

# Automatic Discovery of Parasitic Malware

Abhinav Srivastava and Jonathon Giffin

School of Computer Science, Georgia Institute of Technology, USA  
{abhinav,giffin}@cc.gatech.edu

**Abstract.** Malicious software includes functionality designed to block discovery or analysis by defensive utilities. To prevent correct attribution of undesirable behaviors to the malware, it often subverts the normal execution of benign processes by modifying their in-memory code images to include malicious activity. It is important to find not only maliciously-acting benign processes, but also the actual parasitic malware that may have infected those processes. In this paper, we present techniques for automatic discovery of unknown parasitic malware present on an infected system. We design and develop a hypervisor-based system, Pyrenée, that aggregates and correlates information from sensors at the network level, the network-to-host boundary, and the host level so that we correctly identify the true origin of malicious behavior. We demonstrate the effectiveness of our architecture with security and performance evaluations on a Windows system: we identified all malicious binaries in tests with real malware samples, and the tool imposed overheads of only 0%–5% on applications and performance benchmarks.

## 1 Introduction

Malware instances exhibit complex behaviors designed to prevent discovery or analysis by defensive utilities. In addition to file system and registry changes, malicious software often subverts the normal execution of benign processes by modifying their in-memory code image (*parasitic behavior*). For example, the Conficker worm injects undesirable dynamically linked libraries (DLLs) into legitimate software [38]. In another example, the Storm worm injects code into a user-space network process from a malicious kernel driver to initiate a DDoS attack from the infected computers [23]. Parasitic behaviors help malware execute behaviors—such as spam generation, denial-of-service attacks, and propagation—without themselves raising suspicion. When analyzing a misbehaving system to identify and eradicate malware, it is important both to terminate maliciously-acting but benign processes and to find other software that may have induced the malicious activity.

The visible effects of current attacks against software regularly manifest first as suspicious network traffic. This is due to the monetary gains involved in controlling large networks for botnet, spam, and denial of service attacks [36]. After detecting malicious traffic, network intrusion detection systems (NIDSs) can pinpoint a host within a network or enterprise responsible for that traffic [11, 25]. These network sensors can identify an infected system’s IP address,

network ports, and traffic behaviors. This coarse-grained information permits only coarse-grained responses: an administrator could excise an infected system from the network, possibly for reimaging. Unfortunately, in many common use scenarios, complete disk sanitization results in intolerable losses of critical data not stored elsewhere, even though that data may not have been affected by the infection. On-host analysis changes these brutal remediation techniques by providing a means to appropriately attribute malicious behavior to malicious software. Realizing that a process sending or receiving attack traffic is a hijacked benign program requires cooperation between network sensors and on-host execution monitors. By gaining a better understanding of a malware infection on a system, we can offer opportunities for surgical response.

This paper presents techniques and a prototype system, Pyrenée, for automatic discovery of unknown parasitic malware present on an infected system. Pyrenée correlates network-level events with host-level activities, so it applies exclusively to attacks that send or receive detectably-suspicious traffic. We design Pyrenée to effectively detect parasitic behavior occurring both at the user and kernel level. To remain tamper-resistant from kernel-level malware, we make use of hypervisors or virtual machine monitors (VMMs). Pyrenée’s architecture is comprised of sensors at the network-to-host boundary (*network attribution sensor*), the host level (*host attribution sensor*) and the network level (*network sensor*), as well as a correlation engine that uses information provided by the sensors to identify likely malware on an infected computer.

The sensors cooperate so that Pyrenée can correctly attribute undesirable behaviors to a malicious software infection. Pyrenée uses off-the-shelf network sensors, such as BotHunter [11] or Snort [31], to detect suspicious network traffic and identify hosts with possible malware infections. When a network sensor detects malicious packets, it informs the network-to-host boundary (network attribution) sensor, deployed at the host in a trusted virtual machine (VM). On receiving the information, the network attribution sensor performs secure virtual machine introspection (VMI) to find the process bound to the malicious connection inside the guest VM. Though knowing the end-point of a malicious connection on an infected system significantly reduces the cleaning effort of an administrator, this information is still not complete. The network attribution sensor does not know if the identified process is malicious, or if it is a hijacked benign program victim of the parasitic behavior.

To find the true origin of malicious parasitic behaviors, Pyrenée uses a host-attribution sensor implanted inside the hypervisor. A process can suffer from parasitic behaviors either from another process or an untrusted kernel driver. To counter that, the host-attribution sensor monitors the execution of both user-level processes and untrusted kernel drivers. The host-attribution sensor monitors system calls and their parameters invoked by processes to detect the process-to-process parasitic behavior. To detect untrusted drivers’ parasitic DLL and thread injection behaviors, we contain untrusted drivers in an isolated address space from the kernel. This design provides the host-attribution sensor an

ability to monitor kernel APIs invoked by untrusted drivers and enables it to detect parasitic behaviors originating from the untrusted drivers.

The correlation engine gathers information from all the sensors to identify the true origin of parasitic behaviors. Correlating network information with host information is a key design feature of our system. Taken alone, either approach will have diminished utility in the presence of typical attacks or normal workloads. Network-based detection can identify an infected system but cannot provide finer-grained process-specific information. Host-based detection can identify occurrences of parasitism, but it cannot differentiate malicious parasites from benign symbiotes. For example, debugging software and other benign software, such as the Google toolbar, use DLL injection for legitimate purposes. These observations are critical: A process sending or receiving malicious network traffic may not itself be malware, and a process injecting code into another process may not be malicious. Only by linking injection with subsequent malicious activity observed at the network (or other) layer can we correctly judge the activity at the host.

This paper makes the following contributions:

- We develop a well-reasoned malware detection architecture that finds unknown malware based on its undesirable network use. Our design correlates activity on the network with behaviors at the infected host.
- We correctly attribute observed behaviors to the actual malware responsible for creating those behaviors, even in the presence of parasitic malware that injects code into benign applications. Proper attribution creates the foundation for subsequent surgical remediation of the malware infection.
- Our system works for both the user- and kernel-level malware. To monitor parasitic behaviors at the user-level, we monitor system calls. For kernel-level parasitism, we securely monitor kernel APIs invoked by untrusted drivers.
- Our design satisfies protection and performance goals. We leverage virtualization to isolate security software from the infected Windows system. Our security evaluation shows that our system is able to detect the true origin of parasitic behavior occurring at user or kernel level. The performance evaluation demonstrates that even with runtime on-host monitoring, our performance impact remains only 5% or better.

