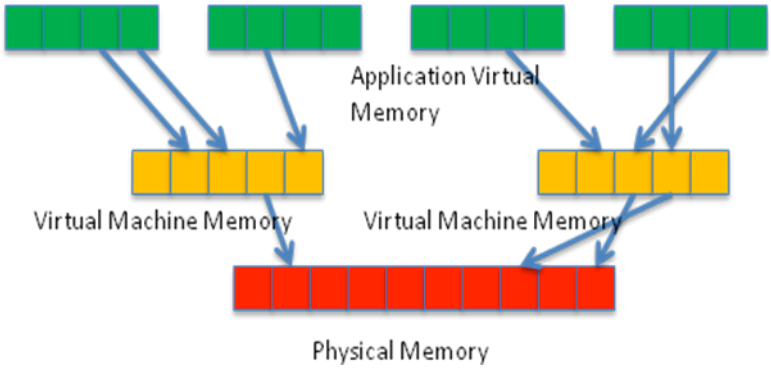

Figure 1. Memory Virtualization support for virtual machines

To understand memory use the virtual machine layer of memory translation 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 appear as random data. This *semantic gap* reflects the difference in view from inside and outside an executing virtual machine [Chen and Noble 2001].

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 [Carrier and Grand 2004]. Root-kits and other malware seek to obscure their presence by manipulating the results from standard collection tools by providing false information [Schreiber 2001]. 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.

The prevailing thought in hardware approaches to forensics [Carrier and Grand 2004], [Petroni et al. 2004] is to use Direct Memory Access (DMA) to retrieve the content of memory. DMA allows peripheral hardware to directly access main memory and bypass the processor, saving time and improving performance [Harvey April 1991]. 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 [Rutkowska 2007][Bahram et al. 2010].

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 [L4Ka Team][Klein et al. 2009].

2. Hardware Based Memory Forensics

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 a PCI card [Molina and Arbaugh 2002]. 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 [Bond and Anderson 2001]. 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 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 an 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 which do not involve file access.

Copilot [Petroni et al. 2004] 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 used 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 rootkits. 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 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 [Corbat\'o F. J. and Vyssotsky 1965]. Support for this separation is now available in the paging structures associated with current processor designs [Intel Corp.].

[Carrier and Grand 2004] 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.

[Baliga et al. 2008] have developed a hardware proof-of-concept for rootkit detection using *system invariance*. The authors have adapted the Diakon software package to guess likely program invariants (i.e. characteristics of the program that do not change) from multiple runs [Ernst et al. 2007]. In contrast to Copilot that utilizes hashing to detect changes in *known* kernel structures, this system detects violations of the detected invariants. The associated PCI card allows their software package to analyze kernel memory for root-kits or malicious code on a separate machine, minimizing performance impact to the host system and providing anti tampering capability. This presents a potential solution to the challenge of pages that mix code and data. The software was 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 [Rutkowska 2007]. In addition, the prohibitive costs and delays associated with the approach makes it difficult to maintain as a viable solution. 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.

3. 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.

3.1 Forensic Toolkits

General solutions for resolving the semantic gap between physical memory, virtual machine memory, and application memory have already been developed and are available in two primary implementations: *XenAccess* [Payne 2008] and MAVMM [Nguyen 2007].

XenAccess provides a programming interface that can extract virtual machine information to an underlying hypervisor. This information can 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.

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 expressly designed to change its functionality when executing in a virtual environment to avoid detection [Moser et al. 2007][Xu et al. 2008].

3.2 Secure Platforms

Unfortunately, many traditional computer security tools are vulnerable to deception through a wide variety of techniques [Apk 1998], [Halflife 1997]. 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 on which to execute security tools. [Li et al. 2008] 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 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[Srivastava and Giffin 2008]. It provides 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 VMware hypervisor has been modified to allow inclusion of security tools encapsulated through the notion of *modules* [Garfinkel and Rosenblum 2003]. The challenges associated with this approach are to develop an operating system independent API that bridges the semantic gap, translate virtual machine addresses into operating system specific addresses, and define a policy engine to handle 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: in general, 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 through the API. If the results differ, the module alerts due the likely presence of a malicious modification. The *user program integrity detector* module 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 it's MAC 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.

