Operating System Interface Obfuscation and the Revealing of Hidden Operations

Abhinav Srivastava[1] Andrea Lanzi[1,2] Jonathon Giffin[1]

[1]School of Computer Science, Georgia Institute of Technology
[2]Dipartimento di Informatica e Comunicazione, Università degli Studi
{abhinav,giffin}@cc.gatech.edu, andrew@security.dico.unimi.it

Abstract

Many software security solutions—including malware analyzers, information flow tracking systems, auditing utilities, and host-based intrusion detectors—rely on knowledge of standard system call interfaces to reason about process execution behavior. In this work, we first obfuscate the Windows and Linux system call interfaces to degrade the effectiveness of these tools. Our attack, called *Illusion*, invokes privileged kernel operations in the kernel at the request of user-level processes without requiring those processes to call the actual system calls corresponding to the operations. The Illusion interface hides system operations from user-, kernel-, and hypervisor-level monitors mediating the conventional system-call interface. Illusion alters neither static kernel code nor read-only dispatch tables, remaining elusive from tools protecting kernel memory. We then consider the problem of Illusion attacks and augment system call data with kernel-level execution information to expose the hidden kernel operations. We present a Xen-based monitoring system, *Sherlock*, that adds kernel execution *watchpoints* to the stream of system call events. Sherlock automatically adapts its sensitivity based on security requirements to remain performant on desktop systems.

1 Introduction

Honeypots and other utilities designed to audit, understand, classify, and detect malware and software attacks often monitor process' behavior at the system-call interface as part of their approach. Past research has developed a widespread collection of system-call based systems operating at user or kernel level [11, 12, 17, 19, 25, 35, 42] and at hypervisor level [20, 36]. In fact, system call monitoring has proved useful enough to see commercialization [8, 31]. Employing reference monitors at the system-call interface makes intuitive sense: absent flaws in the operating system (OS) kernel, it a non-bypassable interface, so malicious code intending to unsafely alter the system will reveal its behavior through the series of system calls that it invokes.

Current malware increasingly [24] makes use of kernel modules or drivers that help the user-level process perform malicious activities by hiding the process' side effects. For example, the rootkits *adore* and *knark* hide processes, network connections, and malicious files by illegitimately redirecting interrupt or system call handling into their kernel modules. Redirection can alter the semantic meaning of a system call—a problem for any system that monitors system calls to understand the behavior of malware. Jiang and Wang [20] addressed this class of attack by making the interrupt descriptor table (IDT), system service descriptor table (SSDT), and system-call handler routines write protected. Systems like that of Jiang and Wang assume that protections against illegitimate alteration of these objects will force malicious software to always follow the standard system-call interface when requesting service from the kernel.

Unfortunately, this assumption does not hold true. In the first of two principal contributions of this paper, we show how malicious code can obfuscate the Windows or Linux system-call interface using only *legitimate functionality* commonly used by kernel modules and drivers. We present a novel attack, *Illusion*, that allows malicious processes to invoke privileged kernel operations without requiring the malware to call the actual system calls corresponding to those operations. In contrast to prior attacks [6] of the sort considered by Jiang and Wang, Illusion alters neither static kernel code nor read-only dispatch tables such as the IAT or SSDT. It only extends the kernel memory by loading a kernel driver, which is a benign operation. During the execution of malware augmented with the Illusion attack, an existing system-call analyzer sees a series of system calls different than those actually executed by the malware.

Kernel malware augmented with Illusion becomes hidden from many common signature- and anomaly-based detectors. Kernel integrity checkers [34,38,52] verify only static kernel code, static data, and dynamic linked data structures such as the process linked list. Illusion does not make any changes to these structures, and hence remains elusive from these detectors. System-call based anomaly detectors [11, 12, 17, 19, 42] monitor sequences of executed system calls to notice unusual patterns. Application-specific monitors compare system-call sequences generated by an application against a normal model of its legitimate behavior. Our Illusion attack augments existing malicious software for which anomaly detectors have no model and provide no monitoring protection. Generic, system-wide monitors that observe all applications' system-call sequences may be evaded using standard mimicry attack constructions [47]. Hence, it is difficult to detect the Illusion attack with system-call based anomaly detectors.

The Illusion attack is possible because current analyzers depend on the standard system-call interface to represent the underlying changes that a process is making to the system. Importantly, they do not take into account the actual execution of the requested system call inside the kernel. The Illusion attack makes use of a malicious kernel module, so the defenses must drop to a lower hypervisor level.

In our second principal contribution, we show how defensive systems can both detect the presence of Illusion attacks and recover the original system call that the malware would have invoked had the Illusion attack been absent. We have developed a prototype system called *Sherlock* to demonstrate the feasibility of our defenses. Sherlock uses Xen and a fully-virtualized Linux guest operating system; defenses for Windows would operate similarly to those we describe for Linux given knowledge of the internal Windows kernel code design. Sherlock tracks the kernel's execution behavior with the help of *watchpoints* inserted along kernel code paths that execute during the service of a system call. Mismatches between the system call invoked and the service routine's execution indicate that an Illusion attack has altered the internal semantics of the system-call interface.

Sherlock is an adaptive defensive system that tunes its own behavior to optimize performance. Sherlock itself does not provide detection capability for attacks other than the Illusion attack, but is designed to augment existing system call analyzers with information about hidden kernel operations. When watchpoints match the expected execution of the requested system call, then traditional system call monitoring indeed observes the correct events. When Sherlock detects the Illusion attack, however, a system call monitor records a faulty series of system calls. Sherlock will then assist the monitor by switching into a deep inspection mode that uses the execution behavior of the kernel to reconstruct the actual system call operations executed by the malware. As soon as the malicious invocation of the system call completes, Sherlock switches back into its high performance mode. During benign execution with watchpoints enabled, Sherlock imposes overheads of 10% on disk-bound applications, 0%–3% on network-bound software, and less than 1% on CPU-bound applications.

In summary, we feel that this paper makes following contributions:

- The Illusion attack: a demonstration that malicious software can obfuscate a commodity kernel's system-call interface using only the legitimate functionality used by benign modules and drivers. The malicious software controls the obfuscation, so every instance of malware using the Illusion attack can create a different system call interface.

