Detecting Kernel-level Rootkits using Data Structure Invariants

Arati Baliga, Vinod Ganapathy, and Liviu Iftode

Abstract—Rootkits affect system security by modifying kernel data structures to achieve a variety of malicious goals. While early rootkits modified control data structures, such as the system call table and values of function pointers, recent work has demonstrated rootkits that maliciously modify *non-control* data. Most prior techniques for rootkit detection have focused solely on detecting control data modifications and, therefore, fail to detect such rootkits.

This article presents a novel technique to detect rootkits that modify both control and non-control data. The main idea is to externally observe the execution of the kernel during an inference phase and hypothesize *invariants* on kernel data structures. A rootkit detection phase uses these invariants as specifications of data structure integrity. During this phase, violation of invariants indicates an infection. We have implemented Gibraltar, a prototype tool that infers kernel data structure invariants and uses them to detect rootkits. Experiments show that Gibraltar can effectively detect previously-known rootkits, including those that modify non-control data structures.

Index Terms—Kernel-level rootkits, non-control data attacks, invariant inference, static and dynamic program analysis.

1 Introduction

Kernel-level rootkits are a form of malicious software that compromise the integrity of the operating system. Such rootkits stealthily modify kernel data structures to achieve a variety of malicious goals, which may include hiding malicious user space objects, installing backdoors, logging keystrokes and disabling firewalls. Recent studies have shown a phenomenal increase in malware that use stealth techniques commonly employed by rootkits. For example, MacAfee Avert Labs [5] reported a 600% increase in the number of rootkits in the three year period from 2004-2006. The most recent list of threat predictions [6], also by MacAfee, contains several recent examples of trojan horses that were used to commit bank fraud. These trojan horses used stealth techniques to hide on a victim's system, disable anti-virus software, prevent signature updates, or include the victim's system into a botnet.

The increase in the number and complexity of rootkits can be attributed to the large and complex attack surface that the kernel presents. The kernel manages several hundred heterogeneous data structures, most of which are critical to its correct operation. A rootkit can subvert kernel integrity by subtly modifying any of these data structures. In particular, kernel data structures that hold control data, such as the system call table, jump tables and function pointers, have long been a popular target for attack by rootkits. However, recent work has demonstrated rootkits that achieve a variety of malicious goals by *modifying non-control data* in the kernel. For example, Petroni *et al.* [34] demonstrated a rootkit that hid malicious user space processes by manipulating linked lists used by the kernel for bookkeeping. In our previous work, we also have demonstrated rootkits that alter non-control data in the kernel [9], such as rootkits that degrade application performance by modifying memory management meta data and those that affect the output of the pseudo random number generator by contaminating entropy pools. Non-control data presents a much larger attack surface than control data, and these rootkits demonstrate the ease with which attackers can subtly modify non-control data structures to subvert the kernel.

To counter rootkits that modify non-control data, Petroni *et al.* [34] proposed a detection architecture in which kernel data structures are periodically compared against a set of *integrity specifications*. These specifications codify semantic properties of kernel data structures; the detection architecture uses specification violation as an indicator of rootkit behavior. While this approach has the advantage of detecting sophisticated rootkits, it also poses a new challenge—that of developing integrity specifications. High-quality specifications can possibly be supplied by a team of experts who have a detailed understanding of kernel data structure semantics. However, commodity operating system kernels typically maintain several hundred, complex data structures. Consequently, specification writers could either fail to supply certain integrity specifications (*e.g.,* because they are unaware of certain specifications) or fail to realize how a rootkit could exploit them.

We propose a novel approach that automatically generates kernel data structure integrity specifications. In our approach, these integrity specifications take the form of *data structure invariants*—properties that must hold for the lifetime of a data structure. The key idea is to monitor the values of kernel data structures during an inference phase in order to hypothesize invariants that are satisfied by these data structures. These invariants can encompass both control and non-control data structures. For example, an invariant could state that the values of elements of the system call table are a constant (an example of a control data invariant). Similarly, an invariant could state that all the elements of the `running-tasks` linked list (used by the Linux kernel for process scheduling) are also elements of the `all-tasks` linked list that is used by the Linux kernel for process accounting (an example of a non-control data

- *All three authors are with the Department of Computer Science, Rutgers University. E-mail:* {`aratib,vinodg,iftode`}*@*`cs.rutgers.edu`

This article is a revised and expanded version of work published in the 24th Annual Computer Security Applications Conference [8].

invariant) [34]. These invariants are then checked during a rootkit detection phase, in which violation of an invariant is assumed to indicate the presence of a rootkit.

To evaluate the proposed approach, we built *Gibraltar*, a rootkit detection tool that automatically infers invariants on kernel data structures. Gibraltar periodically captures snapshots of kernel memory via an external PCI card to reconstruct kernel data structures. It adapts Daikon [19], an invariant inference tool for user-space application programs, to infer invariants on kernel data structures. In experiments with 23 rootkits, including those that modify non-control data, we found that Gibraltar detected *all* rootkits while imposing a runtime monitoring overhead of under 0.5%.

Our experiments demonstrate the feasibility of automated generation of integrity specifications for kernel data structures. The invariants inferred by our approach can serve as the starting point for a team of kernel experts, who can further refine these specifications. However, we found that the automatically-generated invariants were quite precise. For example, during a 42 minute rootkit detection phase, we observed only 82 spurious invariants out of a total of 236,444 automatically-inferred invariants. Nevertheless, the semantic quality of automatically-generated invariants remains unknown. For example, it may be possible for a kernel expert to provide a small set of invariants that encompasses several hundred invariants inferred by our approach. Future work can further explore how the output of our automatic approach compares against invariants written by a kernel expert.

In summary, this article makes the following contributions:

- **Rootkit detection via invariant inference.** It proposes a novel approach that detects rootkits by identifying violations of automatically-inferred kernel data structure invariants. Section 3 presents an overview of our approach, and presents examples of both control and non-control data attacks that were detected in our experiments.
- **Design and implementation of Gibraltar.** Section 4 presents the design and implementation of Gibraltar, a prototype tool that uses the above approach for rootkit detection.
- **Evaluation on real-world rootkits.** Section 5 presents a comprehensive evaluation of Gibraltar on 23 rootkits, which affect both control and non-control data structures. Gibraltar can detect all of them with negligible monitoring overhead.

2 Related Work

The evolution of rootkits and the techniques to detect them has traditionally been an arms race between attackers and defenders. Early rootkits operated by replacing system binaries on disk with infected versions. Rootkits have more recently evolved to infecting the system by modifying kernel code, control data, and non-control data. Gibraltar is the latest in a long list of rootkit detection tools, but is one of only two techniques that can detect malicious modifications of non-control data.

Defending system utilities. Rootkits can hide malicious user-space objects by replacing key system utilities with infected versions. For example, a rootkit can replace the `ps` utility with a version that hides a malicious process from a system administrator. Such rootkits can be detected using a number of prior tools, such as Tripwire [25] and Strider Ghostbuster [10], and several commercial tools (*e.g.,* [1, 14, 45]). Most of these tools operate by comparing the kernel's view of user-space object (*e.g.,* a cryptographic hash of system utilities) against known values. Any inconsistencies are indicative of rootkits. Recent work [11] has aimed to prevent such rootkits by modifying hard disk drives to disallow modifications to critical system utilities unless such changes are authorized, *e.g.,* by physically connecting an authentication token to the hard disk.

In contrast to these techniques, which focus on protecting user-space objects, Gibraltar focuses on rootkits that operate by modifying the kernel. These techniques may be used together with Gibraltar to provide comprehensive protection.

