Countering Persistent Kernel Rootkits Through Systematic Hook Discovery

Zhi Wang, Xuxian Jiang
North Carolina State University

Weidong Cui
Microsoft Research

Xinyuan Wang
George Mason University

Abstract. Kernel rootkits, as one of the most elusive types of malware, pose significant challenges for investigation and defense. Among the most notable are *persistent kernel rootkits*, a special type of kernel rootkits that implant persistent kernel hooks to tamper with the kernel execution to hide their presence. To defend against them, an effective approach is to first identify those kernel hooks and then protect them from being manipulated by these rootkits. In this paper, we focus on the first step by proposing a systematic approach to identify those kernel hooks. Our approach is based on two key observations: First, rootkits by design will attempt to hide its presence from *all* running rootkit-detection software including various system utility programs (e.g., *ps* and *ls*). Second, to manipulate OS kernel control-flows, persistent kernel rootkits by their nature will implant kernel hooks on the corresponding kernel-side execution paths invoked by the security programs. In other words, for any persistent kernel rootkit, either it is detectable by a security program or it has to tamper with one of the kernel hooks on the corresponding kernel-side execution path(s) of the security program. As a result, given an authentic security program, we *only* need to monitor and analyze its kernel-side execution paths to identify the related set of kernel hooks that could be potentially hijacked for evasion. We have built a proof-of-concept system called HookMap and evaluated it with a number of Linux utility programs such as *ls*, *ps*, and *netstat* in RedHat Fedora Core 5. Our system found that there exist 35 kernel hooks in the kernel-side execution path of *ls* that can be potentially hijacked for manipulation (e.g., for hiding files). Similarly, there are 85 kernel hooks for *ps* and 51 kernel hooks for *netstat*, which can be respectively hooked for hiding processes and network activities. A manual analysis of eight real-world rootkits shows that our identified kernel hooks cover all those used in them.

1 Introduction

Rootkits have been increasingly adopted by general malware or intruders to hide their presence on or prolong their control of compromised machines. In particular, kernel rootkits, with the unique capability of directly subverting the victim operating system (OS) kernel, have been frequently leveraged to expand the basic OS functionalities with additional (illicit) ones, such as providing unauthorized system backdoor access, gathering personal information (e.g., user keystrokes), escalating the privilege of a malicious process, as well as neutralizing defense mechanisms on the target system.

In this paper, we focus on a special type of kernel rootkits called *persistent kernel rootkits*. Instead of referring to those rootkits that are stored as persistent disk files and will survive machine reboots, the notion of persistent kernel rootkits here (inherited from [14]) represents those rootkits that will make persistent modifications to run-time OS kernel control-flow, so that normal kernel execution will be somehow hijacked to provide illicit rootkit functionality[1]. For example, many existing rootkits [1, 2] will modify the system call table to hijack the kernel-level control flow. This type of rootkits is of special interest to us for a number of reasons. First, a recent survey [14] of both Windows and Linux kernel rootkits shows that 96% of them are persistent kernel rootkits and they will make persistent control-flow modifications. Second, by running inside the OS kernel, these rootkits have the highest privilege on the system, making them very hard to be detected or removed. In fact, a recent report [3] shows that, once a system is infected by these rootkits, the best way to recover from them is to re-install the OS image. Third, by directly making control-flow modifications, persistent kernel rootkits provide a convenient way to add a rich set of malicious rootkit functionalities.

On the defensive side, one essential step to effectively defending against persistent kernel rootkits is to identify those hooking points (or kernel hooks) that are used by rootkits to regain kernel execution control and then inflict all sorts of manipulations to cloak their presence. The identification of these kernel hooks is useful for not only understanding the hooking mechanism [23] used by rootkits, but also providing better protection of kernel integrity [10, 14, 20]. For example, existing anti-rootkit tools such as [8, 16, 17] all can be benefited because they require the prior knowledge of those kernel hooks to detect the rootkit presence.

To this end, a number of approaches [14, 23] have been proposed. For example, SBCFI [14] analyzes the Linux kernel source code and builds an approximation of kernel control-flow graph that will be followed at run-time by a legitimate kernel. Unfortunately, due to the lack of dynamic run-time information, it is only able to achieve an approximation of kernel control-flow graph. From another perspective, HookFinder [23] is developed to automatically analyze a given malware sample and identify those hooks that are being used by the provided malware. More specifically, HookFinder considers any changes made by the malware as tainted and recognizes a specific change as a hooking point if it eventually redirects the execution control to the tainted attack code. Though effective in identifying specific hooks used by the malware, it cannot discover other hooks that can be equally hijacked but are not being used by the malware.

In this paper, we present a systematic approach that, given a rootkit-detection program, discovers those related kernel hooks that could be potentially used by persistent kernel rootkits to evade from it. Our approach is motivated by the following observation: To hide its presence, a persistent kernel rootkit by design will hide from the given security program and the hiding is achieved by implanting kernel hooks in a number of strategic locations within the kernel-side execution paths of the security program. In other words, for any persistent kernel rootkit, either it is detectable by the security program or it has to tamper with one of the kernel hooks. Therefore, for the purpose of detecting persistent kernel rootkits, it is sufficient to just identify all kernel hooks in the kernel-side execution paths of a given rootkit-detection program.

To identify hooks in the kernel-side execution of a program, we face three main challenges: (1) accurately identifying the right kernel-side execution path for monitoring; (2) obtaining the relevant run-time context information (e.g., the ongoing system call and specific kernel functions) with respect to the identified execution path; (3)

[1] For other types of kernel rootkits that may attack kernel data, they are *not* the focus of this paper and we plan to explore them as future work.

uncovering the kernel hooks in the execution path and extracting associated semantic definition. To effectively address the first two challenges, we developed a context-aware kernel execution monitor and the details will be described in Section 3.1. For the third one, we have built a kernel hook identifier (Section 3.2) that will first locate the run-time virtual address of an uncovered kernel hook and then perform OS-aware semantics resolution to reveal a meaningful definition of the related kernel object or variable.

We have developed a prototype called HookMap on top of a software-based QEMU virtual machine implementation [6]. It is appropriate for two main reasons: First, software-based virtualization allows to conveniently support commodity OSes as guest virtual machines (VMs). And more importantly, given a selected execution path, the virtualization layer can be extended to provide the unique capability in instrumenting and recording its execution without affecting its functionality. Second, since we are dealing with a legitimate OS kernel in a clean system, not with a rootkit sample that may detect the VM environment and alter its behavior accordingly, the use of virtualization software will not affect the results in identifying kernel hooks.