## 2 Related Work

Pyrenée discovers unknown parasitic malware by identifying the true origin of malicious activities. To achieve its goals, it combines information gathered at both the host and the network level. Previous research in both individual areas has developed a collection of solutions to aspects of this problem.

Host-based security software generally either scans unknown programs for patterns that match signatures of known malware [2, 17] or continually monitors behaviors of software searching for unusual or suspicious runtime activity [10, 12, 32]. Pyrenée’s host attribution sensor is closest in spirit to the latter

systems. It monitors the execution behavior of processes and untrusted drivers to identify instances of DLL injection or remote thread injection. Unlike traditional host-based utilities, it does not rely on injection alone as evidence of malware, as benign software sometimes uses injection for benign purposes. A heuristic-based malware detection system that monitors system calls or kernel APIs and detects code injection attacks may produce false positives. For example, DLL injection is used by the Microsoft Visual Studio debugger to monitor processes under development. Likewise, the Google toolbar injects code into `explorer.exe` (the Windows graphical file browser) to provide Internet search from the desktop. Pyrenée uses system-call information only when a network-sensor provides corroborating evidence of an attack.

Pyrenée uses virtualization to isolate its on-host software from an infected system. Virtualization has been used previously in the development of security software, including intrusion detection systems [8, 14, 15, 20, 28], firewalls [35], protection [26, 30, 40], and other areas [6]. Pyrenée’s network attribution sensor is an evolution of the VMwall virtualization-based firewall design [35]. VMwall required packet queuing that introduced delay into network communication; our sensor has no such need and allows network communication to operate at full speed. The sensor makes use of virtual machine introspection (VMI), proposed by Garfinkel and Rosenblum [8], to attribute network communication to processes. Nooks [37] and SIM [33] proposed address space partitioning to isolate drivers and security applications, respectively. Pyrenée also uses address space partitioning to isolate only untrusted drivers from the core kernel and trusted drivers in a different address space.

Backtracker [19] reconstructs the sequence of steps that occurred in an intrusion by using intrusion alerts to initiate construction of event dependency graphs. In a similar way, Pyrenée uses NIDS alerts to initiate discovery of malicious software even in the presence of parasitic behaviors. Technical aspects of Backtracker and Pyrenée differ significantly. Backtracker identifies an attack’s point of entry into a system by building dependencies among host-level events. It assumes that operating system kernels are trusted and hence monitors system calls; it stores each individual system call in its log for later dependency construction. Pyrenée identifies software components responsible for a post-infection attack behavior visible on the network by correlating behaviors at both the network level and host level. On the host, it monitors and stores only high-level parasitic behaviors. It does not trust the OS kernel and assumes that kernel-level malware may be present, and it monitors both system calls and kernel APIs to detect both user- and kernel-level parasitism. Both Backtracker and Pyrenée are useful to remediation in different ways: Pyrenée’s information guides direct removal of malicious processes, while Backtracker’s information helps develop patches or filters that may prevent future reinfection at the identified entry point.

Malware analysis tools [41] have also built upon virtualization. Dinaburg et al. [5] developed an analysis system that, among other functionality, traced the execution of system calls in a manner similar to our host attribution sensor. Martignoni et al. [21] proposed a system that builds a model of high-level malware

behavior based upon observations of low-level system calls. Like that system, Pyrenée uses a high-level characterization of DLL and thread injection identified via low-level system-call monitoring; however, our system does not employ the performance-costly taint analysis used by Martignoni. In contrast to analysis systems, our goal is to provide malware detection via correct attribution of malicious behavior to parasitic malware. We expect that it could act as a front-end automatically supplying new malware samples to deep analyzers.

### 3 Parasitic Malware

Pyrenée discovers parasitic malware. In this section, we present the threat model under which Pyrenée operates and describe common parasitic behaviors exhibited by malware.

#### 3.1 Threat Model

We developed Pyrenée to operate within a realistic threat model. We assume that attackers are able to install malicious software on a victim computer system at both the user and kernel levels. Installed malware may modify the system to remain stealthy. These facts are demonstrated by recent attacks happening at the user and the kernel level. A preventive approach that does not allow users to load untrusted drivers may not be effective because users sometimes unknowingly install untrusted drivers for various reasons, such as gaming or adding new devices. Due to these reasons, we distinguish between trusted and untrusted drivers and isolate untrusted drivers in a separate address space. We assume that the malware will at some point send or receive network traffic that network-level intrusion detection systems (network sensors) are able to classify as malicious or suspicious: this may include traffic related to spam, denial-of-service attacks, propagation, data exfiltration, or botnet command-and-control.

Pyrenée makes use of virtual machine introspection (VMI) in its network attribution sensor. We perform VMI from a high-privilege virtual machine different than the infected system and assume that the high-privilege machine and the underlying hypervisor are within the trusted computing base. VMI requires kernel data structure invariants to hold. Pyrenée does not protect these data structures, but rather assumes that either existing invariant testing solutions protect the structures [1, 27, 34] or introspection is performed in a secure way [3]. We do not attempt to detect illicit hooking, control data attacks, evasion from hypervisor-based monitors, or modification of binaries on disk, as previous research has already studied those threats [5, 18, 28, 40].

#### 3.2 Malware Behaviors

Parasitic malware alters the execution behavior of existing benign processes as a way to evade detection. These malware often abuse both Windows user and

**Table 1.** Different parasitic behavior occurring from user- or kernel-level.

<i>Number</i>	<i>Source</i>	<i>Target</i>	<i>Description</i>
Case 1A	Process	Process	DLL and thread injection
Case 1B	Process	Process	Raw code and thread injection
Case 2A	Kernel driver	Process	DLL and thread alteration
Case 2B	Kernel driver	Process	Raw code and thread alteration
Case 2C	Kernel driver	Process	Kernel thread injection

kernel API functions to induce parasitic behaviors. We consider a malware parasitic if it injects either executable code or threads into other running processes. The parasitic behaviors can originate either from a malicious user-level process or a malicious kernel driver. Table 1 lists the different cases in which malware can induce parasitic behavior, and the following section explains each of those cases in detail.

