SEMANTIC VIEW RE-CREATION FOR THE SECURE MONITORING OF VIRTUAL MACHINES

A Thesis
Presented to
The Academic Faculty

by

Martim Carbone

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2012

SEMANTIC VIEW RE-CREATION FOR THE SECURE MONITORING OF VIRTUAL MACHINES

Approved by:

Professor Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Karsten Schwan
School of Computer Science
Georgia Institute of Technology

Professor Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

Dr. Weidong Cui
Security and Privacy Research Group
Microsoft Research

Professor Jonathon Giffin
School of Computer Science
Georgia Institute of Technology

Date Approved: 13 June 2012

To my dear mother,

for all the love and support.

ACKNOWLEDGEMENTS

This work owes its existence to the support provided by a large number of people over a timespan of several years, and I would like to acknowledge them.

First of all, I would like to thank my advisor, Wenke Lee. Ever since the start of my Ph.D., Professor Lee encouraged me to pursue my interests and gave me the freedom to do so. He always provided me with the material and financial support necessary to conduct my work and grow as a researcher. In times of doubt and uncertainty regarding the future of my research, he was always there to give me support and advice. His mentoring proved invaluable on helping me build my research skills, and his example as a successful, respected researcher has made me always try to aim high with my goals. I will always carry inside me, with great pride, these years spent working with him.

I would also like to acknowledge my thesis committee members: Mustaque Ahamad, Weidong Cui, Jonathon Giffin and Karsten Schwan. I thank them for agreeing to be part of my committee and providing comments and suggestions that helped increase the quality of my work.

During my Ph.D., I have been fortunate to collaborate with several outstanding engineers and researchers in academia and industry. Amongst them, I would first like to acknowledge Dr. Weidong Cui, whose mentoring and support over two summer internships and many months of extended collaboration taught me a lot about research, systems and security. In industry, I would also like to acknowledge the mentoring and support of Matthew Conover, Bruce Montague, Marcus Peinado, Sanjay Sawhney and Diego Zamboni. In academia, I am thankful to Long Lu, Bryan Payne, Monirul Sharif and Abhinav Srivastava.

My Ph.D. experience would not have been the same without the support and friendship of my lab-mates, some of which have become truly great friends. Therefore, I would like to acknowledge Manos Antonakakis, Yizheng Chen, Artem Dinaburg, Brendan Dolan-Gavit, Ikpeme Erete, Prahlad Fogla, Guofei Gu, Yeongjin Jang, Andrea Lanzi, Byoungyoung Lee, Long Lu, Yacin Nadji, Bryan Payne, Roberto Perdisci, Monirul Sharif, Kapil Singh, Chengyu Song, Abhinav Srivastava, Xinyu Xing and Junjie Zhang. Their friendship made all the difference.

The work infrastructure and student support facilities provided by the Georgia Tech Information Security Center, the College of Computing and the Georgia Institute of Technology deserve a distinguished mention, as it proved crucial to my work and quality of life.

Last but not least, I would like to give a special thanks to my family for their continuous love and support throughout all these years that I have been away from home. The decision to leave my home country and come to the United States to pursue a doctorate was not easy for either of us, but you have always encouraged me to aim high and pursue my dreams. And even though thousands of miles keep us apart, I always carry you in my heart and it gives me strength.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The insecurity of modern-day software systems has created the need for security monitoring applications, such as anti-virus tools. These applications conduct passive monitoring of the system's state and active monitoring of the system's events. Two serious deficiencies are commonly found in such applications. First, their lack of isolation from the system being monitored allows malicious software to tamper with or disable them. Second, the lack of secure and reliable monitoring primitives in the operating system compromises their visibility, making them easy to be evaded.

A technique known as Virtual Machine Introspection attempts to solve these problems by leveraging the strong isolation and mediation properties of full-system virtualization. It isolates the monitoring application in a separate, security virtual machine, from where it can securely monitor a guest virtual machine by leveraging the hypervisor's view of resources. This separation creates, however, a problem known as *semantic gap*, which can be defined by the loss of a high-level view of the guest's state and events from the part of the monitoring application. It occurs as a result of the low-level separation enforced by the hypervisor between the guest and the security virtual machine.

This thesis proposes and investigates novel techniques to overcome the semantic gap, advancing the state-of-the-art on the syntactic and semantic guest view re-creation for security applications that conduct passive and active monitoring of virtual machines.

In the space of passive monitoring, we propose a new technique for reconstructing a syntactic view of the guest OS kernel's heap state. By applying a combination of

static code and dynamic memory analysis techniques, we are able to reconstruct a map of the guest OS's dynamic kernel objects. Our key contribution over previous work is the accuracy and completeness of our analysis, which translates into stronger monitoring capabilities for security applications.

Although sufficient for certain types of integrity checking applications, a syntactic view of the guest state is not enough for others that require access to information at a higher level. With this in mind, we propose a technique that combines the security of out-of-VM monitoring with the semantic awareness of in-VM monitoring. By allowing out-of-VM applications to invoke and securely execute API functions inside the monitored guest's kernel, we eliminate the need for the application to know details of the guest's internal data structures. Our key contribution over previous work is the ability to overcome the semantic gap between the monitoring application and the guest OS in a robust and secure manner, by relying on the guest's own code.

A security monitoring solution cannot be complete without an active monitoring component that intercepts and evaluates guest events as they happen. In this space, we propose a new virtualization-based event monitoring technique based on the interception of kernel data modifications as opposed to code execution trapping. Our key contribution over previous work is the ability to monitor high-level operating system events without the need for in-guest components and without the same circumvention problems of code execution hooks, and the ability to automatically re-create the syntactic context of guest kernel memory accesses.

CHAPTER I

INTRODUCTION

1.1 Insecurity of Modern Systems

Computer systems have become a central ingredient of the technological mix that supports modern life. In fifty years, they have evolved from no more than a handful of isolated, room-sized behemoths into a wide variety of interconnected devices that number in the billions [34]. Unfortunately, in recent years, this rapid expansion has been matched by the growing number of security problems that threaten the confidentiality, integrity and availability of digital information.

Malicious computer activity has now long been dissociated from the notion of the idealistic hacker who breaks into computer systems to satisfy his curiosity. Presently, the industry of malicious software, or *malware*, moves an estimated US$100 billion/year worldwide and has strong ties with criminal organizations [27]. Malware spreads through the Internet, and its goals include identity theft, industrial theft, espionage, cyber-warfare, bank and credit card fraud, and SPAM, among others, causing significant losses to governments, businesses and individuals. In 2011, McAfee reported having received over 75 million unique malware samples [61], and the trend has been increasing for the past several years. It is estimated that 35.5% of the nodes connected to the Internet are infected with some type of malware [72].

Clearly, a serious security problem exists and its causes are several. First, security is not always considered an important requirement in the system development process. Second, even when it is, limitations with current programming languages and verification techniques make it difficult to provide hard security guarantees. Third, security is dependent not only on the system itself but also on the education and

awareness of the end user.

This reality has created the consensus in industry and academia that designing and implementing systems that behave 100% securely is an unreachable goal. As a result, a large part of the current security effort is reactive in nature, starting from the assumption that software is vulnerable and will be exploited eventually, and thus needs to be monitored for such.

1.2 Host-based Security Monitoring

Security monitoring applications are commonly found today both at the network and host level. Host-based security monitors reside inside the monitored system and perform passive analysis of state and/or active analysis of events that may indicate a security compromise. Common examples include anti-virus applications, host-based firewalls and memory/disk integrity checkers. We argue that the effectiveness of such monitors can be determined as a measure of the following two requirements. They are based on the original requirements from the *reference monitor* definition given in the Trusted Computer System Evaluation Criteria (TCSEC) [28].

1. *Visibility:* The monitor must have complete and continuous access to the system's state and events, without any risk of circumvention or evasion by a malicious entity;

2. *Isolation:* The monitor must be protected from the monitored system. Its confidentiality, integrity and availability cannot be compromised by a malicious entity that has gained administrative control of the monitored system.

The first requirement ensures that the monitoring application has access to all sources of information necessary for its decision-making, while the second requirement ensures that the application itself is protected. We do not include the *Correctness* reference monitor requirement in this list due to is being a property of the monitoring application itself, and not of the monitoring infrastructure that supports it.

Today's security monitors fall considerably short of complying with such requirements. This stems from the fact that they are designed and implemented as common user-space processes or kernel drivers that reside inside the operating system that is being monitored. Therefore, the monitor's view of the system relies on basic facilities provided by the OS, such as memory mapping, filesystem access and event redirection. Consequently, if the OS kernel is compromised, this view can be tampered with, resulting in an inaccurate/incomplete view. Isolation is also a problem, since malware that has administrative system privileges or access to the OS kernel can easily tamper with or disable the monitoring application. Modern operating systems simply do not provide the visibility and isolation guarantees necessary for the trustworthy deployment of security monitoring applications.

1.3 The Role of Virtualization

Full-system virtualization has been used mainly for workload consolidation and compatibility purposes. It presents, however, interesting possibilities in the area of systems security, and specifically for host-based security monitoring. This direction was pioneered by a technique known as Virtual Machine Introspection (VMI), first proposed by Garfinkel et al. [38]. This approach is also referred to as *out-of-VM monitoring* or *external monitoring*. It improves the *Isolation* and *Visibility* requirements of security monitoring when compared to traditional host-based security monitors.

First, VMI leverages the inter-domain separation enforced by the hypervisor to provide a significantly stronger level of isolation when compared to the process-level isolation implemented by modern OSes. Specifically, it separates the monitoring application from the monitored guest OS by placing each in a distinct VM: a trusted security VM (SVM) and an untrusted guest VM (GVM). This relies on the assumption that the hypervisor cannot be compromised, and its low-level inter-VM isolation cannot be overcome.

Second, VMI taps into the hypervisor's view of the GVM, allowing the monitoring application to passively (i.e., conduct scans) or actively (i.e., trap specific events) monitor the GVM. This level of visibility is greater than the one provided by traditional host-based monitors, since the hypervisor's low-level view of state and events covers the entire guest OS and cannot be circumvented from inside the GVM.

1.4 Challenges

Despite its security benefits, several challenges are associated with introspection. The most significant one concerns the level of visibility of the GVM state provided by the hypervisor to the monitoring application, a phenomenon known as *semantic gap*. As a low-level resource manager, the hypervisor knows nothing of the internal semantics of the GVM. From a passive monitoring perspective, all it sees are memory pages, CPU registers, disk blocks, and other low-level state. From an active monitoring perspective, the events it sees are interrupts, memory exceptions and instruction traps. This data is at a level too low to be useful to most security applications, which are commonly interested in monitoring the system at a higher abstraction level.

Previous work has addressed this problem in different ways. The most straightforward of them is to manually study, extract and encode the syntax and semantics of the GVM state into the monitoring application. The encoded information can then be used to infer higher-level information from the low-level data. This approach is not desirable for several reasons. First, it is time-consuming, as it needs to be done manually. It also lacks robustness, given that even small changes made to the syntax or the semantics of the GVM (i.e., patches) can render the application non-functional until the manual extraction effort is repeated. Other approaches attempt to bridge the semantic gap in a more automated fashion but are prone to a variety of completeness, robustness, and security problems. A detailed discussion of these appears in Chapter 2.

The second major challenge associated with external monitoring is performance. Performance degradation is expected when compared to traditional, non-virtualized host-based monitoring, due to the additional CPU and memory resources consumed by virtualization. Introspection itself also adds to that cost as a result of additional world switches between the virtual machines and the hypervisor involved for most passive and active monitoring operations. It is important, however, that the overhead stays within acceptable limits. This must be considered as a primary requirement when conceptualizing and implementing solutions.

1.5 Dissertation Overview

Given the challenges discussed above, this thesis *proposes and investigates novel techniques to overcome the semantic gap, advancing the state-of-the-art on the syntactic and semantic guest view re-creation for security applications that conduct passive and active out-of-VM monitoring of guest operating systems.*

In this dissertation, we first provide background information and discuss related work in Chapter 2. In Chapter 3 we present our first contribution. We propose an out-of-VM memory analysis technique for automatically reconstructing a syntactic view of the guest OS kernel's heap state [16]. This view includes information regarding the location of objects in memory, their types and contents. By applying a combination of static code analysis of the kernel's source code and dynamic memory analysis on its memory image, our KOP system is able to reconstruct a map of the guest OS's dynamic kernel objects with near-complete coverage and accuracy. We demonstrate how this map can be used by security applications based on passive monitoring to detect the presence of malware. *Our key contribution over previous work is the accuracy and completeness of our syntactic analysis, which translates into stronger monitoring capabilities for certain types of security applications.*

In Chapter 4 we present our second contribution. Although a syntactic view is

enough for certain types of systematic integrity checking applications, others often require a higher semantic-level view. With this in mind, we present a passive monitoring technique that combines the security of out-of-VM monitoring with the semantic view of in-VM monitoring [15]. Our infrastructure, SYRINGE, leverages the guest OS's own code to overcome the semantic gap and collect meaningful guest state information. SYRINGE creates a secure end-to-end guest code execution chain whereby a guest OS function can be securely invoked from the monitoring application, securely executed and have its results securely reported back to the application. *Our key contribution over previous work is the ability to overcome the semantic gap between the monitoring application and the guest OS in a secure and robust manner by relying on the guest's own code.*

In Chapter 5, we present our third contribution. While the two previous contributions focus on passive monitoring, our third contribution is in the context of active monitoring. Solutions for active monitoring of guest OSes to-date face the challenge of having to rely on in-guest components that monitor code execution, creating the possibility of tampering and circumvention. To address this issue, we propose a new virtualization-based event interception primitive for active monitoring based on the interception of data modifications. By intercepting events at the data access level instead of the code execution level, our solution eliminates the need for in-guest components and their shortcomings, significantly reducing the risk of circumvention. *Our key contribution over previous work is the ability to monitor high-level operating system events without any in-guest components and a reduced risk of circumvention, and the ability to automatically re-create the syntactic context of guest kernel memory accesses.*

Finally, Chapter 6 concludes this thesis by summarizing the work done, discussing open problems and future research opportunities in this area and presenting our closing remarks.

CHAPTER II

BACKGROUND AND RELATED WORK

2.1 Isolation

Isolation is a critical requirement for systems security, being one of the defining characteristics of a Trusted Computing Base (TCB). It is necessary to prevent elements of the untrusted portion of the system from corrupting the trusted portion, and thus undermining all security assumptions. Isolation can be achieved through hardware or software techniques, or a combination of both.

Early secure system designs already try to leverage the isolation strength of small, low-level software monitors, such as micro-kernels and virtual machine monitors. In the 1980s, KVM/370 [39, 86] retrofitted security into the existing VM/370 system by using a VMM as the TCB for strong isolation. Other works during that time period proposed new, security-driven OS designs. The security kernel was proposed as a small, verifiable OS subsystem that implements the reference monitor concept [94]. The separation kernel was proposed as a specialized security kernel that put more emphasis on isolation [84]. Micro-kernels perform a complete modularization of the operating system, re-architecting its subsystems as isolated modules running in user-space and have a very small kernel running in supervisor mode to handle elementary privileged operations, such as thread scheduling and memory mapping [2, 44, 56]. This design has the potential to increase reliability and security, but does not perform as efficiently as a monolithic kernel due to the frequency of address space transitions.

Despite this early work on secure systems design, modern commodity OSes, such as Windows and Linux, adopt a straightforward, performance-oriented, monolithic design that does not prioritize isolation between processes and between user and

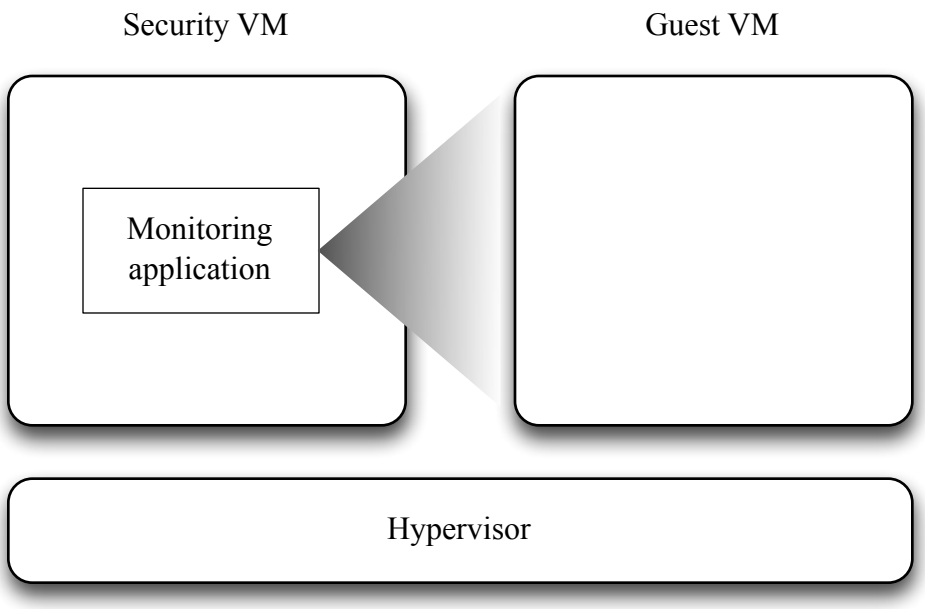

Figure 1: Virtual Machine Introspection, where the monitoring application is placed on a security VM and relies on the hypervisor to monitor a guest VM.

kernel-space. As a result, security monitoring tools and the operating system itself cannot be adequately protected. This problem is one of the main motivations of this thesis and a variety of other works, whose goal it to create a foundation for isolation while still running unaltered, commodity OSes.

The CoPilot system adopts a hardware-based approach by deploying a security monitor inside a PCI card installed on the monitored system. This monitor scans the integrity of a Linux system through DMA while minimizing the risk of tampering [76]. This approach is known as coprocessor-based isolation, as has also been used to protect filesystem integrity checkers [69] and intrusion detection systems [111].

The requirement for special-purpose hardware limits the adoption of coprocessor-based techniques. As a software-based alternative, the seminal work by Garfinkel et al. [38] first introduced the concept of virtual machine introspection (VMI), in which a hypervisor is used to isolate the security application from the monitored system (Figure 1). This is done by placing the application inside its own, trusted security VM running a commodity OS, and having it externally monitor the guest VM with

aid from the hypervisor. The hypervisor's small size and the low-level nature of its management improves significantly upon the isolation provided by modern OSes. This principle is shared with the micro-kernel design, although several differences exist between the two approaches [40, 43]. Various VMI frameworks, such as XenAccess [73], VMwatcher [50], and VMware's VMsafe [104] have since been proposed for different hypervisors, such as Xen [12] and VMware ESX Server [103]. Similarly, a large number of security applications relying on introspection-based isolation have been proposed [74, 50, 78, 15, 97].

In the case where support for multiple guest VMs is not needed, the use of tiny security-oriented hypervisors have been proposed by several works [17, 87, 87]. Tiny hypervisors are about one order of magnitude smaller than full-featured hypervisors. SecVisor addresses the problem of kernel code integrity for Linux systems, by using hypervisor memory protections [87]. BitVisor focuses instead on I/O security, such as storage encryption [91].

Several recent works have tried to use virtualization to retrofit isolation into commodity operating systems, allowing security agents to stay in-guest and thus improve performance. Lares adopts a simplistic approach by placing in-guest a stateless kernel agent and write-protecting its code [74]. SIM goes a step further and allows the protection of stateful kernel-level agents by creating a new address space inside the guest [90]. Overshadow uses virtualization to protect an in-guest application's confidentiality and integrity in the presence of a compromised guest OS, but does not protect the execution of guest OS code and does not address agent availability concerns [22]. Finally, substantial research effort has been devoted to the problem of driver isolation in commodity OSes, as these tend to be the main source of reliability problems [60, 19, 105, 33, 100, 108]. These works adopt a combination of address space relocation, static analysis and instrumentation and inline reference monitoring.

2.2 Passive External Monitoring

The major challenge faced by passive out-of-VM security monitors is overcoming the semantic gap as extracting useful information from the guest's raw memory state. First efforts solved this problem by manually encoding the syntax and semantics of the guest memory state inside the monitoring application. Systems like CoPilot [76] and Livewire [38] monitor the static code and data sections of the guest for signs of attacks. Petroni et al. later extended CoPilot to map and analyze dynamic kernel data structures for semantic integrity constraints [77].

Other works have attempted to address the semantic gap problem in a more systematic fashion. VMwatcher [50] and SBCFI [78] discuss automatic ways of reconstructing the guest's kernel memory state to perform security checks. SBCFI in particular shows how the mapping of kernel objects can be used to check for malicious function pointer values. Gibraltar uses a similar technique to derive and enforce kernel data structure invariants [11]. All these systems, however, can only leverage static type information and therefore require a substantial amount of guest code annotation, resulting in unsatisfactory coverage.

Since the source code and/or symbol information may not always be available, more recent work has approached this problem without assuming access to the source code. Laika [25] uses Bayesian unsupervised learning to automatically infer the location and overall structure of the data objects used by user-level applications. Despite its flexibility, this approach is prone to false positives. Dolan-Gavitt et al. [31] use memory fuzzing to derive robust signatures for kernel objects, which can then be used by a memory analyzer to locate kernel objects in a memory image. Rewards [58] tracks a program's calls to known API functions (with known parameter types) and dynamically propagates this knowledge throughout the program to construct a map of data structures. Howard [92] builds upon Rewards by also being able to identify data structures internal to a program. SigGraph [57] attempts to identify kernel data

structures by brute-force scanning the memory and using graph-based signatures. Unlike KOP, however, it has limited coverage.

A promising approach to solving the semantic gap relies on securely leveraging the guest code itself to extract wanted information. This relieves the monitoring application from having to know details (derived automatically or otherwise) about the guest memory semantics, instead simply using the guest code as a tool to handle it. This idea was first proposed by Joshi et al. [52] as an auxiliary mechanism to their record-and-replay system, and further developed by SADE [23]. SADE stealthily injects a small driver agent into the guest and uses it to call guest OS functions. Its injected agent and the guest code itself are, however, unprotected and thus vulnerable to attack. Virtuoso [30] shares the same insight. It relies on pre-extracted execution traces of guest monitoring functions to automatically generate introspection programs that can be executed in the security VM. These traces must be extracted before any monitoring can be performed, and must be re-extracted whenever the guest OS is updated. Virtuoso also suffers from the fundamental incompleteness of dynamic analysis, which can create significant runtime hazards for the generated introspection programs. VMTS expands on Virtuoso by not relying on execution traces and thus not being vulnerable to code path coverage problems [35]. Its techniques, however, are very dependent on the Linux guest operating system, and it is not clear how well they could be adapted to other OSes, like Windows.

2.3 Active External Monitoring

Active monitoring of applications and operating system events has always played a prominent role with security monitoring solutions. Recent research has leveraged virtualization to isolate and protect the monitoring application, while still maintaining access to a VM's low-level events through the hypervisor.

This idea has been demonstrated by XenAccess to infer high-level filesystem events

from low-level disk block write operations [73]. VMwall uses a similar approach to monitor network events initiated by the guest and correlate them with in-guest state as a means to detect unauthorized activity [97]. Antfarm extended this idea beyond the realm of I/O, by correlating low-level address space switches captured by the hypervisor to high-level creation, destruction and scheduling of processes [51]. VMScope uses an emulator to intercept and analyze all system call-triggering instructions executed in a honeypot environment [49]. Patagonix monitors code execution inside the guest VM by relying on the hypervisor's memory virtualization [59]. It intercept code execution events and verifies the memory pages containing the code by comparing them with a whitelist.

Active techniques that operate completely outside the guest have, in most cases, the disadvantage of having to deal with the semantic gap. The exception is with low-level events that happen to exactly match high-level ones, as is the case with Antfarm (process scheduling) and VMScope (system call invocation). This does not, however, generalize to all types of event. As a result, protected in-guest event hooking infrastructures have been proposed, which allow an event to be captured at a higher level, inside the guest. Lares places hooks in the system call table and protects them by marking the corresponding pages read-only from inside the hypervisor [74]. These hooks trigger a hypervisor exit that gets forwarded to the monitoring application running in the security VM. SIM goes a step further by allowing the monitoring application to stay in-guest and protecting it through the use of additional hypervisor-enforced guest address spaces [90].

One of the more radical ideas have recently been proposed by Srinivasan et al., in which not only the monitoring application is removed from the guest VM, but also the monitored application [95]. System calls from the monitored application are forwarded to the guest OS, allowing it to remain isolated from the security VM while reaping the benefits of having it executing in the same environments as the monitoring

application, and thus effectively reducing the semantic gap problem. This technique is not, however, applicable system-wide, but only to a select few applications.

2.4 Secure Code Execution

The simplest way to enforce secure code execution is to enforce its static integrity: the code must not be modified in runtime. Numerous works have attempted to reach this goal through a variety of techniques, one of which is virtualization. SecVisor uses virtualization to write-protect kernel code pages in the guest's memory and only allows the loading of modules that have been pre-included in a whitelist [87]. NICKLE enforces the same property using a different technique that relies on maintaining multiple versions of code pages [81].