- A hypervisor-based kernel execution monitoring system using watchpoints and models of system call handler execution behavior to reconstruct privileged operations hidden by an Illusion attack.

- Adaptive kernel execution monitoring that balances security and performance by deeply inspecting kernel state only when the kernel executes hidden operations.

2 Related Work

If defenders know *a priori* about offensive technologies used by attackers, then they can develop appropriate remedies. To this end, researchers have performed various attack studies to better understand how attackers can evade host-based security tools. Mimicry attacks [7,47,48] against application-level intrusion detection systems [19] escape detection by making malicious activity appear normal. Baliga et al. [2] proposed a new class of kernel-level stealth attacks that cannot be detected by current monitoring approaches. These studies showed that security software often relies upon assumptions that may not be known and that attackers can escape detection with attacks that violate the assumptions.

We apply this style of attack reasoning to system-call monitors because of their increasing use in the design of security systems. With the increasing popularity, it has become important to know the assumptions on which they are founded. Like previous literature, we also assume the perspective of an attacker who is trying to undetectably execute malicious software. By taking this view, we hope to help defenders understand and defend against the threat of system call API obfuscation.

We use a hypervisor to observe the behavior of a kernel executing within a virtual machine. Virtual machines are becoming widely used for exfiltration detection [5,45], rootkit prevention [50], file protection [53], hidden process detection [23], and in other applications [10,14–16,21,27,41,46,49]. While previous research mainly focused on monitoring memory and disks for signs of an intrusion, Sherlock monitors a kernel's execution to discover hidden operations happening inside the kernel. Jones et al. [22] developed a mechanism for virtualized environments to track kernel operations related to specific process actions. With their system, process creation, context-switch, and destruction can be observed from a hypervisor. Sherlock allows for arbitrary kernel behavior tracking based upon watchpoints inserted into relevant kernel code execution paths.

Insertion of additional state exposure events into existing code has been a recurring idea in past research [30]. Payne et al. [37] developed a framework that inserted arbitrary hooks into an untrusted machine and then protected those hooks using a hypervisor. Xu et al. [51] inserted waypoints into applications; execution of a waypoint adds or subtracts privilege from an application so that it better adheres to access restrictions based on least privilege. Giffin et al. [18] used null system calls to improve the efficiency of a user-level system-call monitor. In a similar way, Windows filter drivers allow hooking and interception of low-level kernel execution events [33]. Sherlock uses its watchpoints to gain a view of system call handler execution behavior, and it expects that these watchpoints will be protected via standard non-writable permissions on memory pages containing kernel code.

Sophisticated attackers may attempt to bypass the portions of kernel code containing watchpoints in a by altering execution flow in a manner similar to that used to execute application-level mimicry attacks [26]. These attacks abuse indirect control flows and could be prevented using techniques such as control flow integrity [1,39,43] that guard indirect calls and jumps. Sherlock is not designed to provide such protections itself.

3 System Call Obfuscation via Illusion Attacks

We first adopt an offensive role and target system-call monitoring systems such as those used to analyze malware, track information flows among processes, audit suspicious execution, and detect attacks against software. Our Illusion attack builds a new system call mechanism using legitimate means that cannot be blocked without disabling a large class of benign kernel drivers.

3.1 Abilities of the Attacker

We use a threat model that reflects current, widespread attacks and state-of-the-art monitoring systems. An attacker can install a kernel driver or module, perhaps via a kernel-level vulnerability or a naïve user, which bypasses [4, 44] preventive techniques such as driver signing [32] as well as various forms of driver verification [50]. However, we assume that the attacker cannot modify existing kernel code. We expect that any vulnerability exploited by the attacker is not so severe as to allow processes to completely bypass the system-call interface when requesting kernel service. The loaded module can access and alter mutable data in the kernel, but it cannot alter read-only data such as the interrupt address table or system service dispatch table. This threat model is sufficient to allow the design of our Illusion attack, though we will revisit this model with additional expectations when presenting the Sherlock defensive system.

3.2 Overview

Illusion attacks obfuscate the sequence of system calls generated by a malicious process requesting service from the kernel. The malicious process must still receive the desired service, else the attack would fail to have an effect beyond CPU resource consumption. The Illusion attack must enable the malicious process to execute the same system call requests that it would make in the absence of the Illusion attack, but the particular calls should not be revealed to a system call monitor.

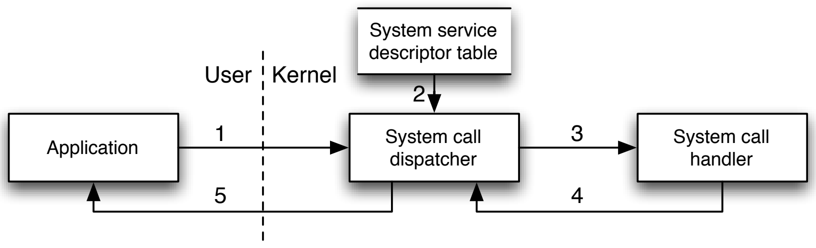

Figure 1: Normal System-call Execution Path (NSEP)

An obvious attack (that will fail in our threat model) directly alters system call dispatch within the kernel. Figure 1 depicts system call dispatch with the following steps. (1) When an application invokes a system call, it issues a software interrupt or system call request. The CPU switches from user to kernel mode and begins executing the system call dispatch function. (2) The dispatch function reads the system call number, which indicates the type of system operation requested, and uses it as an index into a table of function pointers called the system service descriptor table (SSDT). Each entry in the SSDT points at the system call handler function for a particular system call. (3) After reading the pointer value from the SSDT, the dispatch function transfers execution to that target system call handler in the kernel. (4) When the handler completes, it returns its results to the dispatch function, (5) which then copies them back into the application's address space and returns control to the application. We define this flow as the *normal system-call execution path*, or $NSEP$. An attack that alters system call dispatch—steps 2 and 3—will not succeed, as our threat model does not allow an attacker to illegitimately alter the entries in the SSDT without detection.

However, a number of system calls allow *legitimate* dispatch into code contained in a kernel module or driver. Consider `ioctl`: this system call takes an arbitrary, uninterpreted memory buffer as an argument and passes that argument to a function in a kernel module that has registered itself as the handler for a special file. Benign kernel modules legitimately register handler functions for such files; a malicious module performing the same registration exhibits no behaviors different from the benign code. However, a call to `ioctl` will be directed into the malicious module's code together with the buffer passed to `ioctl` as an argument.