**Case 1A:** *Dynamically-linked library (DLL) injection* allows one process to inject entire DLLs into the address space of a second process [29]. An attacker can author malicious functionality as a DLL and produce malware that injects the DLL into a victim process opened via the Win32 API call `OpenProcess` or created via `CreateProcess`. These functions return a process handle that allows for subsequent manipulation of the process. The malware next allocates memory inside the victim using the `VirtualAllocEx` API function and writes the name of the malicious DLL into the allocated region using `WriteProcessMemory`. Malware cannot modify an existing thread of execution in the victim process, but it can create a new thread using `CreateRemoteThread`. The malware passes to that function the address of the `LoadLibrary` API function along with the previously written-out name of the malicious DLL.

**Case 1B:** A *raw code injection* attack is similar to a DLL injection in that user-space malware creates a remote thread of execution, but it does not require a malicious DLL to be stored on the victim’s computer system. The malware allocates memory space as before within the virtual memory region of the victim process and writes binary code to that space. It then calls `CreateRemoteThread`, passing the starting address of the injected code as an argument.

**Case 2A:** A kernel-level malicious driver also shows parasitic behavior by injecting malicious DLLs inside the user-space process. A malicious driver can perform this task in a variety of ways, such as by calling system call functions directly from the driver. A stealthy technique involves Asynchronous Procedure Calls (APCs): a method of executing code asynchronously in the context of a particular thread and, therefore, within the address space of a particular process [22]. Malicious drivers identify a process, allocate memory inside it, copy the malicious DLL to that memory, create and initialize a new APC, alter an existing thread of the target process to execute the inserted code, and queue the APC to later run the thread asynchronously. This method is stealthy as APCs are very common inside the Windows kernel, and it is very hard to distinguish between benign and malicious APCs.

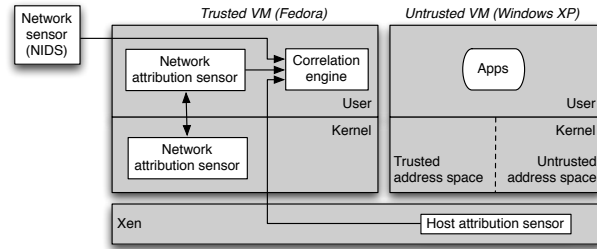


Fig. 1. Architecture of Pyrenée

**Case 2B:** This method is similar to the one explained in Case 2A. The difference lies in the form of malicious code that injected into a benign process. Here, malicious kernel drivers inject raw code into a benign process and execute it using the APC.

**Case 2C:** Finally, a *kernel thread injection* is the method by which malicious drivers execute malicious functionality entirely inside the kernel. A kernel thread executing malicious functionality is owned by a user-level process, though the user-level process had not requested its creation. By default, these threads are owned by the `System` process, however a driver may also choose to execute its kernel thread on behalf of any running process.

Our system adapts well as new attack information becomes available. Though the described methods are prevalent in current attacks, other means of injecting malicious code into benign software also exist. For example, `SetWindowsHookEx`, `AppInit_DLL`, and `SetThreadContext` APIs can be used for malice. Our general technique can easily encompass these additional attack vectors by monitoring their use in the system.

## 4 Architecture

Pyrenée automatically identifies at runtime the malicious code running on an infected system. That objective leads to the following design goals:

- **Accurate Attribution:** Pyrenée combines data from network-based and host-based sensors to avoid false positives, provide process or driver level granularity in malware identification, and to account for evasive behaviors of parasitic malware.
- **Automatic, Runtime Detection:** We design lightweight sensors that incur low overhead, allowing Pyrenée to operate at runtime. We identify malicious code without any human intervention.
- **Resist Direct and Indirect Attacks:** Pyrenée’s tamper-resistant design prevents direct attack by a motivated attacker. We deploy all components of our system outside of an infected operating system.

Pyrenée has a modular design (Figure 1). Its architecture uses the hypervisor to provide isolation between our software and the infected system. To perform accurate detection and identification of malicious code, Pyrenée aggregates information collected from three different sensors. A network sensor identifies inbound or outbound network traffic of suspicion; we use off-the-shelf network intrusion detection systems (NIDS) like BotHunter, Snort, or Bro and will not further discuss this component. The network attribution and host attribution sensors are software programs executing in the isolated high-privilege virtual machine and hypervisor, respectively. A correlation engine, also running in the trusted VM, takes data from all three types of sensors and determines the malicious software present in the victim. Our sensors are lightweight and suitable for on-line detection. The following sections describe the network attribution and host attribution sensors as well as the correlation engine.

#### 4.1 Network Attribution Sensor

The network attribution sensor maps network-level packet information to host-level process identities. Given a network sensor (NIDS) alert for some suspicious traffic flow, the network attribution sensor is responsible for determining which process is the local endpoint of that flow in the untrusted VM. This process may be malicious, or it may be a benign process altered by a parasitic malware infection. We deployed the network attribution sensor in a trusted virtual machine. It has two subcomponents: one in the VM’s kernel space and one in userspace. The kernel component provides high-performance packet filtering services by intercepting both inbound and outbound network packets for an untrusted VM. The userspace component performs virtual machine introspection (VMI) whenever requested by the kernel component.

The kernel component identifies separate TCP traffic flows. Whenever it receives a SYN packet, it extracts both the source and destination IP addresses and ports, which it then passes to the userspace component for further use. The kernel component is a passive network tap and allows all packets flows to continue unimpeded. Though in the current prototype of Pyrenée we only work with TCP flows, our system is able to intercept packets of any protocol.

The userspace component determines which process in the victim VM is the local endpoint of the network flow. When invoked by the kernel component, it performs memory introspection of the untrusted VM to identify the process bound to the source (or destination) port as specified in the data received by the kernel component. To find a process name, it must locate the guest kernel’s data structures that store network and process information. We have reverse engineered part of the Windows kernel to identify these structures, discussed in-depth in Section 5. The userspace component stores the extracted process and network connection information in a database to be used later by the correlation engine. The stored information helps even in the case when malware exits after sending malicious packets.