Trusted computing and its attestation capabilities provide another way of achieving this goal. Sailer et al. demonstrate a TPM-based technique for attesting the integrity of Linux kernel modules [85]. Terra does the same thing to a VMM. There have been works that do not rely on special hardware support, while still performing a type of attestation. Arbaugh et al. pioneered the ideas of secure and trusted boot by proposing a new verifiable boot architecture [8]. Pioneer allows a verifier to attest that the execution of a certain piece of code on an untrusted environment has occurred without any form of tampering [88]. More recent work in this area, such as Flicker, leverages dynamic TPM secure launch capabilities to provide secure code execution capabilities while relying on a minimal TCB [63].

Static code integrity, however, is not enough in most cases. A lot can go wrong with an application or a kernel driver after it is loaded, even if its code is kept intact. The software's control flow, for instance, can be maliciously diverted, as with stack overflow vulnerabilities, or even its non-control data tampered with [21]. As a result, several alternatives that either complement or replace static code integrity have been proposed.

Kiriansky et al. proposed the program shepherding technique to protect systems against application vulnerabilities [55]. It uses an emulator-based trap-and-emulate approach to dynamically monitor the execution of control transfer instructions in a program to ensure that it does not deviate from a certain execution policy. The same effect can be achieved through a technique known as *Inline Reference Monitoring* (IRM), which relies on static or dynamic code instrumentation to monitor the execution of certain types of instructions [32]. This technique has been applied in numerous works in the area of secure code execution. Abadi et al. uses IRM to enforce a code execution property known as control-flow integrity [1], which relies on monitoring and analyzing the target of indirect control-flow instructions, as well as ensuring that the code itself is not modified. IRM can therefore also be used for intra-address space isolation.

2.5 Virtualization

The idea behind operating systems virtualization is not new, dating to the 1960s. It was first implemented in the VM/370 system to enable time-sharing in IBM System/370 mainframes. It the last fifteen years, however, OS virtualization has experienced a great surge of popularity in the enterprise environment due to its abstraction, compatibility, flexibility and isolation capabilities. These have enabled companies to save money by consolidating workloads and thus maximizing resource usage. They have also enabled a variety of other applications related to testing, security, and the new service-oriented computing paradigm known as *cloud computing*.

The basic requirements and mechanics of virtualization were first proposed and formalized by Popek and Goldberg in their seminal article [79]. They proposed *efficiency*, *resource control* and *equivalence* as the fundamental properties of a virtual machine monitor (VMM). The first refers to the native execution of innocuous instructions; the second refers to a VMM's full control and mediation of basic system

resources; and the third refers to the compatibility property of full virtualization. Popek and Goldberg also discuss what it means for a computer architecture to be *virtualizable*, stating that this holds if the set of CPU instructions that modify basic system resources (called *sensitive* instructions) is a subset of those instruction that generate a trap if executed in non-privileged mode (called *privileged* instructions). This makes sense intuitively, as without this ability a VMM would not be able to maintain the illusion of virtualization for the virtual machines it supports and would enable a VM to tamper with other VMs and with the VMM itself. Architectures designed to specifically support virtualization, such as the IBM System/370, are virtualizable. This is not, however, the case with other more recent and popular architectures, such as the Intel x86. The challenges raised by the task of doing full virtualization on top of traditionally non-virtualizable architectures have spawned a variety of techniques whose understanding will be crucial in the following chapters.

2.5.1 CPU Virtualization

Virtualizing an ISA that does not follow the requirements discussed above is a challenging endeavor. VMware was the first to demonstrate transparent full-system virtualization of the x86 architecture through a technique known as *binary translation* [4, 3]. This technique circumvents the problem of x86 not trapping on all of its sensitive CPU instruction by translating them in memory, on-the-fly, into small segments of code that virtualize its behavior. This allows non-sensitive instructions to execute natively on the CPU, whereas the sensitive ones are replaced with its translated equivalents by the VMM. This approach results in excellent performance guest performance.

A different technique, called *para-virtualization*, relies instead on modifying the guest operating system to include explicit hypervisor calls, or *hypercalls*, to the VMM at those points where sensitive operations must be performed. This approach was

implemented by the first versions of Xen [12], but never gained wide adoptions due to its requirement that the guest OS be modified.

In 2006, Intel and AMD decided to tackle the original problem that made solutions like binary translation and para-virtualization a necessity. They extended the x86 architecture to enable *hardware-supported virtualization* by including new instructions and a set of new trapping controls [71]. These controls allow the VMM to trap on all sensitive CPU instructions, in addition to other optional ones. Hardware virtualization eliminates a lot of the VMM software complexity required by binary translation and provides similar performance [3]. As a result, it has been adopted by all major players in the industry and is the virtualization technique on which our work is based.

2.5.2 Memory Virtualization

Memory is one of the main resources that a VMM needs to virtualize in order to provide guests with a transparent and protected view of it. This involves introducing a new layer of addressing in the memory translation chain to control the guest's view of physical memory, *guest physical addresses* (GPA), in addition to the hardware-supported virtual and machine physical addresses (MPA). By controlling the mapping between GPAs and MPAs for each guest, a VMM is able to isolate and manage the memory resources of each guest, while at the same time providing the illusion that each guest has the entire physical address space to itself.

In practice, adding this new addressing layer requires a new level of address translating structures, or page tables, to be created inside the hypervisor. There are several different techniques for determining how these page tables operate and are maintained. The first and most straightforward technique, known as *shadow paging*, creates a collection of shadow page tables (SPTs) inside the hypervisor that mirror their equivalents inside the guest (GPTs) [48]. While GPTs map GVAs to GPAs, their corresponding SPTs map the same GVAs to MPAs. Thus, GPTs are not used

in the actual translation process, the SPTs serving this role. This level of indirection gives the hypervisor total control over guest memory mappings, without breaking the virtualization illusion. Shadow paging works by trapping all address space switches and all modifications made to GPTs by the guest, so that they can be propagated to their corresponding SPT. The number of hypervisor exits generated by classic shadow paging makes it quite performance-intensive, which has prompted the creation of more optimized versions of it, such as the *virtual TLB*.

More recently, CPU vendors have added explicit hardware support for memory virtualization through a technique known as *nested paging* [48, 13]. Through this technique, the hypervisor no longer has to monitor the GPTs to keep them synchronized with SPTs. Nested paging introduces a new type of addressing structure, the nested page table, that translates GPAs into MPAs. Thus, the hypervisor needs no knowledge of guest virtual addresses and the complete address translation chain, from GVA to MPA, is performed by the hardware by walking both the guest page tables and the nested page tables.

This method has the advantage of not requiring any hypervisor exits to maintain the shadow pages, and therefore greatly improves the performance of virtualized systems. It does complicate, however, the design and implementation of virtualization-based memory protection techniques that operate at the guest virtual level. Since the hypervisor no longer has control over a guest's virtual address space, it becomes very easy to evade such techniques by simply remapping entries in the guest page tables. Part of the work presented in Chapter 5 presents a solution to this problem.

CHAPTER III

MAPPING KERNEL OBJECTS FOR SYSTEMATIC INTEGRITY CHECKING

3.1 Motivation

Kernel-mode malware represents a significant threat because of its ability to compromise the security of the kernel and, hence, the entire software stack. Such malware tampers with kernel code and data to hide itself and collect useful information. Attempts have been made to solve this problem by proposing a variety of passive kernel integrity checking applications [38, 76, 77, 78, 11]. The basic idea is to locate, map and analyze the contents of the kernel code and data sections against a security policy to determine whether the kernel has been compromised or not.

Virtual machine introspection allows the integrity checker to be isolated from the monitored system, but presents the semantic gap challenges associated with out-of-VM monitoring. For certain types of systematic integrity checks, such as the ones based on scanning of code sections and analysis of object fields (e.g. function pointers), it is desirable to re-create a mid-level *syntactic* view consisting of the locations and types of kernel code and data objects.

It is relatively easy for an out-of-VM security monitor to re-create a syntactic view of the kernel code and the static portion of the kernel data due their static memory locations. It is, however, much harder to do that for dynamic data objects due to their unpredictable memory locations and volatile nature. Not surprisingly, dynamic data has become one of the most attractive targets for kernel-mode malware [46, 83] and is therefore the focus of our work.

The usual manner of locating a dynamic object is to find a *reference* to it, such as

a pointer. This pointer could, however, be located in another dynamic object, turning this into a recursive problem. Mapping all the dynamic objects involves performing a complete traversal of memory, starting from a set of statically-defined objects and following each pointer reference to the next object, until all have been covered. This process is complicated by challenges related to traversing generic pointers, whose target may vary at run-time, unions and dynamic arrays.

Previous out-of-VM kernel integrity checking solutions either limit themselves to kernel code and static data (e.g., system call tables) [38, 76], or can reach only a fraction of the dynamic kernel data [77, 78, 11], resulting in limited coverage and security. It is straightforward for an attacker, for instance, to implement new kernel-mode malware that tampers only with function pointers in objects that cannot be reached by these systems.

In this chapter, we describe a technique that achieves close to 100% coverage and accuracy in the syntactic view re-creation of the OS kernel's heap state. We demonstrate this technique through a passive kernel monitoring infrastructure, the Kernel Object Pinpointer (KOP). Unlike previous systems, KOP addresses the challenges in pointer-based memory traversal, such as generic pointers, unions and dynamic arrays. KOP first performs *static analysis* on the source code to construct an *extended type graph*. This extended type graph has not only type definitions and global variables but also all candidate target types for generic pointers. Given a memory snapshot, KOP then performs a *memory analysis* based on the extended type graph. KOP traverses the kernel memory, resolving type ambiguities caused by unions or generic pointers with information derived from the static analysis. The output is an *object graph* that contains all the identified kernel objects and their pointers to other objects.

Kernel systematic integrity checking applications can rely on the syntactic view provided by KOP's object graph to perform their tasks. To concretely demonstrate KOP's power, we present such an application that can be used to identify function

pointers manipulated by kernel-mode malware in dynamic kernel objects.

3.2 Previous Approaches

Kernel integrity has been the target of intense security research, given the increasing threat posed by kernel rootkits and other malware. Systems like CoPilot [76] and Livewire [38] passively check the static portion of the kernel memory for integrity violations. More elaborate types of checks were also shown by Petroni et al. [77] to verify the semantic consistency of dynamic kernel structures based on manually created rules.

State-based Control Flow Integrity [78] is similar to KOP as it also traverses the dynamic kernel object graph. By using a simple type graph and manual annotations, it verifies the value of function pointers at each object it finds against some policy (e.g., pointing to a known module). Gibraltar [11] and PoKeR [82] also rely on static type information and manual annotations to traverse the kernel object graph. These systems suffer from three major limitations. First, they cannot follow generic pointers (e.g., `void*`) because they only leverage type definitions and thus do not know the target types of generic pointers. Second, they cannot follow pointers defined inside unions since they cannot tell which union subtype should be considered. Third, they cannot recognize dynamic arrays and thus the objects inside them. Our study shows that these systems may miss up to 72% of the dynamic kernel data. Furthermore, these systems require all linked list constructs to be annotated so that the corresponding objects can be correctly identified by the traversal.

Another line of research has approached the problem of locating data structures without assuming access to the source code. Laika [25] uses Bayesian unsupervised learning to automatically infer the location and overall structure of the data objects used by user-level applications. Despite its flexibility, this approach is prone to false positives. Dolan-Gavitt et al. [31] use memory fuzzing to derive robust signatures

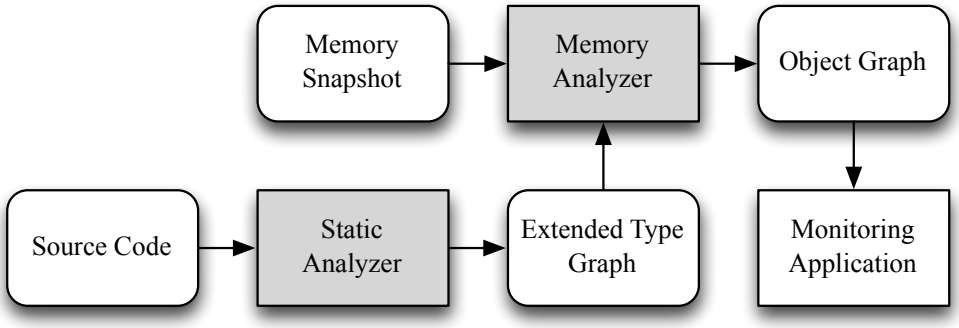

Figure 2: The KOP system architecture

for kernel objects, which can then be used by a memory analyzer to locate kernel objects in a memory image. Rewards [58] tracks a program's calls to know API functions (with known parameter types) and dynamically propagates this knowledge throughout the program to construct a map of data structures. Howard [92] builds upon Rewards by also being able to identify data structures internal to a program. SigGraph [57] attempts to identify kernel data structures by brute-force scanning the memory and using graph-based signatures. Unlike KOP, however, these systems have limited coverage.

Recent work has shifted the problem of monitoring the operating system kernel to that of monitoring the actual hypervisor, which in these works is not considered part of the trusted computing base. HyperSentry [10] and HyperCheck [106] leverage the System Management Mode (SMM) to isolate their integrity checkers.

3.3 The KOP Kernel Object Mapping Infrastructure

3.3.1 Overview

The goal of KOP is to *completely* and *accurately* map all kernel objects in a memory snapshot in order to enable systematic kernel integrity checking. In KOP, we refer to a live instance of a data type (or, a data structure) as an *object*. KOP has two main components: the static analysis component and the memory analysis component. Its

high-level architecture is shown in Figure 2.

KOP first performs static analysis on the kernel source code. It starts with an inter-procedural, inclusion-based points-to analysis [7] to derive a *points-to graph*. This is a directed graph whose nodes are pointers in the program and edges represent *inclusion* relationships. In other words, an edge from pointer x to pointer y means that any object pointers that can be derived from y are also derivable from x. Additionally, the points-to graph is maintained as a pre-transitive graph, i.e., the graph is not transitively closed [42].

Based on the pre-transitive points-to graph, KOP then infers candidate target types for generic pointers. *Generic pointers* are those whose target types cannot be extracted from their definitions. The term includes `void*` pointers as well as pointers defined inside linked list-related structures that are nested inside objects. The final output of KOP's static analysis component is an *extended type graph*. This is a directed graph where each node is either a data type or a global variable, and each edge connects two nodes with a label of (m, n). This means that the pointer field at offset m in the source node points to the target node at offset n. Note that we call this an *extended* type graph because it has edges corresponding to *generic pointer* fields which do not exist in the type graph derived from only type definitions.

Given a memory snapshot, KOP performs memory analysis by using the extended type graph to traverse the kernel memory. The output of the memory analysis component is an *object graph* whose nodes are instances of objects in the memory snapshot and edges are the pointers connecting these objects. Kernel data integrity checks can then be performed based on this object graph.

To help explain KOP's static and dynamic analysis, we will use the source code in Figure 3 as a running example. The code snippet shows the data structures and functions for inserting a `TXT_DATA` object or a `BIN_DATA` object into a singly-linked list (`WrapDataListHead`). The list stores a group of `WRAP_DATA` objects.

```
1:  SLIST_ENTRY WrapDataListHead;

2:  typedef struct _WRAP_DATA {
3:    SLIST_ENTRY List;
4:    int32       Type;
5:    void*       PData;
6:  } WRAP_DATA;

7:  typedef struct _BIN_DATA {
8:    int32       BinLength;
9:    char*       BinData;
10: } BIN_DATA;

11: typedef struct _TXT_DATA {
12:   char*       TxtData;
13: } TXT_DATA;

14: void InsertSList
15: (SLIST_ENTRY *Head,
        SLIST_ENTRY *Entry)
16: {
17:   Entry->Flink = Head->Flink;
18:   Head->Flink = Entry;
19: }

20: void InsertWrapList (int32 type,
         void *data)
21: {
22:   WRAP_DATA *WrapData =
         AllocateWrapData();
23:   WrapData->Type = type;
24:   WrapData->PData = data;
25:   InsertSList(&WrapDataListHead,
         &WrapData->List);
26: }

27: void InsertTxtData(
        TXT_DATA *txt_data)
28: {
29:   InsertWrapList(0, txt_data);
30: }

31: void InsertBinData(
        BIN_DATA *bin_data)
32: {
33:   InsertWrapList(1, bin_data);
34: }
```

Figure 3: The source code for the running example.

3.3.2 Assumptions

In designing KOP, we make two assumptions. First, we assume the kernel memory snapshot is given. There are several ways to obtain a memory snapshot of a running kernel such as taking a full kernel memory dump [67], using a PCI card (Copilot [76]), or taking a snapshot of a virtual machine. This may require parsing a page file. Second, like previous systems [78, 11], we assume the source code of the operating system kernel and kernel-mode drivers loaded in the kernel memory snapshot is available.

3.3.3 Static Kernel Source Code Analysis

KOP's static analysis component takes the kernel's source code as input, and outputs its extended type graph. This graph contains three sets of information: (1) object type definitions, (2) declared types and relative addresses of global variables, and (3) candidate target types for generic pointers. Since it is straightforward to retrieve the

```
_InsertWrapList:                                    #21
 _type, _data      = ENTERFUNCTION                  #21
 t282, {*CallTag}  = CALL* &_AllocateWrapData       #22
 _WrapData         = ASSIGN t282                    #22
 t283              = ADD _WrapData, 4               #23
 [t283]*           = ASSIGN _type                   #23
 t284              = ADD _WrapData, 8               #24
 [t284]*           = ASSIGN _data                   #24
 t285              = ADD _WrapData, 0               #25
 t286              = CONVERT t285                   #25
 CALL* &_InsertSList, &_WrapDataListHead, t286      #25
                       EXITFUNCTION                 #26
```

Figure 4: `InsertWrapList` in medium-level intermediate representation (MIR).

first two sets of information from a compiler, we will focus on how the candidate target types for generic pointers are determined. We first describe how we perform an inter-procedural points-to analysis [7] to construct a points-to graph. We then describe how we derive target types for generic pointers based on the points-to graph and type definitions of local and global variables. Our static analysis is based on the medium-level intermediate representation (MIR) used by the Phoenix compiler framework [68]. This allows us to extend KOP to different target architectures. In Figure 4, we show the MIR for the function `InsertWrapList` of our running example.

3.3.3.1 Points-To Analysis

Our inter-procedural flow-insensitive (i.e., ignoring the control flow within a procedure) points-to analysis is due to Andersen [7]. It computes the set of logical objects that each pointer may point to (referred to as the points-to set for that pointer). The logical objects include local and global variables as well as dynamically allocated objects. Since our goal is to find candidate target types for generic pointers, our points-to analysis must be *field-sensitive* (i.e., distinguishing the fields inside an object). Furthermore, to achieve a good precision, we chose to perform *context-sensitive*

Table 1: Deduction rules used by the original algorithm[7] and KOP.

Rule	Original	KOP
Assign	$x = y \Longrightarrow \langle x, y\rangle$	$(x = y + n, op) \Longrightarrow \langle x, y, n, op\rangle$
Trans	$\langle x, y\rangle, \langle y, z\rangle \Longrightarrow \langle x, z\rangle$	$\langle x, y, n_1, ops_1\rangle, \langle y, z, n_2, ops_2\rangle \Longrightarrow \langle x, z, n_1 + n_2, ops_2 + ops_1\rangle$ where $ops_1 + ops_2$ is a valid call path.
Star-1	$\langle x, \&z\rangle, *x = y \Longrightarrow \langle z, y\rangle$	$\langle x, \&z, n, ops\rangle, (*x = y, op) \Longrightarrow \langle z.n, y, 0, op + rev(ops)\rangle$ where $op + rev(ops)$ is a valid call path.
Star-2	$\langle y, \&z\rangle, x = *y \Longrightarrow \langle x, z\rangle$	$\langle y, \&z, n, ops\rangle, (x = *y, op) \Longrightarrow \langle x, z.n, 0, ops + op\rangle$ where $ops + op$ is a valid call path.

analysis (i.e., distinguishing the calling contexts). The reason is that generic functions such as `InsertSList` from our running example are widely used in OS kernels, and without context-sensitivity, the analysis of such functions would result in very general points-to sets for their arguments. Basically, all list heads and entries that are ever passed to such a generic function would point to each other. Finally, our points-to analysis must scale to a large codebase such as an OS kernel.

Points-to analysis for C programs has been widely studied in the programming languages field [42, 109, 9, 41, 99, 26, 75]. Unfortunately, none of the previous algorithms can meet our requirements without modifications. This is mainly because all the previous solutions chose to sacrifice precision for performance since the points-to analysis used inside compilers is expected to finish within minutes. When designing KOP, we decided to *revise the algorithm proposed by Heintze and Tardieu in [42] to achieve field-sensitivity and context-sensitivity.* Note that the original algorithm is context-insensitive and field-based. In field-based analysis, all instances of a field are treated as one variable, whereas in field-sensitive analysis, each instance is treated separately. Consequently, field-sensitive analysis is more precise.

Next we describe in detail how we achieve field-sensitivity and context-sensitivity in our points-to analysis. We will focus on the changes introduced to the original

Heintze and Tardieu's algorithm. Readers are referred to their chapter [42] for details of their original algorithm.

By using temporary variables, Heintze and Tardieu transform pointer assignments into four canonical forms: $x = y$, $x = \&y$, $*x = y$, and $x = *y$. To handle pointer offsets, We generalize the first two assignment forms to $x = y + n$ and $x = \&y + n$ where n is the pointer offset. To enforce context-sensitivity, we associate each assignment with a variable *op* that specifies the *call* or *return* operation involved in the assignment. Note that *op* is null when the assignment occurs inside a single function.

In [42], given the four canonical assignment forms, an edge in the points-to graph is a pair $\langle src, dst \rangle$, and four deduction rules are used to compute the points-to graph (shown in the left portion of Table 1). To consider pointer offsets and calling context changes, we enhance the semantics of edges to be a four-tuple $\langle src, dst, n, ops \rangle$. For example, given the pointer assignment $\_Entry = t286$ due to the function call at line 26 of Figure 4, the corresponding edge will be $\langle \_Entry, t286, 0, call@file : 25 \rangle$.

Given the enhanced semantics of edges, we change the deduction rules accordingly (shown in the right portion of Table 1). The changes related to field-sensitivity are straightforward. In the **Assign** rule, the pointer offset is simply put in the edge's four-tuple. In the **Trans** rule, the pointer offsets are added up. In the **Star** rules, we create a new node $z.n$ to represent an instance of the pointer field at offset n in logical object z to achieve field-sensitivity. In our deduction rules, whenever we create a new edge, we also check if the sequence of call/return operations involved is valid under context-sensitivity. A sequence is valid if it can be instantiated from a valid call path (i.e., a control flow). We assume there are no recursive functions (we have not observed any in the Windows source code we analyzed). So a valid call path has at most a single call at each call site. Additionally, we do not need to apply any special rules to global variables since we create a single node for each global variable

disregarding the function contexts. This allows information to flow through global variables between different functions.

To avoid the cost of computing the full transitive closure, Heintze and Tardieu propose a new algorithm to maintain a pre-transitive graph and compute the points-to set on-demand. We cannot use their algorithm directly because the edge semantics are different. Compared with the original algorithm, the key differences are two-fold. First, we enforce context-sensitivity by checking if a sequence of call/return operations is *valid*. Second, whenever a cycle is found, the algorithm in [42] merges all the nodes in the cycle, but instead we opted to terminate the path traversal in this case. This is because our edges have more semantics than just pointer inclusions, which ends up affecting the performance of our algorithm. However, since our static analysis runs in an offline manner and only needs to be run once for each OS kernel, we can tolerate the performance degradation. The cycle detection in our pre-transitive graph algorithm and the no-recursive-call policy in enforcing context-sensitivity ensure that our points-to analysis terminates eventually.

Given that it is difficult to define a language that can accurately reflect the use of the C language in real-world programs, a common practice in the programming language field is to empirically evaluate the correctness of a points-to algorithm and its implementation. In KOP, we take the same approach and rely on our evaluation to empirically demonstrate the correctness of our points-to analysis.

3.3.3.2 Inferring Types for Generic Pointers

The output of our points-to analysis is a pre-transitive points-to graph from which we can derive the candidate target types for generic pointers. The key idea is to leverage the type definitions of local and global variables. Before describing our algorithm in detail, we will first use an example to explain the intuition behind it.

The basic idea of our algorithm is illustrated in Figure 5. In the points-to graph

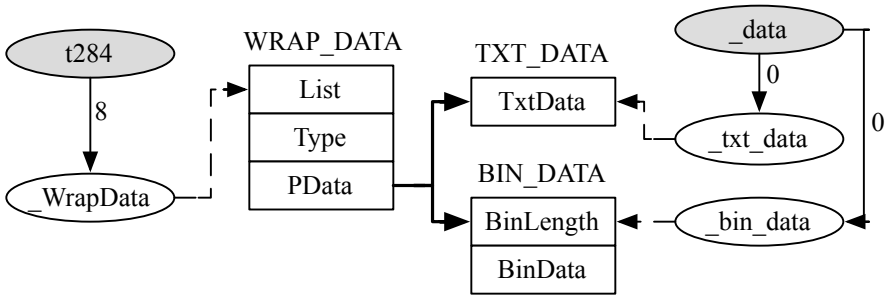

Figure 5: An example for inferring candidate target types of generic pointers. In this example, we derive the types for `WRAP_DATA.PData` from the assignment `*t284 = _data` (see the MIR code in Figure 4). This graph is a mix of the points-to graph and the extended type graph. It illustrates how we derive edges in the extended type graph based on the points-to graph. Ellipse nodes and solid arrows are part of the points-to graph. Rectangular nodes and bold-solid arrows are part of the final extended type graph. The dashed arrows are derived from the type definitions of variables.

of our running example, we have edges from `t284` to `_WrapData` (with pointer offset 8) and from `_data` to `_bin_data` and `_txt_data` (with pointer offset 0). In addition, based on the type definitions, we know that `_WrapData` points to `WRAP_DATA`, `_bin_data` points to `BIN_DATA` and `_txt_data` points to `TXT_DATA`. Then, given the assignment `*t284 = _data`, we can infer that `WRAP_DATA+8`, which is `WRAP_DATA.PData`, may point to either `BIN_DATA` or `TXT_DATA`. The key difference here from classic points-to analysis is that, *although a pointer like `_WrapData` may not point to any logical object, we leverage its type definition to derive the target types for `WRAP_DATA.PData`.* Moreover, with the pointer offsets in the points-to graph, we naturally identify that `WRAP_DATA.List` does not just point to an `SLIST_ENTRY` object but actually a `WRAP_DATA` object. With this, KOP avoids the need for manual annotations in the code for types such as `SLIST_ENTRY`. The extended type graph for our running example is shown in Figure 6. Note that `WrapDataListHead` is a global variable and other nodes are data types.