This argument contains the original system call request in a serialized form. The malware will marshal the original system call into a memory buffer that it subsequently passes to the `ioctl` system call. The handling function within a malicious kernel module will unmarshal the buffer to identify the actual operation requested by the malware, and will then directly call the system call handler for that operation. With this interface illusion in place, the kernel still executes the same operations that the malware instance would have executed without the obfuscation. However, system call monitoring utilities would observe a sequence of `ioctl` requests and would not realize that malicious operations had occurred.

This attack has properties appealing to an attacker:

- The attacker can select from a collection of system calls offering legitimate dispatch to module code. For example, `ioctl` passes arbitrary data to a kernel module by design. The attack can also use more exotic mechanisms, such as shared memory, netlink sockets, or a character device. A particular instance of an Illusion attack may use any or all of these operations in combination as replacements for existing system calls.

- The attacker controls marshaling and unmarshaling operations. Although a system call monitor knowledgeable of a system call marshaling format could recover the original system call sequence by unmarshaling arguments, the attacker can continually change the marshaling format to prevent the monitor from gaining this knowledge. This continual change is similar to the effect of polymorphic engines or packers upon malicious code.

- The attacker only needs to obscure system calls performing operations that would reveal the malicious intent of their malware. Other system calls executed by the malware, including dummy calls that could be inserted for confusion or mimicry, need not be obfuscated. This reduces the rate of calls to the Illusion attack's dispatching operation and increases the apparent normalcy of the system call sequence.

- The attacker can automatically convert a malware instance invoking traditional system calls into a malware instance using an Illusion attack. A straightforward source-to-source transformation can simply replace a malware's system calls with a marshaling operation and a call to an Illusion dispatching system call such as `ioctl`. Similar automated transformation could occur at the binary level at any system call site whose arguments can be recovered via static binary analysis [18].

- The attacker could define new system calls executable through the Illusion attack but having no counterpart in the traditional system-call interface provided by the kernel.

The end result is that while a system-call monitor is able to know that system-calls are occurring, it cannot evaluate the semantic meaning of the operations.

3.3 The Illusion Kernel Module

The Illusion attack creates an *alternative system call execution path* (ASEP) that executes system operations requested by malicious applications. By using an ASEP, malware is able to perform any system operation without raising the actual system call event related to that operation. In order to create an ASEP, we need a kernel module, a cooperating malware instance, and a communications channel between the two. Note that we expect the attacker to successfully load the Illusion kernel module into the victim operating system kernel; this may occur by acquiring a signature for the module, loading the module on a system that does not check signatures, or loading the code via an exploit that circumvents module legitimacy checks. A legitimate signature is the most capable technique, as it evades cutting-edge defenses against kernel-level malicious code [40].

The kernel module is composed of two engines: the protocol engine and the execution engine. The protocol engine unmarshals an argument buffer into a system operation request and interprets the semantics of that request. The execution engine actually executes the requested system call operation. The execution engine can execute the system operations in several different ways, including: (1) a direct call to the corresponding system-call handler, (2) a direct call to any exported kernel function with similar behavior, or (3) execution of code within the kernel module followed by a jump into low-level portions of the kernel's handler, such as device driver code that accesses hardware of the computer.

Figure 2 shows how the ASEP alters a kernel's processing of system call operations. (1) Malware sends an operation request to the protocol engine. (2) The protocol engine obfuscates the request by using the marshaling specification. The marshaling specification describes how the requested operation should be embedded inside the communication call. (3) Once the original request is obfuscated, the malware sends it to the kernel module via the particular communication channel used by this instance of the Illusion attack. (4) The protocol engine at the kernel side receives the request, unmarshals the argument using the specification, and (5) sends the requested operation to the execution engine. (6) The execution engine invokes the appropriate kernel operation, using one of the three execution methods described above. (7) When the function returns, it passes the results to the protocol engine which then returns them to the malware. If needed by the malware, the protocol engine can obfuscate the response, which would then be deobfuscated by the corresponding engine in the user-level malware.

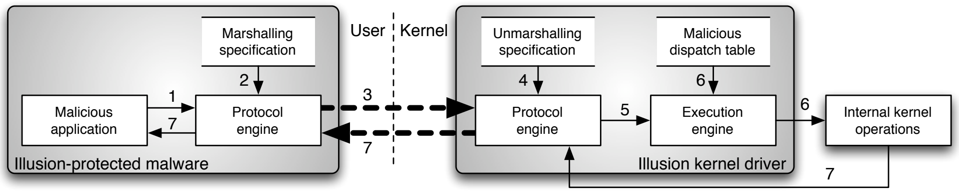

Figure 2: Alternative System-call Execution Path (ASEP)

3.4 Implementation

We have built two different prototypes of the Illusion attack: one for Windows and one for Linux. For validation of the attack's ability to obfuscate system call sequences, we also developed *Blinders*, a Qemu-based system-call tracer for Windows designed using the same principles of Jiang and Wang [20]. We describe details of our Windows implementation, but the Linux attack uses a nearly identical design.

Our Windows Illusion attack uses the `DeviceIoControl` system call as its entry point into the kernel. This function is used to exchange data between a kernel driver and an application through a device. It receives parameters that include a device handle for the device and a memory buffer to pass information to the driver. We used this buffer to pass serialized system operation requests according to the marshaling protocol.

Our Windows kernel driver is composed of two main functions: one each for the protocol engine and the execution engine. The first function interprets the protocol and extracts the information for the requested operation. For the latter function, we implemented all three execution methods described in Section 3.3. For method 1, the kernel driver prepares the stack layout according to the requirements of the system-call handler function. After extracting the handler's function pointer from the system-call dispatch table, it directly calls the pointer. For the second method, the driver directly calls exported kernel functions; for example, a call to the function `ZwCreateFile` performs a file creation operation. The last method is more subtle in that the driver executes some part of the code inside itself and only then jumps into the kernel code midway along the control-flow path of the requested operations. We copied a piece of the function's assembly code from the kernel into the driver and then executed it using the `__asm` directive. We equipped our kernel driver to execute two operations: creation of a file and creation of a registry key.