The network attribution sensor’s task is to determine the end-point of a network connection originated from the guest VM. Recent kernel-level attacks

complicate this task. For example, `srizbi` [16] is a kernel-level bot that executes entirely in the kernel. When untrusted drivers send/receive packets from the kernel, there is no user-space process that can be considered as the end-point of the connection. `Pyrenée` solves this problem by monitoring the execution of untrusted drivers. Since all kernel threads created by drivers are always assigned to a user-level process, that process becomes the end-point of the in-driver connection. To determine the actual driver, we enumerate all threads of the end-point process and match against threads of untrusted drivers.

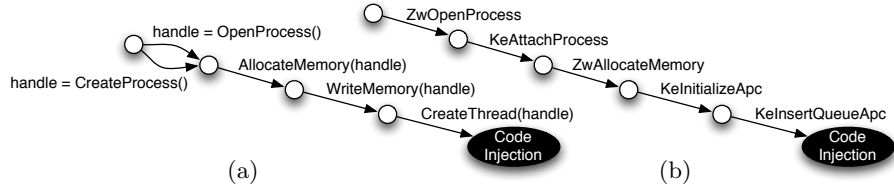
## 4.2 Host Attribution Sensor

The local endpoint of a malicious network flow may itself be a benign program: it may have been altered at runtime by a DLL or thread injection attack originating at a different parasitic malware process or driver. The host attribution sensor, deployed within the hypervisor, identifies the presence of possible parasitic malicious code. We describe the monitoring of both user and kernel level parasitic behaviors in the following sections.

**User-level Parasitism** User-level parasitism occurs when a malicious user process injects a DLL or raw code along with a thread into a running benign process as explained in Cases 1A and 1B. To detect a process-to-process parasitic behavior, the host attribution sensor continuously monitors the runtime behavior of all processes executing within the victim VM by intercepting their system calls [7, 9]. Note that we monitor the *native API*, or the transfers from userspace to kernel space, rather than the Win32 API calls described in Section 3. High-level API monitors are insecure and may be bypassed by knowledgeable attackers, but native API monitoring offers complete mediation for user-level processes.

The host attribution sensor intercepts all system calls, but it processes only those that may be used by a DLL or thread injection attack. This list includes `NtOpenProcess`, `NtCreateProcess`, `NtAllocateVirtualMemory`, `NtWriteVirtualMemory`, `NtCreateThread`, and `NtClose`, which are the native API forms of the higher-level Win32 API functions described previously. The sensor records the system calls' parameter values: *IN* parameters at the entry of the call and *OUT* parameters when they return to userspace. Recovering parameters requires a complex implementation that we describe in detail in Section 5.

The sensor uses an automaton description of malware parasitism to determine when DLL or thread injection occurs. The automaton (Figure 2a) characterizes the series of system calls that occur during an injection. As the sensor intercepts system calls, it verifies them against an instance of the automaton specific to each possible victim process. We determine when the calls apply to the same victim by performing data-flow analysis on the process handle returned by `NtOpenProcess` and `NtCreateProcess`. Should the handle be duplicated (using `NtDuplicateObject`), we include the new handle in further analysis. The sensor communicates information about detected injections to the correlation engine for further use.



**Fig. 2.** Runtime parasitic behavioral models. (a) Process-to-process injection. (b) Driver-to-process injection.

**Table 2.** Permission bits on trusted and untrusted address spaces.

Address Space	Trusted Code	Trusted Data	Untrusted Code	Untrusted Data
Trusted	rx	rw	r	rw
Untrusted	—	rw	rx	rw

**Kernel-level Parasitism** Kernel-level parasitism occurs when a malicious kernel driver injects either a DLL or raw code followed by the alteration of an existing targeted process’ thread (Case 2A and 2B). A kernel-level malicious driver can also create a new thread owned by any process as explained in Case 2C. To detect kernel-to-process parasitic behavior, the host attribution sensor monitors all kernel APIs invoked by untrusted drivers. However, there is no monitoring interface inside the kernel for drivers similar to the system-call interface provided to user applications. To solve this problem, Pyrenée creates a monitoring interface inside the kernel for untrusted drivers by isolating them in a separate address space and monitoring kernel APIs invoked by untrusted drivers through this new interface.

Pyrenée creates a new address space inside the hypervisor transparent to the guest OS and loads all untrusted drivers in this address space. This new address space is analogous to the existing kernel address space, however permissions are set differently. The existing kernel space, called the *trusted page table* (TPT), contains all the core kernel and trusted driver code with read and execute permissions, and untrusted driver code with read-only permissions. The untrusted driver address space, called the *untrusted page table* (UPT), contains untrusted code with read and execute permissions, and trusted code as non-readable, non-writable, and non-executable. Pyrenée also makes sure that the data pages mapped in both the address spaces are non-executable. Table 2 shows the permissions set on UPT and TPT memory pages. With these permission bits, any control flow transfers from untrusted to trusted address space induce page faults thereby enabling the host-attribution sensor to monitor kernel APIs invoked by untrusted drivers.

Pyrenée differentiates between trusted and untrusted drivers at the time of loading to decide in which address space they must be mapped. This differentiation can be made using certificates. For example, a driver signed by Microsoft can be loaded in the trusted address space. However, Microsoft might not rely

on drivers signed by other parties whose authenticity is not verified. With this design, all Microsoft signed drivers are loaded into the trusted address space, and other drivers signed by third party vendors or unsigned drivers, including kernel malware, are loaded into the untrusted address space.

Due to the isolated address space, the host-attribution sensor intercepts all kernel APIs invoked by untrusted drivers and inspects their parameters. The sensor uses an automaton to characterize the parasitic behavior originating from malicious drivers. When the sensor intercepts kernel APIs, it verifies against the automaton to recognize the parasitic behavior. In our current prototype, we create an automaton based on the kernel APC-based code injection (Figure 2b). The host-attribution sensor records the gathered information for future use by the correlation engine.

### 4.3 Correlation Engine

The correlation engine identifies which code on an infected system is malicious based on information from our collection of sensors [39]. The engine has three interfaces that communicate with a NIDS, the host attribution sensor, and the network attribution sensor. Architecturally, it resides in the isolated, high-privilege VM.

The NIDS provides network alert information to the correlation engine's first interface. This information includes the infected machine's IP address, port used in the suspicious flow, and other details. The alert acts as a trigger that activates searches across information from the software sensors. The second interface gathers information from the network attribution sensor, which provides information that maps the malicious network connection identified by the NIDS to a host-level process.