Protecting kernel code and critical data. Modern rootkits operate by directly infecting the operating system kernel. For example, a rootkit can modify kernel code or the system call table to instead execute malicious code. Prior work to detect such rootkits falls under three broad categories:

- *Kernel module validators.* Rootkits often spread as kernel modules that affect kernel code and data after they have been loaded. Such kernel modules can also use techniques such as polymorphism to evade detection by virus scanners. Static analysis of kernel modules (*e.g.,* using symbolic execution [26, 49]) can detect these rootkits, but conservative approximations made by static analysis may result in false alarms. Instead of checking kernel modules, Gibraltar focuses on observing and validating data modifications to kernel memory, including those made by kernel modules.
- *Hardware-based integrity monitors.* The recent addition of trusted platform modules (TPM) to commodity hardware has allowed the development of protocols to verify the integrity of the software stack executing on a remote machine (*e.g.,* [21, 38, 39, 41]). Such techniques can detect certain kinds of rootkits, in particular, those that modify kernel code and critical data structures. However, we are not aware of the use of trusted computing to detect rootkits that modify arbitrary kernel data structures.

Secure co-processors allow remote monitoring of physical memory, and have been used to build rootkit detectors [33, 51]. For example, Co-Pilot uses a co-processor [33] to periodically fetch and ensure the integrity of physical memory pages that contain kernel code and critical data. Gibraltar uses a similar technique (a PCI-card) to periodically fetch snapshots of kernel memory from a target machine, reconstruct data structures in these snapshots and check invariants.

- *Virtual machine introspection.* Virtual machine monitors offer an alternative technique to monitor the integrity of memory pages that contain kernel code and data. In this technique, called virtual machine introspection [7, 22, 27], the virtual machine monitor fetches and forwards memory pages from a guest operating system to a rootkit detector, which can analyze the integrity of these pages. This technique has also been used to *prevent* rootkits from modifying kernel code [36]. Gibraltar can potentially be adapted to use virtual machine introspection.

Attack	Example invariants violated by the attack
(**a**) Entropy pool contamination (§3.1)	poolinfo.tap1 ∈ {26, 103}, poolinfo.tap2 ∈ {20, 76}, poolinfo.tap3 ∈ {14, 51}, poolinfo.tap4 ∈ {7, 25}, poolinfo.tap5 == 1
(**b**) Process hiding (§3.2)	run-list ⊆ all-tasks
(**c**) Adding binary formats (§3.3)	LENGTH(formats) == 2
(**d**) Resource wastage (§3.4)	zone_table[1].pages_min==255, zone_table[1].pages_low==510, zone_table[1].pages_high==765
(**e**) Intrinsic denial of service (§3.5)	max_threads == 14, 336
(**f**) Disabling firewalls (§3.6)	nf_hooks[2][1].next.hook == 0xc03295b0
(**g**) Disabling PRNG (§3.7)	random_fops.read == 0xc028bd48, urandom_fops.read == 0xc028bda8
(**h**) Altering real-time clock (§3.8)	rtc_fops->ioctl == 0xc01a39e0
(**i**) Defeating signature scans (§3.9)	kmem_fops->read == 0xc0186e00, mem_fops->read == 0xc0186c40

Fig. 1. Summary of the attacks discussed in Section 3 and kernel data structure invariants violated by these attacks.

Protecting kernel hooks. Rootkits have recently evolved to hijacking control of the kernel by modifying *kernel control data*, such as function pointers. This attack technique, called *hooking*, has spurred research on tools to detect and protect kernel hooks [35, 46, 47, 50]. For example, SBCFI [35] periodically scans hooks in kernel memory and ensures that they point to pre-approved locations, *e.g.,* addresses of exported kernel functions. As discussed in Section 1, Gibraltar infers invariants over both control and non-control data, and can therefore detect rootkits that use hooking to hijack kernel control flow.

Protecting non-control data. Recent research has demonstrated rootkits that affect system security by modifying arbitrary kernel data [9, 34]. In contrast to rootkits that hijack kernel control flow, these rootkits operate by modifying kernel data structures to hide user-space processes [34], affect application performance, or affect the output of the kernel's pseudo random number generator [9]. The rootkit detection tools discussed above do not monitor non-control data structures and therefore cannot detect such rootkits.

Petroni *et al.* [34] first proposed an architecture to detect rootkits that affect non-control data structures. Their architecture periodically compares kernel data structures against integrity specifications that describe semantic properties of kernel data structures. These specifications must hold during normal execution of the kernel. Violation of any of these specifications indicates the presence of a rootkit. Their paper demonstrated this approach by using two sets of specifications (developed manually) to detect two rootkits.

Gibraltar extends the approach developed by Petroni *et al.* by automating the extraction of integrity specifications. It does so by applying automated techniques that observe kernel data structures over a period of time and hypothesize invariants, violations of which are then used in an anomaly detection phase to identify rootkits. In this respect, Gibraltar closely resembles prior work on software engineering aids that use a similar approach to detect programming errors [13, 16, 24].

Recovery. Finally, recent work has proposed techniques to automatically recover from system infections and errors, including rootkits [17, 20, 32, 42]. In contrast to these techniques, Gibraltar only performs detection and cannot recover from rootkit infections.

3 Rootkit Detection via Invariant Inference

This section motivates the use and effectiveness of data structure invariants at detecting rootkits by presenting nine attacks that employ stealth techniques. These attacks either modify non-control kernel data (cf. Attacks 1-5) or modify kernel control data (cf. Attacks 6-9). Although these attacks were implemented using the Linux kernel, similar attacks should also be applicable to other operating systems. Gibraltar successfully detects each of the attacks discussed in this section. Where applicable, we discuss existing tools that can detect each attack.

For each of the attacks presented in this section, we also describe a data structure invariant (automatically inferred by Gibraltar by observing the execution of an uncompromised kernel) that is violated by the attack (see Figure 1). In addition, we also describe the semantic meaning of each invariant, *i.e.,* the reason why a data structure satisfies the property specified by the invariant in an uncompromised kernel. The invariants listed in this section are examples drawn from several thousand invariants that are automatically inferred by Gibraltar. Particularly noteworthy in the examples below is the heterogeneity of the data structures over which Gibraltar infers invariants. Although these invariants can be examined, interpreted and refined by a security expert, Gibraltar, by default, automatically uses these invariants as specifications of data structure integrity.

3.1 Attack 1: Entropy Pool Contamination

The kernel uses the pseudo random number generator (PRNG) to obtain randomness needed to seed several other security-critical applications. The goal of the entropy pool contamination attack [9] is to contaminate entropy pools and associated polynomials used by the PRNG, so as to degrade the quality of random numbers that it generates.

Attack. The PRNG uses the primary and secondary entropy pools to generate random numbers. The primary pool derives entropy from external events such as keystrokes, mouse movements, disk and network activity. As a request arrives for a random number, the kernel extracts bytes from the primary pool and moves them to the secondary pool. Bytes extracted from the secondary pool are in turn used to provide random numbers to kernel functions and user-level applications.[1]

To ensure that the numbers generated by the PRNG are pseudo random, the contents of the pools are updated using a stirring function each time bytes are extracted from the pools. The stirring function uses a polynomial whose coefficients are

[1] For ease of presentation, we have simplified some details of the attacks presented in Section 3. For full details of each attack, please consult the original references.

specified in five integer fields of a `struct poolinfo` data structure, namely `tap1`, `tap2`, `tap3`, `tap4` and `tap5`. This attack zeroes the coefficients of the polynomial, which renders ineffective part of the algorithm used to extract bytes from the pool. It also writes zeroes constantly into the entropy pools. Consequently, the numbers generated by the PRNG are no longer random.