For our experimental marshaling protocol, the user-space malware constructed a sequence of comma-separated fields. We defined two static fields: the first contained the unique number for a system operation, and the second contained a value that identified the number of parameters for the operation. All following values represented the values of parameters.

We executed a prototype malware instance using our Illusion attack together with Blinders inside Windows XP. We issued a request from the user-space malware component for file and registry key creation. We observed that the system calls issued by the user-space component were identified by Blinders as the incorrect call, as expected. The system-call tracer is only able to see the system call used to communicate with the driver, which was `DeviceIoControl` in this experiment.

We implemented a similar prototype of our Illusion attack for the Linux 2.6 kernel. In this version, the kernel module directly called exported kernel functions to perform requested operations, and it provided unmarshaling for open, read, write, socket, connect, send, and receive requests. The attack used `ioctl` as the hidden system call entry point to the Illusion dispatch functionality. We ran the Linux prototype, executed all operations, and used the standard system call logging tool *strace* to observe the issued system calls. When an experimental malware instance executed any of the hidden calls, *strace* only showed the `ioctl` system call in its log.

3.5 Foundation

The Illusion attack can be understood by reasoning about system call mapping operations.

Definition: Trace Correctness: *Let Σ be the set of the system operations and Σ^* the set of sequences over the alphabet Σ. Let $T = \{t_1, t_2, ..., t_m\} \in \Sigma^*$ be the ordered sequence of system operations actually invoked by an application P, as given by an oracle. Let $V = \{v_1, v_2, ..., v_n\} \in \Sigma^*$ be the ordered sequence of system operations observed by a monitor M observing P through any means possible. The trace V is* correct *iff $m = n$ and $T = V$.*

In a correct trace, there is a one-to-one mapping between a system operation invoked by a process and the operation observed by the monitor. The Illusion attack breaks trace correctness by creating a new one-to-many relation, unknown to M, between the system call invoked and the real system operations executed.

Figure 3 shows two different mappings between a system operation observed by interposing on the normal system call mechanism and the real system operation executed within the kernel. The first mapping is a normal one-to-one relation, where for each system-call invocation there is only one corresponding system operation executed by the kernel. The second mapping, one-to-many, is created by the Illusion attack and is not known to the monitor M. In fact, even if the monitor were aware that system calls were obfuscated through such a relation, it would be unable to reverse the one-to-many relation without additional information to disambiguate among the collection of actual system calls that all map to the same observed call.

3.6 Discussion

Our attack diminishes the ability of system call monitors to understand actual malicious process behavior.

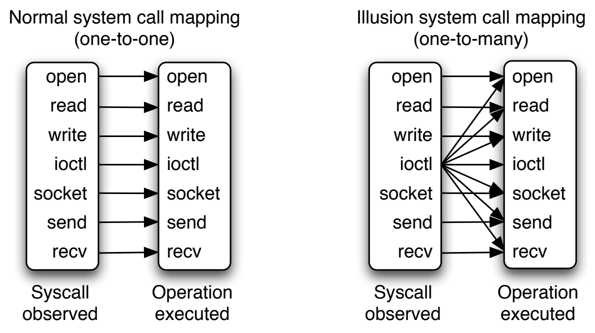

Figure 3: System-calls and System Operations Mapping

Hypervisor-based system-call analyzers: Due to the prevalence of kernel malware, hypervisor-based system-call monitoring tools have been proposed [20, 36]. Although these tools provide tamper-resistance, they still depend on the normal system-call execution path (NSEP) to record system operations. Illusion is able to obfuscate the system call mechanism and will hide the system operations from these tools.

Malware unpackers and analyzers: A similar effect can be seen on system-call based malware unpacking and analysis [9, 28]. Recently, Martignoni et al. [28] proposed an unpacker that monitors the execution of suspected malware and assumes that the code is unpacked when the application issues a dangerous system call. In another work, Martignoni et al. [29] presented a behavioral monitoring system that relies on the system-calls. Once the system sees a malicious system-call, it then tries to collect low-level information corresponding to the call. Since the Illusion attack hides the malicious calls, the behavioral system would never be invoked to collect low-level information.

Anti-virus tools and intrusion detectors: Some anti-virus software and IDS tools [31] use the system-call interception mechanism to detect a malware infection. Such tools modify the system-call dispatch table in order to intercept the system calls. Even in this case, the Illusion attack is able to hide the system call information without raising the sensors set by the security application.

Many common security tools such as kernel memory scanners or system-call based anomaly detectors do not detect Illusion attacks. The Illusion attack does not modify a kernel's static data structures or dispatch tables, which makes it remain stealthy from security tools that monitor these data. Illusion augments malicious software that would not be protected by application-specific anomaly detection systems. Importantly, an attacker can always bolster an unusual system call sequence of Illusion requests with unobfuscated nop system calls to create an undetectable mimicry attack sequence.

4 Sherlock

We developed a hypervisor-based prototype system called *Sherlock* that detects the presence of an Illusion attack and determines the actual system operations executed via the hidden interface. It is designed to achieve the following goals:

- **Secure system:** Sherlock must provide security by exposing hidden operations happening inside the operating system. It uses watchpoints, or state exposure operations within kernel code, to track kernel execution. The watchpoints reveal the hidden behaviors of an Illusion attack's kernel module executing within the kernel.

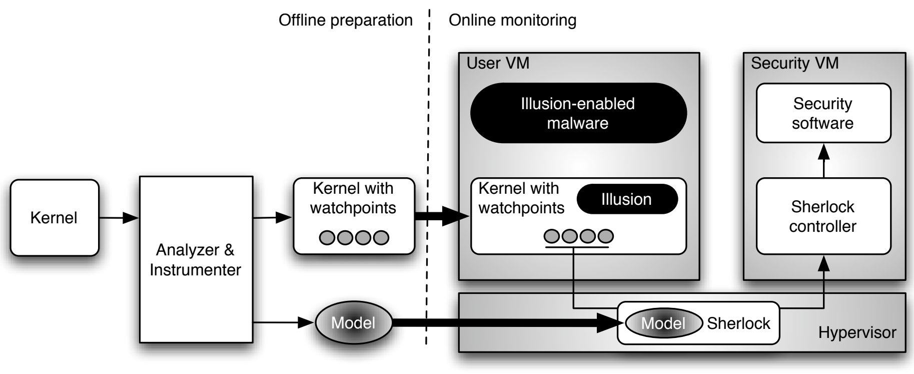

Figure 4: Sherlock's Architecture.