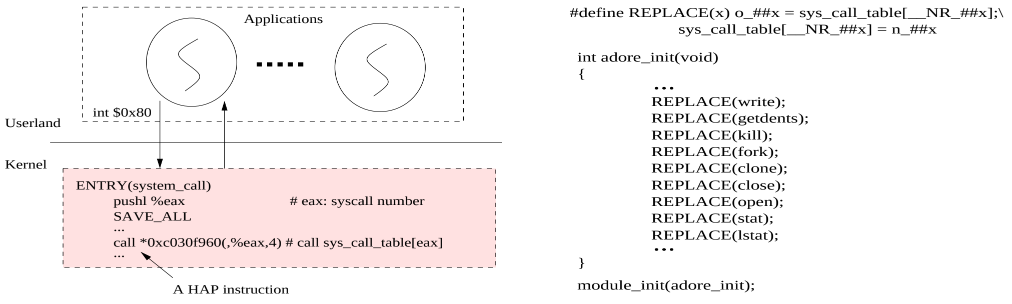
To evaluate the effectiveness of our approach, we ran a default installation of RedHat Fedora Core 5 (with Linux kernel 2.6.15) in our system. Instead of using any commercial rootkit-detection software, we chose to test with three utility programs, *ls*, *ps* and *netstat* since they are often attacked by rootkits to hide files, processes or network connections. By monitoring their kernel-side executions, our system was able to accurately identify their execution contexts, discover all encountered kernel hooks, and then resolve their semantic definitions. In particular, our system identified 35, 85, and 51 kernel hooks, for *ls*, *ps* and *netstat*, respectively. To empirically evaluate the completeness of identified kernel hooks, we performed a manual analysis of eight real-world kernel rootkits and found that the kernel hooks employed by these rootkits are only a small subset of our identified hooks.

The rest of the paper is structured as follows: Section 2 introduces the background on rootkit hooking mechanisms. Section 3 gives an overview of our approach, followed by the description of HookMap implementation in Section 4. Section 5 presents the experimental results and Section 6 discusses some limitations of the proposed approach. Finally, Section 7 surveys related work and Section 8 concludes the paper.

2 Background

In this section, we introduce the hooking mechanisms that are being used by persistent kernel rootkits and define a number of terms that will be used throughout the paper.

There exist two main types of kernel hooks: *code hooks* and *data hooks*. To implant a code hook, a kernel rootkit typically modifies the kernel text so that the execution of the affected text will be directly hijacked. However, since the kernel text section is usually static and can be marked as read-only (or not writable), the way to implant the code hook can be easily detected. Because of that, rootkit authors are now more inclined to implant data hooks at a number of strategic memory locations in the kernel space. Data hooks are usually a part of kernel data that are interpreted as the destination addresses in control-flow transition instructions such as *call* and *jmp*. A typical example of kernel data hook is the system call table that contains the addresses to a number of specific system call service routines (e.g., *sys_open*). In addition, many data hooks may



```
ENTRY(system_call)
       pushl %eax                    # eax: syscall number
       SAVE_ALL
       ...
       call *0xc030f960(,%eax,4) # call sys_call_table[eax]
       ...
```

(a) The system call dispatcher on Linux

```
#define REPLACE(x) o_##x = sys_call_table[__NR_##x];\
                   sys_call_table[__NR_##x] = n_##x

 int adore_init(void)
 {
        ...
             REPLACE(write);
             REPLACE(getdents);
             REPLACE(kill);
             REPLACE(fork);
             REPLACE(clone);
             REPLACE(close);
             REPLACE(open);
             REPLACE(stat);
             REPLACE(lstat);
             ...
 }
 module_init(adore_init);
```

(b) The Linux adore rootkit

Fig. 1. A HAP instruction example inside the Linux system call dispatcher – the associated kernel data hooks have been attacked by various rootkits, including the Linux adore rootkit [1].

contain dynamic content as they are mainly used to hold the run-time addresses of kernel functions and can be later updated because of the loading or unloading of kernel modules. For ease of presentation, we refer to the control-flow transition instructions (i.e., *call* or conditional or un-conditional jumps) whose destination addresses are not hard-coded constants as *hook attach points* (HAPs).

In Figure 1, we show an HAP example with associated kernel data hooks, i.e., the system call table, which is commonly attacked by kernel rootkits. In particular, Figure 1(a) shows the normal system call dispatcher on Linux while Figure 1(b) contains the code snippet of a Linux rootkit – adore [1]. From the control-flow transfer instruction – *call *0xc030f960(,%eax,4)*[2] in Figure 1(a), we can tell the existence of a hook attach point inside the system call dispatcher. In addition, Figure 1(b) reveals that the adore rootkit will replace a number of system call table entries (as data hooks) so that it can intervene and manipulate the execution of those replaced system calls. For instance, the code statement *REPLACE(write)* rewrites the system call table entry *sys_call_table[4]* to intercept the *sys_write* routine before its execution. The corresponding run-time memory location $0xc030f970$ and the associated semantic definition of *sys_call_table[4]* will be identified as a data hook. More specifically, the memory location $0xc030f970$ is calculated as $0xc030f960 + \%eax \times 4$ where $0xc030f960$ is the base address of system call table and $\%eax = 4$ is the actual number for the specific *sys_write* system call. We defer an in-depth analysis of this particular rootkit in Section 5.2.

Meanwhile, as mentioned earlier, there are a number of rootkits that will replace specific instructions (as code hooks) in the system call handler. For instance, the SucKit [19] rootkit will prepare its own version of the system call table and then change the dispatcher so that it will invoke system call routines populated in its own system call table. Using Figure 1(a) as an example, the rootkit will modify the control-flow transfer instruction or more specifically the base address of the system call table $0xc030f960$ to point to a rootkit-controlled system call table. Considering that (1) implanting a code

[2] This instruction is in the standard AT&T assembly syntax, meaning that it will transfer its execution to another memory location pointed to by $0xc030f960 + \%eax \times 4$.