More specifically, for each assignment in the form $*x = y$, we first search for all the reachable nodes in the pre-transitive graph for x and y, separately. We refer to them

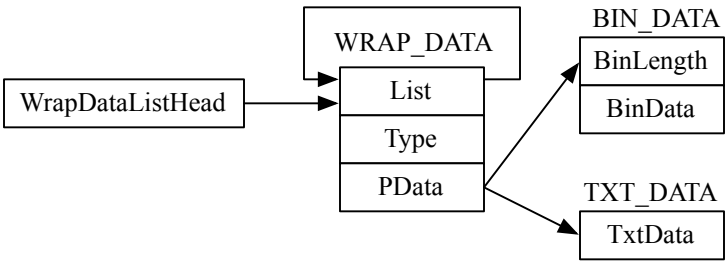

Figure 6: The extended type graph for the running example.

as $TargetSet(x)$ and $TargetSet(y)$. Then for each node a in $TargetSet(x)$ and each node b in $TargetSet(y)$, we check if there is a valid call path from a to b. If there is one, we derive a candidate target type for a pointer field in the data type of a. Similarly, we derive candidate types from assignments of the form $x = *y$. The intuition is that, when y is a generic pointer such as `void*`, it will be cast back to its actual type before the program accesses the data pointed to by it. Specifically, for each assignment, we first search for the nodes that can reach x, referred to as $SourceSet(x)$. Then for each node a in $SourceSet(x)$ and each node b in $TargetSet(y)$, we check if there is a valid call path from a to b. If so, we derive a candidate type for a pointer field in the data type of a.

An inherent problem of flow-insensitive points-to analysis is the imprecision. To mitigate this problem, we introduce a constraint when deriving candidate types for generic pointers in linked list constructs. For example, a pointer field in `SLIST_ENTRY` must point to an `SLIST_ENTRY` structure. This kind of constraint reduces the number of incorrect candidate target types and thus reduces the possibility of errors in the memory analysis. Such constraints do not decreases KOP's coverage because all valid candidate target types are expected to meet this constraint.

3.3.4 Dynamic Memory Snapshot Analysis

KOP's memory analysis component maps kernel data objects and derives the object graph in a given memory snapshot. It does so by using the extended type graph

derived earlier to traverse the kernel memory. We use our running example to explain the basic idea behind our memory traversal algorithm and the challenges we faced. Starting at global variable `WrapDataListHead`, KOP first reaches an object of type `WRAP_DATA` referenced by it. KOP then follows each pointer field defined inside this object. By following the field `WRAP_DATA.List` (a linked list structure), KOP reaches another object of type `WRAP_DATA`, and continues by following each pointer field inside it.

A challenge arises when trying to follow the pointer field `WRAP_DATA.PData`. This field is a generic pointer which, according to the extended type graph, can either point to a `BIN_DATA` object or a `TXT_DATA` object. KOP must determine the type of the object referenced by `WRAP_DATA.PData` in memory. Additionally, the `BIN_DATA.BinData` and `TXT_DATA.TxtData` could be pointers to dynamic arrays. Finally, KOP needs to tolerate identification errors to a certain degree.

In summary, to correctly identify kernel objects, KOP faces three challenges: resolving type ambiguities, recognizing dynamic arrays, and controlling identification errors. In the rest of this section, we describe in detail how we address these challenges. We draw examples from Windows operating systems but our techniques are applicable to other operating systems (e.g., Linux) since they rely on common implementation paradigms used in modern operating systems.

3.3.4.1 Resolving Type Ambiguities

Type ambiguities come from two sources: unions and generic pointers that have multiple candidate target types. We will refer to the range of possible choices in both cases as *candidate types* or *candidates*. KOP is the first system that can resolve type ambiguities in memory traversal.

KOP considers two constraints when determining the correct candidate type. The first is a size constraint. Specifically, operating system kernels (e.g., Windows) store

dynamic kernel data in a set of memory allocation units called *pool blocks*. Each pool block is created by a call to a memory allocation function (e.g., `ExAllocatePool` in Windows). Each kernel object must lie completely within a single pool block. We consider this as a hard constraint. When resolving type ambiguities, KOP rejects any candidate that violates the size constraint.

The second constraint is based on the observation that the data stored by certain data types must have specific properties. Currently, we only apply this constraint to pointer fields. With certain exceptions, pointer fields in kernel objects are either null or assume values in the kernel virtual address range (e.g., $[0x80000000, 0xffffffff]$ for 32-bit Windows). Drivers that directly access user mode memory, for instance, do not meet this condition. Thus, we treat this condition as a soft constraint. We accept candidates that violate this constraint as long as the number of violating pointers is sufficiently small. More precisely, given several candidate types, we compute for each candidate the fraction of pointer fields that violate the constraint and choose the one with the lowest fraction. We discard the candidate if the fraction of invalid pointer values for it is too high (e.g., $> 10\%$).

The two constraints above provide the basis for resolving type ambiguities in our memory traversal. Whenever the traversal reaches an object field with several candidate types, which can be a generic pointer or a union, it tests the constraints for each candidate and selects the one with the best result. The constraints are not only evaluated on the candidates but also, recursively, for their "child" objects (i.e., the objects pointed by the candidates) up to some depth level (e.g., three). By doing so, we improve the accuracy of type ambiguity resolution since we have more data to rely upon when making the decision.

3.3.4.2 Recognizing Dynamic Arrays

Dynamic arrays are widely used in OS kernels and drivers. KOP is the first system with the capability to automatically recognize dynamic arrays in memory traversal. The key idea is to leverage the kernel memory pool boundaries, i.e., a dynamic array must fit into a pool block. Moreover, we note that a dynamic array is usually allocated in two possible ways: it may take up a whole pool block, or it may extend an object whose last field is defined as an array of size 0 or 1. Based on these two observations, KOP checks each allocated pool block to recognize dynamic arrays after the object traversal (without dynamic arrays) is completed.

If a single object is identified at the start of a pool block, KOP analyzes it further to determine if it contains a dynamic array of the first kind. The intuition is that arrays are typically accessed via a pointer to their first element. KOP then tests if the array candidate meets a new size constraint: the size of a pool block must be a multiple of the size of the first object plus some number between 0 and $A - 1$, where A is the pool block alignment. This is a hard constraint. Finally, KOP checks the pointer value constraint for each array element. KOP accepts the candidate dynamic array if a sufficiently large fraction of array elements (e.g., 80%) have a low fraction of invalid pointer values.

KOP checks a pool block for a dynamic array of the second kind if there is an empty space (i.e., no objects were found) trailing an object and the object's last element is an array of size 0 or 1. For such objects, KOP checks the size and pointer value constraints as above.

After identifying dynamic arrays, KOP uses them as roots and reruns the traversal algorithm. This process is repeated until no more dynamic arrays are found.

3.3.4.3 Controlling Object Identification Errors

During the memory traversal, KOP may incorrectly identify an object for three main reasons: (1) choosing the wrong candidate when resolving type ambiguities, (2) mistaking a dynamic array, and (3) program bugs (e.g., dangling pointers). Given the recursive nature of KOP's memory traversal, an incorrect object affects all subsequent traversals following it. Therefore, it is critical to reduce identification errors and prevent them from propagating. To do so, we employ the following two techniques.

First, instead of performing a single complete traversal, KOP traverses the kernel memory in multiple rounds. The key idea is to *identify kernel objects who are more likely to be correct first and use them to preempt those identified later when there is a conflict*. Specifically, KOP performs the memory traversal in three distinct rounds. In the first round, KOP identifies all the global objects and those objects referenced by global pointers. These are the roots used in the traversal and are likely to be correct. In the second round, starting from the objects found in the first round, KOP traverses the kernel memory *but only follows pointer fields that do not have type ambiguities*. We do not infer dynamic arrays in this round either. This way we avoid the identification errors that may be caused by either resolving type ambiguities or inferring dynamic arrays. In the third round, starting from the objects found in the previous rounds, KOP traverses the kernel memory and resolve type ambiguities when necessary. KOP also identifies and traverses dynamic arrays in this round (after the traversal without dynamic arrays is finished). Note that, if two objects identified in the same round conflict with each other, we keep both of them. Currently, we perform a depth-first traversal in each round.

Second, to limit the damage caused by a previous identification error, KOP uses a safe-guard mechanism. Whenever following a typed pointer during the traversal, KOP first checks if the object implied by the pointer type meets the constraints used for resolving type ambiguities (see Section 3.3.4.1). This can be treated as a special

case in which only a single candidate is considered. If the object violates either constraint, KOP discards it and stops that branch of traversal.

3.4 Security Application

Function pointers are commonly used throughout the kernel to perform indirect calls. A popular technique used by malware is to change their values to point to malicious code, an action also known as *hooking*. By doing so, malware can hijack the OS's control flow whenever an indirect call of these function pointers occurs. This allows it to intercept and control certain types of system activity.

A common task in detecting unknown or analyzing known kernel-mode malware is to identify all the function pointers manipulated by the malware. The ideal way to do this is to inspect the values of all function pointers in the kernel and determine if they point to legitimate targets. There are several difficulties with this. First, many function pointers reside in dynamic kernel objects, and therefore do not have a fixed location in memory. As demonstrated in Sections 3.3.3 and 3.3.4, locating all these objects in a memory snapshot is not trivial. Second, inside a single object, not all function pointers can be unequivocally identified. This can happen in the following two scenarios: (1) a field is declared with a primitive type (e.g., `unsigned int`) but effectively used as a function pointer, and (2) a function pointer is defined inside a union. We refer to these as *implicit* function pointers and all the others as *explicit* function pointers. Thus, the task of complete and accurate function pointer identification is a challenge in modern OSes.

To address these problems we built SFPD, the *Subverted Function Pointer Detector*. SFPD relies on KOP to perform a systematic analysis of function pointers in a kernel memory snapshot. Particularly, it leverages KOP's nearly complete memory traversal to identify kernel objects. Due to KOP's greater coverage of the kernel memory, SFPD is able to verify the function pointers of a much larger set of objects

than previous approaches such as SBCFI [78]. SFPD also leverages KOP's points-to analysis to recognize implicit function pointers. SFPD is the first system that can identify malicious implicit function pointers in kernel memory.

SFPD assumes the knowledge of which benign modules are currently loaded, including their binary images. Given a memory snapshot, SFPD first checks if the code of the benign modules was modified. If so, any modified parts of the code are marked as untrusted. The rest of the code is treated as trusted. SFPD then checks every function pointer in the kernel objects found by KOP based on the following policy: *An explicit function pointer must point to trusted code; an implicit function pointer must point to either trusted code or a data object found by KOP; otherwise, the function pointer is marked as malicious.*

This policy is simple but powerful. With it, SFPD can detect malicious implicit function pointers which target malicious code placed in unused blocks of memory. At the same time, by leveraging KOP's high coverage, it effectively avoids the false alarms that would otherwise be caused in two cases. (1) Our flow-insensitive points-to analysis can mistakenly identify data pointers as implicit function pointers, due to imprecision; and (2) data pointers may share the same offset as a function pointer in a union. We are aware that this policy will not be able to identify function pointers manipulated by malware using *return-to-libc* attacks, which are rare in practice.

Additionally, we leverage the traversal information available in KOP to retrieve the traversal path to objects whose function pointers were found to be malicious. Such information is important because this path often reveals the purpose of the function pointer. For instance, simply knowing a function pointer in a `EX_CALLBACK_ROUTINE_BLOCK` object does not tell what it is for. We will, however, know it is used for intercepting process creation events when SFPD shows that it is referenced from a global pointer array in `PspCreateProcessNotifyRoutine`.

3.5 Implementation and Evaluation

We developed a prototype of KOP on Windows. The static analysis component was built using the Phoenix compiler framework [68], and the runtime component is a standalone program. Both components were implemented in C# with a total of 16,000 lines of code. KOP operates in an offline manner on a snapshot of the kernel memory, captured in Windows as a complete memory dump [67]. KOP relies on the Windows Debugger API [66] to resolve symbols, access virtual addresses, and extract information about the pool blocks allocated in the snapshot.

We used the Windows Vista SP1 operating system as our analysis subject. Its kernel and drivers are mostly written in C, with parts in C++ and assembly. Our experiments were performed using a Windows Vista SP1 system loaded with 63 kernel drivers shipped with the OS. We ran this system in a VMware virtual machine with 1GB RAM. In our prototype, we used the following parameters for the memory analysis: tolerance of at most 10% for invalid pointer values in an object, requirement of at least 80% of the dynamic array elements to meet the pointer constraint, and the use of three levels of child objects when evaluating the pointer constraint for a candidate.

Several implementation techniques in the Vista kernel and drivers presented difficulties for KOP. We were able to identify the following cases: (1) the lower bits in some pointers are used to store the reference count, assuming that the target is 8-byte aligned, (2) the relative layout of independent objects is used in cross-object pointer arithmetic (e.g., when independent objects are stored in a single pool block), and (3) implicit type polymorphism in C (e.g., a single object can belong to more than one type). In developing our prototype, we manually adjusted our implementation to handle these cases.

We also implemented a prototype system for SFPD with a total of 1,000 lines of C# code. The relatively small size of our SFPD prototype shows that, given the

infrastructure provided by KOP, it requires only a small amount of extra effort to implement an integrity checking application. In the rest of this section, we present the evaluations of KOP and SFPD.

3.5.1 KOP

KOP's main goal is to completely and accurately map the kernel objects in a memory snapshot. Since we trivially identified all static kernel objects by mapping global variables, we will only evaluate KOP's *coverage* of dynamic kernel objects. We also evaluated KOP's *performance* to demonstrate that it can perform its offline memory analysis in a reasonable amount of time. Before presenting our experimental results on coverage and performance, we will first summarize the results of our static analysis.

3.5.1.1 Static Analysis

We applied KOP's static analysis to the source code of the Vista SP1 kernel and the 63 drivers, with a total of 5 million lines of code. This codebase contains 24423 data types and 9629 global variable definitions. KOP derived the candidate target types for 3228 `void *` pointers, 1560 doubly linked lists, 118 singly linked lists, and 8 triply linked lists (i.e., balanced trees). KOP also identified 3412 implicit function pointers. In our experiments, KOP needed less than 48 hours to complete its static analysis on a 2.2GHz Quad-Core AMD Opteron machine with 32GB RAM. Since KOP only needs to run its static analysis once for an OS kernel and its drivers, this running time is acceptable.

3.5.1.2 Coverage

We measure KOP's coverage by the fraction of the total allocated dynamic kernel memory for which KOP is able to identify the *correct* object type. Ideally, we would use a ground truth that specifies the exact object layout in kernel memory. However, obtaining such a ground truth is extremely difficult and time-consuming. For instance,

the value of a certain field in an object may determine the existence and layout of other objects in the same pool block. Thus, we would need to understand the semantics of each object field to obtain the exact object layout.

Instead, we obtained a ground truth with a slightly coarser granularity. Specifically, we instrumented the kernel to log every pool allocation and deallocation during runtime, along with the call stack, address and size.

We manually inspected the source code for each location on the call stack. This allowed us to identify a call stack location at which the types of the allocated objects could be readily identified in the source code. This was often not the stack location at which the generic allocation function (`ExAllocatePool`) was called, but some location higher in the call stack. We manually analyzed 367 allocation sites and identified the object types that can be allocated at each site. This corresponds to 95% of the allocated pool blocks (94% of the allocated bytes). We were not able to do this for 100% of the pool blocks simply because of the very large number of different allocation sites for the remaining 5%.

Since our ground truth does not specify the exact object layout, we do not know the exact number of objects that exist in the pool blocks. Therefore, we cannot measure KOP's coverage based on the fraction of correctly identified objects. Instead, we measured the coverage based on bytes, since we know the total number of bytes in allocated pool blocks.

For a byte b inside a pool block that is part of our ground truth, we say b is *correctly mapped* if KOP identified a single object which contains b's location and the object type is associated with the pool block. If b is mapped to an object of incorrect type or more than one type, we say it was *incorrectly mapped* by KOP. Finally, if it was not mapped at all, we say it was *missed under ground-truth*. Let CM, IM and MG be the sets of bytes that are classified as correctly mapped, incorrectly mapped and missed under ground-truth, respectively. We define *verified coverage* as

$$\frac{|CM|}{|CM| + |IM| + |MG|}$$

where $|\cdot|$ denotes the set size. We chose the allocation sites for which we computed the ground truth only based on the number of pool blocks they cover and not based on any properties of KOP. Therefore, we believe that the verified coverage has the character of a statistical sample and that it is representative for the overall coverage of KOP.

To gain further confidence, we compute a second measure of coverage. Consider any byte b in a pool block that is not in our ground truth. We say that b is an *unverified mapping* if KOP identified some object at its location and *missed outside of ground-truth* otherwise. Let UM and MOG denote the respective sets. We define *gross coverage* as

$$\frac{|CM| + |UM|}{|CM| + |IM| + |MG| + |UM| + |MOG|}$$

In our coverage experiments, we compared KOP with a *basic* traversal algorithm. Like previous approaches [78, 11], the basic traversal only follows typed pointers and doubly linked lists without resolving type ambiguities and recognizing dynamic arrays. The only difference is that our basic traversal algorithm uses the target types of linked lists automatically derived from KOP's static analysis, while previous approaches relied on manual efforts. To demonstrate KOP's robustness with different workloads, we tested it on two different memory snapshots. One was collected right after the system was booted up, and the other was collected after running a large number of system and user processes on the system for 15 minutes. We refer to these two memory snapshots as the *clean-boot* and *stress-test* snapshot.

The experimental results for the coverage of KOP and the basic traversal algorithm

Table 2: Coverage results for the basic traversal and KOP when applied to the clean-boot and stress-test memory snapshots. CM = Correctly Mapped, IM = Incorrectly Mapped, UM = Unverified Map, MG = Missed in Ground-truth, MOG = Missed Outside Ground-truth, VC = Verified Coverage and GC = Gross Coverage. The numbers in the table are percentages of the total number of bytes.

	Clean-Boot (%) – Total bytes: 42775648						
Type	CM	IM	MG	UM	MOG	VC	GC
Basic	25.4	0.0	68.9	1.4	4.3	**26.9**	**26.8**
KOP	93.7	0.0	0.6	5.3	0.4	**99.3**	**98.9**

	Stress-Test (%) – Total bytes: 50588704						
Type	CM	IM	MG	UM	MOG	VC	GC
Basic	26.6	0.0	68.0	1.4	4.0	**28.1**	**28.0**
KOP	93.8	0.0	0.8	5.0	0.4	**99.2**	**98.8**

are shown in Table 2. The dynamic kernel data has 42.7MB in the clean-boot memory snapshot and 50.6MB in the stress-test snapshot. In both snapshots, KOP's verified coverage and gross coverage are 99%, whereas the basic traversal has only 28%. Since our ground truth covers 94% of the dynamic kernel data, the gross coverage is very close to the verified coverage, as shown in Table 2. We manually investigated some of the cases where KOP either identified the objects incorrectly or missed them completely. We found that they were due to three reasons: KOP incorrectly resolving type ambiguities or recognizing dynamic arrays, dangling pointers and unorthodox Windows kernel implementation techniques that we were not able to identify. In Section 3.6, we will discuss future research directions that can help mitigate these errors.

3.5.1.3 Performance

We measured the running time of KOP when analyzing twelve distinct memory snapshots used in our experiments (including those used on SFPD's evaluation). We used a 4GHz Intel Xeon Duo Core machine with 3GB RAM. The median running time was 8 minutes, including the overhead of reading the memory snapshot stored on the

disk. This running time is acceptable for offline analysis.

3.5.2 SFPD

The goal of SFPD is to identify all malicious function pointers in the kernel memory. To evaluate it, we used it to analyze the memory snapshots of systems infected with kernel-mode malware. Specifically, given a malware sample, we executed it in the Windows Vista SP1 virtual machine used in KOP's evaluations, and then generated a memory snapshot after waiting for a few seconds.

For each memory snapshot, we manually built the ground truth of all malicious function pointers. More precisely, we first manually identified the code regions occupied by the malware based on our instrumentation logs. We then conducted an exhaustive memory search for memory locations pointing to the regions containing the malware's code. We then manually verified each of them to check if they were malicious function pointers.

In our experiments, we tested SFPD with eight real-world kernel malware samples collected from a public database. Running on a 4GHz Intel Xeon Duo Core machine of 3GB RAM, SFPD finishes a scan of a memory snapshot in less than two minutes, excluding time KOP takes to map kernel objects in the snapshot.

Our experimental results for SFPD are shown in Table 3. We do not report results on System Service Dispatch Tables (SSDTs) and the Interrupt Dispatch Table (IDT) because they are static data and therefore are not our focus. We compared SFPD with a baseline algorithm which is similar to previous approaches [78, 11]. This baseline algorithm inspects explicit function pointers based on the kernel objects identified in the basic traversal. SFPD identified all the malicious function pointers for all eight malware samples with *zero* false alarms. However, the baseline algorithm missed malicious explicit function pointers placed by seven of the eight malware samples, as well as *all* the implicit function pointers. This was caused by

Table 3: Results from applying SFPD to nine memory snapshots infected with different real-world malware samples. Function pointers are classified as either *explicit* (E) or *implicit* (I) based on their kind. A/B means that a scheme detects A out of B malicious function pointers.

Malware	Malicious function pointer	Type	Baseline	SFPD
Farfli.G	DRIVER_OBJECT.DriverInit	E	**0**/2	2/2
	DRIVER_OBJECT.MajorFunction[]	E	**0**/30	30/30
	EX_CALLBACK_ROUTINE_BLOCK.Function	E	**0**/1	1/1
	ETHREAD.StartAddress	I	**0**/2	2/2
	ETHREAD.Win32StartAddress	I	**0**/2	2/2
Syspro.A	DRIVER_OBJECT.DriverInit	E	1/1	1/1
	DRIVER_OBJECT.MajorFunction[]	E	28/28	28/28
	FAST_IO_DISPATCH.*	E	21/21	21/21
	FS_FILTER_CALLBACKS.*	E	12/12	12/12
	NOTIFICATION_PACKET.NotificationRoutine	E	1/1	1/1
Cutwail.K	DRIVER_OBJECT.DriverInit	E	**0**/1	1/1
	DRIVER_OBJECT.MajorFunction[]	E	**2**/6	6/6
	EX_CALLBACK_ROUTINE_BLOCK.Function	E	**0**/1	1/1
	ETHREAD.StartAddress	I	**0**/1	1/1
	ETHREAD.Win32StartAddress	I	**0**/1	1/1
Odsrootkit.C	DRIVER_OBJECT.DriverInit	E	**0**/1	1/1
	DRIVER_OBJECT.DriverUnload	E	**0**/1	1/1
Syzor.A	DRIVER_OBJECT.MajorFunction[]	E	4/4	4/4
	ETHREAD.StartAddress	I	**0**/1	1/1
	ETHREAD.Win32StartAddress	I	**0**/1	1/1
Agent.fwz	DRIVER_OBJECT.DriverInit	E	**0**/1	1/1
	DRIVER_OBJECT.MajorFunction[]	E	1/1	1/1
	EX_CALLBACK_ROUTINE_BLOCK.Function	E	**0**/1	1/1
	ETHREAD.StartAddress	I	**0**/1	1/1
	ETHREAD.Win32StartAddress	I	**0**/1	1/1
DriverByPass	DRIVER_OBJECT.DriverInit	E	**0**/1	1/1
	DRIVER_OBJECT.DriverUnload	E	**0**/1	1/1
	DRIVER_OBJECT.MajorFunction[]	E	4/4	4/4
	EX_CALLBACK_ROUTINE_BLOCK.Function	E	**0**/1	1/1
	KAPC.KernelRoutine	E	6/6	6/6
Haxdoor	DRIVER_OBJECT.DriverInit	E	**0**/1	1/1
	DRIVER_OBJECT.MajorFunction[]	E	**0**/2	2/2

the low memory coverage of the basic traversal, as well as its lack of knowledge of implicit function pointers. For instance, the basic traversal fails to identify the `EX_CALLBACK_ROUTINE_BLOCK` object added by the malware because it is referenced by the global variable `PspCreateProcessNotifyRoutine` via a generic pointer.