- **Tamper resistant:** Attackers controlling the operating system should not be able to alter Sherlock's execution. It uses a hypervisor-based design to remain isolated from an infected system. It may protect its watchpoints inside the guest OS by making the kernel's static code write-protected to prevent tampering.
- **High performance:** Sherlock should not unreasonably slow the execution of benign software. Its design permits adaptive execution, where performance-costly operations will be invoked only when an Illusion attack may be underway.

Our system-call API obfuscation detection operates in two phases. In the first phase (offline preparation), we analyze the kernel source code and insert watchpoints along kernel code paths that execute during the service of a system-call. We create models of system-call handler execution behavior to detect Illusion attacks. The second phase (online monitoring) divides the system into three components: an untrusted user virtual machine (VM), a trusted security VM, and a modified Xen hypervisor [3] (Figure 4). The user VM runs a guest operating system (OS) instrumented with watchpoints. Each watchpoint notifies the hypervisor-level Sherlock monitor of kernel execution behavior via a `VMCALL` instruction, and Sherlock determines if the watchpoint is expected or suspicious. Suspicious watchpoints are passed to the security VM for deep analysis and as a way to inform traditional system call monitors that an Illusion attack is underway.

4.1 Threat Model

We expand on the threat model previously described in Section 3.1. Sherlock addresses the problem of Illusion attacks. As Sherlock uses virtualization, we assume that an attacker is unable to directly attack the hypervisor or the security VM, an expectation that underlies much research in virtual machine based security. Sherlock relies on kernel code instrumentation, and so we expect that an Illusion attack's kernel module will be required to call into the existing, instrumented code of the kernel at some point in its handling of hidden operations. Our belief is that while a module may carry some fragments of duplicated kernel code containing no watchpoint instrumentation, it cannot predict all possible configurations of hardware and file systems and must eventually make use of the existing code of a victim's kernel. Although the module could read and duplicate the kernel's existing code, W$\oplus$X protections would prevent the module from executing copied code. While these protections do not address the problem of object hiding via direct kernel object manipulation, such attacks are not Illusion attacks and would be best detected with kernel integrity checkers [34, 52].

Operation	*Functions*
Open	sys_open, sys_open, open_namei
Read	sys_read, sys_read, do_sync_read generic_file_aio_read
Write	sys_write, sys_write, do_sync_write, generic_file_aio_write
Socket	sys_socket, sys_socket, _sock_create, inet_create, tcp_v4_init_sock
Connect	sys_connect, sys_connect, net_stream_connect, tcp_connect_init
Send	sys_send, sys_send, __sock_sendmsg, tcp_sendmsg
Receive	sys_recv, sys_recv, __sock_recvmsg, tcp_recvmsg

Table 1: Watchpoint Placement

4.2 Exposing Kernel Execution Behavior

Sherlock monitors the kernel's execution behavior with the help of watchpoints inserted along the kernel code paths. These watchpoints expose kernel-level activities and are implemented using the `VMCALL` instruction. In order to distinguish among watchpoints, each watchpoint passes an unique identifier to Sherlock that helps Sherlock identify the operation being performed. For example, in the case of a *read* operation, a watchpoint on the execution path of the read system-call handler sends a unique watchpoint `do_sync_read` whenever it executes. When the hypervisor component receives the `do_sync_read` watchpoint, it infers that a read operation is being performed.

We chose where to place watchpoints by performing a reachability analysis of kernel code. Our analysis finds all functions reachable from the start of the system-call handler down to the low-level device drivers. This analysis yields a set of functions that may legitimately execute subsequent to the entry of the system call handler. Inserting watchpoints in all these functions would greatly impact the performance of the operating system. We instead find a set of functions that dominate all other functions in the reachability graph, with separate dominator functions chosen from separate OS components. A function v in the call-graph for a system-call handler dominates another function w if every path from the beginning of the call-graph to function w contains v. We define a component as a (possibly) separately-linked code region, namely the core kernel and each distinct module. For example, the components involved in the execution of a *read* operation are the core system-call handler, the filesystem driver, and the disk driver. We insert a watchpoint in the middle of each selected dominator. Continuing the *read* example, we insert watchpoints in `sys_read`, `do_sync_read` and `generic_file_aio_read`. Table 1 shows the list of operations and corresponding functions where watchpoints were placed.

We currently perform this analysis manually by looking into the source code of the kernel. During this process, we inserted watchpoints inside the kernel functions corresponding to different system-calls and ran different user-space applications to stimulate our watchpoints. This process has given us confidence that the watchpoints are at the correct locations. To remove errors that may get introduced by the manual analysis, an automated approach is required. Ganapathy et al. [13] presented a system that automatically analyzes the kernel source code and finds locations to insert hooks. This automated system may also be suitable for selection of Sherlock watchpoint instrumentation locations.

4.3 Modeling System Call Handler Execution

Sherlock's goal is to reveal hidden system operations. It must distinguish between a normal system call and a suspicious system call. A call is suspicious if it was altered by a one-to-many mapping, as described in Section 3.5. We detect altered system call operations by comparing the sequence of watchpoints executed by the running kernel against an automaton model describing the expected behavior of each system call handler function in the kernel.

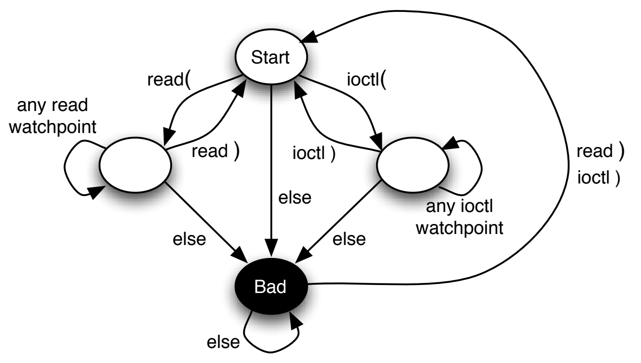

Figure 5: A portion of Sherlock's Büchi automaton used to distinguish between benign and suspicious operations. All states are accepting. The figure shows the portions of the automaton corresponding to the system calls `read` and `ioctl`; the patterns for other system calls are similar. Whenever the automaton enters in the *Bad* state, the internal kernel operations are not matched with a requesting system call, an indicator that an Illusion attack is underway.