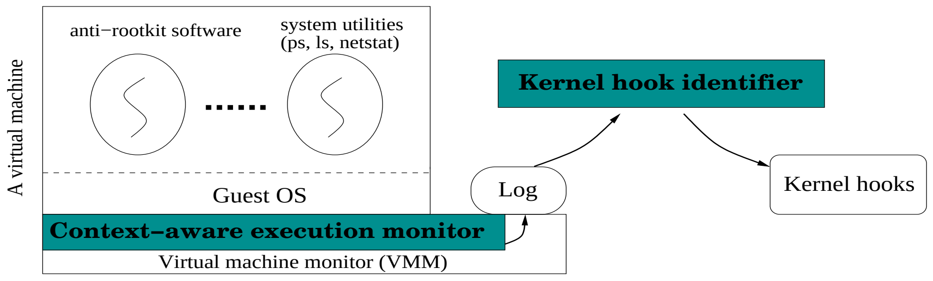

anti−rootkit software
system utilities (ps, ls, netstat)
A virtual machine
Guest OS
Context−aware execution monitor
Virtual machine monitor (VMM)
Log
Kernel hook identifier
Kernel hooks

Fig. 2. A systematic approach to discovering kernel hooks

hook will inflict kernel code modifications, which can be easily detected, and (2) every kernel instruction could be potentially overwritten for code hook purposes, we in this paper focus on the identification of kernel data hooks. Without ambiguity, we use the term kernel hooks to represent kernel data hooks throughout the paper.

Finally, we point out that kernel hooks are elusive to identify because they can be widely scattered across the kernel space and rootkit authors keep surprising us in using new kernel hooks for rootkit purposes [7, 18]. In fact, recent research results [23] show that some stealth rootkits use previously unknown kernel hooks to evade all existing security programs for rootkit detection. In this paper, our goal is to systematically discover all kernel hooks that can be used by persistent kernel rootkits to tamper with and thus hide from a given security program.

3 System Design

The intuition behind our approach is straightforward but effective: a rootkit by nature is programmed to hide itself especially from various security programs including those widely-used system utility programs such as *ps*, *ls*, and *netstat*. As such for an infected OS kernel, the provided kernel service (e.g., handling a particular system call) to any request from these security software is likely manipulated. The manipulation typically comes from the installation of kernel hooks at strategic locations somewhere *within* the corresponding kernel-side execution path of these security software. Based on this insight, if we can develop a system to comprehensively monitor the kernel-side execution of the same set of security programs within a clean system, we can use it to exhaustively uncover all kernel hooks related to the execution path being monitored. Figure 2 shows an architectural overview of our system with two main components: *context-aware execution monitor* and *kernel hook identifier*. In the following, we will describe these two components in detail.

3.1 Context-Aware Execution Monitor

As mentioned earlier, our system is built on top of an open-source virtual machine implementation, which brings the convenient support of commodity OSes as guest VMs. In addition, for a running VM, the *context-aware execution monitor* is further designed to monitor the internal process events including various system calls made by running processes. As suggested by the aforementioned insight, we need to only capture those kernel events related to security software that is running inside the VM. Note that the main purpose of monitoring these events is to understand the right execution context inside the kernel (e.g., “which process is making the system call?”). With that, we can then accurately instrument and record all executed kernel instructions that are relevant to the chosen security software.

However, a challenging part is that modern OS kernels greatly complicate the capture and interpretation of execution contexts with the introduction of “out of order” execution (mainly for improving system concurrency and performance reasons). The “out of order” execution means that the kernel-side execution of any process can be asynchronously interrupted to handle an incoming interrupt request or temporarily context-switched out for the execution of another unrelated process. Notice that the “out of order” execution is considered essential in modern OSes for the support of multi-tasking and asynchronous interrupt handling.

Fortunately, running a commodity OS as a guest VM provides a convenient way to capture those external events [3] that trigger the “out of order” executions in a guest kernel. For example, if an incoming network packet leads to the generation of an interrupt, the interrupt event needs to be emulated by the underlying virtual machine monitor and thus can be intercepted and recorded by our system. The tricky part is to determine when the corresponding interrupt handler ends. For that purpose, we instrument the execution of *iret* instruction to trace when the interrupt handler returns. However, additional complexities are introduced for the built-in support of *nested interrupts* in the modern OS design where an interrupt request (IRQ) of a higher priority is allowed to preempt IRQs of a lower priority. For that, we need to maintain a shadow interrupt stack to track the nested level of an interrupt.

In addition to those external events, the “out of order” execution can also be introduced by some internal events. For example, a running process may voluntarily yield the CPU execution to another process. For that, instead of locating and intercepting all these internal events, we need to take another approach by directly intercepting context switch events occurred inside the monitored VM. The interception of context switch events requires some knowledge of the OS internals. We will describe it in more details in Section 4.

With the above capabilities, we can choose and run a particular security program (or any rootkit-detection tool) inside the monitor. The monitor will record into a local trace file a stream of system calls made by the chosen program and for each system call, a sequence of kernel instructions executed within the system call execution path.

3.2 Kernel Hook Identifier

The context-aware execution monitor will collect a list of kernel instructions that are sequentially executed when handling a system call request from a chosen security program. Given the collected instructions, the kernel hook identifier component is developed to identify those HAPs where kernel hooks are involved. The identification of

[3] Note that the external events here may also include potential debug exceptions caused from hardware-based debugger registers. However, in this work, we do not count those related hooks within the debug interrupt handler.

potential HAPs is relatively straightforward because they are the control-flow transfer instructions, namely those *call* or *jmp* instructions.

Some astute readers may wonder "wouldn't static analysis work for the very same need?" By statically analyzing kernel code, it is indeed capable of identifying those HAPs. Unfortunately, it cannot lead to the identification of the corresponding kernel hooks. There are two main reasons: (1) A HAP may use registers or memory locations to resolve the run-time locations of the related kernel hooks. In other words, the corresponding kernel hook location cannot be determined through static analysis. (An example is already shown in Figure 1(a).) (2) Moreover, there exists another complexity that is introduced by the loadable kernel module (LKM) support in commodity OS kernels. In particular, when a LKM is loaded into the kernel, not only its loading location may be different from previous runs, but also the module text content will be updated accordingly during the time when the module is being loaded. This is mainly due to the existence of certain dependencies of the new loaded module on other loaded modules or the main static kernel text. And we cannot resolve these dependencies until at run-time.

Our analysis shows that for some discovered HAPs, their run-time execution trace can readily reveal the locations of associated kernel hooks. As an example, in the system call dispatcher shown in Figure 1(a), the HAP instruction – *call *0xc030f960(,%eax,4)*, after the execution, will jump to a function which is pointed to from the memory location: $0xc030f960 + \%eax \times 4$, where the value of *%eax* register can be known at run-time. In other words, the result of the calculation at run-time will be counted as a kernel hook in the related execution path. In addition, there also exist other HAPs (e.g., *call *%edx*) that may directly call registers and reveal nothing about kernel hooks but the destination addresses the execution will transfer to. For that, we need to start from the identified HAP and examine in a backwards manner those related instructions to identify the source, which eventually affects the calculated destination value and will then be considered a kernel hook. (The detailed discussion will be presented in Section 4.2.) In our analysis, we also encounter some control-flow transfer instructions whose destination addresses are hardcoded or statically linked inside machine code. In this case, both static analysis and dynamic analysis can be used to identify the corresponding hooks. Note that according to the nature of this type of hooks (Section 2), we consider them as code hooks in this paper.