The baseline algorithm is able to *detect the existence* of all the eight real-world malware samples we tested since it is enough to identify one malicious function

pointer (including entries in SSDTs or IDT not shown in Table 3) to determine that a system is infected. However, it is straightforward to create a new rootkit that only tampers with function pointers missed by the baseline algorithm. For instance, a rootkit can simply hook an `EX_CALLBACK_ROUTINE_BLOCK` object pointed by `PspCreateProcessNotifyRoutine` so that its code can be executed whenever a process is created.

SFPD relies on KOP to identify all the implicit function pointers. In Table 3, we can see that SFPD successfully identified all the malicious implicit function pointers. The `ETHREAD.StartAddress` and `ETHREAD.Win32StartAddress` pointer fields refer to the initial function for starting a thread. By identifying these function pointers, SFPD was able to reveal that the malware created kernel threads to run itself.

3.6 Discussion

KOP's static analysis could be improved to solve the kernel implementation corner cases discussed in Section 3.5 in a more general way. For example, tracking the arithmetic and logical operations associated with pointer values could provide a general means to identify bit manipulations in pointers. Likewise, identifying the use of casts in assignments could help automatically determine type polymorphism cases. These improvements could make the task of porting KOP to a different OS easier.

The techniques used in KOP's memory analysis are also not perfect. Currently KOP relies on its knowledge of pointer fields to select a candidate from the range of possibilities. There are cases where this knowledge may not be sufficient to make the correct choice. It is very hard, for instance, to tell apart small objects with very few or no pointers at all, which may lead to inaccuracies in the traversal.

One possibility to mitigate these problems is to increase the scope of our static analysis to determine domain constraints for other types of fields in addition to pointers. For example, a unicode string should always be terminated by two consecutive

null bytes, and enumerated (`enum` in C) types can only assume a statically-defined set of values. Such information would be very useful for increasing the precision of resolving type ambiguities.

One must also consider the possibility of an attacker trying to disrupt KOP's traversal by polluting the kernel memory. He could, for instance, intentionally break the internal structure of key kernel objects by tampering with the values stored at pointer fields. As a result, our traversal may incorrectly identify these objects due to pointer field mismatches. This attack is not as simple as it sounds, however, since the attacker has to carry it out in a way that the modifications do not destabilize the whole kernel and crash the system. Our current system can tolerate this kind of attack up to a certain point, since it checks the pointer-value constraints in a flexible way. However, it will not be able to do so if a very large number of pointers inside an object is manipulated. A more robust improvement could come from pre-determining which fields can be tampered with without crashing the system and ignoring them when matching pointer fields.

3.7 Summary

Dynamic kernel data have become a common target for malware looking to evade traditional code and static data-based integrity monitors. Previous out-of-VM solutions for inspecting dynamic kernel data can reach only a fraction of it, leaving holes for well-engineered malware to evade. Thus, it is imperative that integrity protection systems be able to accurately and completely map kernel objects in the memory.

In this chapter we presented this dissertation's first contribution: a set of techniques capable of re-creating a syntactic view of the guest kernel's heap state with very high coverage and accuracy. We demonstrated these techniques by implementing KOP, a passive monitoring infrastructure that can be used by systematic integrity checking applications. KOP uses a combination of static source code and dynamic

memory analysis techniques to achieve its goals. Our evaluation of KOP showed substantial coverage gains over previous approaches. We also implemented an integrity checking application based on KOP to detect malicious function pointers. Our evaluation of this application involving real-world and artificial malware samples demonstrated that KOP's high coverage and accuracy translate into the ability to detect kernel integrity violations missed by previous approaches.

CHAPTER IV

OVERCOMING THE SEMANTIC GAP THROUGH GUEST-ASSISTED INTROSPECTION

4.1 Motivation

Out-of-VM monitoring provides good security by placing the monitoring application in an isolated security VM (SVM), from where it can securely monitor a guest VM using virtual machine introspection [38, 73, 50, 77, 78]. It does not usually rely on internal guest components to perform its monitoring, as these components can be maliciously tampered with. VMI lacks robustness, however, due to the semantic gap problem. Any changes made to the syntax (i.e., internal disposition and location of fields) or semantics (i.e., the meaning of the data stored in each field) of monitored GVM data structures across different software releases can break introspection-based tools, which rely on pre-determined and, in many cases, reverse engineered knowledge. This is especially true for undocumented data structures, which are extremely common in closed-source operating systems and applications. This problem does not exist with in-guest monitoring, since it allows monitoring applications to directly call functions provided by the guest OS API to get the information it needs (e.g., the list of active processes on the system) [23]. This method naturally accommodates changes made to data structure syntax and semantics across releases, as it uses the guest's own code, which is changed accordingly by the software vendor and has a public, documented API. This approach lacks the security of the out-of-guest approach, however, since the application and the guest OS can be easily tampered with by malware to report fake monitoring results, or be simply disabled. To fully protect an in-guest monitoring application is a very hard problem, and has only been shown

for small agents operating under limiting constraints [74, 90] or without ensuring the application's availability [22].

In this chapter we present SYRINGE, an infrastructure for monitoring VMs that combines the advantages of out-of-guest and in-guest approaches, allowing the semantic gap to be overcome with security and robustness. SYRINGE satisfies these two requirements by placing the monitoring application in an isolated SVM, as done by the out-of-guest approach, but still leveraging the GVM's own code for monitoring, as done by the in-guest monitoring. For this to work, (1) the SVM-resident monitoring application must be able to call GVM functions and (2) the security of the GVM's code execution must be verifiable. These problems are respectively addressed by two techniques: *function call injection* and *localized shepherding*.

Function call injection allows a monitoring application to be placed in the SVM and still be able to invoke functions in the GVM, by carefully interrupting the GVM's execution and manipulating the contents of its virtual CPU and memory through introspection. Localized shepherding monitors the execution of the invoked guest code against attacks by using a combination of on-the-fly static code checking and inline reference monitoring through dynamic instrumentation. This allows it to detect attacks such as hooking [46] and return-oriented programming [47, 89]. It also enforces atomic code execution to prevent unauthorized tampering with temporary execution state.

SYRINGE combines function call injection and localized shepherding to create a robust VM monitoring infrastructure with strong security properties. It avoids the semantic gap inherent to introspection by using guest OS API code instead of directly parsing and reading data structures in memory. As such, changes in the syntax and semantics of guest data structures commonly performed by patches and new software releases do not affect SYRINGE, as long as the public exported API remains unaltered. SYRINGE's design is OS-independent, enabling the creation of a

wide range of monitoring applications.

4.2 Previous Approaches

Secure monitoring of virtual machines has received much attention in the past 10 years from academia and industry. The seminal work by Garfinkel et al. first introduced the concept of virtual machine introspection (VMI), whereby the state of a GVM is passively analyzed by a scanner placed in a separate VM [38]. Various VMI frameworks, such as XenAccess [73], VMwatcher [50], and VMsafe [104], have since been proposed for different hypervisors. Multiple VMI-based solutions have also been proposed to address specific problems such as tracking the execution of guest processes [51], identifying covertly executing binaries [59], verifying semantic integrity constraints [77], detecting persistent control-flow integrity violations [78], and detecting past vulnerability exploitations through record and replay [52]. These solutions, although satisfying their security goals, are vulnerable to the semantic gap problem inherent to introspection. Others, such as Laika, assume no such previous knowledge and apply machine learning to the problem—an approach that is prone to false positives despite its flexibility [25].

The protection of in-guest monitors has also been explored. Lares [74] and SIM [90] protect a small agent inside the guest using hypervisor memory protection and additional address spaces. To ensure security, however, the agent is subject to significant limitations that would not allow such schemes to be used with sophisticated monitoring tools, such as AV scanners. SYRINGE uses a hybrid in-guest/out-of-guest approach for its monitoring infrastructure. SADE dynamically injects a small agent into the guest that can call internal guest functions, but does not protect the agent or the execution of guest code [23]. Overshadow uses virtualization to protect an in-guest application's confidentiality and integrity in the presence of a compromised guest OS, but does not protect the execution of guest OS code and does not address

agent availability concerns [22].

The Virtuoso project shares SYRINGE's basic insight of leveraging the guest's code to minimize the semantic gap [30]. It does so, however, using techniques very different from ours. Basically, it relies on pre-extracted execution traces of guest monitoring functions to automatically generate introspection programs that can be executed in the SVM. These traces must be extracted before any monitoring can be performed, and must be re-extracted whenever the guest OS is updated. Virtuoso also suffers from the fundamental incompleteness of dynamic analysis, which can create significant runtime hazards for the generated introspection programs. SYRINGE shepherds the guest's own internal execution, thus avoiding the hazards of execution trace replaying and the need for a recurrent learning phase.

A brief discussion of the technique underlying function call injection was first presented by Joshi et al. [52], and a more basic variant was later proposed by SADE [23]. Thus, despite our more in-depth investigation and explanation, we do not claim FCI as a contribution. Program shepherding was first proposed by Kiriansky et al. to protect systems against application vulnerabilities [55]. It dynamically monitors the execution of control transfer instructions in the program to ensure that it does not deviate from a certain control-flow integrity policy. We do not, however, require every control transfer instruction to be checked, just indirect ones. Direct ones are already implicitly verified by our code integrity checker. Furthermore, our approach also ensures atomic execution by handling the relevant instructions. Finally, SYRINGE's shepherding is not done system or application-wide, but is localized to monitoring thread resulting from the injected function call and is activated/deactivated on-demand, minimizing the performance impact. Abadi et al. use static binary instrumentation to enforce control-flow integrity (CFI) [1]. CFI has greatly impacted subsequent work on software security, including SYRINGE's localized shepherding.

Secure code execution has received substantial attention from the community in

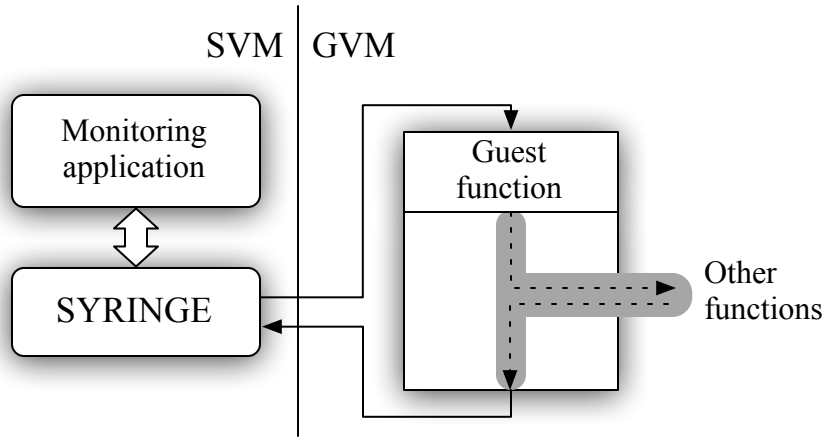

Figure 7: High-level view of SYRINGE. Straight arrows represent function call injection and dashed arrows represent the monitoring thread. The gray background surrounding it represents the localized shepherding of the monitoring thread.

the context of attestation. Pioneer, for instance, allows a verifier to attest that the execution of a certain piece of code on an untrusted environment has occurred without any form of tampering [88]. Some of the techniques that it uses, such as interrupt disabling, are also used by SYRINGE. More recent work in this area, such as Flicker [63], leverages new hardware support for trusted computing and Intel's Trusted Execution Technology.

4.3 The SYRINGE VM Monitoring Infrastructure

4.3.1 Overview

SYRINGE was designed as an infrastructure to be used by applications to securely and robustly monitor a GVM (Figure 7). Its design process started from a basic in-guest monitoring architecture, which, as discussed previously, already incorporates the desired robustness. We then focused on determining what additions and modifications should be done to it so as to make it secure. In more concrete terms, this meant securing the two high-level entities involved in in-guest monitoring: (1) the monitoring application and (2) the execution of guest OS functions invoked by the

application.

Protecting the monitoring application. In this work we assume that the monitoring application is a user-space program. This assumption is based on the fact that most real-world monitoring applications such as AV scanners, intrusion detection systems and system diagnostic tools are implemented in user-space application. Fully protecting a user-space application (monitoring or otherwise) running inside an untrusted guest OS is a hard problem. As demonstrated by Chen et al., it is possible to use virtualization to protect the confidentiality and integrity of its code/data [22]. However, the control that the guest OS has over the application's resources (CPU time, memory, etc) means that it is extremely difficult to ensure the application's availability on an untrusted guest OS. In other words, it would be easy for an attacker who has compromised the guest OS to disable the application, or deny it essential computing resources controlled by the OS. For these reasons, in SYRINGE we opted to remove the monitoring application from the GVM, placing it in an isolated, trusted SVM. This move allowed us to secure the application, but disrupted its ability to invoke guest OS functions. We solved this problem with the *function call injection* technique. Function call injection enables the monitoring application to be moved out of the GVM, but still retain the ability to invoke guest functions by injecting function calls into the GVM. This technique works by interrupting the GVM's execution at a pre-determined point and manipulating the contents of its virtual CPU and memory using introspection, setting it to the desired target function with the desired parameters. In its current form, SYRINGE only supports the injection of function calls to kernel functions.

Protecting the execution of the invoked guest OS functions. We refer to the execution thread triggered inside the guest as a result of the function call injection as the *monitoring thread*. To protect the execution of the monitoring thread

we introduce a novel technique: *localized shepherding*. This technique basically performs on-demand monitoring of the control-flow integrity of the monitoring thread by using on-the-fly instrumentation, in accordance to a policy that we defined to address the most common attacks that rely on control-flow manipulation. Together with function call injection, localized shepherding also ensures the *atomic execution* of the monitoring thread. This property is necessary to prevent malicious threads from tampering with the monitoring thread's local state when their executions are interleaved. Atomic execution is implemented by disabling interrupts at the start of the monitoring thread and shepherding interrupt-related instructions to prevent them from being re-enabled.

SYRINGE *was not* designed as a general security system. Its goal is not to defend the guest against attacks in general. SYRINGE focuses on the task of determining whether the data returned by the monitoring thread to the monitoring application results from an untampered execution. If SYRINGE detects any form of tampering with the monitoring thread, such as a control-flow violation, it will not attempt to repair it. For safety, it will allow the monitoring thread to continue executing unshepherded, but will notify the monitoring application in the SVM that the results returned by the function should not be trusted. An attacker can exploit this fact to disrupt SYRINGE's monitoring, effectively causing a DoS. The monitoring application, however, will know at this point that the system has been compromised, at which point the best course of action may be to restore the GVM to a previous snapshot or employ another type of remediation procedure.

The atomic execution property enforced by SYRINGE creates some functional limitations. First, SYRINGE cannot shepherd guest code that relies on asynchronous code execution, such as I/O or Deferred Procedure Calls (DPCs). This prevents certain types of exceptions, such as page faults, from being handled properly. We do a detailed discussion of these limitations in Section 4.7.

4.3.2 Assumptions

In this work, we assume an underlying x86 architecture running a hypervisor with two virtual machines: a monitored guest virtual machine (GVM); and a secure virtual machine (SVM) in which SYRINGE will be deployed. This assumption *does not* mean that our work cannot be generalized to environments executing a larger number of VMs, as is common in cloud computing. Our whitelisting-based code integrity approach also assumes previous access to legitimate copies of the binaries composing the guest OS's kernel. On Windows, this includes the kernel executive (NTOS) and other kernel-level modules. It *does not* include 3rd-party modules. We believe this to be a reasonable assumption, given that this set of binaries is manageable in size and relatively homogeneous for each particular OS version. A database of such binaries can be easily created and automatically updated, for instance, when patches are issued by the OS vendor. We also assume access to the public Windows kernel API, which includes the function prototype and parameter type definitions. This API is easily accessible online and we do not consider it as part of the semantic gap.

Knowledge of the base address in guest memory for each loaded whitelisted binary is also assumed. This information can be obtained through a variety of methods and heuristics that are orthogonal to this work, and are thus not detailed here. We further assume in our threat model that the GVM can be fully compromised by an attacker, including its kernel. The system hardware, hypervisor, and SVM constitute our trusted computing base.

4.3.3 Function Call Injection

Function call injection (FCI) secures the monitoring application by placing it in an isolated SVM, while still keeping its ability to invoke GVM functions. This is the first piece of our solution to the problem of creating a secure and robust VM monitoring infrastructure. FCI essentially provides the ability for code running in one VM to call

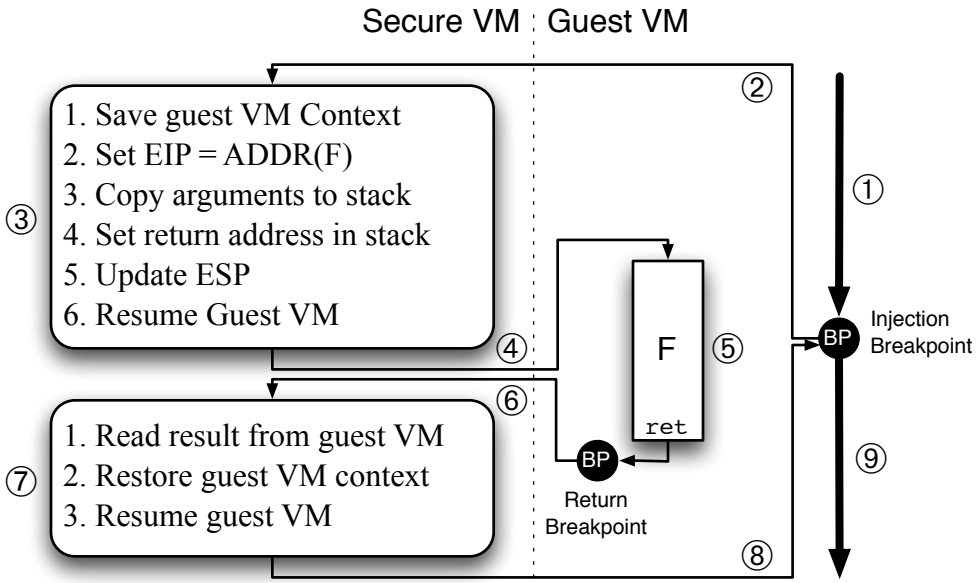

Figure 8: Injecting a call to guest function F. (1) The GVM executes normally until it reaches an injection context; (2) a breakpoint placed at the injection address transfers execution to the SVM and suspends the GVM; (3) The SVM saves the guest VCPU context and sets its EIP to point to F's start address and copies F's arguments to the stack, also updating its ESP; (4) the guest VCPU is resumed and function F starts execution, as if it had just been called by guest code; (5) F is executed; (6) control is returned to the SVM through another breakpoint placed at F's return address; (7) the guest VCPU's context is set by the SVM to the saved context (8 and 9) when resumed and it continues running from the point where it was originally interrupted.

a function in another VM and retrieve its results. FCI uses simple VM introspection techniques. It can be viewed as a type of inter-VM Remote Procedure Call (RPC), but without the need for an RPC server running on the destination.

SYRINGE currently assumes that the GVM only has one virtual CPU (VCPU) in order to ensure atomicity for the monitoring thread. Multiple VCPU support would require the virtualization infrastructure to be able to suspend individual VCPUs during the guest's execution. This is not the case, however, with ESX/VMsafe. Although this assumption may be limiting for certain types of VMs, we believe it to be a consequence of a platform limitation, rather than a fundamental flaw in our approach.

The operation of FCI is shown in Figure 8. The first step in FCI is to interrupt the execution of the guest so that a function call can be injected. Pre-selected *injection contexts* designate the execution contexts under which the guest must be interrupted so that a function call injection may occur. An injection context is a tuple (P_S, A_I), where P_S represents a *surrogate process* and A_I is an *injection address*. FCI can only happen when process P_S is currently active in a guest virtual CPU (VCPU) *and* the instruction at A_I is about to be executed by that same VCPU. A surrogate process is identified by the physical address of its page directory table (stored in the CR3 register). Each surrogate process can have its own injection addresses, or they can be shared between multiple surrogates. Injection addresses can be selected in the guest's kernel-space, for injecting calls to kernel functions, or in user-space for injecting calls to user-level API functions. Multiple distinct injection contexts can be used. To minimize injection delay, it is important to choose injection contexts that are reached frequently enough in the GVM's normal execution. They must also not be easily circumvented by a malicious entity in control of the guest OS. Details concerning our choice of injection context and how these requirements were met are given in Section 4.4.

SYRINGE interrupts the GVM by using VMsafe page-table level breakpoints placed at the injection addresses. We call these *injection breakpoints*. This type of breakpoint cannot be detected or tampered with by the guest OS because it is implemented at the hypervisor level, and is therefore transparent to the guest. It works by marking the memory page where the injection address is located as non-executable. This way, whenever the instruction corresponding to the injection address is executed, a trap is triggered, the GVM is suspended, and control transferred to the hypervisor, and then to SYRINGE in the SVM. SYRINGE then checks if the current CR3 value of the guest's VCPU corresponds to that of a surrogate process associated with the injection address where the execution was interrupted. In case

it does, SYRINGE determines that an injection context has been reached. Injection contexts are only made active (i.e., the hypervisor-level breakpoints are activated) when SYRINGE has requests queued for function call injections, otherwise the system runs normally without any performance penalty. In its current form, SYRINGE allows only one monitoring thread to be running in the GVM. In other words, function call injections cannot overlap each other.

Let us assume a function call $F(A_0, ..., A_n)$, i.e., a call to the guest kernel function F with arguments A_i. Let us also assume a `stdcall` or `cdecl` calling convention, so that arguments are placed on the stack, in reverse order. When the injection breakpoint is triggered, SYRINGE first saves the guest VCPU's context so that it can be restored later and then sets the VCPU's EIP register to F's starting address. F's offset in its corresponding binary can be extracted from the binary's export table. Knowledge of the binary's base address in memory is listed as part of our assumptions and is obtained when SYRINGE is initialized.

Next, the stack needs to be appropriately set with arguments A_i and a return address. This is done by using ordinary memory introspection to map the guest memory region corresponding to the value of the VCPU's ESP register and making the necessary changes. Arguments are handled according to their evaluation semantics. Call-by-value arguments are copied directly onto the stack. Call-by-reference arguments require a more careful treatment. The data buffer referenced by the argument must be copied to the guest and the reference itself must be placed on the stack as an argument. SYRINGE provides two ways of doing this. The simplest way is to place the data structure at the bottom of the current stack frame and push a reference to it. Another possibility is to allocate a special memory buffer inside the guest (for example, by injecting another function call to a memory allocation function) and use it to store the referenced data structure. This is useful in the case where the structure

is too large to be placed on the stack. The return address is set to a special memory location inside the guest containing another VMsafe execution breakpoint—the *return breakpoint*—placed by SYRINGE. This can be any memory location whose page does not contain valid code, so as to avoid unnecessary VM switches caused by execution of code.

Once the stack is set, the value of ESP is updated to accommodate the arguments and return address. To ensure atomicity, the guest's VCPU state is modified so that regular guest interrupts, hardware breakpoints, instruction tracing exceptions and performance monitoring interrupts (PMIs) are disabled when the monitoring thread starts executing. This is done by clearing the IF (Interrupt Flag) bit in the guest VCPU's EFLAGS register, bits 1 and 8 in `IA32_DEBUGCTL`; bits 0, 1, 32–34 in `IA32_PERF_GLOBAL_CTR`; and bits 0–9, 13 in `DR7`.

Finally, at this point, the guest VCPU is resumed. F then begins to execute as if it had been called with arguments A_i from inside the surrogate process, at the injection address. This is our monitoring thread. At this point, the localized shepherding component (described in Section 4.3.4) takes over and shepherds the monitoring thread. When the final `RET` instruction is reached, the return breakpoint is triggered, suspending the VCPU and passing control back to SYRINGE in the SVM. At this point, the result of F's execution is read from the `eax` register and returned to the monitoring application. If any memory buffers have been passed by reference on the stack or heap to receive results from the function, it is the monitoring application's responsibility to retrieve their contents. Finally, SYRINGE restores the original VCPU context that was saved when the guest OS was first interrupted and resumes the GVM, which continues its original execution thread from the point where it was interrupted.