Invariants. Figure 1(a) shows the invariants that Gibraltar identifies for the coefficients of the polynomial that is used to stir entropy pools in an uncompromised kernel (the `poolinfo` data structure shown in this figure is represented in the kernel by one of `random_state->poolinfo` or `sec_random_state->poolinfo`). The coefficients are initialized upon system startup, and must never be changed during the execution of the kernel. The attack violates these invariants when it zeroes the coefficients of the polynomial. Gibraltar detects this attack when the invariants are violated.

3.2 Attack 2: Process Hiding

The goal of this attack is to hide a malicious user-space process from the system utilities, such as `ps`. The attack operates by modifying the contents of the kernel linked lists used for process accounting and scheduling [2, 34].

Attack. This attack relies on the fact that process accounting utilities, such as `ps`, and the kernel's task scheduler consult different process lists. The process descriptors of all tasks running on a system belong to a linked list called the `all-tasks` list (represented in the kernel by the data structure `init_tasks->next_task`). This list contains process descriptors headed by the first process created on the system. The `all-tasks` list is used by process accounting utilities. In contrast, the scheduler uses a second list, called the `run-list` (represented in the kernel by `run_queue_head->next`), to schedule processes for execution.

The process hiding attack removes the process descriptor of a malicious user-space process from the `all-tasks` list (but not from the `run-list` list). This ensures that the process is not visible to process accounting utilities, but that it will still be scheduled for execution.

Invariant. Figure 1(b) presents the invariant automatically discovered by Gibraltar, which states that all elements of `run-list` must also be elements of the `all-tasks` list. When a rootkit attempts to remove a task from the `all-tasks` list, this invariant is violated, and is therefore detected by Gibraltar. This attack was previously described by Petroni *et al.* [34] as an example of a non-control data attack. They also describe an invariant enforcement tool to detect such attacks. However, in their approach, data structure invariants were supplied manually by a security expert. Gibraltar extends Petroni *et al.*'s work by developing an automated approach to produce data structure invariants.

3.3 Attack 3: Adding Binary Formats

The goal of this attack is to invoke malicious code each time a new process is created on the system [44]. While rootkits typically achieve this form of hooking by modifying kernel control data, such as the system call table, this attack works by inserting a new binary format into the system.

Attack. This rootkit adds a malicious handler to support a new binary format. The binary formats supported by a system are maintained by the kernel in a global linked list called `formats`. The binary handler, specific to a given binary format, is also supplied when a new format is registered.

When a new process is created on the system, the kernel creates the process address space, sets up credentials and in turn calls the function `search_binary_handler`, which is responsible for loading the binary image of the process from the executable file. This function iterates through the `formats` list to look for an appropriate handler for the binary that it is attempting to load. As it traverses this list, it invokes each handler in it. If a handler returns an error code Enoexec, the kernel considers the next handler on the list; it continues to do so until it finds a handler that returns the code Success.

This attack works by inserting a new binary format in the `formats` list and supplying the kernel with a malicious handler that returns the error code Enoexec each time it is invoked. Since the new handler is inserted at the head of the `formats` list, the malicious handler is executed each time a new process is executed.

Invariants. Gibraltar infers the invariant shown in Figure 1(c) on the `formats` list on our system, which has two registered binary formats. The size of the list is constant after the system starts, and changes only when a new binary format is installed. As this attack inserts a new binary format it changes the length of the `formats` list violating the invariant in Figure 1(c); consequently, Gibraltar detects this attack.

3.4 Attack 4: Resource Wastage

This attack creates artificial pressure on the memory subsystem [9], thereby forcing the memory management algorithms to constantly free memory by swapping pages to disk. In spite of the availability of free memory, this memory is not used either by the kernel or by user-space applications. Continuous swapping to disk also affects the performance of the system.

Attack. The kernel's memory management unit ensures that there are always free pages in memory to fulfill allocation requests made both from the kernel and user-space applications. To do so, it employs *memory balancing algorithms*, which use three watermarks to estimate memory pressure, namely, the fields `pages_min`, `pages_low` and `pages_high`, of a `struct zone_struct` data structure (`zone_table[1]`, in Figure 1(d)). When the number of free pages in the system drops below the `pages_low` watermark, the kernel asynchronously swaps unused pages to disk. This process continues until the number of pages reaches the `pages_high` watermark. In contrast, if the number of free pages available drops below the `pages_min` watermark, the kernel synchronously swaps pages to disk.

This attack manipulates the three watermarks and sets their values close to the number of free pages in the system. Consequently, the number of free pages frequently drops below the `pages_min` and `pages_low` watermarks, forcing the kernel to continuously swap pages to disk, thereby creating synthetic memory pressure in the system.

Invariants. Gibraltar identifies the invariants shown in Figure 1(d) for the three watermarks. These values are initialized upon system startup, and typically do not change in an uncompromised kernel. The attack sets the `pages_min`, `pages_low` and `pages_high` watermarks to $210,000$, $215,000$ and $220,000$ respectively. The values of these watermarks are close to 225,280, which is the total number of pages available on our system. Gibraltar detects this attack because the invariants shown in Figure 1(d) are violated.

3.5 Attack 5: Intrinsic Denial of Service

The goal of this attack is to degrade application performance by throttling the number of processing threads that an application can create to perform tasks in parallel. It works by corrupting data used by the `clone` system call in Linux. As a result, tis attack stealthily causes a measured degree of denial of service because resources beyond a certain threshold become temporarily unavailable to applications, which in turn experience a slowdown.

Attack. The kernel relies on the process creation mechanisms to satisfy user requests. In particular, server applications are designed to be multi-process or multi-threaded; they constantly create new processes/threads to service requests obtained from clients. This attack changes the `max_threads` variable used by the clone system call. This variable is used to check if the total number of processes created on the system exceeds the total number of processes that can possibly be created. This check within the `clone` system call is incorporated to curtail fork bombs. The `max_threads` variable is in the static kernel has a default value of $14,336$, which is therefore an upper limit on the total number of processes that can exist on a system. During our attack, the total number of processes existing at the time of attack was 33. The attack changes the value of `max_threads` to 40, thereby severely limiting the number of new processes that can be created on the system. Once the number of processes exceeds 40, subsequent system calls to create new processes receive an error message, indicating the temporary unavailability of the resource. Although applications are typically programmed to handle this error code, they experience a slowdown because this attacks limits their ability to create new processes. This attack resembles an intrinsic denial of service attack, where the service is unable to function at its full capacity.

Invariants. The invariant inferred by Gibraltar on the `max_threads` variable is shown in Figure 1(e). The attack modifies the value of this variable and violates the invariant and is therefore detected by Gibraltar.

3.6 Attack 6: Disabling Firewalls

The goal of this attack is to stealthily disable firewalls installed on the system [9]; a user is unable to determine that firewalls have been disabled using the `iptables` utility. Instead, `iptables` shows the firewall rules that were created for the system, and the firewall appears to be enabled.