Finally, after identifying those kernel hooks, we also aim to resolve the memory addresses to the corresponding semantic definitions. For that, we leverage the symbol information available in the raw kernel text file as well as loaded LKMs. More specifically, for main kernel text, we obtain the corresponding symbol information (e.g., object names and related memory locations) from the related *System.map* file. For kernel modules, we derive the corresponding symbol information from the object files (e.g., by running the *nm* command)[4]. If we use Figure 1(a) as an example, in an execution path related to the *sys_open* routine, the hook's memory address is calculated as *0xc030f974*. From the symbol information associated with the main kernel text, that memory ad-

[4] We point out that the *nm* command output will be further updated with the run-time loading address of the corresponding module. For that, we will instrument the module-loading instructions in the kernel to determine the address at run-time.

dress is occupied by the system call table (with the symbol name *sys_call_table*) whose base address is *0xc030f960*. As a result, the corresponding kernel hook is resolved as *sys_call_table[5]*[5] where 5 is actually the system call number for the *sys_open* routine.

4 Implementation

We have built a prototype system called HookMap based on an open-source QEMU $0.9.0$ [6] virtual machine monitor (VMM) implementation. As mentioned earlier, we choose it due to the following considerations: (1) First, since we are dealing with normal OS kernels, the VM environment will not affect the results in the identified kernel hooks; (2) Second, it contains the implementation of a key virtualization technique called dynamic binary translation [6, 4], which can be leveraged and extended to select, record, and disassemble kernel instruction sequences of interest; (3) Third, upon the observation of VM-internal process events, we need to embed our own interpretation logic to extract related execution context information. The open-source nature of the VM implementation provides great convenience and flexibility in making our implementation possible. Also, due to the need of obtaining run-time symbols for semantic resolution, our current system only supports Linux. Nevertheless, we point out that the principle described here should also be applicable for other software-based VM implementations (e.g., VMware Workstation [4]) and other commodity OSes (e.g., Windows).

4.1 Context-Aware Execution Logging

One main task in our implementation is that, given an executing kernel instruction, we need to accurately understand the current execution context so that we can determine whether the instruction should be monitored and recorded. Note that the execution context here is defined as the system call context the current (kernel) instruction belongs to. To achieve that, we have the need of keeping track of the lifetime of a system call event. Fortunately, the lifetime of a system call event is well defined as the kernel accepts only two standard methods in requesting for a system call service: *int $0x80* and *sysenter*. Since we are running the whole system on top of a binary-translation-capable VMM, we can conveniently intercept these two instructions and then interpret the associated system call arguments accordingly. For this specific task, we leverage an “out-of-the-box” VM monitoring framework called VMscope [11] as it already allows to real-time capture system calls completely outside the VM. What remains to do is to correlate a system call event and the related system call return event to form its lifetime. Interested readers are referred to [11] for more details.

Meanwhile, we also face another challenge caused by the “out-of-order” execution (Section 3). To address that, we monitor relevant external events (e.g., interrupts) as well as internal events (e.g., context switches) to detect run-time changes of the execution context. The main goal here is to avoid the introduction of “noises” – unnecessary kernel executions – into the execution path for monitoring and analysis. Fortunately, with a software-based VM implementation, we are able to intercept all these external events

[5] The calculation is based on the following: $(0xc030f974 - 0xc030f960)/4 = 5$, where 4 represents the number of bytes occupied by a function pointer.

as they need to be eventually emulated by the underlying VMM. However, an interesting part is to handle the nested interrupts scenario where a shadow interrupt stack should be maintained at the VMM layer to keep track of the nested level of the ongoing interrupt. For the internal events, our prototype sets a breakpoint on a kernel function that actually performs context-switching. On Linux, the related function is called *__switch_to* and its location is exported by kernel and can be found in the *System.map* file.

With the above capabilities, our system essentially organizes the kernel instruction execution into a stream of system calls and each system call contains a sequence of kernel instructions executed within this specific context. Furthermore, to facilitate later identification and analysis of kernel hooks, for each kernel instruction in one particular context, we further dump the memory locations as well as registers, if any, involved in this instruction. The additional information is needed for later kernel hook identification, which we describe next.

4.2 Kernel Hook Identification

Based on the collected sequence of kernel instructions, the kernel hook identifier locates and analyzes those control-flow transfer *call* or *jmp* instructions (as HAP instructions) to uncover relevant kernel hooks. As a concrete example, we show in Table 1 a list of identified HAPs, associated system call contexts, as well as those kernel hooks that are obtained by monitoring kernel-side execution of the *ls* command. Note that a (small) subset of those identified kernel hooks have already been used by rootkits for file-hiding purposes (more in Section 5).

As mentioned earlier, for an HAP instruction that will read a memory location and jump to the function pointed by a memory location, we can simply record the memory location as a kernel hook. However, if an HAP instruction directly calls a register (e.g., *call *%edx*), we need to develop an effective scheme to trace back to the source – a kernel hook that determines the value of the register.

We point out that this particular problem is similar to the classic problem addressed by dynamic program slicing [5, 24]: Given an execution history and a variable as the input, the goal of dynamic program slicing is to extract a slice that contains all the instructions in the execution history that affected the value of that variable. As such, for the register involved in an identified HAP instruction, we apply the classic dynamic program slicing algorithm [5] to find out a memory location that is associated with a kernel object (including a global static variable) and whose content determines the register value. To do that, we follow the algorithm by first computing two sets for each related instruction: one is $DEF[i]$ that contains the variable(s) defined by this instruction, and another is $USE[i]$ that includes all variables used by this instruction. Each set can contain an element of either a memory location or a machine register. After that, we then examine backwards to find out the memory location that is occupied by a kernel object and whose content determines the register value. In the following, we

[6] A different version of *ls* can result in the execution of *sys_getdents64* instead of *sys_getdents*, which leads to one variation in the identified kernel hooks – *sys_call_table[220]* instead of *sys_call_table[141]*. A similar scenario also happens when identifying another set of kernel hooks by monitoring the *ps* command (to be shown in Table 2).