To know when system calls are legitimately invoked by an application, Sherlock inserts two watchpoints in each system-call handler function to know the start and end of the system operation. When a benign application invokes a system operation, execution reaches the system-call handler using NSEP and the first watchpoint in that handler executes. Sherlock's hypervisor component receives the watchpoint and traverses an opening bracket "(" transition indicating which system call was invoked. It then receives subsequent watchpoints that had been inserted along the control-flow path of the system operation. Every time it receives a watchpoint, it verifies that the watchpoint was expected given the invoked system call. It performs this step repeatedly until it sees the watchpoint ")" corresponding to the end of the system call request. However, if any received watchpoint should not have occurred given the system call request, then an Illusion attack may have obfuscated the system call interface. The language of allowed watchpoint sequences is omega-regular, so a finite-state automaton can easily characterize the expected and suspicious sequences. Figure 5 shows a portion of the automaton used by Sherlock to distinguish between benign and suspicious operations, with all individual system calls aggregated into a single model. Whenever a watchpoint executes and is not the part of the system operation, Sherlock will take the *else* transition and reach the *suspicious* state. At the suspicious state, any operation can be executed until the system call completes, taking Sherlock back to the *start* state. A kernel runs forever, so Sherlock's model is a Büchi automaton with all states final.

4.4 Adaptive Design

Sherlock tunes its performance to match the current security condition of the monitored kernel. Its hypervisor-level component receives watchpoints from the guest operating system and operates the Büchi automaton to identify the suspicious activities. If the automaton is not in a suspicious state, then the kernel is executing in an expected way. Sherlock's hypervisor component immediately discards all received watchpoints, as they would provide no additional information to a traditional system call monitor at the standard interface given the absence of an Illusion attack. Otherwise Sherlock concludes that this is a suspicious activity and starts logging subsequent watchpoints until the conclusion of the system call. During this process, it activates performance-costly inspection capabilities: it pauses the user VM and extracts parameters involved in the operation at each watchpoint. This data gathering helps Sherlock rebuild information about the actual system operation underway. Sherlock unpauses the user VM when an analyzer in the security VM completes the inspection of the state of the VM.

4.5 Demultiplexing of Events

The hypervisor component of Sherlock sees watchpoints as a stream of events generated by the interleaved execution of multiple processes. In order to verify these events using the automaton, Sherlock first demultiplexes them by identifying the VM and process that generated each event in the data stream. In an environment where multiple guest domains are running, it distinguishes among different VMs using the VM identifier. In order to distinguish among different processes, Sherlock uses the value present in the hardware `CR3` register, which contains the physical address of the page directory used by the executing process. This value is unique for each active process and is updated with every context switch. Relying on the CR3 and VM identifier values are safe because modification of these values are privileged operations and can only be performed by the hypervisor.

4.6 Sherlock Controller

The security VM has a user-level controller process that controls Sherlock's operation and runs infrequent, performance-costly operations. The controller starts and stops the user VM as needed to perform parameter extraction and system call identification during an active Illusion attack. Whenever the hypervisor-level component of Sherlock detects the Illusion attack, it informs the controller about the active attack and passes all the hidden system operations which were executed during the attack. The controller passes the information to existing security software, pauses the user VM, performs detailed analysis of the unexpected watchpoint, and then restarts the guest.

Consider again our example of the *read* system call and corresponding in-kernel execution behavior. Suppose malware uses the Illusion attack to invoke the *read* via arguments marshalled through the *ioctl* system call. The Illusion kernel module may invoke the read operation by directly calling the ext3 filesystem driver's read function `do_sync_read`. As Table 1 indicates, we inserted a watchpoint in this function. Sherlock receives the beginning marker event `"ioctl("` followed by the watchpoint `do_sync_read`. Since these events correspond to two different operations, they do not match and the automaton enters in the *Bad* state. Sherlock's hypervisor component reports this mismatch to the controller along with an indication that *read* is believed to be the actual system call operation underway. This report activates the controller's deep analysis mode. It pauses the user VM, extracts the parameters for the *read* operation from the user VM, and passes the computed system call to higher-level monitoring utilities and security software in the security VM.

5 Evaluation

We paired Sherlock with the Illusion attack to measure its impact on a typical system. We first provide an empirical false positive and false negative analysis of Sherlock in Section 5.1. We expect Sherlock to slightly diminish system performance and carry out experiments measuring its effect in Section 5.2. Lastly, we analyze the system's robustness to a knowledgeable attacker in Section 5.3.

5.1 False Positive & False Negative Analysis

We empirically analyze the ways in which Sherlock may erroneously believe interface obfuscation is underway or fail to recognize the presence of an Illusion attack.

A false positive may arise if our model of kernel system call handlers is incorrect and omits allowed behavior. A fortunate property of false positives of this sort is that they do not occur due to fundamental weaknesses in Sherlock's technique. For example, Linux uses the `write` system-call for both writing to a

Single Watchpoint	*Time (μs)*
Sherlock without Hidden Operation	3.201
Sherlock with Hidden Operation	39.586

Table 2: Single watchpoint processing time

file and writing to a socket. If a `write` is invoked to write to the socket, it invokes watchpoints both corresponding to the write operation and socket operation. In this case, Sherlock sees watchpoints corresponding to two different system operations; a naïve automaton model would cause Sherlock to conclude that this is suspicious activity. Refinements in such a naïve model will remove these false positives and improve Sherlock for all future use of the system.

A false negative occurs when Sherlock does not report a hidden operation that happens inside the guest OS. This situation arises when an attacker performs an operation that does not trigger our watchpoints. As we disallow modifications to kernel code, the attacker may have either (1) brought a full copy of the kernel code they require as part of their kernel module, or (2) manipulated kernel code pointers to cause execution to jump in-and-out of small fragments of existing code in the kernel. If the Illusion attack kernel module attempts accesses to hardware, then it is likely that the attacker will call back into the existing device driver contained in the kernel, which our watchpoints have instrumented. We expect that abuse of indirect control flows could be thwarted by guards at the point of the indirect flows [1,39,43].