Attack. This attack overwrites hooks in the Linux `netfilter` framework, which is a packet filtering framework in the Linux kernel. It provides hooks at multiple points in the networking stack, and was designed for kernel modules to register callbacks for packet filtering, packet mangling and network address translation. The `iptables` command line utility enforces firewall rules through the `netfilter` framework. Pointers to the `netfilter` hooks are stored in a global table called `nf_hooks`. This attack overwrites the hooks for the IP protocol, and instead sets them to point to the attack function, thereby effectively disabling the firewall. The table where the firewall rules are stored is unaltered and therefore displayed by `iptables` when the user manually inspects the firewall.

Invariants. Gibraltar inferred the invariant shown in Figure 1(f) for `netfilter`. The attack overwrites the hook with the attack function, thereby violating the invariant that function pointer `nf_hooks[2][1].next.hook` is a constant.

As this attack modifies kernel function pointers, it can also be detected by SBCFI [35], which automatically extracts and enforces kernel control flow integrity. In fact, function pointer invariants inferred by Gibraltar implicitly determine a control flow integrity policy that is equivalent to SBCFI. However, in contrast to SBCFI, Gibraltar can also detect non-control attacks (*e.g.,* Attacks 1-5).

3.7 Attack 7: Disabling the PRNG

This attack overwrites the addresses of the functions registered with the virtual file system layer by the PRNG [9]. The overwritten values point to functions that always return zero or an attacker-defined sequence when random bytes are requested from the PRNG; the PRNG's functions are never executed.

Attack. The kernel provides two devices `/dev/random` and `/dev/urandom` from which random numbers can be read. The data structures used to register the device functions are `random_fops` and `urandom_fops`, both of which are variables of type `struct file_operations`. These data structures have function pointers to the various functions provided by the PRNG. The attack replaces the genuine function pointers for the `read` function within these data structures. After the attack has infected the kernel, every byte read from the two devices simply returns a zero. The original PRNG functions are never called.

Invariants. The invariants inferred by Gibraltar on our system for the `random_fops` and `urandom_fops` are shown in Figure 1(g). The attack code changes the values of the above two function pointers, violating the invariants. As with Attack 6, this attack can also be detected using SBCFI.

3.8 Attack 8: Altering Real-time Clock Behavior

The real-time clock (RTC) on a system provides the system time and features, such as setting an alarm clock, for scheduled execution of applications at later points in time. This attack alters the behavior of the real time clock so that alarms registered by certain applications, such as anti-virus software and other intrusion detection systems, are never triggered. This disables scheduled virus scans and other defense activities carried out on the system, thus making it vulnerable to attacks.

Attack description. The real time clock in a computer system is powered by a small battery or accumulator and continues to tick even when the system is turned off. It can be programmed to issue periodic interrupts or to issue an interrupt when the clock reaches a certain value. The Linux kernel uses the RTC to retrieve the date and time. The RTC driver provides the device /dev/rtc to applications, which they can use to access the RTC. The system time can be set by an administrator using the clock utility.

The RTC is used by applications that rely on periodic execution of tasks. A classic example of such an application is an anti-virus software. A user typically schedules complete disk scans for viruses and worms when the system is not in use because it is a time-consuming process. For example, a periodic scan of the system might be scheduled to run every Sunday morning at 3:00am. The anti-virus program relies on the RTC to issue an interrupt when the clock reaches this time.

The goal of this attack is to disable the scheduled execution of the anti-virus program. Applications set such an alarm by using the ioctl system call on the /dev/rtc device. This attack works by overwriting the function pointer for the ioctl system call, which is stored within the data structure rtc_fops. The malicious function can selectively disable alarms only for certain applications of interest, such as the anti-virus software, thereby other regular applications function flawlessly. The system continues to be vulnerable to attacks because anti-virus and intrusion detection systems do not run at their scheduled times. This attack makes the system more vulnerable to future attacks.

Invariants. The invariant inferred by Gibraltar on the rtc_fops->ioctl variable is shown in Figure 1(h). Since the attack installs its own ioctl handler, this invariant is violated and the attack is detected. As with Attack 6, this attack can also be detected by SBCFI.

3.9 Attack 9: Defeating In-memory Signature Scans

This attack defeats malware detectors that use in-memory signature scans by providing them with a fake view of memory. The attack achieves this goal by installing malicious read functions for the /dev/kmem and the /dev/mem devices, which provide interfaces for reading and writing to the kernel virtual address space and the system physical memory, respectively.

Attack description. /dev/mem and /dev/kmem are character devices provided by a Linux system that allow read and write access to system memory. Only privileged users are allowed to read or write to these devices. The device /dev/kmem accesses data from the kernel virtual memory, while /dev/mem reads data from the system physical memory. Reads from these devices return the memory contents existing at the respective memory locations, while writes allow patching memory with supplied data. Rootkits have also used the /dev/kmem interface to patch the running kernel (*e.g.,* SucKIT [40]).

The high-level objective of this attack is similar to that of the Shadow Walker rootkit [43], which utilizes the split TLB architecture of the Intel Pentium processor to modify the kernel's page-fault handler to return fake memory contents. However, this attack achieves the same objective by overwriting function pointers registered by the /dev/mem and /dev/kmem devices. These function pointers are stored in the virtual file system layer in the data structure kmem_fops and mem_fops, of type struct file_operations. The malicious handlers for the read function can present a counterfeit view of the memory pages, thus thwarting all detection software that uses these interfaces to scan memory for malware signatures.

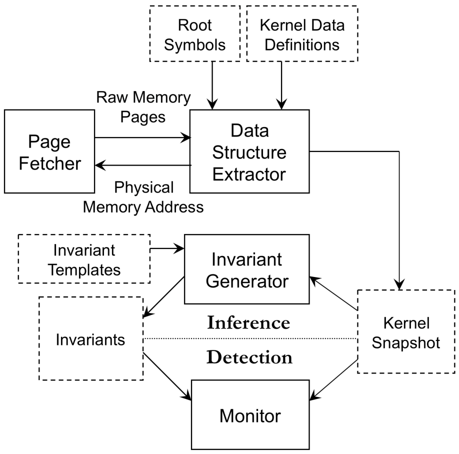


Fig. 2. Design of Gibraltar. Boxes with solid lines show components of Gibraltar, while boxes with dashed lines show the input or output of different components.

Invariants. The /dev/mem and /dev/kmem devices register their respective read function handlers with the Linux VFS layer. The invariants inferred by Gibraltar for the respective data structures are shown in Figure 1(i). When the attack replaces these function handlers to point to its own malicious functions, the invariants shown in Figure 1(i) are violated and hence the attack is detected.

4 Design and Implementation of Gibraltar

Gibraltar must be physically isolated from the monitored system because it aims to detect kernel-level rootkits. In our implementation, Gibraltar executes on a separate machine, called the *observer*, and monitors the execution of the *target* machine. Both the observer and the target are interconnected via a secure back-end network using the Myrinet PCI intelligent network cards [3].[2] The back end network allows Gibraltar to remotely access the target kernel's physical memory, from which it infers data structure invariants. Both coprocessor and VMM-based monitors use similar techniques to read the target's memory. Consequently, Gibraltar can be easily adapted to work with either infrastructure.

Gibraltar operates in two modes, namely, an *inference* mode and a *detection* mode. In the inference mode, Gibraltar infers invariants on data structures of the target's kernel. Inference happens in a controlled environment on an uncompromised target, *e.g.,* a fresh installation of the kernel on the target machine. In the detection mode, Gibraltar checks whether the data structures on the target's kernel satisfy the inferred invariants.