The third interface collects information from the host attribution sensor. In its process-to-process injection report, the host attribution sensor passes identifiers of injecting and victim processes, a handle for the victim of the injection, and other data. When receiving this information, the correlation engine uses VMI to retrieve detailed data about the victim and injecting processes, including their name, their component DLLs, and their open files. Should the victim process not have an identifier, as is the case for victims created via `NtCreateProcess`, the engine uses the victim's process handle to recover information about the victim. Section 5 provides low-level details of this data extraction. In the kernel-to-process injection report, it passes details of the victim process, such as the process identifier, name, and the name of the malicious kernel driver.

Based on the information provided by sensors, the correlation engine constructs a list of malicious processes and drivers. It matches attack information provided by a NIDS with network flow endpoint records generated by the network attribution sensor. When it finds a match, it extracts the name of the process bound to the malicious connection. Using information from the host attribution sensor, it determines whether or not the process has suffered from a parasitic attack. When it finds an injection, it extracts the name of the injecting

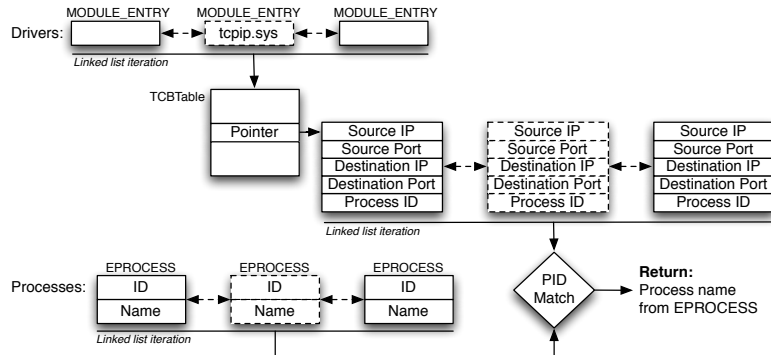


Fig. 3. Network connection to host-level process correlation in Windows

process or driver and adds it to the list of malicious code. Finally, it identifies other benign processes infected by the malware by searching again within the host attribution sensor’s records. The correlation engine periodically purges records generated by the network and host attribution sensors.

## 5 Low-Level Implementation Details

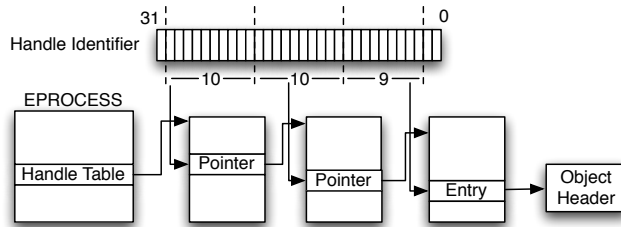
Pyrenée is an operating prototype implemented for Windows XP SP2 victim systems hosted in virtual machines by the Xen hypervisor version 3.2. The high-privilege VM executing our software runs Fedora Core 9. Implementing Pyrenée for Windows victim systems required technical solutions to challenging, low-level problems.

### 5.1 Fast Network Flow Discovery

The network attribution sensor intercepts all inbound and outbound network flows of untrusted virtual machines. To intercept packets at a fast rate, we deployed the sensor’s packet filter inside the trusted VM’s kernel-space; we developed the packet filter as a Linux kernel module. To capture packets before they exit the network, we set up our untrusted VMs to use a virtual network interface bridged to the trusted VM. We inserted a hook into a bridge-based packet filtering framework called *ebtables* [4] to view packets crossing the bridge. Whenever the sensor’s kernel component receives a TCP SYN packet from the hook, it notifies the userspace component to perform introspection.

### 5.2 Introspection

The network attribution sensor identifies local processes that are the endpoints of network flows via virtual machine introspection. This requires the sensor’s userspace component to inspect the runtime state of the victim system’s kernel state. Unfortunately, Windows does not store network port and process name



**Fig. 4.** Handle resolution in Windows converts a 32-bit handle identifier into a structure for the object referenced by the handle. Resolution operates in a manner similar to physical address resolution via page tables.

information in a single structure. A network driver (`tcpip.sys`) manages network connection related information. To locate the data structure corresponding to `tcpip.sys`, the sensor’s userspace component iterates across the kernel’s list of loaded drivers to find the structure’s memory address. The driver maintains a pointer to a structure called `TCBTable`, which in turn points to a linked list of objects containing network ports and process IDs for open connections. To convert the process ID to a process name, the component iterates across the guest kernel’s linked list of running processes. Figure 3 illustrates the complete process of resolving a network connection to a host-level process name.

The correlation engine uses VMI across handle tables to identify the names of processes that receive DLL or thread injection from other, potentially malicious, software. The engine knows handle identifiers because the host attribution sensor observes *IN* parameters to the Windows system calls used as part of an injection, and these parameters include handles. All handles used by a process are maintained by the Windows kernel in handle tables, which are structured as shown in Figure 4.

To resolve a handle to a process name, the correlation engine uses the handle to find the corresponding `EPROCESS` data structure in the Windows kernel memory. Since it knows the process ID of an injecting process, the engine can find that process’ handle table. It searches the table for the specific object identifier recorded by the host attribution sensor. As a pleasant side-effect, this inspection of the handle table will additionally reveal the collection of files and registries currently open to the possibly malicious injecting process.

### 5.3 System Call Interpositioning and Parameter Extraction

Pyrenée’s host attribution sensor requires information about system calls used as part of DLL or thread injection. We developed a system call interpositioning framework deployable in Xen; this framework supports inspection of both *IN* and *OUT* system call parameters. An *IN* parameter’s value is passed by the caller of a system call while an *OUT* parameter’s value is filled after the execution of the system call inside the kernel.

Windows XP uses the fast x86 system-call instruction `SYSENTER`. This instruction transfers control to a system-call dispatch routine at an address specified in the `IA32_SYSENTER_EIP` register. Unfortunately, the Intel VTx hardware virtualization design does not allow the execution of `SYSENTER` to cause a VM to exit out to the hypervisor. As a result, our host attribution sensor must forcibly gain execution control at the beginning of a system call. It alters the contents of `IA32_SYSENTER_EIP` to contain a memory address that is not allocated to the guest OS. When a guest application executes `SYSENTER`, execution will fault to the hypervisor, and hence to our code, due to the invalid control-flow target.