5.2 Performance

As a software mechanism that intercepts events that may occur at high rates, we expect Sherlock's monitoring to impact the execution performance of guest applications. To measure its performance, we carried out several experiments involving CPU-bound, disk-bound, and network-bound workloads. For all experiments, we used Fedora Core 5 in both the security VM and user VM running above Xen 3.0.4 operating in fully virtualized (HVM) mode. Our test hardware contained an Intel Core 2 Duo processor at 2.2 GHz, with VT-x, and with 2 GB of memory. We assigned 512 MB of memory to the untrusted user VM. All reported results show the median time taken from five measurements. We measured microbenchmarks with the x86 `rdtsc` instruction and longer executions with the Linux `time` command-line utility.

We measured the time to process a single watchpoint both with and without an Illusion attack underway. Whenever a watchpoint executes, the control reaches the hypervisor and Sherlock's hypervisor component checks whether or not the watchpoint is due to a hidden operation. If so, it then pauses the guest VM and sends information to the Sherlock controller. Otherwise, it discards the watchpoint and sends control back to the VM. Table 2 shows the result of our experiment. The adaptive behavior of Sherlock is well visible: its watchpoint processing is fast for the benign case, and an order of magnitude more costly when deep inspection is performed due to the detection of a hidden operation. We note that hidden operation processing shows a best-case cost, as our experiment performed a no-operation inspection into the guest OS. An analyzer performing a more detailed analysis may take more time before resuming the guest VM's execution.

Sherlock expects multiple watchpoints for each operation. We next measured the time to execute a single operation in the presence of watchpoints. Since we only instrumented the code of `open`, `read`, `write`, `socket`, `connect`, `send`, and `recv` system calls, we only measured the time to execute these operations. We wrote a sample test program to invoke each of these calls and measured execution times in three ways: without Sherlock, with Sherlock during benign execution, and with Sherlock during an Illusion attack. Table 3 presents the result of this experiment, and it also shows the number of watchpoints inserted for each operation and the number of watchpoints executed during an Illusion attack. It can be seen from the table that for `open` and `read` operations, Sherlock is able to detect the presence of the Illusion attack

Operation	*Normal VM Time*	*Sherlock Time*	*Sherlock + Hidden Op. Time*	*# of watchpoints*	
				Inserted	*Executed*
Open	17.946	26.439	61.752	3	1
Read	19.419	30.537	60.970	4	1
Write	27.025	37.807	90.656	4	2
Socket	24.515	45.558	115.106	5	3
Connect	1879.894	1905.336	1984.419	4	2
Send	717.391	746.416	838.440	4	2
Receive	8.377	20.958	79.488	4	2

Table 3: Single operation processing time (μs)

Operations	*Normal VM (sec)*	*Sherlock (sec)*	*% Overhead*
Disk I/O	45.731	49.902	9.12
Network I/O (Virtual Network)	29.005	29.608	2.07
Network I/O (Physical Network)	212.054	213.352	0.61
CPU Bound	102.004	103.383	0.01

Table 4: Performance measurements. "Normal VM" indicates Xen without Sherlock monitoring; "Sherlock" includes monitoring time.

even with the execution of single watchpoint.

Finally, we tested Sherlock with real workloads to measure its overhead seen by the guest operating system's applications. We carried out experiments with disk-, network-, and CPU-bound workloads. In our disk I/O experiment, we copied the 278 MB kernel source code tree from one directory to another using Linux's `cp` command. Table 4 shows the time taken by this operation and it can be seen that Sherlock creates 9.2% overhead. To test Sherlock against network workloads, we performed two different experiments. In the first experiment, we transferred a file of size 200 MB over HTTP between virtual machines using Xen's virtual network. We used a small HTTP server, `thttpd`, to serve the file. We repeated the file transfer operation on the same file using the physical network to a nearby machine. We measured the time taken by these file transfers with and without Sherlock and show results in Table 4. It is evident that Sherlock's overhead is very low.

Finally, we performed a CPU bound operation. Since we mainly instrumented I/O related system calls (both disk and network), we did not expect significant overhead with CPU workloads. To measure CPU bound cost, we used `bzip2` utility to compress a tar archive of the Linux source tree. The result of this operation is again shown in Table 4, and it is again clear that Sherlock's overhead on CPU bound loads is negligible.

As expected during the execution of these benign applications, Sherlock did not report any hidden operation. These results show that Sherlock is indeed an adaptive system that creates small overhead during the execution of benign applications. Given this appealing performance, it appears well suited to production desktop environments.

5.3 Security Analysis

We analyze Sherlock from the perspective of an attacker who wants to bypass our security system in order to execute malicious code.

Sherlock depends upon watchpoints to know about the state of the executing kernel. It implements a watchpoint using the `VMCALL` instruction, which gets triggered when a watchpoint is hit inside the guest VM. An attacker may try to confuse Sherlock by executing fake or nop `VMCALL` instructions, which may lead to false alarms. Sherlock defeats these kind of mimicry attacks by locating the code position (instruction pointer) from where a `VMCALL` instruction is invoked. With this knowledge, any `VMCALL` instruction that does not correspond to the set of our watchpoints can be discarded.

An attacker cannot tamper with the watchpoints or static kernel code. However, she can unload a device driver that contains our watchpoint and replace it with a new device driver. In this way, the attacker can execute the driver functions to perform hidden operations without hitting any of the watchpoints. Sherlock thwarts this attack by keeping the memory locations where the device drivers containing watchpoints are loaded in the kernel memory (this information is kept in the `/proc/module` file). Any unloading and loading (memory update) operations in those memory areas are intercepted by Sherlock to counter the attack.