Gibraltar consists of four key components, as shown in Figure 2. The *page fetcher* responds to requests by the *data*

[2] Prior work [37] shows that both PCI and co-processor-based techniques to read the contents of main memory can be bypassed for AMD processors. These attacks operate by inserting illegal entries into the memory map of the memory controller (*i.e.,* the northbridge). In the worst case, attempts to read the contents of memory on a compromised machine can return values chosen by the attacker. Gibraltar can also operate with other techniques that can securely fetch memory pages from the target machine, *e.g.,* VMM-based monitors [22].

structure extractor to fetch kernel memory pages from the target. The data structure extractor, in turn, analyzes raw physical memory pages and extracts values of data structures. To do so, it requires data type definitions of the target's kernel and a set of root symbols that it uses to traverse the target's kernel memory pages. Both these inputs are obtained via an offline static analysis of the source code of the kernel version executing on the target machine. The data structure extractor outputs a (partial) kernel *snapshot*, *i.e.,* the set of kernel data structures in the physical memory pages obtained from the target.[3] The *invariant generator* processes snapshots and hypothesizes likely invariants, which represent properties of individual data structures (*i.e.,* memory objects), as well as collections of data structures. Examples of objects include scalars, such as integer variables or array elements, and fields within aggregate data structures, such as C `struct`s. Collections of data structures represent objects of the same type grouped together *e.g.,* linked lists. During detection, the *monitor* uses the inferred invariants as specifications of kernel data structure integrity. The monitor raises an alert when a kernel data structure invariant is violated. The following sections elaborate on the design of each of these components.

4.1 The Page Fetcher

Gibraltar executes on the observer, which is isolated from the target system. Gibraltar's page fetcher is a component that takes a physical memory address as input, and obtains the corresponding memory page from the target. The target runs a Myrinet PCI card with an enhanced version of the original firmware, which directly interprets and services requests from the page fetcher, without intervention from the target's kernel. For each request, the firmware on the target initiates a DMA request for the requested page, and sends the contents of the physical page to the observer upon completion of the DMA.

4.2 The Data Structure Extractor

The data structure extractor analyzes raw physical memory pages received from the page fetcher and outputs kernel snapshots. It has two key responsibilities: *locating* (Section 4.2.1) and *naming* (Section 4.2.2) data structures.

4.2.1 Locating data structures

The extractor uses a set of *root symbols* and *type definitions* to locate data structures in raw physical memory pages received from the target. Root symbols denote kernel data structures whose physical memory locations are fixed. All data structures on the target's heap are reachable from root symbols. In our implementation for targets running Linux, we used the symbols in the `System.map` file of the target's kernel as the set of roots. Second, it uses a set of *type definitions* of the data structures in the target's kernel. Type definitions are used as described below to recursively identify all reachable data structures. We automatically extracted 1,292 type definitions by analyzing the source code of the target's Linux-2.4.20 kernel using a CIL module [30].

```
Input: (a) $R$: addresses of roots;
(b) Data structure definitions.
Output: Set of all data structures reachable from R.
worklist = $R$;
visited = $\phi$;
snapshot = $\phi$;
while worklist is not empty do
    addr = remove an entry from worklist;
    visited = visited $\cup$ {addr};
    $M$ = physical memory page containing addr;
    obj = object at address addr in $M$;
    snapshot = snapshot $\cup$ value of obj;
    foreach pointer $p$ in obj do
        if $p \notin$ visited
            worklist = worklist $\cup$ {$p$};
return snapshot;
```

Fig. 3. Algorithm used by the data structure extractor.

```
struct task_struct {...
    struct list_head
        CONTAINER(struct task_struct,run_list) run_list;
...}
```

Fig. 4. An example showing the `CONTAINER` annotation. The field `run_list` within the structure `task_struct` points to the `run_list` field of another `task_struct`.

Figure 3 presents the algorithm that the data structure extractor uses to locate data structures in physical memory. The extractor first adds the addresses of the roots to a worklist and issues requests to the page fetcher for memory pages containing the roots. It extracts the values of the roots from these pages, and uses their type definitions to identify pointers to previously unseen data structures. For example, if a root is a C `struct`, the data structure extractor adds all pointer-valued fields of this `struct` to the worklist to locate more data structures in the kernel's physical memory. This process continues in a recursive fashion until the traverser identifies all the data structures target kernel's memory reachable from its roots. A complete set of data structures reachable from the roots is called a *snapshot*. The data structure extractor periodically probes the target and outputs snapshots.

The data structure extractor may require assistance when it encounters certain pointer-valued fields during traversal. A particularly important case is when the traverser encounters pointers to linked lists. In the Linux kernel, linked lists are implemented using the `list_head` data structure. Kernel objects that must be organized as a linked list simply include the `list_head` data structure. Figure 4 presents an example of a `task_struct`, in which the field `run_list` is of type `list_head`. Objects of type `task_struct` are linked together as a list using the `next` and `prev` fields, which are members of the `list_head` structure. The kernel provides functions to add, delete, and traverse `list_head` data structures. To traverse a list of `task_struct` structures, the kernel locates the `list_head` structures within the `task_struct` structure in the list and computes a pointer to the next `task_struct` object in the list using pointer arithmetic. This is a commonly-used idiom in the Linux, and is codified in the `container_of` macro.

[3] The snapshot is partial because our implementation of the data structure extractor currently ignores dynamically-allocated arrays, opaque pointers and untagged unions (see Section 4.2). Moreover, the data structure extractor may encounter inconsistent data structures views because it fetches memory pages in an asynchronous fashion.

We added code annotations to assist the data structure extractor when it encounters pointer-valued fields, such as those in the list_head structure. The CONTAINER annotation, shown in Figure 4, explicitly specifies the type of the container data structure (struct task_struct) and the field within this type (run_list) that the list_head pointers point to. The extractor uses this annotation when it encounters the run_list field, and locates the container task_struct data structure. The CONTAINER annotations therefore disambiguate the semantics of the list_head pointer to the data structure extractor. In our experiments, we annotated all 163 occurrences of the list_head data structure in the Linux-2.4.20 kernel. Gibraltar may also require assistance to disambiguate opaque pointers (void *), dynamically-allocated arrays and untagged unions. For example, the extractor would require the length of a dynamically-allocated array in order to traverse and locate objects in the array. Our current prototype does not support traversal of dynamic arrays, opaque pointers and untagged unions. However, recent work has addressed this shortcoming using pointer analysis of the kernel [12].

The algorithm depicted in Figure 3 may encounter inconsistencies, such as pointers to non-existent objects, as it processes raw physical memory pages. This is because the page fetcher obtains pages from the target asynchronously, *i.e.,* without halting the target. Consequently, operations such as deallocation may invalidate pointers. Such invalid pointers are problematic because the data structure extractor will incorrectly fetch and parse the (invalid) memory region referenced by the pointer. In turn, this memory region may contain values that resemble pointers, which the data structure extractor will recursively follow to identify more spurious objects. To prevent our implementation from extracting a large number of spurious objects, we placed an upper bound on the number of objects traversed by the extractor (as in prior work [35]). In our experiments, we found that on an idle system, the number of data structures in the kernel varies between 40,000 and 65,000 objects. We therefore placed an upper bound of 150,000. The data structure extractor aborts the collection of new objects when this threshold is reached. In our experiments, this threshold was reached only when the system was heavily loaded. On average, the data structure extractor takes about 100 seconds to gather a single kernel snapshot. Note that by placing an upper bound on the number of extracted objects, the data structure extractor may fail to extract certain kernel data structures. In turn, Gibraltar may fail to infer invariants on these data structures (or detect invariant violations).