Table 1. File-hiding kernel hooks obtained by monitoring the *ls -alR /* command in RedHat Fedora Core 5

execution path	#	Hook Attach Points (HAPs)		Kernel Hooks
		address	instruction	address
sys_write	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[4]
	2	0xc014e5a3	call *0xec(%ecx)	selinux_ops[59]
	3	0xc014e5c9	call *%edi	tty_fops[4]
	4	0xc01c63c6	jmp *0xc02bfb40(,%eax,4)	dummy_con[33]
	5	0xc01fa9d2	call *0xc(%esp)	tty_ldisc_N_TTY.write_chan
	6	0xc01fd4f5	call *0xc8(%ecx)	con_ops[3]
	7	0xc01fd51e	call *0xd0(%edx)	con_ops[5]
	8	0xc01fd5fa	call *%edx	con_ops[4]
	9	0xc01fd605	call *0xc4(%ebx)	con_ops[2]
	10	0xc0204caa	call *0x1c(%ecx)	vga_con[7]
sys_open	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[5]
	2	0xc014f024	call *0xf0(%edx)	selinux_ops[60]
	3	0xc0159677	call *%esi	ext3_dir_inode_operations[13] (ext3.ko)
	4	0xc015969d	call *0xbc(%ebx)	selinux_ops[47]
	5	0xc019ea96	call *0xbc(%ebx)	capability_ops[47]
sys_close	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[6]
	2	0xc014f190	call *%ecx	ext3_dir_operations[14] (ext3.ko)
	3	0xc014f19a	call *0xf4(%edx)	selinux_ops[61]
sys_ioctl	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[54]
	2	0xc015dbcf	call *%esi	tty_fops[8]
	3	0xc015de16	call *0xf8(%ebx)	selinux_ops[62]
	4	0xc01fc5a1	call *%ebx	con_ops[7]
	5	0xc01fc5c9	call *%ebx	tty_ldisc_N_TTY.n_tty_ioctl
sys_mmap2	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[192]
	2	0xc0143e0e	call *0xfc(%ebx)	selinux_ops[63]
	3	0xc0143ebc	call *0x2c(%edx)	selinux_ops[11]
	4	0xc0144460	call *%esi	mm→get_unmapped_area
	5	0xc019dc50	call *0x18(%ecx)	capability_ops[6]
	6	0xc019f5d5	call *0xfc(%ebx)	capability_ops[63]
sys_fstat64	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[197]
	2	0xc0155f33	call *0xc4(%ecx)	selinux_ops[49]
sys_getdents[6]	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[141]
	2	0xc015de80	call *0xec(%ecx)	selinux_ops[59]
	3	0xc015decc	call *0x18(%ebx)	ext3_dir_operations[6] (ext3.ko)
	4	0xc016b711	call *%edx	ext3_dir_inode_operations[3] (ext3.ko)
sys_getdents64	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[220]
	2	0xc015de80	call *0xec(%ecx)	selinux_ops[59]
	3	0xc015decc	call *0x18(%ebx)	ext3_dir_operations[6] (ext3.ko)
	4	0xc016b711	call *%edx	ext3_dir_inode_operations[3] (ext3.ko)
sys_fcntl64	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[221]
	2	0xc015d7a7	call *0x108(%ebx)	selinux_ops[66]

will walk-through the scheme with an example. (For the classic algorithm, interested readers are referred to [5] for more details.)

```
#line machine code       instruction            DEF        USE
====  =============    ====================    ========    =====
i-1 : ...
i+0 : 89 c3              mov    %eax,%ebx        %ebx       %eax
i+1 : 83 ec 04           sub    $0x4,%esp        %esp       %esp
i+2 : 8b 80 c4 00 00 00  mov    0xc4(%eax),%eax  %eax       mem[%eax+0xc4], %eax
i+3 : f6 c2 03           test   $0x3,%dl         eflags     %dl
i+4 : 89 04 24           mov    %eax,(%esp)      mem[esp]   %eax
i+5 : 74 0e              je     c016b713                    eflags
i+6 : 8b 40 24           mov    0x24(%eax),%eax  %eax       mem[%eax+0x24], %eax
i+7 : 8b 50 0c           mov    0xc(%eax),%edx   %edx       mem[%eax+0xc],  %eax
i+8 : 85 d2              test   %edx,%edx        eflags     %edx
i+9 : 74 04              je     c016b713                    eflags
i+10: 89 d8              mov    %ebx,%eax        %eax       %ebx
i+11: ff d2              call   *%edx            %eip       %edx
i+12: ...
```

Fig. 3. Discovering a kernel hook based on dynamic program slicing

Figure 3 shows some sequential kernel instructions[7] of a kernel function *__mark_inode_dirty* that are executed in the *sys_getdent64* context of the *ls* command. In particular, the sequence contains an HAP instruction – *call *%edx* – at the memory location $0xc016b711$ (line $i + 11$ in Figure 3). Note that since we monitor at run-time, we can precisely tell which memory locations/registers are defined and/or used. As a result, we directly derive the corresponding destination address (contained in the *%edx* register), which is $0xc885bca0$ – the entry point of a function *ext3_dirty_inode* within a LKM named *ext3.ko*. Obviously, it is the destination address the HAP instruction will transfer to, *not* the relevant kernel hook. Next, our prototype further expands the associated semantics of every executed instruction i to compute the two sets $DEF[i]$ and $USE[i]$ and the results are shown in Figure 3. With the two sets defined for each instruction, we can then apply the dynamic slicing algorithm. Specifically, from the HAP instruction (line $i + 11$), the USE set contains the *%edx* register, which is defined by the instruction at line $i + 7$. This particular instruction is associated with a USE set having two members: *%eax* and *mem[%eax+0xc]*. It turns out the *%eax* points to the kernel object *ext3_dir_inode_operations* and $0xc$ is an offset from the kernel object. After identifying the responsible kernel object, the slicing algorithm then outputs *ext3_dir_inode_operations[3]* as the corresponding kernel hook and terminates. In Table 1, this is the fourth kernel hook identified in the *sys_getdent64* context. Note that this particular kernel object is a jump table containing a number of function pointers. The offset $0xc$ indicates that it is the fourth member function in the object as each function pointer is four bytes in size. (The first four member functions in the kernel object are in the offsets of $0x0$, $0x4$, $0x8$, and $0xc$, respectively.)

5 Evaluation

In this section, we present the evaluation results. In particular, we conduct two sets of experiments. The first set of experiments (Section 5.1) is to monitor the execution

[7] These instructions are in the AT&T assembly syntax, where source and destination operands, if any, are in the reverse order when compared with the Intel assembly syntax.

of various security programs and identify those kernel hooks that can be potentially hijacked for hiding purposes. The second set of experiments (Section 5.2) is to empirically evaluate those identified hooks by analyzing a number of real-world rootkits and see whether the used kernel hooks are actually a part of the discovered ones.