6 Conclusions

In this paper, we described our two principal contributions: first, we implemented an attack called *Illusion* that obfuscates the system-call interface and uses a legitimate mechanism to invoke system operations without revealing the system call for that operation. We considered the problem where an existing system-call based analyzer is not able to see the operations performed by the Illusion attack due to its reliance on the system-call mechanism. Second, we presented a system named *Sherlock* that addresses the problem of hidden operations created by the Illusion attack. It detects the presence of the Illusion attack and exposes hidden operations with the help of watchpoints inserted inside the kernel code path of the guest OS. The adaptive design of Sherlock helps maintain the acceptable performance overhead imposed on the user-space applications.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2005.

[2] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2007.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[4] Blorge. Faulty drivers bypass Vistas kernel protection. `http://vista.blorge.com/2007/08/02/faulty-drivers-bypass-vistas-kernel-protection/`. Last accessed 11 Sep 2008.

[5] K. Borders, X. Zhao, and A. Prakash. Siren: Catching evasive malware. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[6] A. Chakrabarti. An introduction to Linux kernel backdoors. `http://www.infosecwriters.com/hhworld/hh9/lvtes.txt`. Last accessed 11 Sep 2008.

[7] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium*, Baltimore, MD, Aug. 2005.

[8] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. Subdomain: Parsimonious server security. In *Systems Administration Conference*, New Orleans, LA, Dec. 2000.

[9] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *15th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, Virginia, Oct. 2008.

[10] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th Symposium on Operating System Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[11] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2003.

[12] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for UNIX processes. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1996.

[13] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the Linux security modules framework. In *12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, Virginia, Nov. 2005.

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[15] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2003.

[16] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS support and applications for trusted computing. In *9th Hot Topics in Operating Systems (HOTOS)*, Lihue, HI, May 2003.

[17] J. Giffin, S. Jha, and B. Miller. Detecting manipulated remote call streams. In *11th USENIX Security Symposium*, San Francisco, CA, Aug. 2002.

[18] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2004.

[19] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[20] X. Jiang and X. Wang. Out-of-the-box monitoring of VM-based high-interaction honeypots. In *10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Surfers Paradise, Australia, Sept. 2007.

[21] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based 'out-of-the-box' semantic view. In *14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2007.

[22] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *USENIX Annual Technical Conference*, Boston, MA, June 2006.

[23] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based hidden process detection and identification using Lycosid. In *ACM Conference on Virtual Execution Environments (VEE)*, Seattle, WA, Mar. 2008.

[24] Kimmo Kasslin. Kernel malware: The attack from within. `www.f-secure.com/weblog/archives/kasslin_AVAR2006_KernelMalware_paper.pdf`. Last accessed 11 Sep 2008.

[25] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Symposium on Operating System Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[26] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *14th USENIX Security Symposium*, Baltimore, MD, Aug. 2005.

[27] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux kernel integrity measurement using contextual inspection. In *2nd ACM Workshop on Scalable Trusted Computing*, Alexandria, VA, Nov. 2007.

[28] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, generic, and safe unpacking of malware. In *23rd Annual Computer Security Applications Conference (ACSAC)*, Miami, FL, Dec. 2007.

[29] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Boston, MA, Sept. 2008.

[30] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu. Probing the guts of kprobes. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2006.

[31] McAfee Security. System Call Interception. `http://www.mcafee.com/us/_tier2/products/_media/mcafee/wp_systemcallinterception.pdf`. Last accessed 11 Sep 2008.

[32] Microsoft. Digital Signatures for Kernel Modules on Systems Running Windows Vista. `http://www.microsoft.com/whdc/winlogo/drvsign/kmsigning.mspx`. Last accessed 11 Sep 2008.

[33] Microsoft. Introduction to File System Filter Drivers. `http://msdn.microsoft.com/en-us/library/ms790748.aspx`. Last accessed 11 Sep 2008.

[34] Microsoft. PatchGuard. `http://en.wikipedia.org/wiki/Kernel_Patch_Protection`. Last accessed 11 Sep 2008.

[35] D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer. Exploiting execution context for the detection of anomalous system calls. In *10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Surfers Paradise, Australia, Sept. 2007.

[36] K. Onoue, Y. Oyama, and A. Yonezawa. Control of system calls from outside of virtual machines. In *ACM Symposium on Applied Computing*, Fortaleza, Ceara, Brazil, Mar. 2008.

[37] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[38] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *15th USENIX Security Symposium*, Vancouver, BC, Canada, Aug. 2006.

[39] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2007.

[40] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Boston, MA, Sept. 2008.

[41] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a MAC-based security architecture for the Xen open-source hypervisor. In *Annual Computer Security Applications Conference (ACSAC)*, Tucson, AZ, Dec. 2005.

[42] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[43] M. Sharif, K. Singh, J. Giffin, and W. Lee. Understanding precision in host based intrusion detection: Formal analysis and practical models. In *10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Surfers Paradise, Australia, Sept. 2007.

[44] SoftPedia. Download Tool to Bypass Driver Signing on 32-bit and 64-bit Windows Vista. `http://news.softpedia.com/news/Download-Tool-to-Bypass-Driver-Signing-on-32-bit-and-.64-bit-Windows-Vista-61405.shtml`. Last accessed 11 Sep 2008.

[45] A. Srivastava and J. Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Boston, MA, Sept. 2008.

[46] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems congurable. In *Symposium on Operating System Design and Implementation (OSDI)*, Seattle, WA, Oct. 2006.

[47] K. M. Tan, K. S. Killourhy, and R. A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Recent Advances in Intrusion Detection (RAID)*, Zurich, Switzerland, Oct. 2002.

[48] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *9th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Nov. 2002.

[49] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. Constructing services with interposable virtual hardware. In *1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.

[50] J. Wilhelm and T. c. Chiueh. A forced sampled execution approach to kernel rootkit identification. In *10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Surfers Paradise, Australia, Sept. 2007.

[51] H. Xu, W. Du, and S. J. Chapin. Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths. In *7th International Symposium on Recent Advances in Intrusion Detection*, Sophia Antipolis, France, Sept. 2004.

[52] M. Xu, X. Jiang, R. Sandhu, and X. Zhang. Towards a VMM-based usage control framework for OS kernel integrity protection. In *12th ACM Symposium on Access Control Models and Technologies (SACMAT)*, Sophia Antipolis, France, June 2007.

[53] X. Zhao, K. Borders, and A. Prakash. Towards protecting sensitive files in a compromised system. In *3rd IEEE Security in Storage Workshop*, 2005.