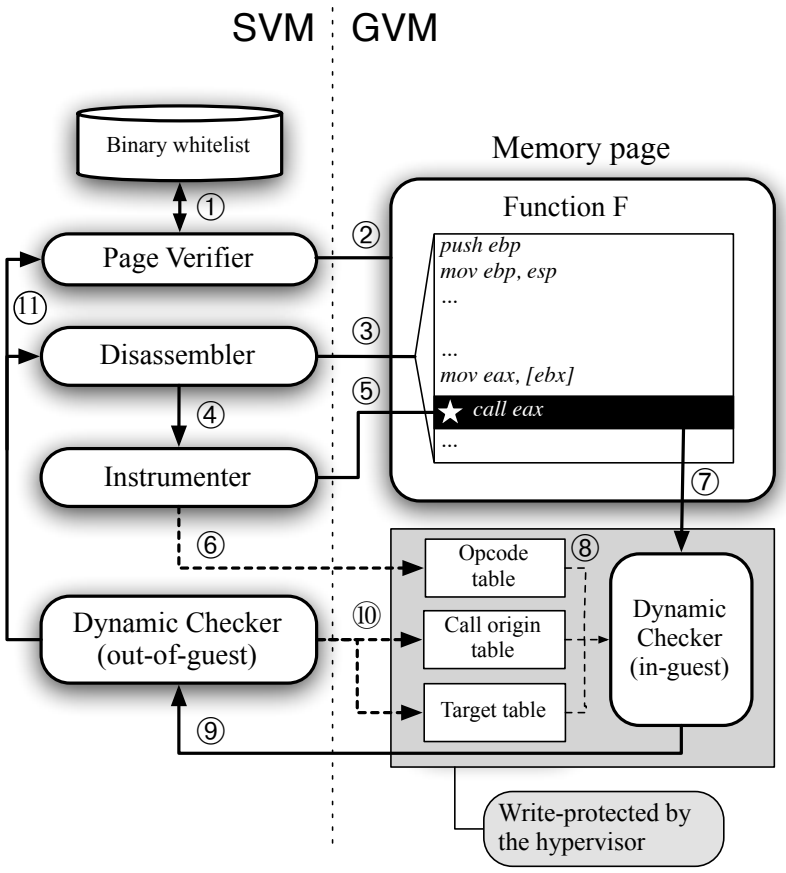

Figure 9: Localized shepherding of function F. (1) Page Verifier pre-builds a whitelist database of the OS kernel binaries. (2) Upon injection, the Page Verifier verifies the code regions of the target page against the whitelist database, (3) the Disassembler recursively disassembles the target function, recording the locations of critical instructions and (4) passing them to the Instrumenter. (5) The Instrumenter patches all critical instructions with int3 breakpoints and (6) updates the in-guest opcode table. (7) When triggered, a critical instruction breakpoint transfers control to SYRINGE's in-guest Dynamic Checker. (8) It consults the in-guest tables to determine whether it can evaluate the instruction by itself. (9) If not, it passes control to the out-of-guest dynamic checker, which (10) updates the in-guest call origin and target tables and, if necessary, (11) re-invokes the Disassembler to analyze new code and the Page Verifier, if the control-flow has transitioned into a new page. This process is conducted recursively for all subsequent function calls.

4.3.4 Localized Shepherding

Localized shepherding is the second piece of our solution to the problem of creating a secure and robust VM monitoring infrastructure. Localized shepherding monitors

the control-flow integrity and ensures the atomic execution of the monitoring thread. Figure 9 illustrates this process and the role played by each component.

Control-flow integrity is monitored by: (1) checking that all guest code executed by the monitoring thread matches the pre-compiled whitelist database of OS API binaries and (2) dynamically evaluating indirect control-flow transferring instructions in accordance to a pre-specified policy. Action (1) guarantees the integrity of direct branches while action (2) monitors the integrity of indirect branches. Thus, together, they cover all control-flow transfers. Action (1) further ensures that non-control-flow related instructions are not modified by an attacker. If a control-flow integrity violation is detected, SYRINGE allows the execution of the monitoring thread to proceed but sends an alert to the monitoring application in the SVM. This alert indicates that malicious tampering has been detected during the execution and therefore the results returned by the monitoring thread cannot be trusted.

Code integrity checking (action (1)) is performed by the Page Verifier component through binary whitelisting. As stated in our assumptions, we assume previous access to legitimate copies of the binaries composing the OS API (both user-space libraries and kernel modules). These binaries are analyzed in an offline manner by the Page Verifier. Based on the metadata and content of each binary's PE sections, the Page Verifier constructs a database containing the location, size, and SHA1 hash corresponding to each code (executable) section in each binary. This information is used at runtime to check the integrity of the code being executed in the guest. Immediately before a function call to F is injected, SYRINGE activates the Page Verifier to check the integrity of all the code present in the page where F's starting address is located. If the page contains a mixture of code and data, each code region in the page is checked individually. A SHA1 hash is calculated for each code region and is compared against its corresponding whitelisted hash. Any discrepancies indicate that the code has been modified. This allows SYRINGE to detect *code patching* attacks, where an attacker

Table 4: SYRINGE's shepherding policy and handler types for critical instructions. The top part shows those instructions related to control-flow integrity monitoring. The bottom part shows those related to atomic execution enforcement.

Instruction	Description	Policy/Action	Handler
Control-flow integrity			
`CALL r/m32`	Indirect function calling	Target must be in trusted code regions. Update in-guest call-origin table	In-guest and Out-of-guest
`JMP r/m32`	Indirect jumping	Target must be in current module	In-guest and Out-of-guest
`RET`	Function returning	Target must be present in pseudo-shadow stack	In-guest
Atomic execution			
`STI/CLI`	Interrupt enabling/disabling	Skip instruction	In-guest
`POPF`	EFLAGS popping	Emulate, clearing the IF and TF bits in the EFLAGS	In-guest
`WRMSR`	Writing to MSR IA32_DEBUGCTL or IA32_PERF_GLOBAL_CTR	Emulate, clearing bits 1 and 8 in the IA32_DEBUGCTL MSR and bits 0, 1, 33–34 in IA32_PERF_GLOBAL_CTR	In-guest
`MOV DR7, *`	Writing to DR7 debug register	Emulate, clearing bits 0–9 and 13 in the DR7 register	In-guest

maliciously modifies the guest code, and notify the monitoring application. This process is repeated whenever the control flow of the monitoring thread transitions into a new page. After being checked and before execution is allowed to begin, code pages are marked as write-protected again by using VMsafe. This marking avoids the need for future checks and prevents time-of-check-time-of-use (TOCTOU) race conditions.

Indirect branch integrity monitoring (action (2)) is performed together by the Disassembler, Instrumenter, and Dynamic Checker components. These components employ a combination of dynamic recursive disassembly, code instrumentation and

reference monitoring. The Disassembler performs a recursive disassembly of the function, stopping at indirect control transfer instructions and direct function calls. During this disassembly, it records the location of all instructions whose execution needs to be trapped and evaluated at runtime to ensure control-flow integrity. We refer to these instructions as *critical instructions* and they are shown in the top part of Table 4. When the Disassembler is done analyzing the function, the Instrumenter instruments all critical instructions so that they can be evaluated by SYRINGE before being executed. This instrumentation consists of an `int3` instruction that overwrites the first byte of the critical instruction. The overwritten byte is recorded by SYRINGE in a write-protected in-guest *opcode table*. The entry #3 of the guest Interrupt Descriptor Table (IDT) is set to point to the in-guest component of the Dynamic Checker. The guest's IDT is write-protected by SYRINGE. The Disassembler is invoked only for those cases where the target in question has not been analyzed previously; otherwise, cached results are used by the Instrumenter for performance.

The Dynamic Checker is responsible for evaluating critical instructions according to our control-flow integrity policy. It has an in-guest component, which implements handlers for those critical instruction invocations that do not need to be handled in the SVM. In-guest handling of critical instructions greatly favors performance in comparison to out-of-guest handling, as it does not require VM switches. The in-guest handlers are injected into the guest by SYRINGE and are write-protected with support from the hypervisor. This protection is effective because in-guest handlers do not require any persistent state to be maintained and are present only when a monitoring thread is being executed. Thus, code write-protection suffices to ensure their good behavior. The in-guest component is invoked by all critical instructions. It determines the type of instruction by consulting the opcode table and whether the instruction can be handled in-guest or not. If not, it generates a trap so that the Dynamic Checker's out-of-guest component can handle it.

Direct CALL instructions do not need to be dynamically evaluated, but are instrumented nevertheless so that their targets can be properly scanned and instrumented before execution is allowed to continue. This instrumentation is only needed until the instruction's first execution, however, and is then removed. They are handled out-of-guest. The target of indirect CALLs must be evaluated dynamically, every execution. The in-guest handler first determines if the computed target of the indirect CALL has been analyzed before, by consulting an in-guest *target table*, maintained by SYRINGE. This table is write-protected inside the guest. If so, then the target is legitimate and execution is allowed to proceed. If not, the in-guest handler generates a trap and passes control to the out-of-guest handler. The later then applies the following policy: the target must be located inside the authorized memory ranges containing the whitelisted system code, as determined previously by the Page Verifier. If this policy is satisfied, the target is added to the target table. The handling of all CALL instructions (both direct and indirect) also includes adding the address of the CALL to an in-guest *call origin table*. This table is also write-protected and is used for the evaluation of RET instructions, explained later. Indirect JMP instructions are handled exactly as described for indirect CALLs with the one following policy difference: their targets must be located inside the current module. The policies used for indirect CALLs and JMPs can detect a large portion of attacks that rely on *hooking* [46] function and code pointers to hijack control-flow.

All RET instructions are handled in-guest, except for the last one. The handler evaluates the RET by comparing its target against all the addresses contained in the call origin table. The RET is considered legitimate if a match is found. This model differs from a shadow stack in that, for a particular RET, the address of its originating CALL is not necessarily at the top of the stack. As a result, our model allows a RET to return to the origin point of any CALLs that were executed previously by the monitoring thread. A complete shadow stack implementation would require the RET

in-guest handler to have write access to the call-origin table, and thus open way to attacks. Thus, we decided against it. Despite not being ideal, the number of allowed return targets is significantly constrained so we believe that this policy is powerful enough to detect most return address manipulation attacks such as *return-oriented programming* [47, 89].

Localized shepherding must also ensure that the monitoring thread is executed atomically. As described in Section 4.3.3, FCI clears the IF flag in the VCPU's EFLAGS register, thereby ensuring that the monitoring thread will start executing with interrupts disabled. Localized shepherding must ensure that they remain disabled throughout its entire execution. The Instrumenter patches another set of critical instructions that can affect atomic execution, and are shown in the bottom part of Table 4. Instructions `CLI` and `STI` are commonly used in OSes to execute critical code sections atomically by temporarily disabling interrupts. SYRINGE's policy is to simply skip these instructions. Thus, they are simply overwritten with a `NOP` by the Instrumenter for the duration of the monitoring thread's execution. All other critical instructions are patched with `INT3`, and are handled in-guest by the Dynamic Checker. The handler for `POPF` pops the stack into the guest's `EFLAGS` and clears the IF and TF flags. SYRINGE ensures that hardware debugging and instruction tracing facilities remain disabled by handling instructions `WRMSR`, when the destination is MSRs `IA32_DEBUGCTL` or `IA32_PERF_GLOBAL_CTR`; and `MOV`, when the destination is the CPU debugging control register `DR7`. Bits 1 and 8 are cleared in `IA32_DEBUGCTL`; bits 0, 1, 32–34 in `IA32_PERF_GLOBAL_CTR`; and bits 0–9, 13 in `DR7`.

With regard to multiprocessing, our assumption that the GVM has just one VCPU guarantees that simultaneous code execution in other CPUs is not an issue.

When the monitoring thread finishes executing (i.e., executes the final `RET` instruction) and SYRINGE reassumes control, the guest is restored to its original state. At this point, all patched critical instructions are un-patched and the IDT is restored

to its original content. Likewise, if external interrupts were enabled when the GVM was interrupted by the FCI component, they will again be enabled when the guest is resumed. This localized, on-demand variant of shepherding satisfies our secure monitoring properties while at the same time does not affect the guest's normal performance when monitoring operations are not being conducted.

Software exceptions present a challenge for localized shepherding. SYRINGE is capable of shepherding exceptions that happen during the execution of the monitoring thread, as long as these exceptions are handled synchronously. This shepherding is done by installing a read-breakpoint in the memory page containing the IDT, so that any attempts to access a descriptor (as should happen during an exception) are trapped and transfer control to SYRINGE. From this point on, the process is the same as the one described above for regular function call invocations.

Exceptions requiring asynchronous activity, such as I/O, cannot be shepherded by SYRINGE. This is a fundamental limitation of SYRINGE. The page fault exception is an especially important case, given their common occurrence in modern OSes. These are often triggered because the code to be executed has not yet been brought to memory (on-demand paging) or has been paged out to disk previously. One option to deal with this problem is to call the guest OS function twice: first without shepherding, to page-in the missing pages; and then with shepherding, to collect the results. Another option is to use a function like `nt!MmProbeAndLockPages` in Windows during SYRINGE's initialization to ensure that sections of the virtual address space (e.g., Windows's NTOS) are entirely in RAM. Non-maskable interrupts (NMIs) are not handled, since they usually indicate a fatal hardware error that would require the GVM to be rebooted or restored to a previous snapshot.

4.4 Implementation Details

SYRINGE was implemented as a Linux library using approximately 3,500 lines of C and Python code. The function call injection component makes up one third of the code, while the localized shepherding code is the rest. VMware's ESX Server 4.1 was used as the hypervisor and VMware's VMsafe was used as our introspection infrastructure [104]. VMsafe natively provides the introspection and breakpoint functionality used by SYRINGE. Despite page-level breakpoints not being provided by other open-source introspection infrastructures and hypervisors (such as XenAccess [73]), we would like to emphasize that the mechanics of this technique are simple and well understood and could therefore be incorporated into them.

When selecting injection contexts, we kept the following requirements in mind: (1) the injection point cannot be maliciously tampered with or disabled, (2) it must be placed in a location safe for calling OS API functions, (3) it must be triggered frequently enough to minimize the injection delay and (4) it must not be easily circumvented. In our prototype, we chose the OS's system call dispatcher as the injection address for kernel functions, and we allow all processes running in the system to act as surrogates. This choice satisfies the above requirements as follows: (1) it uses a hypervisor-level guest-transparent breakpoint, which prevents its disabling by an attacker. Requirements (2) and (3) are satisfied because the system call dispatcher has natural role in the OS as a kernel entry-point, high-level routine dispatcher and is executed at every system call. Requirement (4) is satisfied by ensuring that the system call dispatcher is executed whenever a system call is executed. This is possible, since the address of the dispatcher is architecturally bound to the MSR_SYSENTER_EIP register, which can be monitored for changes in runtime.

We selected as our return breakpoint address location the start of the `.data` section of the kernel executive module (NTOS).

4.5 Evaluation

We conducted a performance and security evaluation of SYRINGE. Our host machine was an Intel Core i7 870 2.93GHz with 4 CPU cores, 8GB of RAM, running VMware ESX Server 4.1. The GVM was configured with 1 VCPU, 1GB of RAM, running Windows XP SP2. The SVM was configured with 1 VCPU, 1 GB of RAM, running Linux CentOS 5.5.

4.5.1 Security

We now analyze and evaluate SYRINGE's security properties. Again, SYRINGE's goal is not to to act as a general attack prevention system. Its goal is to be able to tell, based on the localized shepherding of the monitoring thread, whether the results returned by the invoked guest function can be trusted or not and notify the monitoring application. So it is possible for an attacker to cause a monitoring DoS by repeatedly attacking the monitoring thread, but not without SYRINGE knowing about it. At this point, remediation rather than monitoring becomes the main concern.

SYRINGE applies a mixture of prevention and detection techniques against attacks directed at itself and the monitoring thread.

4.5.1.1 Attacks against SYRINGE's components

Attacks against the monitoring application and SYRINGE's out-of-guest components are prevented by the isolation between the GVM and SVM. In-guest components are protected as follows. For FCI, the injection and return breakpoints cannot be tampered with or disabled, since they operate at the hypervisor level. Our choice of injection context (the system call dispatcher), combined with the continuous monitoring of the MSR_SYSENTER_EIP register, ensure that it cannot be easily circumvented.

For localized shepherding, several components are involved. The `INT3` instrumentation used to trap on critical instructions is protected by the write-protection

of guest code, which prevents the guest from modifying pages containing code being shepherded. This `INT3` relies on the guest's IDT to pass control to the dynamic checker's in-guest component. The IDT is write-protected from inside the hypervisor, preventing any modifications from inside the guest. The dynamic checker's in-guest component consists of a code segment and three tables: the call origin table, the opcode table and the target table. All three tables are write-protected, and can only updated by SYRINGE's out-of-guest components in the SVM. Since the code does not rely on any data maintained by the guest OS (only the three tables) and does not itself maintain any persistent data across dynamic checker invocations, write-protecting its code is enough to protect it against attacks. It is conceivable that an attacker could attempt to modify guest page table mappings so that invocations to the in-guest dynamic checker could fail. While this attack is possible, the memory pages containing the dynamic checker and the three tables are locked in memory and its corresponding mappings are constantly monitored by SYRINGE. Any changes to these mappings, being unexpected, are interpreted as an attack and the monitoring application is notified.

4.5.1.2 Attacks against the monitoring thread

SYRINGE's localized shepherding guards the monitoring thread against attacks using a combination of prevention and detection techniques.

Attacks attempting to patch the guest code cannot succeed given that, before being executed, all code is checked by the Page Verifier against the binary whitelist database and then write-protected. This step occurs atomically to eliminate the possibility of a time-of-check-time-of-use race condition, where the code could be modified after being checked and before being write-protected.

Table 5: Security evaluation results. SYRINGE was able to detect all the attacks and notify the monitoring application.

Attack	Target	Result
Code Patching	`nt!ZwQueryInformationProcess`	**Detected** by *Page Verifier*
Hooking	`KeServiceDescriptorTable[]`	**Detected** by *Dynamic Checker*
Return-into-libc	Return address on stack	**Detected** by *Dynamic Checker*

Indirect control-flow instructions are patched by the Instrumenter and are evaluated dynamically. Attacks directed at these instructions have their power severely restricted by SYRINGE's control-flow integrity policy. Function pointers cannot point anywhere outside whitelisted code, indirect jumps cannot point anywhere outside their own module and function returns must target the instruction following a CALL executed previously in the monitoring thread . This policy greatly restricts the effectiveness of function pointer hooking, which in most cases rely on injected code; and jump-oriented [14, 20] and return-oriented programming [47, 89], which need access to a large code base to extract a good variety of gadgets. More fine-grained policies can be integrated into SYRINGE by using techniques such as alias analysis.

To empirically validate our claims above, we simulated one code patching, one hooking and one return-into-libc attack. Results are show in Table 5. A function call to `ZwQueryInformationProcess` was injected and then shepherded. The first attack patched function `ZwQueryInformationProcess`'s code, and was detected by SYRINGE's Page Verifier, which is responsible for checking the integrity of code sections before they are executed. Hooking was performed on the system service descriptor table, on the entry corresponding to the `NtQueryInformationProcess` system call. Since the hooked address is used by an indirect function call instruction inside the system call dispatcher, it was trapped and evaluated by the Dynamic Checker, which detected the attack, since the address pointed to injected code. Return-into-libc was also detected by the Dynamic Checker, when it could not find the function's return

destination in the call origin table, indicating that it had been modified.

The atomicity property enforced by SYRINGE makes it difficult for attacks to tamper with the monitoring thread's temporary state in the stack or heap. This state is only valid during the time when the monitoring thread is executing, and we can be sure that no other (potentially malicious) thread will be running during that time, and thus cannot directly tamper with it. The only way to do so would be to exploit a software vulnerability, such as a stack or heap overflow, in the shepherded code, so that the monitoring thread itself does the tampering. This could be difficult however, given that the arguments passed to the top-level function in the monitoring thread are controlled by SYRINGE. The attack would have to rely on modified global OS data that the target code relies on, and be careful so as to not affect control-flow.

4.5.2 Performance

In this section, we investigate the performance of the two building blocks of SYRINGE: function call injection and localized shepherding. In all cases, the reported results are wall-clock times, derived from the division of the host CPU timestamp counter (TSC, accessed by the `rdtsc` instruction) by the clock frequency. Care was taken to ensure that the accessed TSC was not virtualized, that its frequency did not vary, and that its value was synchronized across CPU cores. In all experiments, five samples were taken for each measurement and the average was used.

Function call injection was evaluated by injecting a function call to a Windows kernel function and measuring the time between when the injection starts and the target guest function starts running (steps 2–4 in Figure 8). For this experiment, no parameters were passed to the function. The entire operation, consisting of elementary CPU and memory introspection operations, followed by a VM switch into the guest, consumed an average of 0.7ms, with a very low variance. We also measured the triggering delay for our selected injection context, the OS system call dispatcher.

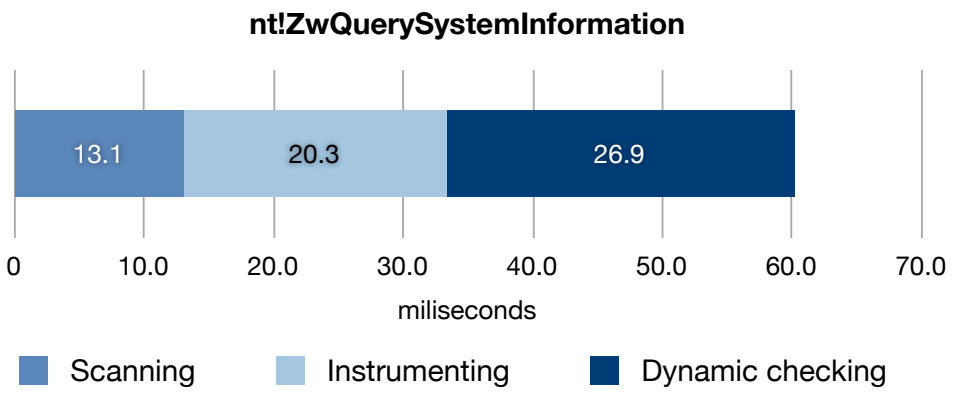

Figure 10: Shepherding execution time breakdown for the Windows executive's `ZwQuerySystemInformation` function, when used for a common monitoring task: obtain the list of active modules in the guest. In the scanning phase represented above, 3163 bytes of code were disassembled by the Disassembler and 12 4KB code pages were verified and write-protected by the Page Verifier. In the instrumenting phase, 53 critical instructions and 23 direct calls were patched/unpatched by the Instrumenter. Finally, in the dynamic checking phase, 17 critical instruction executions and 9 direct calls were handled by the out-of-guest Dynamic Checker and 316 critical instructions executions were handled by the in-guest Dynamic Checker.

This delay indicates the amount of time that a monitoring application has to wait between its request for a function call to be injected and the moment when it is actually injected. As expected, results varied widely, ranging from a minimum of 18ms to a maximum of 51ms, averaging at 33ms out 10 samples taken. We consider this number to be acceptable for most monitoring applications, indicating that the system call dispatcher satisfies the execution frequency requirement for an injection context. This frequency is expected to vary, however, depending on the nature and intensity of the workload being run inside the GVM.

Localized shepherding was evaluated by injecting a function call to a guest OS function and measuring the execution time consumed by each shepherding component. Performance measurements corresponding to the run with the median execution time and other shepherding statistics are shown in Figure 10. For this experiment, we selected a function from the Windows kernel executive commonly used

for monitoring: `ZwQuerySystemInformation`. This function is a wrapper for the `NtQuerySystemInformation` system call, which itself is a function that, in Windows XP, may call any of over 60 other functions that process and return a wide variety of system information and statistics. This makes `ZwQuerySystemInformation` an ideal candidate for evaluating SYRINGE's performance.

The scanning and instrumenting phases were consumed by inter-VM page copying operations that VMsafe uses for memory introspection. Performance could be improved by using sharing-based introspection such as used by XenAccess. The dynamic checking phase used about 50% of the total execution time, as shown in Figure 10, totaling 28.5ms. This time is almost entirely consumed by context switches between the SVM and the GVM for critical instructions that need to be handled out-of-guest, and first-time execution of direct calls, which are also handled out-of-guest. Considering that these 28.5ms correspond to just 17 critical instructions and 9 direct calls being handled out-of-guest (out of 325), averaging 0.91ms per instruction, the importance of in-guest handling cannot be overemphasized. If we were to handle all critical instructions out-of-guest, the execution overhead created would make SYRINGE impractical for use in any virtualized environment. Of the 17 instructions handled out-of-guest, 1 was an indirect call executed by the system call dispatcher and 16 were indirect jumps triggered by two different instructions. These 17 executions were handled out-of-guest because the in-guest Dynamic Checker, by consulting the in-guest target table, determined that their targets were being reached for the first time and so needed to be analyzed and instrumented by SYRINGE. These were included in the target table to indicate that all future indirect `CALL`s and `JMP`s instructions targeted at those addresses could be handled in-guest. This allowed the following 88 executions of these instructions to be analyzed in-guest. Direct and indirect `CALL` instructions, in their first execution (which has to be handled out-of-guest),

also have the in-guest call origin table updated, so that `RET` instructions can be evaluated in-guest. In our example, 211 `RET` instructions were executed and handled in-guest. The execution of the actual guest code consumes an insignificant amount of time when compared to shepherding, and is therefore not shown in Figure 10.

4.6 Monitoring Application

After measuring the performance of its individual components and its security properties, we evaluated SYRINGE in the context of a rudimentary monitoring application. This application, named *SYRMod*, uses SYRINGE to periodically obtain a list of the user and kernel modules loaded in the current process' address space. The calling interval can be defined by the applications's user. This module list is obtained by injecting a call to and shepherding guest OS kernel function `ZwQuerySystemInformation`. This is a generic wrapper function that can be used to extract information from a Windows system.

Despite the simplicity of this application, it serves to illustrate the interface exposed by SYRINGE to higher-level applications. It also helps us to gain better insight into how SYRINGE impacts the guest OS's overall performance. The code excerpt below shows how SYRMod uses SYRINGE to invoke `ZwQuerySystemInformation` in the guest.

```
1: res = VmCallGuestFunction(ntos_base + 0x26ff8,
2:         "nt!ZwQuerySystemInformation",
3:         _callback_app_generic, TRUE, 4,
4:         4, VAL, 0xb,
5:         4, VAL, &appbuf,
6:         4, VAL, 0x8000,
7:         4, STACKREF, &size);
```

Line 1 specifies the function address as an offset from NTOS's base address, line 2 specifies the function name, line 3 specifies the callback to be used, a boolean

indicating that the function should be shepherded, and the number of arguments. Lines 4–7 specify the arguments, with each line specifying the argument's size, its evaluation semantics and the argument itself. Callback activation and function result fetching are transparently handled by SYRINGE.