Inside the hypervisor, the sensor processes all faults due to its manipulation of the register value. It records the system call number (stored in the `eax` register), and it uses the `edx` register value to locate system-call parameters stored on the kernel stack. The sensor extracts *IN* parameters with a series of guest memory read operations. It uses the `FS` segment selector to find the *Win32 thread information block* (TIB) containing the currently-executing process' ID and thread ID. It then modifies the instruction pointer value to point at the original address of the system-call dispatch routine and re-executes the faulted instruction.

We use a two-step procedure to extract values of *OUT* parameters at system-call return. In the first step, we record the value present in an *OUT* parameter at the beginning of the system call. Since *OUT* parameters are passed by reference, the stored value is a pointer. In order to know when a system call's execution has completed inside the kernel, we modify the return address of an executing thread inside the kernel with a new address that is not assigned to the guest OS. This modification occurs when intercepting the entry of the system call. In the second step, a thread returning to usermode at the completion of a system call will fault due to our manipulation. As before, the hypervisor receives the fault. Pyrenée reads the values of *OUT* parameters, restores the original return address, and re-executes the faulting instruction. By the end of the second step, the host attribution sensor has values for both the *IN* and *OUT* system-call parameters.

#### 5.4 Address Space Construction and Switching

We create isolated address space for untrusted drivers using the Xen hypervisor and the Windows XP 32-bit guest operating system, though our design is general and applicable to other operating systems and hypervisors. We allocate memory for UPT page tables transparent to the guest OS inside the hypervisor. We then map untrusted driver code pages into the UPT and trusted kernel and driver code into the TPT. We mark all untrusted driver code pages in TPT as non-executable and non-writable and mark all trusted code pages in UPT as non-executable, non-writable, and non-readable.

Pyrenée switches between the two address spaces depending upon the execution context. It manipulates the `CR3` register: a hardware register that points to the current page tables used by memory management hardware and inaccessible to any guest OS. When an untrusted driver invokes a kernel API, execution

faults into the hypervisor due the non-executable kernel code in the UPT. Inside the hypervisor, Pyrenée verifies the legitimacy of the control flow by checking whether the entry point into the TPT is valid. If the entry point is valid, it switches the address space by storing the value of `TPT_CR3`, the trusted page table base, into `CR3`. If the entry point is not valid, Pyrenée records this behavior as an attack and raises an alarm. Similarly, control flow transfers from TPT to UPT fault because untrusted driver code pages are marked non-executable inside the TPT. On this fault, Pyrenée switches the address space by storing the untrusted page table base, `UPT_CR3`, in the `CR3` register.

Pyrenée identifies the legitimate entry points into the TPT by finding the kernel and trusted drivers' exported functions. These exported functions' names and addresses are generated from the PDB files available from Microsoft's symbol server. Pyrenée keeps this information in the hypervisor for the host-attribution sensor.

### 5.5 Interception of Driver Loading

Pyrenée requires knowledge of drivers' load addresses to map their code pages into either the UPT or TPT. Since Windows dynamically allocates memory for all drivers, these addresses change. Moreover, Windows uses multiple mechanisms to load drivers. Pyrenée intercepts all driver loading mechanisms. It rewrites the kernel's binary code on driver loading paths automatically at runtime. It modifies the direct call instruction to the `ObInsertObject` kernel function by changing its target to point to a location in the guest which is not assigned to the guest VM; it stores the original target. With this design, during the driver loading process execution faults into the hypervisor. On the fault, Pyrenée extracts the driver's load address securely from the driver object and resumes the execution at the original target location. This design provides complete interpositioning of driver loading.

## 6 Evaluation

We tested our prototype implementation of Pyrenée to evaluate its ability to appropriately identify malicious software on infected systems, its performance, and its avoidance of false positives. To generate alerts notifying the correlation engine of suspicious network activity in our test environment, we ran a network simulator that acted as a network-based IDS.

### 6.1 User-level Malware Identification

We tested Pyrenée's ability to detect process-to-process parasitic behaviors with the recent `Conficker` worm [38]. `Conficker` employs DLL injection to infect benign processes running on the victim system. We executed `Conficker` inside a test VM monitored by Pyrenée and connected to a network overseen by our NIDS simulator. When executed, the worm ran as a process called `rund1132.exe`. The

host attribution sensor recorded DLL injection behavior from `rundll32.exe` targeting specific `svchost` processes.

When our NIDS simulator sent the IP addresses and port numbers for outbound malicious traffic to Pyrenée’s correlation engine, the engine then determined what malicious code on the host was responsible. It searched the network attribution sensor’s data to extract the name of the process bound to the connection’s source port, here `svchost.exe`. It then searched the host attribution sensor’s data and found that `svchost.exe` was the victim of a parasitic DLL injection from `rundll32.exe`. The correlation engine also found the names of other executables infected by the malware, and it generated a complete listing that could be sent to a security administrator.

We repeated these tests with the `Adclicker.BA` trojan and successfully detected its parasitic behavior.

## 6.2 Kernel-level Malware Identification

We evaluated Pyrenée’s ability to detect kernel-level parasitism by testing it with the recent Storm worm [23]. Storm is kernel-level malware that exhibits parasitic behaviors by injecting malicious DLLs into the benign `services.exe` process, causing `services.exe` to launch DDoS attacks. We loaded Storm’s malicious driver in the test VM. Since the driver is untrusted, Pyrenée loaded it into the separate isolated address space. On the execution of the driver’s code, all kernel APIs invoked by the driver were verified and logged by Pyrenée’s host attribution sensor. The sensor found that the driver was performing injection via APCs, and it recorded both the parasitic behavior and the victim process.

When our network simulator flagged the traffic made by `services.exe`, the correlation engine gathered the data collected by the host and network attribution sensors. The network attribution sensor determined `services.exe` to be the end-point of the connection, and the host attribution sensor identified the parasitism of the malicious driver.

## 6.3 Performance

We designed Pyrenée to operate at runtime, so its performance cost on an end user’s system must remain low. We tested our prototype on an Intel Core 2 Quad 2.66 GHz system. We assigned 1 GB of memory to the untrusted Windows XP SP2 VM and 3 GB combined to the Xen hypervisor and the high-privilege Fedora Core 9 VM. We carried out CPU and memory experiments using a Windows benchmark tool called `PassMark Performance Test` [24]. We measured networking overheads using `IBM Page Detailer` [13] and `wget`. Our experiments measured Pyrenée’s overhead during benign operations, during active parasitic attacks, and during the isolation of a heavily-used driver in the UPT. We executed all measurements five times and present here the median values.