The secure platform approach has also be applied in the context of a honeypot (i.e. a system specifically designed to lure attackers) [Jiang and Wang 2007]. The approach relocates forensic instrumentation out of the virtual machine and into a hypervisor. This allows system instructions 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 the 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 [Jiang et al. 2010b]. 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.

3.3 Customized Hypervisors

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

[Baliga et al. 2006]. 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. [Litty and Lie 2006] demonstrate this concept on application code while [Quynh and Takefuji 2007] show a similar approach 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 [Bahram et al. 2010] [Sparks and Butler Jul 2005]. 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 [Portokalidis et al. 2006], [Jones et al. 2006]. 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 [Portokalidis et al. 2006]. 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..

Recall that 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 [Rhee et al. 2009]. 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 between 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. It provides 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 execution of a portion of a virtual machine's kernel execution into the hypervisor. This provides an extra layer of security if the virtual machine is compromised by ensuring that rootkits do not have equal privilege with the virtual machine kernel. 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 [Appel and Li 1991]: every page must be marked as executable, even if only a tiny fragment is executable code. [Riley et al. 2008] 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 [Riley et al. 2009]. 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. [Baliga et al. 2008] have proposed a similar scheme to that of [Riley et al. 2009] 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 files. Both implementations were able to successfully detect 27 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 [Baliga et al. 2008] 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 [Seshadri et al. 2007]. 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, except the approved 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 hardware 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 ran57% slower than native performance. Application level performance was strongly impacted: two benchmarks, kernel build and Postmark, reported 66% and 51% performance impacts respectively.

Integrity Monitoring. The embedding of an arbitrary code segment into the normal control flow a program (ie. 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 [Litty et al. 2008] [Shacham 2007]. Control flow integrity verification is an extension of integrity monitors discussed in [Erlingsson and Schneider 2000], [Grizzard 2006]. The primary idea is to verify that the target of any branch in a programs 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 [Erlingsson and Schneider 2000] implemented within a hypervisor [Petroni and Hicks 2007]. Validation of static kernel segments (ie 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 monitor is executed 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, [Abadi et al. 2009] 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 provided through compiler techniques and appropriate runtime protection.

4. Related Work:

Apart from forensic analysis, other aspects of computer security can also benefit from the use virtualization. One approach is to utilize the technology to create virtual networks for testing potentially disruptive network services. The most successful of these implementations is PlanetLab, which employs hundreds of machines worldwide that researchers can build experiments upon [Chun et al. 2003]. In addition, a number of similar implementations have been reported, such as: vBet [Jiang and Xu 2003] Netbed/Emulab [White et al. 2002], ORBIT [Raychaudhuri et al. 2005] and OneLab [Cappello et al. 2005].

Simply browsing the internet exposes a system to an exponentially growing number of attacks. One mitigation technique has sought to use virtualization to provide a *clean slate* approach that prevents persistent infection of client side applications such as web-browsers. No attempt is made to detect the presence of malicious software, each instance of the web browser is launched in its own virtual machine clearing all changes made by the application and simultaneously removing any infection [Wang et al. pending][Jiang et al. 2010a]. A similar clean state approach used by [Kuhn and Taylor submitted] non-deterministically refreshes a server to a known gold-standard configuration, simultaneously removing undetected kernel-level compromises and root-kits. This approach non-deterministically changes network properties to hide virtual machines around the network and may also utilize camouflage to provide disinformation [Kanter and Taylor submitted][Taylor et al. Submitted].

5. Conclusion

This survey has considered the research challenges associated with the forensic analysis of memory in virtualized environments. This work has benefited from a long history associated with hardware root-kit detection methods. Unfortunately, the high cost of hardware development and evolution has prevented the emergence of an enduring solution. Virtualized environments, offer many of the opportunities associated with hardware methods by providing a protected environment to view the operation of a virtual machine. 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. Unfortunately, a quantitative comparison of techniques provided by the literature has proved elusive: no common collective base of root-kits, applications, and kernel versions has emerged that can form a ground-truth for cross technology comparisons.

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.

Notice

References

1. ABADI M., BUDIU M., ERLINGSSON U., and LIGATTI J. Control-flow integrity principles, implementations, and applications. *ACM Trans.Inf.Syst.Secur.* 13: 1 (2009), 4:1-4:40.