5.1 Kernel Hooks

In our experiments, we focus on three types of resources that are mainly targeted by rootkits: files, processes, and network connections. To enumerate related kernel hooks, we correspondingly chose three different utility programs – *ls*, *ps*, and *netstat*. These three programs are from the default installation of Red Hat Linux Fedora Core 5 that runs as a guest VM (with $512MB$ memory) on top of our system. Our testing platform was a modest system, a Dell PowerEdge 2950 server with Xeon 3.16Ghz and 4GB memory running Scientific Linux 4.4. As mentioned earlier, the way to choose these programs is based on the intuition that to hide a file (, a process, or a network connection), a persistent kernel rootkit needs to compromise the kernel-side execution of the *ls* (, *ps*, or *netstat*) program.

In our evaluation, we focus on those portions of collected traces that are related to the normal functionality of the security program (e.g., the querying of system states of interest as well as the final result output) and exclude other unrelated ones. For example, if some traces are part of the loading routine that prepares the process memory layout, we consider them not related to the normal functionality of the chosen program and thus simply ignore them. Further, we assume that the chosen security program as well as those dependent libraries are not compromised. Tables 1, 2, and 3 contain our results, including those specific execution contexts of related system calls. Encouragingly, for each encountered HAP instruction, we can always locate the corresponding kernel hook and our manual analysis on Linux kernel source code further confirms that each identified kernel hook is indeed from a meaningful kernel object or data structure.

More specifically, these three tables show that most identified kernel hooks are part of jump tables defined in various kernel objects. In particular, there are three main kernel objects containing a large collection of function pointers that can be hooked for hiding purposes: the system call table *sys_call_table*, the SELinux-related security operations table *selinux_ops*, as well as the capability-based operations table *capability_ops*. There are other kernel hooks that belong to a particular dynamic kernel object. One example is the function pointer *get_unmapped_area* (in the *sys_mmap2* execution path of Table 2) inside the *mm* kernel object that manages the process memory layout. Note that this particular kernel hook cannot be determined by static analysis.

More in-depth analysis also reveals that an HAP instruction executed in different execution contexts can be associated with different kernel hooks. One example is the HAP instruction located in the system call dispatcher (Figure 1(a)) where around 300 system call service routines are called by the same HAP instruction. A kernel hook can

[8] Different versions of *ps* invokes different system calls to list files under a directory. In our evaluation, the 3.2.7 version of ps uses the *sys_getdents* system call while the version 3.2.3 uses another system call – *sys_getdents64*. Both system calls work the same way except one has a kernel hook *sys_call_table[141]* while another has *sys_call_table[220]*.

Table 2. Process-hiding kernel hooks obtained by monitoring the *ps -ef* command in RedHat Fedora Core 5

execution path	# kernel hooks	Details
sys_read	17	sys_call_table[3], selinux_ops[5], selinux_ops[59], capability_ops[5], kern_table[336], timer_pmtmr[2], proc_info_file_operations[2], proc_file_operations[2], proc_sys_file_operations[2], proc_tty_drivers_operations[2], tty_drivers_op[0], tty_drivers_op[1], tty_drivers_op[2], tty_drivers_op[3], proc_inode.op.proc_read, simple_ones[1].read_proc, simple_ones[2].read_proc
sys_write	11	sys_call_table[4], selinux_ops[59], dummy_con[33], tty_fops[4], con_ops[2], con_ops[3], con_ops[4], con_ops[5], vga_con[6], vga_con[7], tty_ldisc_N_TTY.write_chan
sys_open	20	sys_call_table[5], selinux_ops[34], selinux_ops[46], selinux_ops[47], selinux_ops[60], selinux_ops[88], selinux_ops[112], capability_ops[46], capability_ops[47], pid_base_dentry_operations[0], proc_sops[0], proc_sops[2], proc_root_inode_operations[1], proc_dir_inode_operations[1], proc_self_inode_operations[10], proc_sys_file_operations[12] , proc_tgid_base_inode_operations[1], proc_tty_drivers_operations[12], ext3_dir_inode_operations[13] (ext3.ko), ext3_file_operations[12] (ext3.ko)
sys_close	10	sys_call_table[6], selinux_ops[35], selinux_ops[50], selinux_ops[61], pid_dentry_operations[3], proc_dentry_operations[3], proc_tty_drivers_operations[14], proc_sops[1], proc_sops[6], proc_sops[7]
sys_time	2	sys_call_table[13], timer_pmtmr[2]
sys_lseek	2	sys_call_table[19], proc_file_operations[1]
sys_ioctl	5	sys_call_table[54], tty_fops[8], selinux_ops[62], con_ops[7], tty_ldisc_N_TTY.n_tty_ioctl
sys_mprotect	3	sys_call_table[125], selinux_ops[64], capability_ops[64]
sys_getdents[8]	3	sys_call_table[141], selinux_ops[59], proc_root_operations[6]
sys_getdents64	3	sys_call_table[220], selinux_ops[59], proc_root_operations[6]
sys_mmap2	8	sys_call_table[192], selinux_ops[63], selinux_ops[11], capability_ops[6], capability_ops[63], ext3_dir_inode_operations[3] (ext3.ko), ext3_file_operations[11], mm→get_unmapped_area
sys_stat64	16	sys_call_table[195], selinux_ops[34], selinux_ops[46], selinux_ops[47], selinux_ops[49], selinux_ops[88], selinux_ops[112], capability_ops[46], capability_ops[47], ext3_dir_inode_operations[13] (ext3.ko), pid_base_dentry_operations[0], pid_dentry_operations[3], proc_root_inode_operations[1], proc_self_inode_operations[10], proc_sops[0], proc_tgid_base_inode_operations[1]
sys_fstat64	2	sys_call_table[197], selinux_ops[49]
sys_geteuid32	1	sys_call_table[201]
sys_fcntl64	2	sys_call_table[221], selinux_ops[66]