After `ZwQuerySystemInformation` returns, SYRMod is notified by SYRINGE. SYRMod then uses regular introspection to retrieve the results from the guest OS's memory, parsing and printing the list of modules to the SVM's standard output. A sample of this output is shown below. For space reasons, we show only its first few lines. The complete list's correctness was verified.

```
Calling nt!ZwQuerySystemInformation
...
Callback invoked for nt!ZwQuerySystemInformation
Return value: 0

Number of modules: 102

\WINDOWS\system32\ntkrnlpa.exe
Base address: 804d7000
Size: 001f6280

\WINDOWS\system32\hal.dll
Base address: 806ce000
Size: 00020380
...
```

No security alerts were raised during the shepherding phase, indicating that no integrity violations were detected with the code's execution, and that the results (the module listing, in this case) can be trusted. This example illustrates how SYRINGE can be used to monitor parts of the GVM's state with security and robustness, as required.

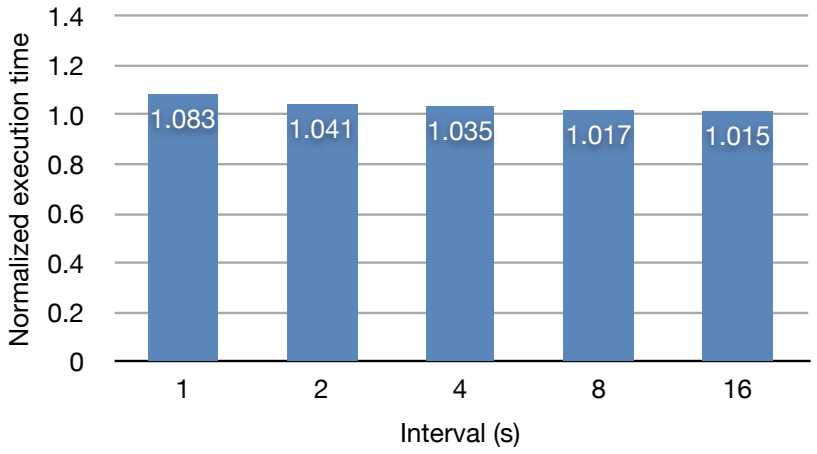

Figure 11: Normalized execution time for the decompression of the Linux kernel source code tree in the GVM. The interval between successive calls to `ZwQuerySystemInformation` is varied.

We next evaluated SYRMod's performance impact on the guest. We varied the time interval between successive callings of `ZwQuerySystemInformation` and measured the time taken inside the guest to decompress the 2.6.33.7 Linux kernel source tree, a 64MB `.tar.bz2` file. The performance penalty comes from the suspension of all guest activity when the monitoring thread is running. Results show that, for a calling period of 1 second, the measured overhead is 8% (Figure 11). This period is obviously dependent on the nature of the monitoring application and the type of information being retrieved, so it is difficult to generalize these results. We believe, however, that for many monitoring applications, a 1 second period is considered low enough so that our results indicate an acceptable performance penalty.

4.7 Discussion

The techniques used by SYRINGE have limitations. Perhaps the most noticeable one is SYRINGE's intrinsic inability to shepherd the complete execution of certain operations. Specifically, operations involving any type of asynchronous code execution, such as reading a file from disk, cannot be shepherded by SYRINGE. This limitation

is inherent to the shepherding technique. We do not see it as a deal-breaker, though, since I/O and asynchronous primitives (e.g., deferred procedure calls) are not commonly used by information querying functions, such as `ZwQuerySystemInformation`. In the near future, we plan to expand our evaluation to demonstrate how SYRINGE can be used to build a full-fledged VM monitoring application, calling guest functions other than `ZwQuerySystemInformation`.

The handling of page fault exceptions, which are very common in Windows and triggered by on-demand paging, is also affected by this problem. In this case, our current solution relies on first performing a non-shepherded call injection to the target guest function so that all necessary pages can be swapped in, and then following it with a shepherded call injection. Attacks targeting the non-shepherded call, for instance, by modifying the code read from disk, would later be detected during the shepherded call.

The effect of guest OS internal synchronization mechanisms can also be problematic for SYRINGE. Due to the disabling of interrupts, it is possible for the shepherded code to be blocked indefinitely by a spinlock or a semaphore taken by another thread, creating a deadlock situation. In our tests we only rarely came across such a situation, but a more careful investigation on how to prevent and detect such occurrences is needed. One possible solution would be to embed specific knowledge of guest OS synchronization primitives into the shepherding algorithm with the goal of determining whether the execution is stalled or not. If affirmative, shepherding can be disabled, the execution allowed to continue, and the monitoring application can be notified of the problem and choose to re-invoke the function at a later time.

We did not discuss function call injection and shepherding for user-space functions due to the difficulty of finding user-space injection contexts that cannot be easily circumvented or disabled by an attacker (for instance, by terminating the surrogate process). Aside from this obstacle, however, user-space code can be invoked through

function call injection and shepherded using the same techniques that we described for kernel code. One necessary addition would be to shepherd the mode-switching instructions that system call invocations rely on: `SYSENTER` and `SYSEXIT` on Intel x86 CPUs for the most recent OSes. We plan to investigate methods that cannot be easily circumvented for reliably injecting calls to guest user-space functions.

Due to SYRINGE's reliance on guest OS's functions, it could be argued that its monitoring capabilities are not as powerful as those of regular memory introspection. The first is restricted to the results returned by a finite set of guest OS functions, while the second can in principle retrieve any information from the guest's memory. This argument assumes that SYRINGE aims to completely replace regular introspection, which is not true. We envision SYRINGE as an information extraction tool that, due to its resilience to the semantic gap, can be used to aid many uses of regular introspection. For example, given a process base image file name, a monitoring application can use SYRINGE to determine the process' base address in memory by using guest kernel functions `PsGetCurrentProcessImageFileName`, to identify the correct process control block; `PsGetCurrentProcess`, to get the control block's address and `PsGetProcessSectionBaseAddress` to extract the process' base address from it. Using the result, the monitoring application can then use regular memory introspection to inspect the process's code.

Finally, the control-flow integrity policy used by SYRINGE's shepherding for indirect control flow transfers is generous. For instance, it will allow indirect calls to any target situated anywhere inside the whitelisted code. Still, this policy is sufficient to block kernel injected-code attacks, which is the most common type. A more precise policy could be constructed by using techniques such as points-to analysis to derive a smaller set of possible destinations for indirect control flow transfers and include that knowledge in the dynamic checker's policy [16].

4.8 Summary

In this chapter we proposed a secure and robust infrastructure for passive monitoring of virtual machines. SYRINGE leverages the guest's own code, thus overcoming the semantic gap inherent to introspection and achieving good robustness. Security is achieved by removing the monitoring application from the guest through function call injection and verifying the execution of the guest code using localized shepherding. We have implemented SYRINGE using the VMsafe introspection API to monitor a guest OS running Windows XP. We evaluated the performance and security of SYRINGE, showing that all simulated attacks were detected. Finally, we built and demonstrated a prototype application, SYRMod, which uses SYRINGE to periodically obtain the list of loaded guest modules. SYRMod showed that for a calling interval of 1 second, the performance overhead imposed by our system is 8%.

CHAPTER V

SECURE DATA-DRIVEN ACTIVE MONITORING OF GUEST OPERATING SYSTEMS

5.1 Motivation

Although passive security monitoring is effective at detecting certain classes of attacks, it cannot by itself be considered a complete security solution. First, it is mostly limited to detecting attacks which have already happened and not preventing them. Second, it is incapable of detecting attacks that are transient in nature if the scanning period is higher than the attack's lifetime. For these reasons, active monitoring of events is considered an important part of security applications, such as anti-virus tools.

Active security monitoring works by intercepting certain types of events as they happen, for instance, by diverting code execution flow, and analyzing their execution context. This information can then be used to decide on a response action, for instance, allowing the event to proceed, blocking it, or raising a security alert. One common example is file creation: anti-virus tools commonly hook the OS filesystem code so that they can be notified when a file is being created, and analyze the contents of that file for known malware signatures.

Traditional active monitoring is done by placing *code execution hooks* at strategic points inside the operating system and having them redirect control-flow into a monitoring agent that resides inside the monitored system. These hooks can be implemented through a variety of techniques, such as dynamically patching in-memory code or modifying control-related data, such as function pointers. The agent can be deployed as a kernel driver, a user-space process or a combination of both.

This architecture presents two major problems. First, as discussed previously, the monitoring agent can be tampered with or disabled. Second, the technique of relying on code execution traps to intercept events is vulnerable to circumvention by malware. Even if the hooks themselves can be protected in memory [74], malware can circumvent them by simply copying the hooked code, removing the hook and execution the code, or directly invoking the non-hooked lower-level functions [98]. In the extreme case, sophisticated malware can choose to simply not invoke any code and directly modify the kernel data objects to mimic the execution of the intended event. In this case, no amount of code execution hooks would prevent circumvention. We identify these as being the two major security problems with modern secure active monitoring.

In this chapter we present the Data Access Reporting Platform (DARP) as a solution to these problems. DARP solves the first problem by leveraging the basic introspection technique of removing the monitoring application from inside the monitored OS, and placing it in a separate, trusted virtual machine. Second, and most importantly, DARP makes use of a novel virtualization-based data-oriented event interception primitive: *data access hooks*. This primitive works by intercepting low-level write operations to monitored regions of guest memory. DARP translates these operations into syntactic-level kernel object events, thus partially bridging the semantic gap. It presents this syntactic view to active monitoring applications, who can use it to infer higher-level events.

The benefit of this data-oriented approach relies on the insight that a very large number of code execution paths (or even none) can be used to process an event, and the difficulty of insuring that all of these paths will be hooked explains how code hooks can be easily circumvented. All these execution paths, however, produce the same end-result: a specific pattern of data object modifications. Therefore, by monitoring data modifications instead of code execution to infer events, and taking

certain additional precautions, we argue that it is possible for a security application to greatly reduce the risk of active monitoring being circumvented by malware. The obvious downside to this approach when compared to code execution hooks is the substantial increase in the number of trapped operations. DARP makes uses of several optimizations to amortize the volume and cost of data write interceptions.

This chapter also presents a monitoring application that uses DARP to monitor a very common type of OS event: file opening. Our application uses DARP to monitor the creation of certain kernel object and translates these into high-level file open events.

5.2 Previous Approaches

Active monitoring of application and operating system events has always played a prominent role with security monitoring solutions. All of the top anti-virus products perform system-wide active monitoring to prevent or detect malware infections [101, 62, 53, 65]. Host-based intrusion detection systems (IDS) have also traditionally relied on active monitoring to implement techniques such as online/offline analysis of system call execution patterns [45, 36, 37, 93, 80].

These systems suffer from serious security limitations by first, not taking any measures to protect the monitoring application; and second, relying on code execution hooks to intercept events. The first allows malware to tamper with or disable the monitoring application while the second allows it to evade the active monitoring.

Recent research has tried to address the first issue by leveraging virtualization to isolate and protect the monitoring application. Lares places the monitoring application in a trusted security VM, from which the untrusted guest VM is monitored [74]. Events are captured through in-guest memory-protected code hooks and forwarded by the hypervisor to the security VM. SIM goes a step further by allowing the security application to stay in-guest and protecting it through the use of additional

hypervisor-enforced guest address spaces [90]. Despite its advances, both of these systems rely on code execution hooks and therefore can be circumvented.

Other works adopt an architectural approach to monitoring certain types of guest events. Antfarm, for instance, does not rely on any in-guest components, instead monitoring low-level architectural events from inside the hypervisor [51]. One of these events, the address space switch (load CR3), can be used to monitor the creation, destruction and scheduling of processes in the guest VM. VMScope records all system calls executed in a honeypot environment by using an emulator to intercept CPU instructions associated with their invocation [49]. Patagonix monitors code execution inside the guest VM [59] by relying on the hypervisor's memory virtualization. It intercepts code execution events and verifies the memory pages containing the code by comparing them with a whitelist. This type of monitoring is sometimes referred to as *architectural introspection* and presents a relatively low risk of circumvention, given the level at which the monitoring is performed. Its disadvantage is that it cannot be generalized to other types of high-level events: only those that have a corresponding low-level architectural equivalent.

More recently, Srinivasan et al. proposed a scheme for relocating suspect processes from the guest VM to the security VM, while still having it interact with the guest's kernel [95]. This approach is limited to the monitoring of a single process and does not support system-wide monitoring, as is commonly required by security applications.

Finally, a significant body of research has been produced on areas related to fine-grained active monitoring of systems and applications for tasks such as dynamic malware analysis [110, 70, 29] and inline reference monitoring [105, 1, 5, 33, 18, 24, 19, 6]. These works, despite also performing a certain type of active monitoring, have assumptions, constraints and goals that are different from ours.

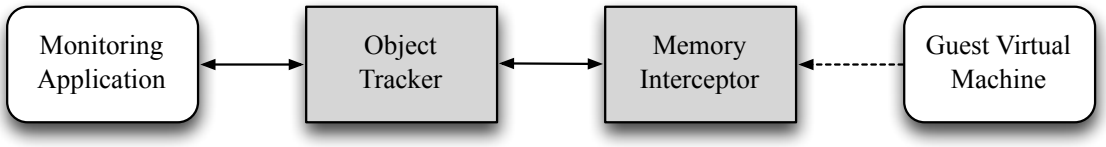

Figure 12: A conceptual view of our solution's architecture, with DARP's high-level components highlighted in grey.

5.3 The DARP Active Monitoring Infrastructure

5.3.1 Overview

The Data Access Reporting Platform (DARP) is a virtualization-based infrastructure for general and secure active monitoring of virtual machines. DARP is used by active monitoring applications running on a security VM to monitor events happening inside an untrusted guest VM. Our infrastructure implements the idea of data access hooks, which are basically a trapping mechanism that is triggered whenever data is modified at certain regions of guest memory. DARP leverages this idea to monitor changes made to parts of the guest kernel's object graph, and reports those *object events* to the monitoring application, which can then use them to infer high-level system events.

DARP is composed of two high-level modules (Figure 12): a memory access interceptor, responsible for intercepting memory writes targeting objects belonging to monitored types; and an object tracker, responsible for converting memory access events into object events, and sending them to the monitoring application.

Object accesses occurring inside the guest VM are trapped by the memory access interceptor running inside the hypervisor. This component leverages the hypervisor's control over a VM's memory to write-protect the guest pages containing monitored objects. We refer to monitored regions of guest memory as *watches*. Watch creation/removal requests are sent by the object tracker component as changes are made to the monitored object graph. At each memory write interception, DARP collects information such as the target address and the contents of the write, and builds a

memory access tuple containing this information. This tuple is then sent to the object tracker.

The object tracker translates memory access tuples into syntactic-level kernel object events. These represent the creation, deletion or modification of a kernel object of a given type. This inference requires the tracker to maintain a shadow object graph that mirrors the location, type and disposition of all guest objects currently being monitored. Incoming tuples are processed against this shadow graph and changes corresponding to the memory access are made. This processing often requires the use of introspection to traverse new object hierarchies in the guest and may result in multiple object events, as well as multiple watch creations and destructions. Once the tracker finishes processing the access, it generates a list of all the object events generated as a result and sends them to the monitoring application.

DARP provides applications with a general, data-driven, syntactic view of the activity happening inside a VM. Given its focus on monitoring data rather than code execution, DARP-based active monitoring is immune to most types of code hook circumvention techniques. With additional security measures in place, as discussed in Section 5.6, we argue that DARP can withstand even the most advanced hook circumvention techniques.

5.3.2 Assumptions

In DARP we make the same basic assumptions as other introspection works. Specifically, we assume that the hypervisor and the security VM are trusted and constitute our system's TCB. The guest VM is untrusted and may be under full control of malware. We also assume access to the guest OS's source code and detailed symbol information. This data is used by DARP to obtain the structure of object types and compute the target types for generic pointers, in a manner similar to the one described in Chapter 3. We also assume the ability to perform a small number of

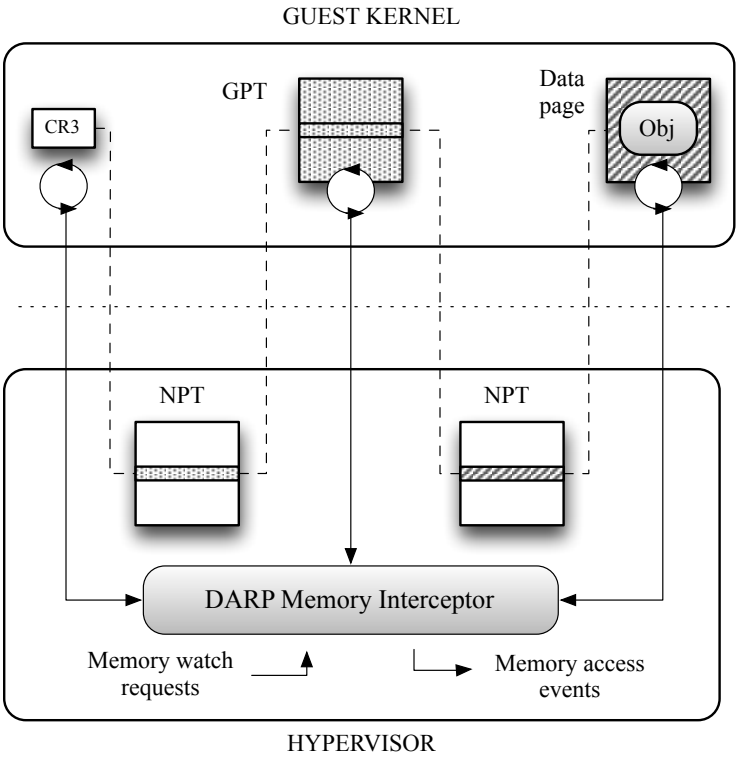

Figure 13: DARP's hypervisor-based memory interceptor monitoring a kernel object. The nested page table (NPT) entries corresponding to the guest physical page and all guest page tables (GPTs) used in the translation are marked as read-only. Changes to any of these, including the guest CR3, trigger the memory interceptor.

changes to the guest OS kernel. Finally, we assume that DARP runs on a system that supports Intel/AMD latest virtualization extensions, including hardware-supported memory virtualization.

5.3.3 Memory Write Interception

The interception of memory writes constitutes the low-level foundation of our active monitoring infrastructure. The component responsible for it is called *memory interception component*. It interacts with higher-level components in two ways. First, it receives tuples consisting of a guest virtual address and a size specified in bytes. This tuple represent a guest virtual address range that DARP wants to monitor for

changes, which we will refer to as a *memory watch*. Second, whenever a write operation is intercepted, it sends to higher-level components an memory write tuple representing the operation. It consists of a guest virtual address, a buffer containing the data that was written and the size of the data. This interface is illustrated in Figure 13.

The mechanics of memory write interception rely on the memory virtualization performed by the hypervisor to isolate different VMs and virtualize their view of memory. To achieve this, the hypervisor introduces an additional layer of indirection in the address translation process, which we leverage to implement our interception capabilities. A description of the most common memory virtualization techniques used by hypervisors is given in Chapter 2. Given the increasing popularity of hardware memory virtualization support and its advantages in terms of performance and software complexity reduction, we have chosen to implement DARP's memory interception on top of hardware-supported nested paging.

When using nested paging, guest memory management inside the hypervisor must be done at the guest physical level. This contrasts with shadow paging, in which hypervisor-based shadow page tables translate guest virtual addresses (GVA) directly into machine physical addresses (MPA). Nested page tables translate guest physical addresses (GPA) into machine physical addresses, and therefore have no knowledge of a guest's virtual address space and how it is mapped. Therefore, in order to protect a guest virtual address range, the first step is to perform a guest page table walk in order to determine the base GPA of the page containing the watch range. For simplicity, we will assume that the range is contained inside a single 4KB page. This walk procedure can be done using the hypervisor's memory mapping capabilities. After the target GPA is determined, the corresponding entry in the nested page table is marked as read-only by clearing the entry's write bit. This will effectively cause a hypervisor exit every time the guest operating system writes anywhere in that physical page.

This is not ideal, since not only the writes targeting the watch range (which may be very small) but the entire page will cause a hypervisor exit. Unfortunately, the current Intel x86 CPU architecture limits the memory protection granularity to a single-page, or 4KB.

When a DARP-caused hypervisor exit happens, DARP first emulates the trapped instruction from inside the hypervisor. This is necessary so that the guest's virtual processor can be resumed at the next instruction. Then, DARP reads the faulting GPA from the structure containing information regarding the hypervisor exit and determines whether the address range affected by the write operation intersects with any region of virtual memory that is currently being watched. In such case, it constructs a memory access tuple containing the GVA of the accessed memory watch, the new data that was written, and invokes DARP's object tracking component to process it. Finally, the interrupted virtual processor is resumed.

The procedure described above assumes that the kernel guest virtual to physical mapping is fixed for all virtual address spaces active inside the guest. This is clearly not true, since the same virtual page may be mapped to different physical pages in different address spaces and even in a single address space, the virtual to physical mapping may change across time due to memory swapping. Malware could also simulate a page swap to evade DARP's monitoring. It is interesting to observe that these difficulties are present with nested paging, but not with shadow paging. As a result, in order to effectively monitor all accesses targeting a certain guest virtual memory region, all the guest page table entries involved in translating the region's virtual addresses in all the guest's currently active address spaces need to be monitored. The mechanics through which this is done resembles a lightweight version of the shadow paging algorithm. First, we trap all address space switches, which in the Intel x86 architecture is done by writing to the CR3 register. Second, for each active address space, we perform a guest page table walk to determine the GPAs of all page table

entries involved in translating the virtual address range, as well as the data page itself. We install special watches in all these page table entries, so that DARP can be notified whenever these entries are modified. When this happens, DARP recomputes the target of the PTE and adjust all PTE and data memory protections accordingly to reflect the change.

A legitimate concern is whether the performance benefits of nested paging are nullified by the lightweight shadow paging performed by DARP. This is not the case, since DARP only traps write operations targeting PTEs related to active memory watches, whereas shadow paging does it for all PTEs in all page tables. Its performance overhead is thus only a fraction of that present with traditional shadow paging.

5.3.4 Kernel Object Tracking

The DARP infrastructure, at its highest level, tracks the creation, deletion and modification of kernel objects. Below, it interacts with the memory interception component by sending down sequences of memory watches that it wants to monitor, and receiving memory write tuples. The object tracking component is responsible for translating these low-level write operation into mid-level object events that it can then send to the monitoring application. The monitoring application can then use these events to infer high-level system events, such as process creation or file opening.

To track kernel objects, it is necessary for DARP to understand which object types it has to monitor, the precise layout of these types, and how they can be located in the kernel heap. The first information is provided by the application as a list of types, and the second one as a list of object edges. Symbol files are located by DARP every time a guest VM boots or a kernel module is loaded. On these events, a notification is sent to the object tracker containing several key attributes of the loaded binary, such as its size, checksum and timestamp. DARP can then use this information to

locate the correct symbol file associated with the loaded binary.

Information regarding the location of objects in memory is given to DARP as a sequence of edge tuples composed of a source field and a destination field. Each of these fields refers to an actual kernel object field, the tuple thus representing an edge in the kernel object graph. They encode the information required to locate a dynamic object in the guest's memory, starting from a static root, i.e., a global variable.

DARP's object tracker uses this information to maintain a *shadow object graph*, which represents a subgraph of the entire guest kernel object graph. This subgraph contains not only the objects that the application wants to monitor, but also those whose monitoring is required in the manner specified by the edge tuples. The object tracker translates memory event notifications into shadow graph object events. These operations may include the creation and/or destruction of objects. When this happens, the monitoring application needs to be notified and the shadow graph be updated. These updates are then reflected down to the memory interception component as a sequence of memory watch creation and removal operations, depending on which objects were created and which were removed from the shadow graph. As an aid to our object tracker, we modified the guest OS's kernel to notify DARP of all pool allocations and de-allocations related to the object types being monitored. This allows the object tracking algorithm to be simplified with regard to knowing when certain objects are not being used anymore and can be removed from the object graph. In a typical scenario, a monitored edge field is written to and the write notification is translated by the object tracker into a write to field F of object O. The object tracker knows that this field is being monitored, and is a pointer to an object of type T, which is also being monitored. If the new contents of the field are different from the old contents, a change to the graph is being made. In this case, object T at location A is being replaced by another object of type T at location B. One object-level operation can be inferred: the creation of T_B. The removal of T_A is

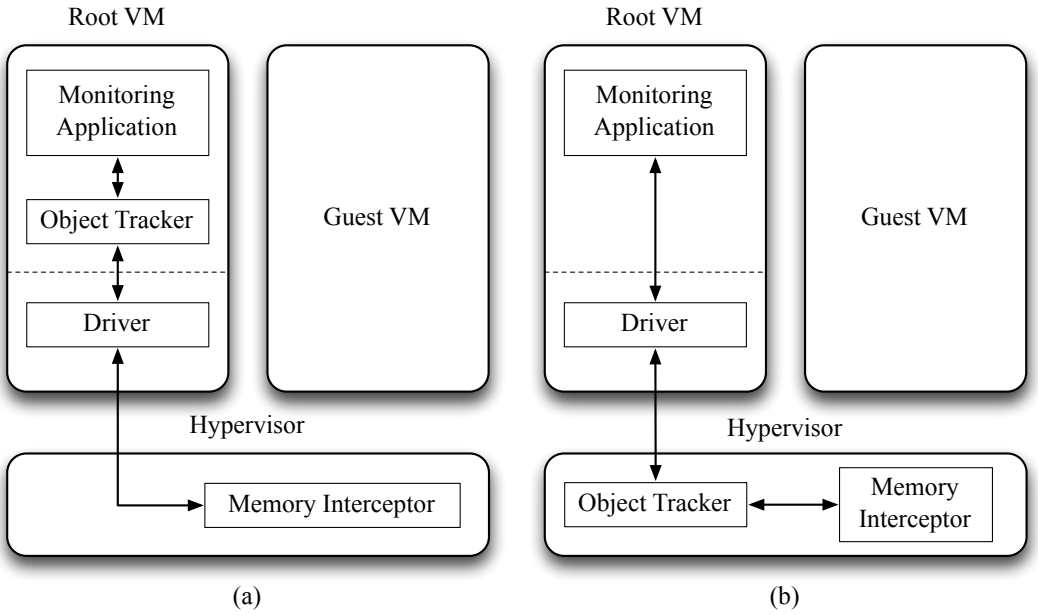

Figure 14: Two DARP design alternatives were implemented, varying the location of the object tracker.

inferred if/when a notification of removal of the kernel pool block containing T_A is received from the guest. Object events are sent to the monitoring application, and memory watches are adjusted to reflect the addition/removal of objects.