2. ADAMS K., AGESEN O. A comparison of software and hardware techniques for x86 virtualization. (2006), 2-13.

3. AGESEN O. Method and system for implementing subroutine calls and returns in binary translation sub-systems of computers. 09/688,091: 6,711,672,B1 (2000), .

4. APK. Interface promiscuity obscurity. *Phrack.* 8: 53 (1998).

5. APPEL AW., LI K. Virtual memory primitives for user programs. *SIGOPS Oper.Syst.Rev.* 25: Special Issue (1991), 96-107.

6. 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.* 0: (2010), 82-91.

7. BALIGA A., IFTODE L., and CHEN X. Automated containment of rootkits attacks. *Comput Secur.* 27: 7-8 (2008), 323.

8. BALIGA A., CHEN X., and IFTODE L. Paladin: Automated detection and containment of rootkit attacks. (2006).

9. 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.

10. BELADY LA., PARMELEE RP., and SCALZI CA. The IBM history of memory management technology. *IBM J.Res.Dev.* 25: 5 (1981), 491-504.

11. BOND M., ANDERSON R. API-level attacks on embedded systems. *Computer.* 34: 10 (2001), 67-75.

12. CAPPELLO F., DJILALI S., FEDAK G., HERAULT T., MAGNIETTE F., NÉRI V., and LODYGENSKY O. Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. *Future Generation Comput Syst.* 21: 3 (2005), 417.

13. CARRIER BD., GRAND J. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation.* 1: 1 (2004), 50.

14. CASE A., CRISTINA A., MARZIALE L., RICHARD GG., and ROUSSEV V. FACE: Automated digital evidence discovery and correlation. *Digital Investigation.* 5: Supplement 1 (2008), S65.

15. CHEN PM., NOBLE BD. When virtual is better than real. (2001), 133.

16. CHUN B., CULLER D., ROSCOE T., BAVIER A., PETERSON L., WAWRZONIAK M., and BOWMAN M. PlanetLab: An overlay testbed for broad-coverage services. *SIGCOMM Comput.Commun.Rev.* 33: 3 (2003), 3-12.

17. CORBAT\'O F. J., VYSSOTSKY VA. Introduction and overview of the multics system. (1965), 185-196.

18. DENNING PJ. Virtual memory. *ACM Comput.Surv.* 2: 3 (1970), 153-189.

19. ERLINGSSON U., SCHNEIDER FB. SASI enforcement of security policies: A retrospective. (2000), 87-95.

20. 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.* 69: 1-3 (2007), 35-45.

21. GARFINKEL T., ROSENBLUM M. A virtual machine introspection based architecture for intrusion detection. *Network and Distributed System Security Symposium.* (2003), .

22. GLASER EL., COULEUR JF., and OLIVER GA. System design of a computer for time sharing applications. (1965), 197-202.

23. GRIZZARD JB. Towards self-healing systems: Re-establishing trust in compromised systems. (2006).

24. HALFLIFE. Bypassing integrity checking systems. *Phrack.* 7: 51 (1997), .

25. HARVEY AF. DMA fundamentals on various PC platforms. Application Note 011: (April 1991).

26. HAY B., NANCE K. Forensic examination of volatile system data using virtual introspection. *SIGOPS Oper.Syst.Rev.* 42: 3 (2008), 74-82.

27. HOERNES GE., HELLERMAN L. An experimental 360/40 for time-sharing. *Datamation.* 14: 4 (1958), 39-42.

28. INTEL CORP. Intel virtualization technology specification for the IA-32 architecture.

29. 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.* (2010a), 1.

30. 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.* 13: 2 (2010b), 12:1-12:28.

31. JIANG X., WANG X. Out-of-the-box monitoring of VM-based high-interaction honeypots. (2007), 198-218.

32. JIANG X., XU D. vBET: A VM-based emulation testbed. (2003), 95-104.

33. JONES ST., ARPACI-DUSSEAU AC., and ARPACI-DUSSEAU RH. Antfarm: Tracking processes in a virtual machine environment. (2006), 1-1.

34. KANTER M., TAYLOR S. *MILCOM.* (submitted), .

35. KIVITY A., KAMAY Y., LAOR D., and LUBLIN U, LIGUORI. KVM: The linux virtual machine monitor. *OLS.* (2007), 225-2250230.

36. 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.