**Table 3.** Results of CPU performance tests for unmonitored execution and for Pyrenée’s monitoring with and without parasitic behaviors present; higher absolute measurements are better. Percentages indicate performance loss.

<i>Operations</i>	<i>Unmonitored</i>	<i>Parasitic Behavior</i>			
		<i>Present</i>	<i>%</i>	<i>Absent</i>	<i>%</i>
Integer Math (MOps/sec)	126.5	92.5	26.88	124.8	1.34
Floating Point Math (MOps/sec)	468.4	439.5	6.17	444.3	5.14
Compression (KB/sec)	1500.9	1494.7	0.41	1496.0	0.32
Encryption (MB/sec)	4.21	4.19	0.48	4.20	0.24
String Sorting (Thousand strings/sec)	1103.3	1072.2	2.82	1072.3	2.81

**Table 4.** Results of memory performance tests for unmonitored execution and for Pyrenée’s monitoring with and without parasitic behaviors present; higher absolute measurements are better. Percentages indicate performance loss.

<i>Operations</i>	<i>Unmonitored</i>	<i>Parasitic Behavior</i>			
		<i>Present</i>	<i>%</i>	<i>Absent</i>	<i>%</i>
Allocate Small Block (MB/sec)	2707.4	2322.3	14.22	2704.1	0.12
Write (MB/sec)	1967.0	1931	1.83	1942.9	1.23

First, we measured Pyrenée’s overhead on CPU-bound and memory intensive operations. Tables 3 and 4 list a collection of benchmark measurements for execution in a VM with and without Pyrenée’s monitoring. For executions including Pyrenée, we measured performance both during execution of a DLL injection attack against an unrelated process and during benign system operation. Our system’s performance in the absence of parasitic behavior is excellent and largely reflects the cost of system-call tracing. Experiments including the execution of an injection attack show diminished performance that ranges from inconsequential to a more substantial performance loss of 27%. The additional overhead measured during the attack occurred when Pyrenée’s host sensor identified injection behavior and harvested state information for its log. This overhead is infrequent and occurs only when parasitic behaviors actually occur.

Next, we measured Pyrenée’s performance during network operations. Using the *IBM Page Detailer*, we measured the time to load a complex webpage (<http://www.cnn.com>) that consisted of many objects spread across multiple servers. The page load caused the browser to make numerous network connections—an important test because Pyrenée’s network attribution sensor intercepts each packet and performs introspection on SYN packets. The result, shown in Table 5, demonstrates that the overhead of the network attribution sensor is low. We next executed a network file transfer by hosting a 174 MB file on a local networked server running `thttpd` and then downloading the file over HTTP using `wget` from the untrusted VM. Table 5 shows that Pyrenée incurred less than 3% overhead on the network transfer; we expect that this strong performance is possible because its packet interception design does not require it to queue and delay packets.

**Table 5.** Results of the network performance tests for unmonitored execution and for Pyrenée’s monitoring without parasitic behaviors present; smaller measurements are better. Percentages indicate performance loss.

<i>Operations</i>	<i>Unmonitored</i>	<i>Pyrenée</i>	<i>%</i>
Page Loading (sec)	3.64	3.82	4.95
Network File Copy (sec)	38.00	39.00	2.63

**Table 6.** Effect of isolating the tcpip.sys driver on CPU operations for unmonitored execution and for Pyrenée’s monitoring without parasitic behaviors present; higher measurements are better. Percentages indicate performance loss.

<i>Operations</i>	<i>Unmonitored</i>	<i>Pyrenée</i>	<i>%</i>
Integer Math (MOps/sec)	126.5	122.0	3.55
Floating Point Math (MOps/sec)	468.4	434.8	7.17
Compression (KB/sec)	1500.9	1467.5	2.23
Encryption (MB/sec)	4.21	4.11	2.38
String Sorting (Thousand strings/sec)	1103.3	1060.8	3.85

**Table 7.** Effect of isolating the tcpip.sys driver on memory performance for unmonitored execution and for Pyrenée’s monitoring without parasitic behaviors present; higher measurements are better. Percentages indicate performance loss.

<i>Operations</i>	<i>Unmonitored</i>	<i>Pyrenée</i>	<i>%</i>
Allocate Small Block (MB/sec)	2707.4	2649.8	2.12
Write (MB/sec)	1967.0	1922.0	2.29

**Table 8.** Effect of isolating the tcpip.sys driver on network performance for unmonitored execution and for Pyrenée’s monitoring without parasitic behaviors present; smaller measurements are better. Percentages indicate performance loss.

<i>Operations</i>	<i>Unmonitored</i>	<i>Pyrenée</i>	<i>%</i>
Network File Copy (sec)	38.00	51.00	34.21

Finally, we measured the cost of our driver isolation strategy by isolating a heavily-used driver in the UPT, forcing a high volume of page faults handled by our hypervisor-level code. We isolated the networking driver `tcpip.sys` and repeated our previous CPU, memory, and network performance measurements in the new setting without active parasitic behaviors. We anticipated that CPU and memory overheads would remain similar, but that network operations would experience decreased performance. Tables 6, 7, and 8 provide evidence that our intuition was correct. Given that the moderate performance cost of isolating a driver in the UPT is borne only by operations invoking that driver’s functionality, we believe that it represents a feasible deployment strategy for unknown and untrusted drivers. The clear performance gain to be had by relocating known-benign drivers in the TPT provides an incentive for driver authors to produce verifiably-safe drivers acceptable to a driver-signing authority.

## 6.4 False Positive Analysis

Pyrenée finds malicious code present on an infected system whenever it receives an alert from a NIDS; it does not detect attacks directly on its own. Hence, false positives will be exhibited by Pyrenée only when it identifies a benign processes' binary or a driver as malicious. We see two possible reasons for such behavior.