4.2.2 Naming data structures

Gibraltar uses two schemes to name each data structure extracted from the target's memory. The first scheme assigns a *pathname* to each data structure, which reflects the path from one of the root symbols to the data structure. For example, Gibraltar represents the head of the all-tasks linked list, described in Section 3.2, using the name init_tasks->next_task (here, init_tasks is a root symbol). The second scheme names a data structure using its virtual memory address in the physical memory pages received from the target. This naming scheme is particularly useful during invariant inference, when it helps identify cases where the same name may represent different data structures in multiple snapshots. Such a scenario may arise because of deallocation and reallocation. For example, suppose that the kernel deallocates (and reallocates, at a different address) the head of the all-tasks linked list. As the name init_tasks->next_task will be associated with different virtual memory addresses before and after allocation, it represents different data structures.

Each of these naming schemes has distinct advantages. Pathnames are useful to generate meaningful invariants for data structures whose paths persist across multiple snapshots. They have the important advantage of being portable across machine reboots: the data structure can be identified using its path name. In contrast, names based on virtual memory addresses are not portable across machine reboots. However, virtual memory addresses can be used to express invariants for both persistent and transient data structures; Section 4.5 discusses this issue in further detail.

4.3 The Invariant Generator

During invariant inference, Gibraltar uses the output of the data structure extractor to infer likely data structure invariants. These invariants are used as specifications of data integrity.

We adapted the Daikon [19] tool to infer likely data structure invariants. Daikon is a software engineering aid that infers likely program invariants using dynamic program analysis. Daikon's front-end instruments programs to emit a *trace file* as it executes. Each trace file contains the values of variables at selected *program points*, such as the entry points and exits of functions. Several trace files may be obtained by executing the program on a test suite, and are then fed to Daikon's inference engine, which analyzes these traces to infer likely invariants—properties of variables that hold across all executions of the program. Daikon generates invariants that conform to one of several *templates*. For example, the template x == const checks whether the value of a variable x equals a constant value const, where const represents a symbolic constant. Daikon also infers invariants over *collections* of objects. For example, it may infer that the field bar of all objects of type struct foo has the value 5.

We developed a new front-end to convert kernel snapshots into a format that can be processed by Daikon's inference engine. Our front-end converts each snapshot into the equivalent of a single Daikon trace file. The front-end outputs a variable declaration and observed variable values for each memory object in a snapshot. To record the values of complex objects (such as C structs), we flatten the objects and record the values of each of their fields. Figure 5 presents a simplified example: Figure 5(a) shows the declaration of a variable named proc_fs_type->next (using the pathname-based naming scheme) of type file_system_type. This declaration also contains the names and types of the fields of the data structure; hashcode indicates that the type is a pointer. The type hashcode is intended for unique object identifiers like memory addresses (pointers) or the return value of Java's Object.hashcode method. Figure 5(b) shows the values of each of the fields observed in the snapshot. Daikon's

```
DECLARE
file_system_type::: \          file_system_type::: \
proc_fs_type->next             proc_fs_type->next
name hashcode                  name 3223787403
fs_flags int                   fs_flags 16
read_super hashcode            read_super 3223233888
owner hashcode                 owner 0
next hashcode                  next 3223882668
fs_supers0_next hashcode       fs_supers0_next 3250705524
fs_supers0_prev hashcode       fs_supers0_prev 3250705524
```

(a)Variable declaration. **(b)Variable values.**

Fig. 5. Declaration and values observed for an object of type `file_system_type`.

inference engine hypothesizes likely invariants for this variable by reasoning about its value across multiple snapshots. We use the term *object invariants* to refer to properties of individual memory objects.

Our front-end can also convert snapshots into a format that allows Daikon to infer invariants over collections of objects. We call such invariants *collection invariants*. To do so, the front-end converts each memory snapshot into the equivalent of a Daikon trace file that contains the equivalent of one "program point" for each collection of objects. The front-end records the values of all objects belonging to that collection. The inference engine hypothesizes invariants for all objects in that collection by observing their values across multiple snapshots. In preliminary experiments with Gibraltar, we found that the number of objects in a collection may be overly large, which may in turn cause the invariant inference engine to exhaust available memory on our machine. We avoided this problem by configuring the front-end to split a large collection of objects into smaller sub-collections.

Daikon does not infer properties on linked lists. However, linked lists are ubiquitous in the Linux kernel and can be exploited subtly by rootkits, as demonstrated in Section 3. As done in prior work [18], our front-end therefore linearizes linked lists into arrays, over which Daikon infers invariants. Figure 6 presents examples of four kernel linked lists of type `task_struct`, linearized into arrays. In each of these examples, addresses of the objects in each list are recorded as elements of the array. This representation suffices to infer a restricted family of invariants over linked lists, which we discuss next. In the current prototype of Gibraltar, we restrict ourselves to linked lists with heads in the static data region.

Invariants and Invariant Templates

Daikon infers invariants that conform to 75 different templates [19]. In the discussion below, and in the experimental results reported in Section 5, we focus on five templates. In the templates below, `var` denotes either a scalar variable or a field of a structure.

(**1**) **Membership template (`var` $\in$ {`a, b, c`}).** This template corresponds to invariants that state that `var` only acquires a fixed set of values (in this case, `a`, `b` or `c`). If this set is a singleton {`a`}, denoting that `var` is a constant, then Daikon expresses the invariant as `var == a`.

(**2**) **Non-zero template (`var != 0`).** The non-zero template corresponds to invariants that determine that a `var` is a non-NULL value (or not 0, if `var` is not a pointer).

```
list:::struct task_struct
init_tasks->next_task
[4160716800 3250692096 3250675712 3250667520 3250987008
 3250978816 3250888704 4153163776 4153663488 4155981824
 4154613760 4154474496 4151255040 4150353920 4150247424
 4149018624 4149002240 4148912128 4148871168 4148207616
 4147781632 4151115776 4147806208 4147773440 4147830784
 4147757056 4147675136 4147642368 4147470336 4146806784
 4152762368  3224125440]

init_tasks->prev_task
[4152762368 4146806784 4147470336 4147642368 4147675136
 4147757056 4147830784 4147773440 4147806208 4151115776
 4147781632 4148207616 4148871168 4148912128 4149002240
 4149018624 4150247424 4150353920 4151255040 4154474496
 4154613760 4155351040 4151402496 4155981824 4153663488
 4153163776 3250888704 3250978816 3250987008 3250667520
 3250675712 3250692096 4160716800 3224125440]

init_tasks->thread_group0->next
[3250978816 3250987008 3250667520 3250675712 3224125440]

init_tasks->thread_group0->prev
[3250675712 3224125440 3250978816 3250987008 3250667520]
```

Fig. 6. Output of the front-end to generate linked list invariants on objects of type task_struct.

(**3**) **Bounds template (`var` $\geq$ `const`), (`var` $\leq$ `const`).** This template corresponds to invariants that determine lower and upper bounds of the values that `var` acquires.

The three example templates discussed above correspond to invariants over variables and fields of C `struct` data structures. These invariants can be inferred over individual objects, as well as over collections of data structures, *e.g.,* the fields `bar` of all objects of type `struct foo` have value `5`. Invariants over collections describe a property that hold for *all members* of that collection across *all snapshots*.

(**4**) **Length template (`LENGTH(var) == const`).** This template describes invariants over lengths of linked lists.

(**5**) **Subset template (`coll`$_1$ $\subset$ `coll`$_2$).** This template represents invariants that describe that the collection `coll`$_1$ is a subset of collection `coll`$_2$. This is used, for instance, to represent invariants that describe that every element of one linked list is also an element of another linked list.