37. KUHN S., TAYLOR S.
Increasing attacker workload with virtual machines. *MILCOM.* (submitted), .

38. L4KA TEAM. L4Ka pistachio kernel.

39. LI Q., HAO Q., XIAO L., and LI Z. VM-based architecture for network monitoring and analysis. (2008), 1395-1400.

40. LITTY L., LAGAR-CAVILLA HA., and LIE D. Hypervisor support for identifying covertly executing binaries. (2008), 243-258.

41. LITTY L., LIE D. Manitou: A layer-below approach to fighting malware. (2006), 6-11.

42. MEYER RA., SEAWRIGHT LH. A virtual machine time-sharing system. *IBM Systems Journal.* 9: 3 (1970), 199-218.

43. MOLINA J., ARBAUGH WA. Using independent auditors as intrusion detection systems. (2002), 291-302.

44. MOSER A., KRUEGEL C., and KIRDA E. Exploring multiple execution paths for malware analysis. *Security and Privacy, IEEE Symposium on.* 0: (2007), 231-245.

45. NGUYEN AM. MAVMM: Lightweight and purpose built VMM for malware analysis. *Computer Security Applications Conference, Annual.* (2007), 441.

46. PAYNE B. XenAccess. 0.5: (2008).

47. PETRONI J,NICK L., HICKS M. Automated detection of persistent kernel control-flow attacks. (2007), 103-115.

48. 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).

49. PETRONI J,NICK L., FRASER T., MOLINA J., and ARBAUGH WA. Copilot - a coprocessor-based kernel runtime integrity monitor. (2004), 179-194.

50. 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.* 3: 4 (2006), 197-210.

51. POPEK GJ., GOLDBERG RP. Formal requirements for virtualizable third generation architectures. *Commun ACM.* 17: 7 (1974), 412-421.

52. 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.* 40: 4 (2006), 15-27.

53. QUYNH NA., TAKEFUJI Y. Towards a tamper-resistant kernel rootkit detector. (2007), 276-283.

54. RAYCHAUDHURI D., SESKAR I., OTT M., GANU S., RAMACHANDRAN K., KREMO H., SIRACUSA R., LIU H., and SINGH M. 3: (2005), 1664.

55. RHEE J., RILEY R., XU D., and JIANG X. Defeating dynamic data kernel rootkit attacks via VMM-based guest-transparent monitoring. (2009), 74-81.

56. RILEY R., JIANG X., and XU D. Multi-aspect profiling of kernel rootkit behavior. (2009), 47-60.

57. RILEY R., JIANG X., and XU D. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. (2008), 1-20.

58. ROSENBLUM M., GARFINKEL T. *Computer title={Virtual machine monitors: current technology and future trends.* 38: 5 (2005), 39.

59. RUTKOWSKA J. Beyond the CPU: Defeating hardware based RAM acquisition. *BlackHat.* (2007).

60. SCHATZ B. BodySnatcher: Towards reliable volatile memory acquisition by software. *Digital Investigation.* 4: Supplement 1 (2007), 126.

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

62. 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.

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

64. SIMON M., SLAY J. Enhancement of forensic computing investigations through memory forensic techniques. *Availability, Reliability and Security, International Conference on.* 0: (2009), 995-1000.

65. SPARKS S., BUTLER J. Shadow walker - raising the bar for rootkit detection. *Phrack.* 11(63): (Jul 2005), 8.

66. SRIVASTAVA A., GIFFIN J. Tamper-resistant, application-aware blocking of malicious network connections. (2008), 39-58.

67. STRACHEY C. Time sharing in large fast computers. *International Conference on Information Processing.* (1959), 336-341.

68. TAYLOR S., HENSON M., KANTER M., KUHN S., MCGILL K., and NICHOLS C. Bear--A resilient operating system for scalable multi-processors. *IEEE.* (Submitted).

69. 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.* 38: 5 (2005), 48-56.

70. WANG J., JAJODIA S., HUANG Y., and GHOSH A. On-demand virtual work system. (pending), .

71. WHITE B., LEPREAU J., STOLLER L., RICCI R., GURUPRASAD S., NEWBOLD M., HIBLER M., BARB C., and JOGLEKAR A. An integrated experimental environment for distributed systems and networks. *SIGOPS Oper.Syst.Rev.* 36: SI (2002), 255-270.

72. 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.