In more complicated cases, the procedure described above can involve the traversal of long chains of objects, including arrays and linked lists. DARP has special logic to handle these structures.

5.4 Implementation Details

Our infrastructure was implemented in a Windows virtualized environment. The DARP memory interception component was implemented inside Microsofts Hyper-V, and directly leverages some its functionality, like page protection and instruction emulation. The DARP application was implemented as a C++ program running in the root partition, which runs Windows Server 2008 R2. The guest partition ran Window 7 SP1.

Two design alternatives were explored when implementing the object tracker. In design 1 (Figure 14(a)), the object tracker was implemented as a DLL library, linked to the application, running in the root's user-space. In design 2 (Figure 14(b)), the object tracker was implemented inside Hyper-V, co-located with the memory interceptor. The reason for us choosing to implement these two versions was related to the trade-offs involved in each. For instance, it is expected that design 1 should be less efficient than design 2, given that messages will have to be sent from the hypervisor all the way to user-space. On the other hand, design 1 should be more reliable and secure than design 2, given that it does not add new (and potentially vulnerable) code into the hypervisor. We discuss these trade-offs more extensively in Section 5.7.

Communication between root user-space and the hypervisor is handled by a kernel driver installed on the root partition. To send information down, our DARP driver invokes a special DARP hypercall and passes arguments through a shared memory buffer. Communication in the opposite direction is performed asynchronously. The driver registers an IRQ on the root partition and communicates it to the hypervisor. To send information, the hypervisor injects an interrupt into the root VM, whose handler is implemented by the DARP driver. The handler copies the data from a different shared memory buffer and passes it to user-space. Communication between the driver and user-space is done through IOCTL calls.

The guest pool allocation/deallocation notifications were implemented by modifying the Windows kernel's main heap allocation functions: `ExAllocatePoolWithTag` and `ExFreePoolWithTag`. A `VMCALL` instruction is executed by these functions in case the tag associated with the pool block being allocated/freed matches any of certain pre-determined tags. Malicious manipulation of pool blocks, pool tags or the `VMCALL` invocation inside the guest will be detected by DARP, since it can detect if an object does not fall inside a pool block or if its tag is not the expected one for that particular

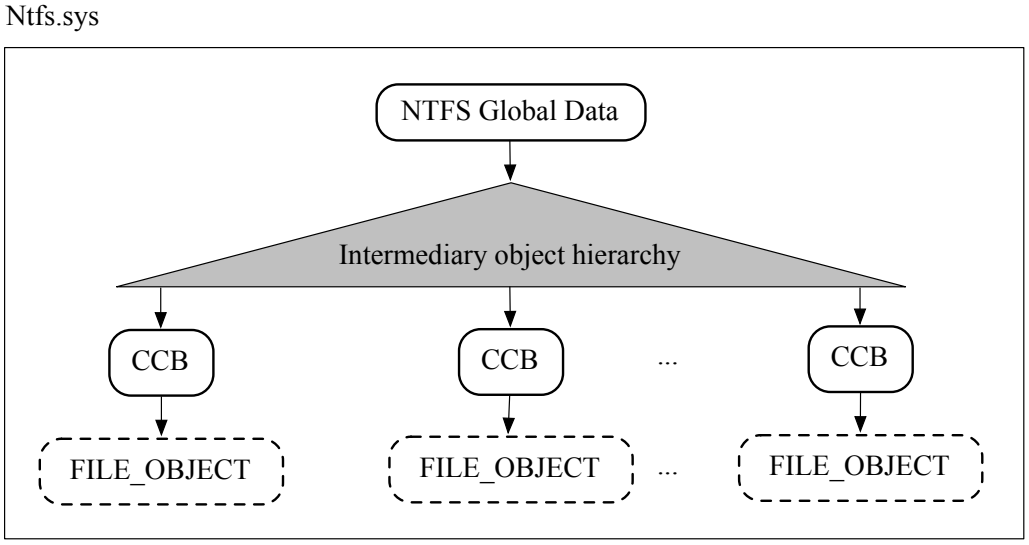

Figure 15: Abstract view of the NTFS object hierarchy being monitored for file open events. The creation of CCB objects is interpreted as a file opening event. Additional information about the file can be extracted from the corresponding FILE_OBJECT.

object type. In such cases, the monitoring application can be alerted.

As a performance optimization, we further modified the kernel heap allocator so that objects belonging to monitored types are allocated separately from all other object in the kernel. A monitored object will not share the same page as a non-monitored object. The goal here is to prevent unnecessary hypervisor exits, since memory protection is enforced at the 4KB-page level on the x86 architecture. The performance benefit of this optimization will be measured and discussed in Section 5.7.

5.5 Monitoring Application

We implemented a simple monitoring application to illustrate DARP's active monitoring capabilities, as well as to evaluate its high-level performance impact on the monitored system. We derived the object edge and event tuples required to monitor the creation and opening of files on a Windows 7 system. This choice was made due to the importance of file access monitoring in modern anti-virus tools.

The first step towards building this application involved deriving which kernel

data structures are directly used by the guest OS to create/open files. Our study led us to the OS's filesystem driver, Ntfs.sys, and its internal data object hierarchy. The NTFS driver implements the filesystem operations and is directly used by the kernel's executive (NTOS) to perform disk I/O. This driver creates a series of dynamic data objects responsible for storing various types of metadata about opened files. One of these objects, the NTFS Current Control Block (CCB), is created whenever a file is opened. This may be a result from a file being created, or a pre-existing file being opened. A CCB object contains a pointer to an object of type FILE_OBJECT. This object aggregates the file's metadata and cached data, and can be used by an application to derive additional information about the file. An abstracted version of this object hierarchy is shown in Figure 15.

The set of rules is encoded within the application itself and passed down to DARP's object tracker, along with necessary symbol information that is automatically extracted from symbol files. These rules focus on the monitoring of CCB objects, along with all intermediary objects up to one or more static global variables. The actual rules could not be listed here due to NDA restrictions regarding the Windows source code.

Whenever DARP's object tracker identifies the creation of a new CCB object inside the NTFS module, it sends a message to the monitoring application consisting of the name of the file and a pointer to its corresponding FILE_OBJECT, both stored as fields of the CCB. The FILE_OBJECT pointer can be used by the application to perform memory introspection on the guest and derive additional information regarding the file. At this point, the application simply prints out the name of the file being opened.

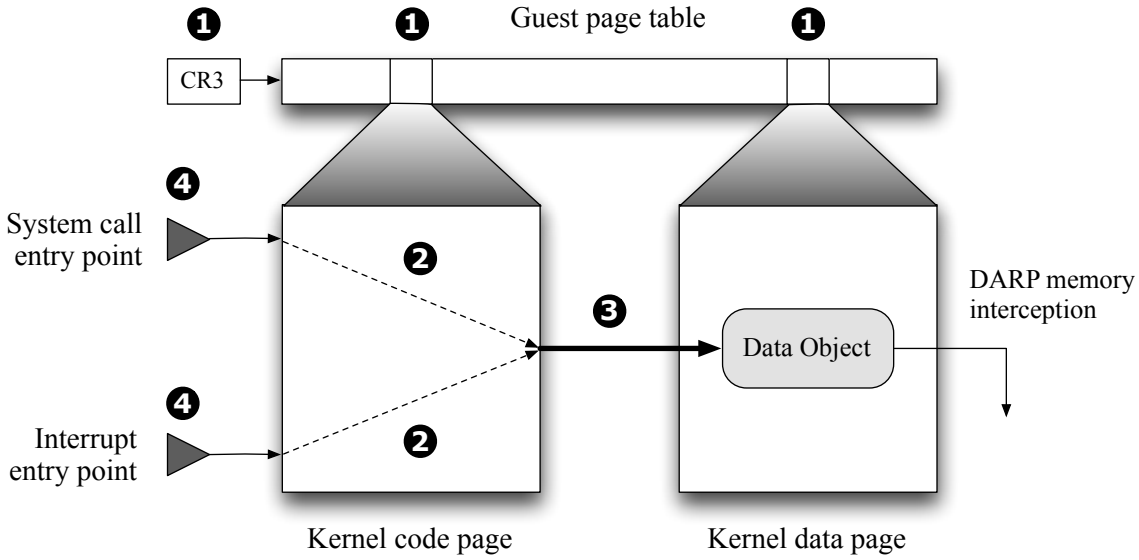

Figure 16: Security analysis of DARP's active monitoring, illustrating all major OS components involved in an object write operation, along with possible points of circumvention.

5.6 Security Analysis

A security analysis of DARP's active monitoring techniques first requires identifying all major guest OS components involved in an object write operation. These consist of the address translation structures, kernel entry points, the kernel code and the data objects themselves (Figure 16). Let us now assume that kernel event E involves the modification of kernel data objects $D(E)$. $D(E)$ is encoded by the set of edge and event rules passed down from the application to DARP. We define property P as follows: *if E occurs, all changes made to $D(E)$ will be intercepted.* DARP's goal is to implement P. In other words, there can be no possibility of circumvention by an attacker who wants to trigger E without notifying the application. Enforcing this property requires looking carefully at the guest kernel components shown in Figure 16.

At the level of address translation structures, it is possible for an attacker to achieve circumvention be remapping code and/or data pages to his own code and

data, which is not monitored. This can be done by directly modifying the existing page tables or creating new ones and modifying the CR3 value (Figure 16-(1)). DARP prevents this attack by monitoring guest page tables and CR3 switches in the manner described in Section 5.3.3. Whenever an address space switch happens or changes are made to guest page tables that affect DARP's guest physical monitoring, memory protections are automatically transferred from the old physical pages to the new ones. The same level of protection can be achieved with kernel code by deploying any of well-known code integrity techniques [87, 59, 81].

The kernel's control-flow integrity could also be corrupted by an attacker so as to divert the execution flow into his own malicious code, which in turn accesses non-monitored data objects (Figure 16-(2)). For this reason, kernel CFI is a necessary condition to prevent circumvention. We realize, however, that complete kernel CFI is currently an unsolved research problem, and that a weaker form of code integrity might suffice to thwart the vast majority of attacks. We propose that static code integrity might suffice for DARP. This would still leave out control-flow data as an attack target, but in order to successfully circumvent DARP's monitoring without compromising the stability of the operating system, an attacker would have to find attack points that mediate all possible code execution paths that access the objects in question. We think that the likelihood of this happening is low. Write-protecting the kernel code also prevents attacks against static data references targeting global data, which serves as a root to locate dynamic objects in the heap (Figure 16-(3)). Finally, the CFI of kernel entry points (Figure 16-(4)) must also be considered, given that a sophisticated attacker could simply replace the entire kernel executive, or specific kernel drivers, with his own malicious versions. These entry points consist mainly of the kernel system call dispatcher and the Interrupt Descriptor Table (IDT).

In summary, achieving property P requires (1) integrity of addressing structures, (2) kernel control-flow integrity, (3) kernel static data access integrity and (4) kernel

entry-point integrity. One might ask, if all these measures are in places, whether the motivation still exists for DARP's data access hooks and whether old-fashioned code execution hooks would not do the job just as well, with significantly less performance overhead. The answer is that measures (1 - 4) are a necessary, but not sufficient condition for P and this can be shown through a counter-example. Assuming (1 - 4) are in place, an attacker could still load a malicious driver and directly modify data objects associated with a certain type of events without relying on any kernel code. This would circumvent code execution hooks even if (1 - 4) are in place. Thus, DARP is still required.

It should also be emphasized that in its role as a monitoring infrastructure, it is not DARP's responsibility to determine exactly which objects $D(E)$ correspond to event E. This is the task of the application, which will transmit this knowledge to DARP in the form of the edge tuples. DARP will then ensure that all changes performed to $D(E)$ are intercepted and reported to the monitoring application. The application developer must therefore ensure that it selects objects that are intrinsic to the handling of E, and not simply objects used by the OS for bookkeeping purposes.

5.7 Evaluation

We performed a series of experiments to evaluate DARP's performance and security when being used by the monitoring application described in Section 5.5. Our physical environment consisted of a PC based on an Intel Core i7 2.93 GHz CPU with 4 cores and 8 GB of RAM, running Windows Server 2008 R2 and the Hyper-V hypervisor. We tested a single guest VM running Windows 7 SP1 with 1GB of RAM. A description of our experiments and results follows.

5.7.1 Performance

To evaluate the performance of DARP when being used by the security application, we used the guest VM boot time as a macro-benchmark. Booting an operating system

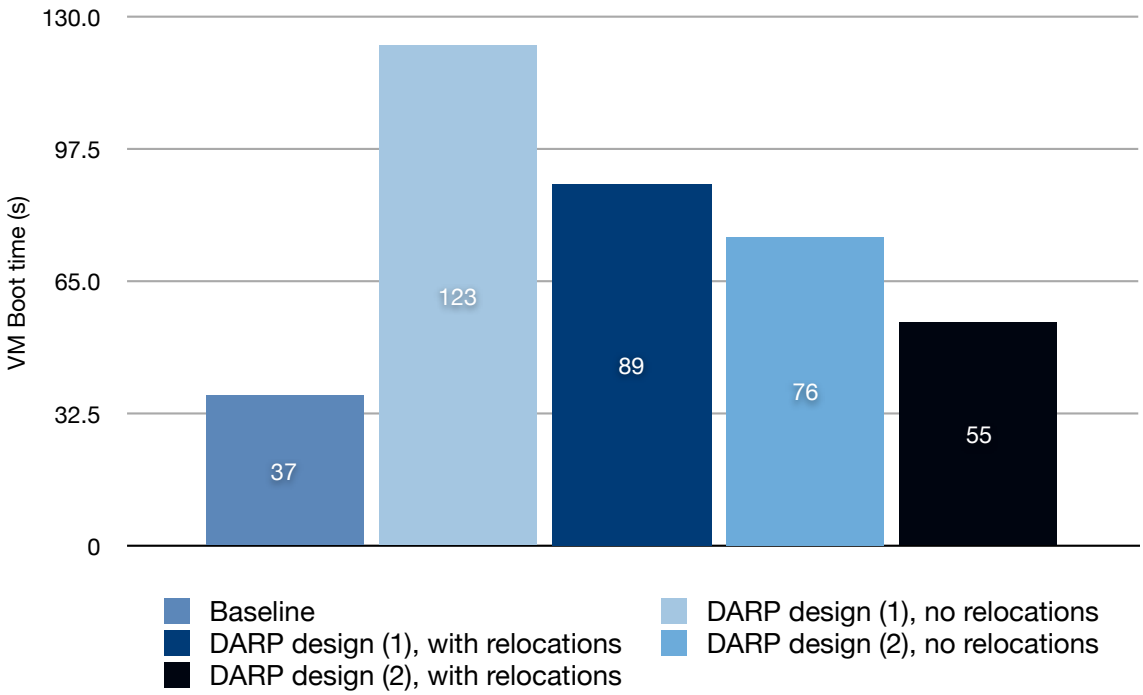

Figure 17: Time measurements for the guest VM boot process under five different experimental configurations. With the baseline, no DARP is being used. With the other four configurations, DARP is being used by our file monitoring application, varying the design and whether object relocations are being used.

and starting its processes is a file-I/O intensive procedure, being a suitable candidate to test the efficiency of our file monitoring application. We define *boot time* as the elapsed amount of time from the moment when the VM is turned on to that when the Windows 7 login prompt is shown.

In order to evaluate the relative efficiency of our two distinct design options, along with the effects of our implemented guest-based object relocation, we conducted tests with five different configurations. The first consists of the guest VM boot time without any DARP kernel modifications or DARP monitoring being performed. We refer to this configuration as our *baseline*. Two other configurations include DARP's user-space design, where the object tracker is implemented as a DLL in the root partition (design 1), and the other two include DARP's hypervisor-based design, where the object tracker is inside the hypervisor (design 2). We tested each design with and

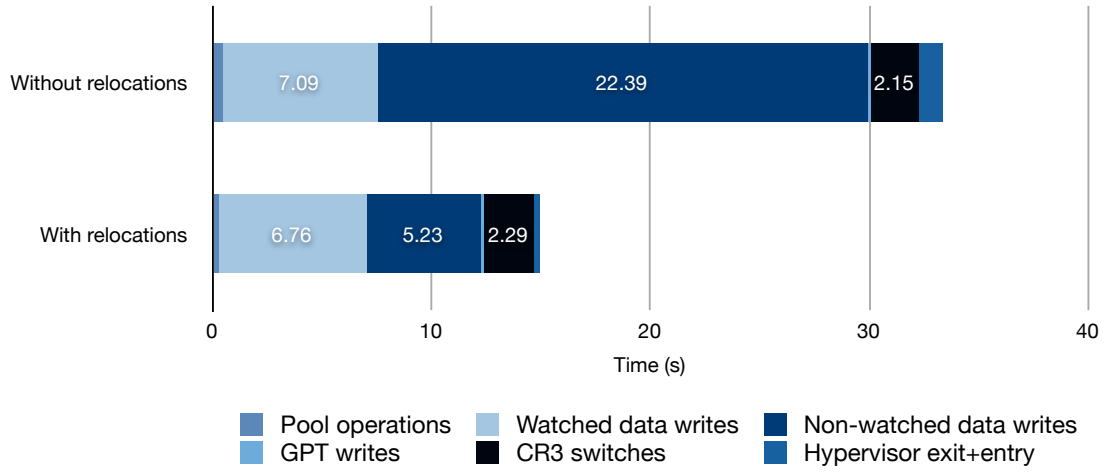

Figure 18: Performance breakdown for the overhead introduced by DARP's different components when monitoring files opened during the guest VM boot assuming that the object tracker is deployed inside the hypervisor.

without the object relocation optimization. Three measurements were performed for each configuration and the median result was used.

Our results are shown in Figure 17. As expected, the performance advantages of placing the object tracker inside the hypervisor, as opposed to root user-space, are quite substantial. This design approach eliminates the need for DARP's memory interceptor to send a message to root user-space (an expensive operation) every time that a monitored object field is modified, as occurs with design (1). Instead, with design (2), a local function call is made. As a result, according to our measurements, overhead reduces from 232% to 105% if relocations are not being used; or from 140% to 48% if they are.

Relocating monitored guest objects to dedicated memory pages has also shown significant contributions to performance. Doing so reduces the number of non-monitored memory interceptions that happen when a non-monitored object resides in the same memory page as a monitored one. For design (1), this optimization improves the overhead from 232% to 140%; and for design (2), from 105% to 48%. The nature of the performance gain obtained by using relocations is illustrated in Figure 18, which

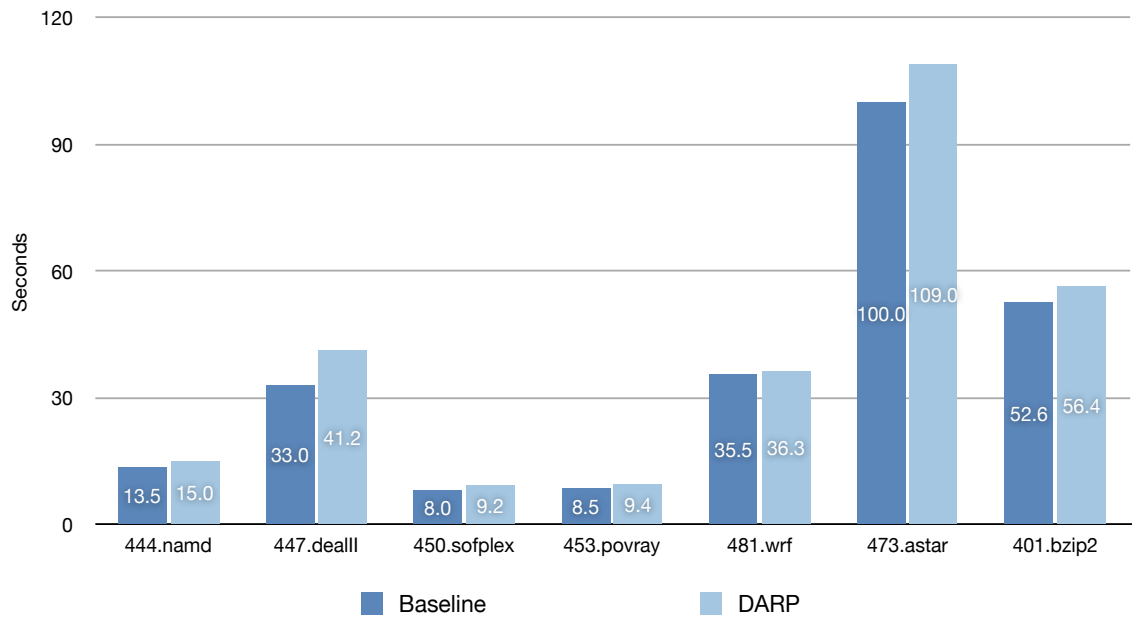

Figure 19: Measuring the performance effects of system-wide DARP-based file monitoring on SPEC2006 benchmarks. We compare time measurements obtained from running the benchmarks on our baseline configuration with our most efficient DARP configuration—design (2) with relocations enabled.

shows the overhead breakdown for all the different DARP components involved in the monitoring process for design (2). The overhead is clearly dominated by the handling of non-watched memory writes when relocations are not being used. These 22.39 seconds correspond to 5230569 occurrences, each consuming an average of 12542 CPU cycles. In comparison, only 24543 watched memory write occurrences were recorded: a mere 0.4% of the total. The use of relocations reduced the number of non-watched memory writes to 1220179: a reduction of 77%. It should be observed that even in this case, non-watched writes still dominate over watched writes with regard to frequency: the later ones remain only 2% of the total. However, given the cost of each watched write—846979 cycles in average—these dominate non-watched ones with respect to time.

When combining object relocation with the use of an in-hypervisor object tracker,

the total overhead imposed by DARP for monitoring file opening operations is reduced from 232% to 48%: an improvement of 79%. This data emphasized the importance of optimizations such as these for DARP's performance and its applicability in the real world.

In addition to using the guest OS boot time as a macro-benchmark, we also measured how the system-wide DARP-based file monitoring implemented by our application affects a set of standardized benchmarking applications. For this purpose, we ran a subset of SPEC2006 on a guest system running nothing but default system processes on the background. Our results are shown in Figure 19. They indicate a much smaller overhead when compared to our previous VM boot experiment. This can be expected, since SPEC2006 is primarily CPU-bound whereas VM boot is disk-bound. We did, however, measure a slight increase when comparing baseline runs with DARP runs of SPEC. The overhead can be attributed to a combination of file activity being carried out by system processes as well as the processing of VM exits caused by CR3 changes and handling of guest pool allocation/removal.

5.7.2 Security

We performed a security evaluation of DARP and our implemented file monitoring application by testing its resilience to several circumvention techniques commonly employed by malware. We selected three common hook evasion techniques: code execution hook removal, system call interface bypass and high-level function bypass. The first one simply overwrites the hook in memory (e.g., in the system call table), effectively removing it [102]. The second relies on a kernel driver to received IOCTL commands from a malicious process and directly invoke the system call handler, thus bypassing any hooks that might have been place in the system call interface [98]. The third applies the same technique, but one layer below, bypassing any hooks placed in the system call handler or, for instance, on an Exported Addresses Table (EAT) of a

Table 6: Results testing DARP's resilience to several common hook circumvention techniques used by malware, when monitoring file open operations.

Type of Circumvention	Example Attacks	Intercepted
Code execution hook removal	Peligroso, DeepDoor [102]	Yes
System call interface bypass	Illusion [98]	Yes
High-level function bypass	Rustock [54]	Yes

kernel driver [54].

For each case, we constructed an artificial attack for our Windows 7 VM based on the published descriptions of each malware. These attacks do not implement the full functionality of the malware, but the basic circumvention technique, according to the descriptions. Each attack applies its corresponding evasion technique to the task of doing an NTFS file open. For each case, we determined whether DARP was able to intercept the file opening operation despite the attempt at circumvention. The results are shown in Table 6. In all cases, DARP was able to intercept the file opening operation.