Table 3. Network-hiding kernel hooks obtained by monitoring the *netstat -atp* command in RedHat Fedora Core 5

execution path	# kernel hooks	Details
sys_read	8	sys_call_table[3], selinux_ops[59], seq_ops.start, seq_ops.show, seq_ops.next, seq_ops.stop, proc_tty_drivers_operations[2]
sys_write	12	sys_call_table[4], selinux_ops[59], dummy_con[33], con_ops[2], con_ops[3], con_ops[4], con_ops[5], tty_fops[4], tty_ldisc_N_TTY.write_chan, vga_con[6], vga_con[7], vga_ops[8]
sys_open	19	sys_call_table[5], selinux_ops[34], selinux_ops[35], selinux_ops[47], selinux_ops[50], selinux_ops[60], selinux_ops[61], selinux_ops[112], capability_ops[47], ext3_dir_inode_operations[13] (ext3.ko), pid_dentry_operations[3], proc_root_inode_operations[1], proc_dir_inode_operations[1], proc_sops[0], proc_sops[1], proc_sops[2], proc_sops[6], proc_sops[7], tcp4_seq_fops[12]
sys_close	9	sys_call_table[6], selinux_ops[35], selinux_ops[50], selinux_ops[61], proc_dentry_operations[3], proc_tty_drivers_operations[14], proc_sops[1], proc_sops[6], proc_sops[7],
sys_munmap	2	sys_call_table[91], mm→unmap_area
sys_mmap2	6	sys_call_table[192], selinux_ops[11], selinux_ops[63], capability_ops[6], capability_ops[63], mm→get_unmapped_area
sys_fstat64	2	sys_call_table[197], selinux_ops[49]

also be associated with multiple HAP instructions. This is possible because a function pointer (contained in a kernel hook) can be invoked at multiple locations in a function. One such example is *selinux_ops[47]*, a kernel hook that is invoked a number of times in the *sys_open* execution context of the *ps* command. In addition, we observed many one-to-one mappings between an HAP instruction and its associated kernel hook. Understanding the relationship between HAP instructions and kernel hooks is valuable for real-time accurate enforcement of kernel control-flow integrity [14].

5.2 Case Studies

To empirically evaluate those identified kernel rootkits, we manually analyzed the source code of eight real-world Linux rootkits (Table 4). For each rootkit, we first identified what kernel hooks are hijacked to implement a certain hiding feature and then checked whether they are a part of the results shown in Tables 1, 2, and 3. It is encouraging that for every identified kernel hook[9], there always exists an exact match in our results. In the following, we explain two rootkit experiments in detail:

[9] Our evaluation focuses on those kernel data hooks. As mentioned earlier, for kernel code hooks, they can be scattered over every kernel instruction in the corresponding system call execution path.

Table 4. Kernel hooks used by real-world rootkits (‡ means a code hook)

rootkit	**kernel hooks based on the hiding features**		
	file-hiding	process-hiding	network-hiding
adore	sys_call_table[141] sys_call_table[220]	sys_call_table[141] sys_call_table[220]	sys_call_table[4]
adore-ng	ext3_dir_operations[6]	proc_root_operations[6]	tcp4_seq_fops[12]
hideme.vfs	sys_getdents64‡	proc_root_operations[6]	N/A
override	sys_call_table[220]	sys_call_table[220]	sys_call_table[3]
Synapsys-0.4	sys_call_table[141]	sys_call_table[141]	sys_call_table[4]
Rial	sys_call_table[141]	sys_call_table[141]	sys_call_table[3], sys_call_table[5] sys_call_table[6]
knark	sys_call_table[141] sys_call_table[220]	sys_call_table[141] sys_call_table[220]	sys_call_table[3]
kis-0.9	sys_call_table[141]	sys_call_table[141]	tcp4_seq_fops[12]

The *adore* rootkit This rootkit is distributed in the form of a loadable kernel module. If activated, the rootkit will implant 15 kernel hooks in the system call table by replacing them with its own implementations. Among these 15 hooks, only three of them are responsible for hiding purposes[10]. More specifically, two system call table entries – *sys_getdents* (sys_call_table[141]) and *sys_getdents64* (sys_call_table[220]) – are hijacked for hiding files and processes while another one – *sys_write* (sys_call_table[4]) – is replaced to hide network activities related to backdoor processes protected by the rootkit. A customized user-space program called *ava* is provided to send hiding instructions to the malicious LKM so that certain files or processes of attackers' choices can be hidden. All these three kernel hooks are uncovered by our system, as shown in Tables 1, 2, and 3, respectively.

The *adore-ng* rootkit As the name indicates, this rootkit is a more advanced successor from the previous *adore* rootkit. Instead of directly manipulating the system call table, the *adore-ng* rootkit subverts the jump table of the virtual file system by replacing the directory listing handler routines with its own ones. Such replacement allows it to manipulate the information about the *root* file system as well as the */proc* pseudo-file system to achieve the file-hiding or process-hiding purposes. More specifically, the *readdir* function pointer (*ext3_dir_operations[6]*) in the root file system operations table is hooked for hiding attack files, while the similar function (*proc_root_operations[6]*) in the */proc* file system operations table is hijacked for hiding attack processes. The fact that the kernel hook *ext3_dir_operations[6]* is located in the loadable module space (*ext3.ko*) indicates that this rootkit is more stealthier and these types of kernel hooks are much more difficult to uncover than those kernel hooks at static memory locations (e.g., the system call table). Once again, our system successfully identified these stealth kernel hooks, confirming our observation in Section 1. Further, the comparisons between those

[10] The other 12 hooks are mainly used to provide hidden backdoor accesses. One example is the sys_call_table[6] (sys_close), which is hooked to allow the attacker to escalate the privilege to root without going through the normal authorization process.

hooks used by rootkits (Table 4) and the list of hooks from our system (Tables 1, 2, and 3) indicate that only a small subset of them have been used.

6 Discussion

Our system leverages the nature of persistent kernel rootkits to systematically discover those kernel hooks that can potentially be exploited for hiding purposes. However, as a rootkit may implant other kernel hooks for other non-hiding features as its payload, our current prototype is ineffective in identifying them. However, the prototype can be readily re-targeted to those non-hiding features and apply the same techniques to identify those kernel hooks. Also, our system by design only works for persistent kernel rootkits but could be potentially extended for other types of rootkits as well (e.g,. persistent user-level rootkits).

Our current prototype is developed to identify those kernel hooks related to the execution of a chosen security program, either an anti-rootkit software or a system utility program. However, with different programs as the input, it is likely that different running instances will result in different sets of kernel hooks. Fortunately, for the rootkit author, he faces the challenge in hiding itself from all security programs. As a result, our defense has a unique advantage in only analyzing a single instantiated execution path of a rootkit-detection program. In other words, a persistent kernel rootkit cannot evade its detection if the hijacked kernel hooks are not a part of the corresponding kernel-side execution path. There may exist some "in-the-wild" rootkits that take chances in only evading selected security software. However, in response, we can monitor only those kernel hooks related to an installed security software. As mentioned earlier, to hide from it, persistent kernel rootkits will hijack at least one of these kernel hooks.