The last two example templates are used to describe properties of kernel linked lists. As reported in Section 5, in our experiments, invariants that conformed to the Daikon templates sufficed to detect all the control and non-control data attacks that we tested. However, to accommodate for rootkits that only violate invariants that conform to other kinds of templates, we may need to extend Gibraltar with more templates in the future. Daikon supports an extensible architecture that allows new templates to be supplied, thereby allowing Gibraltar to detect more attacks.

4.4 The Monitor

During detection, the monitor ensures that the data structures in the target's memory satisfy the invariants obtained during inference. As with the invariant generator, the monitor obtains snapshots from the data structure extractor, and checks the data

```
init_fs->root->d_sb->s_dirty.next->i_dentry.next
->d_child.prev->d_inode->i_fop.read == 0xeff9bf60
```

Fig. 7. Example of a transient invariant. The name of the variable changes across reboots.

structures in each snapshot against the invariants. This ensures that any malicious modifications to kernel memory that cause the violation of an invariant are automatically detected.

4.5 Persistent and Transient Invariants

The invariants inferred by Gibraltar can be categorized as either *persistent* or *transient*. Persistent invariants represent properties that are valid across reboots of the target machine, provided that the target's kernel is not reconfigured or recompiled between reboots. An invariant is persistent if and only if the expression that references the object persists across reboots *and* the property represented by the invariant holds across reboots. All the examples presented in Section 3 are persistent invariants.

In contrast, a transient invariant either expresses a property of an object whose pathname does not persist across reboots or represents a property that does not hold across reboots. For example, consider the invariant in Figure 7, which expresses a property of a `struct file_operations` object. This invariant is transient because it does not persist across reboots. The expression that references this object changes across reboots as it appears at different locations in kernel linked lists. Consequently, the number of `next` and `prev`s that appear in the expression differs across reboots.

The distinction between persistent and transient invariants is important because it determines the number of invariants that must be inferred each time the target machine is rebooted. To find the number of persistent invariants inferred by Gibraltar, we repeatedly rebooted the system and executed an inference workload (Section 5.1 presents details of the workload) until the number of persistent invariants remained constant. After eight reboots, we found that the total number of persistent invariants reported as true by Daikon was 26,946. Invariants inferred by Gibraltar over the kernel static area also persist across reboots, and total 209,498 invariants. In contrast, a single run of the inference workload yielded a total of 718,940 invariants for data structures allocated dynamically on the kernel's heap.

Even though the number of persistent invariants is much smaller than the number of transient ones, persistent ones have the advantage that they do not have to be re-learnt after the system is rebooted. Moreover, they sufficed to detect all the rootkits in our test suite. Therefore, the primary focus of our experiments is on persistent invariants.

5 Experimental Results

This section presents the results of experiments to test the effectiveness and performance of Gibraltar at detecting rootkits that modify both control and non-control data structures. We focus on three concerns:

Rootkit/Attack	**Data structures affected**
Rootkits from Packet Storm [4]	
Adore-0.42, All-root, Kbd, Synapsys-0.4 Linspy2, Modhide, Phide, Rial, Kis 0.9 Rkit 1.01, Shtroj2, THC Backdoor	System call table
Adore-ng	Vfs hooks, udp recvmsg
Knark 2.4.3	System call table, proc hooks
Rootkits from research literature [9]	
Disabling firewall (§3.6)	Netfilter hooks
Disabling PRNG (§3.7)	Vfs hooks
Altering real-time clock (§3.8)	Vfs hooks
Defeating signature scans (§3.9)	Vfs hooks

Fig. 8. Linux-based rootkits that modify control data. This table shows the data structures modified by the rootkit. Gibraltar successfully detects all the above rootkits. The invariants violated are all Object invariants, detected by the Membership(constant) template.

- **Detection accuracy.** We tested the effectiveness of Gibraltar by using it to detect both publicly-available rootkits as well as those proposed in the research literature [9, 34, 44]. Gibraltar detected all these rootkits (Section 5.3).
- **Spurious alerts.** When operating in the detection mode, Gibraltar raises an alert when it observes an invariant violation. An alert is spurious if the invariant violation was not as a result of a malicious change to a data structure. In our experiments, we observed that 0.035% of the persistent invariants raised spurious alerts (Section 5.4).
- **Performance.** We measured Gibraltar's performance and found that it imposes a negligible monitoring overhead (Section 5.5).

All the experiments reported in this section were performed on a target system with a Intel Xeon 2.80GHz processor with 1GB RAM, running a Linux-2.4.20 kernel. Infrastructure limitations prevented us from upgrading to the latest version of the Linux kernel. The observer also had an identical configuration.

5.1 Experimental Methodology

Our experiments with Gibraltar were conducted using three workloads, as described below. We ran Gibraltar in the inference mode and executed an *inference workload*, which emulated user behavior on the target system. We then used Gibraltar's invariant generator to hypothesize invariants using the kernel snapshots collected during the execution of the inference workload. The numbers reported in this article are based upon invariants inferred over fifteen kernel snapshots collected by Gibraltar. We also configured Gibraltar to collect thirty kernel snapshots and inferred invariants over these snapshots. However, we observed that fewer than 0.01% of the invariants changed when we used a larger set of kernel snapshots for invariant inference. We then configured Gibraltar to run in the detection mode using the automatically-inferred invariants. We executed a *malicious workload*, comprising 23 rootkits, and studied the effectiveness of Gibraltar at detecting these rootkits. Finally, we studied the number of spurious alerts by executing a *benign workload* on the target. All three workloads are described in detail below.

Attack Name	**Data Structures Affected**	**Invariant Type**	**Template**
Entropy Pool Contamination (§3.1)	struct poolinfo	Collection	Membership
Hidden Process (§3.2)	all-tasks list	Collection	Subset
Linux Binfmt (§3.3)	formats list	Collection	Length
Resource Wastage (§3.4)	struct zone_struct	Object	Membership (constant)
Intrinsic Denial of Service (§3.5)	max_threads	Object	Membership (constant)

Fig. 9. Rootkits that modify non-control data [9, 34, 44]. This table also shows the data structure modified by the attack, the type of the invariant violated by the attack, and the template that this invariant conforms to.

(**1**) **Inference workload.** We chose Lmbench [29], a micro-benchmark suite used to measure operating system performance, as our inference workload. Lmbench measures bandwidth and latency for common operations performed by applications, such as copying to memory, reading cached files, context switching, networking, file system operations, process creation, signal handling and IPC operations. This benchmark exercises several kernel subsystems and therefore modifies several kernel data structures as it executes.

(**2**) **Malicious workload.** This workload comprised 23 rootkits (see Figure 8 and Figure 9). We installed the rootkits one at a time, and determined whether Gibraltar could detect the infection. We uninstalled each rootkit after recording alerts generated by Gibraltar, and then installed the next rootkit from the workload.

(**3**) **Benign workload.** To count the number of spurious alerts, we designed a workload consisting of several benign applications: (1) copying the Linux kernel source code from one directory to another; (2) editing a text document; (3) compiling the Linux kernel; (4) downloading eight video files from the Internet; and (5) performing file system read/write and meta data operations using the IOZone benchmark [31]. This workload ran for 42 minutes on the target, during which time we ran Gibraltar in detection mode using the automatically-inferred invariants. The benign workload contained real application tasks and was chosen to be completely different from the inference workload.