First, a NIDS may have false positives when distinguishing between benign and malicious traffic, and it may mis-characterize benign traffic as malicious. In this case, when the NIDS sends an alert along with the network-related information, the network attribution sensor will identify the process that is bound to the connection, and the correlation engine will mark that process as malicious. Certainly, this is a false positive. Fortunately, this problem will diminish over time as NIDS' false positive rates decrease [11]. Even in the case of such false positives, Pyrenée helps an administrator meaningfully look into the actual problem by locating the endpoint of the network traffic. We feel that this design is stronger than an alternative that stores a whitelist of benign parasitic applications and considers malicious parasitic behaviors to be those initiated by non-whitelisted applications. The alternative design requires a whitelist that may not be feasible to generate.

Second, Pyrenée could identify a benign process as malicious when a NIDS correctly generates an alert. Absent implementation bugs, this could only be possible if the network attribution sensor or the host attribution sensor collect incorrect information. Benign parasitic behaviors, such as injections caused by debugging, will not appear to be malicious unless the debugged process is using the network in a way that appears to the NIDS as an attack.

## 7 Conclusions

We demonstrated the usefulness of identifying malicious code present on an infected system during attacks. We presented techniques and a prototype system, Pyrenée, for the automatic discovery of unknown malicious code. Pyrenée correlates network-level events to host-level activities with the help of multiple sensors and the correlation engine. When alerted by a NIDS, our system discovered malicious code, even in the presence of parasitic malware, by correlating information gathered from the host and network attribution sensors. Real malware samples showed that Pyrenée correctly identified malicious code. Our performance analysis demonstrated that our solution was suitable for real world deployment.

**Acknowledgment of Support and Disclaimer.** We thank our shepherd, Davide Balzarotti, and our anonymous reviewers for their extremely helpful comments. This material is based upon work supported by National Science Foundation contract number CNS-0845309. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the NSF or the U.S. Government.

## References

1. A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structures invariants. In *ACSAC*, Anaheim, CA, Dec. 2008.
2. M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.
3. M. Christodorescu, R. Sailer, D. Schales, D. Sgandurra, and D. Zamboni. Cloud security is not (just) virtualization security. In *Cloud Computing Security Workshop*, Chicago, IL, Nov. 2009.
4. Community Developers. Ebttables. <http://ebtables.sourceforge.net/>. Last accessed Apr. 15, 2010.
5. A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *ACM CCS*, Alexandria, VA, Oct. 2008.
6. G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, Boston, MA, Dec. 2002.
7. 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.
8. T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, San Diego, CA, Feb. 2003.
9. J. Giffin, S. Jha, and B. Miller. Detecting manipulated remote call streams. In *11<sup>th</sup> USENIX Security Symposium*, San Francisco, CA, Aug. 2002.
10. J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *NDSS*, San Diego, CA, Feb. 2004.
11. G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting malware infection through IDS-driven dialog correlation. In *USENIX Security Symposium*, Boston, MA, Aug. 2007.
12. S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
13. IBM. Ibm page detailer. <http://www.alphaworks.ibm.com/tech/pagedetailer/download>. Last accessed Apr. 15, 2010.
14. X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based ‘out-of-the-box’ semantic view. In *ACM CCS*, Alexandria, VA, Nov. 2007.
15. S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based hidden process detection and identification using Lycosid. In *ACM VEE*, Seattle, WA, Mar. 2008.
16. K. Kasslin. Evolution of kernel-mode malware. [http://igloo.engineeringforfun.com/malwares/Kimmo\\_Kasslin\\_Evolution\\_of\\_kernel\\_mode\\_malware\\_v2.pdf](http://igloo.engineeringforfun.com/malwares/Kimmo_Kasslin_Evolution_of_kernel_mode_malware_v2.pdf). Last accessed Apr. 15, 2010.
17. J. Kephart and W. Arnold. Automatic extraction of computer virus signatures. In *Virus Bulletin*, Jersey, Channel Islands, UK, 1994.
18. G. H. Kim and E. H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *ACM CCS*, Fairfax, VA, Nov. 1994.
19. S. T. King and P. M. Chen. Backtracking intrusions. In *ACM SOSP*, Bolton Landing, NY, Oct. 2003.
20. L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, San Jose, CA, Aug. 2008.

21. L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *RAID*, Boston, MA, Sept. 2008.
22. MSDN. Asynchronous procedure calls. [http://msdn.microsoft.com/en-us/library/ms681951\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms681951(VS.85).aspx). Last accessed Apr. 15, 2010.
23. OffensiveComputing. *Storm Worm Process Injection from the Windows Kernel*. <http://www.offensivecomputing.net/?q=node/661>. Last accessed Apr. 15, 2010.
24. Passmark Software. PassMark Performance Test. <http://www.passmark.com/products/pt.htm>. Last accessed Apr. 15, 2010.
25. V. Paxson. Bro: A system for detecting network intruders in real-time. In *Usenix Security*, San Antonio, TA, Jan. 1998.
26. 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.
27. 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 *USENIX Security Symposium*, Vancouver, BC, Canada, Aug. 2006.
28. N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *ACM CCS*, Alexandria, VA, Nov. 2007.
29. J. Richter. Load your 32-bit DLL into another process's address space using injlib. *Microsoft Systems Journal*, 9(5), May 1994.
30. R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *RAID*, Boston, MA, Sept. 2008.
31. M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of USENIX LISA*, Seattle, WA, Nov. 1999.
32. 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.
33. M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *ACM CCS*, Chicago, IL, Nov. 2009.
34. A. Srivastava, I. Erete, and J. Giffin. Kernel data integrity protection via memory access control. In *Technical Report GT-CS-09-05*, Georgia Institute of Technology, Atlanta, GA, 2009.
35. A. Srivastava and J. Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *RAID*, Boston, MA, Sept. 2008.
36. S. Staniford, V. Paxson, and N. Weaver. How to Own the internet in your spare time. In *USENIX Security Symposium*, San Francisco, CA, Aug. 2002.
37. M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *ACM SOSR*, Bolton Landing, NY, Oct. 2003.
38. ThreatExpert. Conficker/downadup: Memory injection model. <http://blog.threatexpert.com/2009/01/confickerdownadup-memory-injection.html>. Last accessed Apr. 15, 2010.
39. A. Valdes and K. Skinner. Probabilistic alert correlation. In *RAID*, Davis, CA, Oct. 2001.
40. Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *ACM CCS*, Chicago, IL, Nov. 2009.
41. C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5(2), Mar. 2007.