5.8 Discussion

As shown by our evaluation results, DARP's performance overhead is significant. Despite very large improvements achieved through the use of relocation-based optimizations and a new design, the measured overhead is 48%. Nevertheless, our performance analysis suggests several venues for further improving this number, provided that more aggressive changes are made to the guest operating system and the underlying architecture.

A substantial part of the overhead comes from the handling of non-watched writes. Our results show that, in the best scenario, 98% of all intercepted writes are non-watched, and the time taken to handle each of those is dominated by the instruction

emulator. One venue towards optimization is the reduction or elimination of non-watched interceptions. These are a result of the way kernel objects are structured combined with the effects of page-level granularity of memory access control on modern CPU architectures. Superfluous interceptions could be eliminated through one of two possible ways: (1) CPU manufacturers including support for byte-granularity memory access control; or (2) OS vendors partitioning kernel objects so that their monitored fields are kept separate from non-monitored fields. There has been research exploring the benefits of the second approach and its results are promising [96, 107].

Further performance improvements could likely be achieved by redesigning parts of the OS data structure hierarchy to reduce the number of intermediary objects between a global variable and a monitored object. In our experiments, for instance, a considerable number of intermediary NTFS data structures in addition to CCB need to be monitored due to the way in which the object hierarchy is structured. Reducing the number of intermediary levels between the global variables and the CCB objects would results in a reduced number of both watched and non-watched memory write interceptions.

Surprisingly to us, the cost of hypervisor exits and entries is almost negligible when compared to the cost of DARP's internal components. The instruction emulator used by DARP, for instance, averaged 11607 cycles per emulated instruction, 92.5% of the total non-watched memory handling cost. On the other hand, the cost of a single hypervisor exit entry in modern CPUs is on the order of hundreds of cycles [64].

DARP's role as an infrastructure is to re-create the syntactic context of guest kernel memory write operations, translating them into object events that can then be used by a monitoring application to infer high-level events. This puts the burden of bridging the gap between syntax and semantics on the application, and requires access to the operating system's source code: something that in many cases is simply not available. One way of addressing this issue would be for OS vendors to provide

and intermediary layer that sits between the application and DARP. This layer's responsibility would be to encode the rules for the most common types of events monitored by certain classes of monitoring applications (such as AV tools), along with context information associated with that event, and perform the syntactic-to-semantic translation for the application. The application's only responsibility would then be interpreting and processing the high-level events that it receives.

5.9 Summary

In this chapter we proposed, evaluated and discussed a novel technique for secure active monitoring of virtual machines. This technique is based on monitoring kernel data modifications as opposed to hooking code execution, with the goal of reducing the risk of circumvention. We proposed DARP, an active monitoring infrastructure based on this technique. DARP monitors modifications to certain kernel object types requested by a monitoring application and is able to automatically reconstruct the syntactic context of each write operation without the need for in-guest monitoring components. This information is sent to the application who can use it to infer high-level guest OS events associated with those objects.

We implemented DARP using Hyper-V and Windows Server 2008 R2, running a Windows 7 guest OS. We also implemented a prototype application for monitoring file open events inside the guest. Our performance evaluation shows a 48% overhead when measuring the guest VM boot time. Our mediation evaluation confirmed that DARP was able to intercept all the file openings that happened during the VM's boot process.

CHAPTER VI

CONCLUSIONS

6.1 Summary

Traditional security monitoring applications, such as anti-virus tools, suffer from isolation and mediation problems. The first may result in the application or its trusted computing base being compromised by malware. The second may impact the application's visibility of the monitored system's state and compromise its ability to identify security violations. As a result, the application's effectiveness and therefore the security of the monitored system may be endangered.

Full-system virtualization technology has been proposed as a solution to these problems through a technique known as Virtual Machine Introspection. VMI leverages the isolation and mediation capabilities of virtualization by relocating the security application to a separate VM and using the hypervisor's low-level view of guest VM state and events. This low-level view creates, however, a challenging semantic gap problem, forcing the application to make use of semantic view re-creation techniques to reconstruct a meaningful, high-level view of the monitored system.

This dissertation has proposed a number of novel semantic view re-creation techniques for passive and active monitoring, which can be used by introspection-based monitoring applications. These techniques address several limitations of existing techniques. Our contributions are summarized as follows.

First, we proposed a novel technique for syntactically re-creating a view of the guest VM's kernel heap state that can be used to perform certain types of integrity checks. It applies a combination of static code analysis of the kernel's source code and dynamic memory analysis on its memory image, and is able to reconstruct a map of

the guest OS's dynamic kernel objects with near complete coverage and accuracy. We created an infrastructure, called KOP, that implements this technique. KOP achieves a level of coverage and accuracy of kernel object that far surpasses those of previous solutions. *Our key contribution over previous work is the accuracy and completeness of our syntactic analysis, which translates into stronger monitoring capabilities for integrity checking applications.*

Second, we proposed a novel technique for secure and robust extraction of semantically meaningful information from a guest OS by directly leveraging its kernel code. Using a combination of virtual CPU introspection and dynamic code shepherding, our implemented infrastructure, SYRINGE, enables monitoring applications residing out-of-guest to invoke and shepherd in-guest functions. This technique relieves the application from having to know details about the guest's internal semantics, instead relying on the guest's own API functions to retrieve the information. *Our key contribution over previous work is the ability to overcome the semantic gap between the monitoring application and the guest OS by relying on the guest's own code, resulting in better robustness for out-of-VM passive monitoring applications.*

Third and last, we proposed a novel technique for bridging the semantic gap in active monitoring of VMs in a manner that is completely out-of-VM and cannot be easily circumvented. Our implemented infrastructure, DARP, leverages virtualization's control of guest memory to intercept kernel object modifications rather than code execution. By doing so, DARP eliminates the need for in-guest components and their shortcomings, significantly reducing the risk of circumvention. DARP presents a view of object events at the syntactic level to monitoring applications, which can use them to infer high-level events. *Our key contribution over previous work is the ability to monitor high-level operating system events without having to rely on any in-guest components and with strong guarantees against the circumvention of our hooks.*

6.2 Opportunities for Future Work

There remains significant room for improvement on the techniques proposed in this thesis. More importantly, the emergence of new computing paradigms such as cloud and mobile computing introduces new requirements and constraints that create entirely new challenges within the scope of our work.

6.2.1 Monitor-aware Operating Systems

Despite several modern OSes being virtualization-aware, for instance, through the use of para-virtualization and enlightenments, none of them were designed or implemented with the expectation that they would be inspected by an out-of-VM monitoring application. This fact is the underlying cause of many of the functionality and performance obstacles faced in our work. A very promising direction would involve creating or retrofitting modern OSes with the awareness of external monitoring, making the necessary design and implementation adjustments to reach this goal. In our DARP work, we did this partially by modifying the OS allocator to place all monitored object types in special memory pages, and achieving a substantial performance boost as a result. Further changes, however, can bring even more benefits.

6.2.2 Scalable Monitoring

Scalability has not played a significant role in our work, which has focused on VM monitoring in a dual-VM setting. With the advent of cloud computing, however, scalability becomes a primary concern. Modern clouds deploy tens or hundreds of thousands of VMs spread along server farms with thousands of physical machines, and all in principle need to be monitored. The techniques discussed in this dissertation cannot be applied unaltered to each individual VM in such an environment, or otherwise the bottlenecks created would create chaos. Research exploring the adaptation of our techniques to achieve scalability by leveraging distinct aspects of the cloud, such as redundancy, elasticity and distribution, could be very promising.

6.2.3 Cloud Security

Cloud computing introduces new trust relationships that create new challenges with regard to the confidentiality, integrity and availability of computation. The mutually distrustful relationship between the cloud provider and its clients, and between distinct clients, adds a new dimension to virtualization security, especially VM monitoring. For example, a cloud provider cannot be allowed to snoop into a client's data inside a VM. At the same time, however, we do want a monitoring infrastructure to be in place so that the client himself, or a third-party authorized by him, can monitor the security of his VMs. Changes may need to be made to classic introspection techniques to take this scenario into account. Also interesting is the related area of VM integrity and resource attestation, i.e., allowing costumers to verify in a trustworthy manner that a certain software stack is being used, and being given the agreed-upon amount of resources by the cloud provider.

6.2.4 Lightweight Isolation

Full-system virtualization has been widely adopted by data centers and clouds due to its resource consolidation, throughput and compatibility benefits. In the realm of smartphones, tablets and desktops, however, these requirements come second to others such as usability, responsiveness and resource consumption. In this case, a full-fledged hypervisor may not be the ideal choice as an isolation mechanism. An interesting research direction involves adapting the work presented here to frameworks that provide more lightweight types of isolation, such as sandboxing, software fault isolation and tiny hypervisors.

Tiny hypervisors support only a single VM (i.e., the host system itself) and treat most events (such as I/O) as pass-through, resulting in a very small performance impact. Security applications of such hypervisors include the protection of certain

security-sensitive OS data, and the creation of a trusted path that a user can leverage to input secure sensitive information, like passwords, in face of a compromised OS [112]. The transparent, low-impact nature of such tiny hypervisors, combined with their potential security benefits, would make them an interesting addition to desktop and mobile operating systems.

6.3 Closing Remarks

This thesis addressed important research problems in the area of systems security monitoring. Specifically, first we proposed, implemented and evaluated novel infrastructural techniques that improve the current state-of-the-art on semantic view re-creation for passive and active monitoring of virtual machines. We dramatically improved the coverage and accuracy of syntactic kernel heap state reconstruction, improving the effectiveness of integrity checking applications. Second, we enabled security applications to extract semantically meaningful information from a monitored OS in a secure and robust manner, by leveraging the OS's own code. Third, we proposed a novel, general technique for semantic view re-creation in active monitoring of guest OSes that does not make monitoring applications prone to the same evasion problems as with previous systems. These three contributions, together, constitute an advance in the area of systems security monitoring. Open problems in the area involve re-structuring modern OSes to better support external monitoring, as well as the scalability, security and performance challenges presented by new computing paradigms, such as cloud and mobile computing.

REFERENCES

[1] Abadi, M., Budiu, M., and Ligatti, Ú. E. J., "Control-Flow Integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*, 2005.

[2] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M., "Mach: A new kernel foundation for unix development," in *Proceedings of the Summer 1986 USENIX Conference*, 1986.

[3] Adams, K. and Agesen, O., "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th International Conference on Architectural support for Programming Languages and Operating Systems*, 2006.

[4] Agesen, O., Garthwaite, A., Sheldon, J., and Subrahmanyam, P., "The evolution of an x86 virtual machine monitor," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 4, 2010.

[5] Akritidis, P., Cadar, C., Raiciu, C., Costa, M., and Castro, M., "Preventing memory error exploits with wit," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[6] Akritidis, P., Costa, M., Castro, M., and Hand, S., "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *Proceedings of the 18th USENIX Security Symposium*, 2009.

[7] Andersen, L. O., *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[8] Arbaugh, W. A., Farber, D. J., and Smith, J. M., "A secure and reliable bootstrap architecture," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997.

[9] Avots, D., Dalton, M., Livshits, B., and Lam, M. S., "Improving software security with a c pointer analysis," in *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[10] Azab, A. M., Ning, P., Wang, Z., Jiang, X., Zhang, X., and Skalsky, N. C., "Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[11] Baliga, A., Ganapathy, V., and Iftode, L., "Automatic inference and enforcement of kernel data structure invariants," in *Proceedings of the 24th Annual Computer Security Applications Conference*, 2008.

[12] 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," in *Proceedings of the 19th ACM symposium on Operating systems principles*, 2003.

[13] Bhatia, N., "Performance Evaluation of Intel EPT Hardware Assist," tech. rep., VMware Inc., 2009.

[14] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z., "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[15] Carbone, M., Conover, M., Montague, B., and Lee, W., "Secure and Robust Monitoring of Virtual Machines through Guest-Assisted Introspection." To appear in 15th International Symposium on Research on Attacks, Intrusions and Defenses, 2012.

[16] Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., and Jiang, X., "Mapping Kernel Objects to Enable Systematic Integrity Checking," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[17] Carbone, M., Zamboni, D., and Lee, W., "Taming virtualization," *IEEE Security & Privacy Magazine*, vol. 6, no. 1, 2008.

[18] Castro, M., Costa, M., and Harris, T., "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[19] Castro, M., Costa, M., Martin, J.-P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., and Black, R., "Fast Byte-Granularity Software Fault Isolation," in *Proceedings of the 22nd symposium on Operating systems principles*, 2009.

[20] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M., "Return-oriented programming without returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[21] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K., "Non-control-data attacks are realistic threats," in *Proceedings of the USENIX Security Symposium*, 2005.

[22] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. K., "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Proceedings of 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[23] Chiueh, T., Conover, M., Lu, M., and Montague, B., "Stealthy deployment and execution of in-guest kernel agents," in *Blackhat Technical Security Conference*, 2009.

[24] Costa, M., Castro, M., Zhou, L., Zhang, L., and Peinado, M., "Bouncer: Securing software by blocking bad input," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

[25] Cozzie, A., Stratton, F., Xue, H., and King, S. T., "Digging for data structures," in *8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[26] Das, M., "Unification-based pointer analysis with directional assignments," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 2000.

[27] David B., "The threats of the Age of cyber-warfare: Eugene Kaspersky on cybercrime." `http://privacy-pc.com/articles/the-threats-of-the-age-of-cyber-warfare-eugene-kaspersky-on-cybercrime.html`, 2012 (accessed July 2012).

[28] Department of Defense, "Trusted Computer System Evaluation Criteria," December 1985.

[29] Dinaburg, A., Royal, P., Sharif, M., and Lee, W., "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[30] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W., "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, 2011.

[31] Dolan-Gavitt, B., Srivastava, A., Traynor, P., and Giffin, J., "Robust Signatures for Kernel Data Structures," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[32] Erlingsson, U., *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2004.

[33] Erlingsson, Ú., Abadi, M., and Vrable, M., "Xfi: Software guards for system address spaces," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[34] Evans, D., "The Internet of Things: How the Next Evolution of the Internet Is Changing Everything." http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf, April 2011 (accessed July 2012).

[35] Fu, Y. and Lin, Z., "Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection," in *Proceedings of 2012 IEEE Symposium on Security and Privacy*, 2012.

[36] Garfinkel, T., "Traps and pitfalls: Practical problems in system call interposition based security tools," in *Proceedings of the 10th Annual Symposium on Network and Distributed Systems Security*, 2003.

[37] Garfinkel, T., Pfaff, B., and Rosenblum, M., "Ostia: A delegating architecture for secure system call interposition," in *Proceedings of the 11th Symposium on Network and Distributed System Security*, 2004.

[38] Garfinkel, T. and Rosenblum, M., "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of the 10th Annual Symposium on Network and Distributed Systems Security*, 2003.

[39] Gold, B., Linde, R., Peller, R. J., Schaefer, M., Scheid, J., and Ward, P. D., "A security retrofit for VM/370," in *AFIPS National Computing Conference*, 1979.

[40] Hand, S., Warfield, A., Fraser, K., Kotsovinos, E., and Magenheimer, D., "Are virtual machine monitors microkernels done right?," in *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, 2005.

[41] Hardekopf, B. and Lin, C., "The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[42] Heintze, N. and Tardieu, O., "Ultra-fast aliasing analysis using cla: A million lines of c code in a second," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001.

[43] Heiser, G., Uhlig, V., and LeVasseur, J., "Are virtual-machine monitors microkernels done right?," *ACM SIGOPS Operating Systems Review*, vol. 40, Jan. 2006.

[44] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S., "Modular system programming in minix 3," *;LOGIN*, April 2006.

[45] Hofmeyr, S. A., Forrest, S., and Somayaji, A., "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, 1998.

[46] Hoglund, G. and Butler, J., *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[47] Hund, R., Holz, T., and Freiling, F., "Return-Oriented Rootkits: Bypassing Code Integrity Protection Mechanisms," in *Proceedings of the 18th USENIX Security Symposium*, 2009.

[48] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2012.

[49] Jiang, X. and Wang, X., ""Out-of-the-box" Monitoring of VM-based High-Interaction Honeypots," in *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection*, 2007.

[50] Jiang, X., Wang, X., and Xu, D., "Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and Communications Security*, 2007.

[51] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Antfarm: Tracking processes in a virtual machine environment," in *Proceedings of the 2006 USENIX Annual Technical Conference*, 2006.

[52] Joshi, A., King, S. T., Dunlap, G. W., and Chen, P. M., "Detecting past and present intrusions through vulnerability-specific predicates," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[53] Kaspersky Lab, "Kaspersky Anti-Virus." `http://www.kaspersky.com`, 2012 (accessed July 2012).

[54] Kasslin, K., "Windows Kernel Malware." `http://www.cse.tkk.fi/fi/opinnot/T-110.6220/2010_Spring_Malware_Analysis_and_Antivirus_Tchnologies/luennot-files/T-110.6220_Kernel.pdf`, 2011 (accessed July 2012).

[55] Kiriansky, V., Bruening, D., and Amarasinghe, S., "Secure Execution Via Program Shepherding," in *Proceedings of the 11th USENIX Security Symposium*, 2002.

[56] Liedtke, J., "On -kernel construction," in *15th ACM Symposium on Operating System Principles*, 1995.

[57] Lin, Z., Rhee, J., Zhang, X., Xu, D., and Jiang, X., "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[58] Lin, Z., Zhang, X., and Xu, D., "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, 2010.

[59] Litty, L., Lagar-Cavilla, H. A., and Lie, D., "Hypervisor support for identifying covertly executing binaries," in *Proceedings of the 17th USENIX Security Symposium*, 2008.

[60] Mao, Y., Chen, H., Zhou, D., Wang, X., Zeldovich, N., and Kaashoek, M. F., "Software fault isolation with api integrity and multi-principal modules," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[61] McAfee, Inc., "McAfee Threats Report: Fourth Quarter 2011." `http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2011.pdf`, 2011 (accessed July 2012).

[62] McAfee, Inc., "McAfee VirusScan." `http://www.mcafee.com`, 2012 (accessed July 2012).

[63] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Isozaki, H., "Flicker: An execution infrastructure for tcb minimization," in *Proceedings of the ACM European Conference on Computer Systems*, 2008.

[64] McDougall, R., "Virtualizing Oracle Databases with VMware." `http://www.slideshare.net/rjmcdougall/virtualizing-oracle-databases-with-vmware`, 2012 (accessed July 2012).

[65] Microsoft Corporation, "Microsoft Security Essentials." `http://windows.microsoft.com/mse`.

[66] Microsoft Corporation, "Debugger Engine and Extensions API." `http://http://msdn.microsoft.com/en-us/library/ff540525.aspx`, 2012 (accessed July 2012).

[67] Microsoft Corporation, "Overview of memory dump file options for Windows Server 2003, Windows XP, and Windows 2000." `http://support.microsoft.com/kb/254649`, 2012 (accessed July 2012).

[68] Microsoft Corporation, "Phoenix Compiler Framework." `http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx`, 2012 (accessed July 2012).

[69] Molina, J. and Arbaugh, W. A., "Using independent auditors as intrusion detection systems," in *Proceedings of the 4th International Conference on Information and Communications Security*, 2002.

[70] Moser, A., Kruegel, C., and Kirda, E., "Exploring multiple execution paths for malware analysis," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007.

[71] Neiger, G., Santoni, A., Leung, F., Rodgers, D., and Uhlig, R., "Intel(r) virtualization technology: Hardware support for efficient processor virtualization," *Intel Technology Journal*, vol. 10, no. 3, 2006.

[72] Panda Security, "PandaLabs Quarterly Report." `http://press.pandasecurity.com/wp-content/uploads/2012/05/Quarterly-Report-PandaLabs-January-March-2012.pdf`, 2012 (accessed July 2012).

[73] Payne, B. D., Carbone, M., and Lee, W., "Secure and flexible monitoring of virtual machines," in *Proceedings of the 23rd Annual Computer Security Applications Conference*, 2007.

[74] Payne, B. D., Carbone, M., Sharif, M., and Lee, W., "Lares: An architecture for secure active monitoring using virtualization," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[75] Pearce, D. J., Kelly, P. H. J., and Hankin, C., "Efficient field-sensitive pointer analysis for c," in *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2004.

[76] Petroni Jr., N. L., Fraser, T., Molina, J., and Arbaugh, W. A., "Copilot: a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[77] Petroni Jr., N. L., Fraser, T., Walters, A., and Arbaugh, W. A., "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *Proceedings of the 15th USENIX Security Symposium*, 2006.

[78] Petroni Jr., N. L. and Hicks, M., "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications security*, 2007.

[79] Popek, G. J. and Goldberg, R. P., "Formal Requirements for Virtualizable Third Generation Architectures," *Communications of the ACM*, vol. 17, no. 7, 1974.

[80] Provos, N., "Improving host security with system call policies," in *Proceedings of the 12th USENIX Security Symposium*, 2003.

[81] Riley, R., Jiang, X., and Xu, D., "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.

[82] Riley, R., Jiang, X., and Xu, D., "Multi-Aspect Profiling of Kernel Rootkit Behavior," in *Proceedings of the 4th European Conference on Computer Systems*, 2009.

[83] Rootkit.com, "Rootkit.com." `http://www.rootkit.com`, (accessed July 2012).

[84] Rushby, J., "Design and verification of secure systems," in *Proceedings of the 8th ACM Symposium on Operating System Principles*, 1981.

[85] Sailer, R., Zhang, X., Jaeger, T., and van Doorn, L., "Design and implementation of a tcg-based integrity measurement architecture," in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[86] Schaefer, M. and Gold, B., "Program confinement in KVM/370," in *Proceedings of the Annual ACM Conference*, 1977.

[87] Seshadri, A., Luk, M., Qu, N., and Perrig, A., "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

[88] Seshadri, A., Perrig, A., Luk, M., van Doorn, L., Shi, E., and Khosla, P., "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," in *Proceedings of the 20th ACM Symposium on Operating System Principles*, 2005.

[89] Shacham, H., "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and Communications Security*, 2007.

[90] Sharif, M., Lee, W., Cui, W., and Lanzi, A., "Secure In-VM Monitoring Using Hardware Virtualization," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[91] Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y., and Kato, K., "BitVisor: A thin hypervisor for enforcing I/O device security," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2009.

[92] Slowinska, A., Stancescu, T., and Bos, H., "Howard: a dynamic excavator for reverse engineering data structures," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[93] Somayaji, A. and Forrest, S., "Automated response using system-call delays," in *Proceedings of the 9th USENIX Security Symposium*, 2000.

[94] S.R., J. A., Gasser, M., and Schell, R., "Security kernel design and implementation: An introduction," *IEEE Computer*, vol. 16, no. 7, 1983.

[95] Srinivasan, D., Wang, Z., Jiang, X., and Xu, D., "Process out-grafting: An efficient "out-of-vm" approach for fine-grained process execution monitoring," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[96] Srivastava, A., Erete, I., and Giffin, J., "Kernel data integrity protection via memory access control," tech. rep., School of Computer Science, Georgia Institute of Technology, 2009.

[97] Srivastava, A. and Giffin, J., "Tamper-resistant, application-aware blocking of malicious network connections," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.

[98] Srivastava, A., Lanzi, A., Giffin, J., and Balzarotti, D., "Operating system interface obfuscation and the revealing of hidden operations," in *Detection of Intrusions and Malware and Vulnerability Assessment*, 2011.

[99] Steensgaard, B., "Points-to analysis in almost linear time," in *Symposium on Principles of Programming Languages*, 1996.

[100] Swift, M. M., Bershad, B. N., and Levy, H. M., "Improving the reliability of commodity operating systems," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[101] Symantec Corporation, "Norton AntiVirus." `http://us.norton.com/antivirus`, (accessed July 2012).

[102] Tereshkin, A., "Rootkits: Attacking personal firewalls." `http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Tereshkin.pdf`, 2006 (accessed July 2012).

[103] VMware Inc., "VMware ESXi and ESX Info Center." `http://www.vmware.com/products/vsphere/esxi-and-esx/index.html`, December 2010 (accessed July 2012).

[104] VMware Inc., "VMware VMsafe partner program." `http://www.vmware.com/technical-resources/security/vmsafe.html`, December 2010 (accessed July 2012).

[105] Wahbe, R., Lucco, S., Anderson, T., and Graham, S., "Efficient software-based fault isolation," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, 1993.

[106] Wang, J., Stavrou, A., and Ghos, A., "Hypercheck: A hardware-assisted integrity monitor," in *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection*, 2010.

[107] Wang, Z., Jiang, X., Cui, W., and Ning, P., "Countering kernel rootkits with lightweight hook protection," in *16th ACM Conference on Computer and Communications Security*, 2009.

[108] Williams, D., Reynolds, P., Walsh, K., Sirer, E. G., and Schneider, F. B., "Device Driver Safety Through a Reference Validation Mechanism," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[109] Wilson, R. P. and Lam, M. S., "Efficient context-sensitive pointer analysis for c programs," in *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, 1995.

[110] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E., "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[111] Zhang, X., van Doorn, L., Jaeger, T., Perez, R., and Sailer, R., "Secure coprocessor-based intrusion detection," in *Proceedings of the Tenth ACM SIGOPS European Workshop*, 2002.

[112] Zhou, Z., Gligor, V. D., Newsome, J., and McCune, J. M., "Building verifiable trusted path on commodity x86 computers," in *Proceedings of 2012 IEEE Symposium on Security and Privacy*, 2012.