Meanwhile, it may be argued that our results from monitoring a running instance of a security program could lead to false positives. However, the fact that these kernel hooks exist in the kernel-side execution path suggest that each one could be equally exploited for hooking purposes. From another perspective, we point out that the scale of our results is manageable since it contains tens, not hundreds, of kernel hooks.

Finally, we point out that our current prototype only considers those kernel objects or variables that may contain kernel hooks of interest to rootkits. However, there also exist other types of kernel data such as non-control data [9] (e.g., the *uid* field in the process control block data structure or the doubly-linked process list), which can be manipulated to contaminate kernel execution. Though they may not be used to implement a persistent kernel rootkit for control-flow modifications, how to extend the current system to effectively address them (e.g., by real-time enforcing kernel control flow integrity [10]) remains as an interesting topic for future work.

7 Related Work

Hook Identification The first area of related work is the identification of kernel hooks exploitable by rootkits for hiding purposes. Particularly, HookFinder [23] analyzes a given rootkit example and reports a list of kernel hooks that are being used by the malware. However, by design, it does not lead to the identification of other kernel hooks that are not being used but could still be potentially exploited for the same hiding purposes. From another perspective, SBCFI [14] performs static analysis of Linux kernel source code and aims to build a kernel control-flow graph that will be followed by a legitimate kernel at run-time. However, the graph is not explicitly associated with those kernel hooks for rootkit hiding purposes. Furthermore, the lack of run-time information could greatly limit its accuracy. In comparison, our system complements them with the unique capability of exhaustively deriving those kernel hooks for a given security program, which could be potentially hijacked by a persistent rootkit to hide from it.

Hook-based Rootkit Detection The second area of related work is the detection of rootkits based on the knowledge of those specific hooking points that may be used by rootkits. For example, existing anti-rootkit tools such as VICE [8], IceSword [16], System Virginity Verifier [17] examine known memory regions occupied by these specific hooking points to detect any illegitimate modification. Our system is designed with a unique focus in uncovering those specific kernel hooks. As a result, they can be naturally combined together to build an integrated rootkit-defense system.

Other Rootkit Defenses There also exist a number of recent efforts [12, 13, 15, 20–22] that defend against rootkits by detecting certain anomalous symptoms likely caused by rootkit infection. For example, The Strider GhostBuster system [21] and VMwatcher [12] apply the notion of cross-view detection to expose any discrepancy caused by stealth rootkits. CoPilot [15] as well as the follow-up work [13] identify rootkits by detecting possible violations in kernel code integrity or semantic constraints among multiple kernel objects. SecVisor [20] aims to prevent unauthorized kernel code from execution in the kernel space. Limbo [22] characterizes a number of run-time features that can best distinguish between legitimate and malicious kernel drivers and then utilizes them to prevent a malicious one from being loaded into the kernel. Our system is complementary to these systems by pinpointing specific kernel hooks that are likely to be chosen by stealth rootkits for manipulation.

8 Conclusion

To effectively counter persistent kernel rootkits, we have presented a systematic approach to uncover those kernel hooks that can be potentially hijacked by them. Our approach is based on the insight that those rootkits by their nature will tamper with the execution of deployed rootkit-detection software. By instrumenting and recording possible control-flow transfer instructions in the kernel-side execution paths related to the deployed security software, we can reliably derive all related kernel hooks. Our experience in building a prototype system as well as the experimental results with real-world rootkits demonstrate the effectiveness of the proposed approach.

9 Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments that helped to improve the presentation of this paper. This work was supported in part by NSF Grants CNS-0716376, CNS-0524286, and CCF-0728771.

References

1. The adore Rootkit. *http://lwn.net/Articles/75990/*.
2. The Hideme Rootkit. *http://www.sophos.com/security/analyses/viruses-and-spyware/trojhidemea.html*.
3. The Strange Decline of Computer Worms. *http://www.theregister.co.uk/2005/03/17/f-secure_websec/print.html*.
4. VMware. *http://www.vmware.com/*.
5. H. Agrawal and J. R. Horgan. Dynamic Program Slicing. *Proceedings of ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, 1990.
6. F. Bellard. QEMU, a Fast and Portable Dynamic Translator. *Proc. of USENIX Annual Technical Conference 2005 (FREENIX Track)*, July 2005.
7. J. Butler. R$\hat{2}$: The Exponential Growth of Rootkit Techniques. *http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Butler.pdf*.
8. J. Butler. VICE 2.0. *http://www.infosecinstitute.com/blog/README_VICE.txt*.
9. S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. Iyer. Non-Control-Data Attacks Are Realistic Threats. *Proc. USENIX Security Symposium*, August 2005.
10. J. B. Grizzard. Towards Self-Healing Systems: Re-Establishing Trust in Compromised Systems. Ph.D. thesis, Georgia Institute of Technology, May 2006.
11. X. Jiang and X. Wang. "Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots. *Proceedings of the 10th Recent Advances in Intrusion Detection (RAID 2007)*, September 2007.
12. X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, October 2007.
13. N. Petroni, T. Fraser, A. Walters, and W. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. *Proc. of the 15th USENIX Security Symposium*, August 2006.
14. N. Petroni and M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. *Proc. of ACM CCS 2007*, October 2007.
15. N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. *Proc. of the 13th USENIX Security Symposium*, August 2004.
16. PJF. IceSword. *http://www.antirootkit.com/software/IceSword.htm, http://pjf.blogcn.com/*.
17. J. Rutkowska. System Virginity Verifier. *http://invisiblethings.org/papers/ hitb05_virginity_verifier.ppt*.
18. J. Rutkowska. Rootkits vs. Stealth by Design Malware. *http://invisiblethings.org/papers/rutkowska_bheurope2006.ppt*.
19. sd. Linux on-the-fly kernel patching without LKM. *Phrack, 11(58):article 7 of 15*, 2001.
20. A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Guarantee Lifetime Kernel Code Integrity for Commodity OSes. In *Proc. of the ACM SOSP 2007*, October 2007.
21. Y. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting Stealth Software with Strider GhostBuster. *Proc. of the 2005 International Conference on Dependable Systems and Networks*, June 2005.
22. Jeffrey Wilhelm and Tzi-cker Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. *Proceedings of the 10th Recent Advances in Intrusion Detection (RAID 2007)*, September 2007.
23. H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. *Proc. of ISOC NDSS 2008*, February 2008.
24. X. Zhang, R. Gupta, and Y. Zhang. Precise Dynamic Slicing Algorithms. *Proc. of the IEEE/ACM International Conference on Software Engineering*, May 2003.