5.2 Invariants

As discussed in Section 4, the invariants inferred by Gibraltar include both properties of both individual objects and collections of objects, *e.g.,* all objects of the same type or all nodes belonging to a linked list. Gibraltar inferred a total of 26,946 persistent invariants on individual objects as well as on collections of objects on the kernel heap. These invariants conformed to the five templates discussed in the previous section; the length and subset invariants apply only to linked lists. Gibraltar inferred 209,498 invariants on the kernel static data region, thereby yielding a total of 236,444 persistent invariants. Gibraltar also inferred 428,046 transient invariants on dynamically-allocated kernel data structures to yield a total of 718,940 invariants.

5.3 Detection Accuracy

We tested Gibraltar's detection accuracy using 23 rootkits, listed in Figure 8. These included rootkits that modify both control and non-control data.

- **Detecting control data modifications.** We used fourteen publicly-available rootkits [4] that modify kernel data structures to test the effectiveness of Gibraltar. We also included four rootkits that have been proposed in the research literature [9] (Attacks 6-9 from Section 3); these rootkits modify kernel function pointers.

Gibraltar was successfully able to detect all the above rootkits. Each of these rootkits violated a persistent invariant that conformed to the template `var == constant`. Because these rootkits modify kernel control flow, they can also be detected by SBCFI. However, as discussed in Section 3, the invariants on control data structures inferred by Gibraltar implicitly determine a control flow integrity policy that is equivalent to SBCFI.

The most common form of invariant violated by publicly available rootkits that modify control data is `var == constant`. This is because these rootkits hijack control of the kernel by overwriting otherwise immutable function pointers. Gibraltar can possibly detect all such rootkits by restricting invariants to the `var == constant` template. Doing so can possibly improve the performance and precision of Gibraltar. However, detecting advanced rootkits that violate properties of non-control data requires more powerful invariants, as discussed at length in Section 3. Consequently, we currently use five templates during invariant inference.

- **Detecting non-control data modifications.** We used five non-control data attacks discussed in prior work [9, 34, 44] to test Gibraltar. These attacks and the invariants that they violate were discussed in detail in Section 3. Figure 9 summarizes these attacks; it shows the data structures modified by the attack, the invariant type (collection/object) violated, and the template that classifies the invariant. Each of the invariants that was violated was a persistent invariant, which survives a reboot of the target machine.

5.4 Spurious Alerts

An invariant violation reported by Gibraltar is spurious if the violation was not caused by a malicious data structure modification. Spurious alerts therefore identify invariants whose precision must be improved using manual refinement or an enhanced inference workload. To measure the number of spurious alerts, we conducted an experiment in which we executed Gibraltar in detection mode using a benign workload on the target system. For this experiment, we configured Gibraltar to detect violations of persistent invariants, numbering 236,444 in total. During the 42-minute duration of this workload, we observed 85 spurious alerts on a total of 82 unique invariants. Note that we directly used the automatically-inferred invariants for detection. Despite this fact, only 0.035% of the persistent invariants contributed to spurious alerts, thereby suggesting that the persistent invariants inferred by Gibraltar are of relatively high precision. Future work on manual refinement of invariants can further improve their precision and reduce the number of spurious alerts.

We also measured the number of spurious alerts by configuring Gibraltar to use both persistent and transient invariants, numbering 718,940 in total. In this case, the number of spurious warnings rose drastically. For the same 42-minute duration, we observed a total of 4,673 spurious alerts (*i.e.,* 0.65% of the invariants resulted in spurious alerts). This experiment suggests that the precision of transient invariants is lower than persistent invariants, and that future work is needed to improve their precision.

5.5 Performance

We measured three aspects of Gibraltar's performance: (1) inference time, *i.e.,* the time taken by Gibraltar to observe the target and infer invariants; (2) detection time, *i.e.,* the time taken for an alert to be raised after the rootkit has been installed; and (3) performance overhead, *i.e.,* the overhead on the target system as a result of periodic page fetches via DMA.

(**1**) **Inference time** is calculated as the cumulative time taken by Gibraltar to gather kernel data structure snapshots and infer invariants when executing in inference mode. Overall, the process of gathering 15 snapshots of the target kernel's memory required approximately 25 minutes, followed by 31 minutes to infer invariants, resulting in a total of 56 minutes. Inference is currently a time-consuming process because our current prototype invokes Daikon to infer invariants after collecting all the kernel snapshots. Inference time can potentially be reduced by invoking Daikon after obtaining each snapshot so that the invariant set is built in parallel as Gibraltar fetches more snapshots.

(**2**) **Detection time** is the interval between the installation of the rootkit and Gibraltar detecting that an invariant has been violated. As Gibraltar traverses the data structures in a snapshot and checks invariants over each data structure, detection time is proportional to the number of objects in each snapshot and the order in which they are encountered by the traversal algorithm. Gibraltar's detection time varied from a minimum of fifteen seconds, when there were 41,254 objects in the snapshot, to a maximum of 132 seconds, when there were 150,000 objects in the snapshot. On average, we observed a detection time of approximately 20 seconds.

(**3**) **Monitoring overhead.** The Myrinet PCI card fetches raw physical memory pages from the target using DMA. Because DMA increases contention on the memory bus, the target's performance will potentially be affected. We measured this overhead using the Stream benchmark [28], a synthetic benchmark that measures sustainable memory bandwidth. Measurement is performed over four vector operations, namely, copy, scale, add and triad and averaged over 100 executions. The vectors were chosen so that they clear the last-level cache in the system, forcing data to be fetched from main memory.

Func.	Time (Monitoring **OFF**)			Time (Monitoring **ON**)			Overhead
	Avg.	Min	Max	Avg.	Min	Max	
Copy	0.2260	0.2259	0.2271	0.2272	0.2269	0.2277	0.48%
Scale	0.2239	0.2237	0.2242	0.2251	0.2248	0.2254	0.49%
Add	0.3316	0.3313	0.3321	0.3329	0.3326	0.3336	0.39%
Triad	0.3295	0.3292	0.3298	0.3308	0.3304	0.3314	0.37%

Fig. 10. Results from the Stream microbenchmark. All numbers reported are across 100 iterations of the benchmark.

Figure 10 presents the bandwidth measurements for these four vector operations, both with Gibraltar's monitoring turned off, and turned on. Bandwidth measurements and time taken for the four vector operations are shown. This figure shows the maximum and minimum time taken for each operation, and the average over 100 executions. As this figure shows, Gibraltar imposes a negligible overhead of 0.49% on the operation of the target system.

6 Limitations

While we are encouraged by Gibraltar's ability to detect rootkits, our current prototype has several limitations, which we plan to remedy in future work.

- **Inconsistent data structures.** Gibraltar fetches pages from the target system in an asynchronous fashion. Consequently, the data structure extractor may encounter inconsistent data structures during traversal. For example, it may encounter a linked list in which the the `next` pointer of a node was updated by the kernel but the `prev` pointer was not yet updated. In such cases, the data structure extractor will traverse stale pointer values and view the resulting data structure as valid. Such inconsistencies introduce noise in the inference process and may result in spurious alerts during detection. In addition to false positives, this noise may also prevent an important set of data structure invariants from being inferred, thereby causing Gibraltar to miss attacks that violate those invariants.

This limitation can potentially be remedied by a proxy driver in the target kernel that notifies and blocks Gibraltar's page fetcher if a data structure is in the process of being modified. For example, this proxy can identify program points (*e.g.,* function entry points and exits) at which it is "safe" to fetch pages, acquire locks on behalf of Gibraltar, and allow the page to be transferred to the observer. It can subsequently release the lock after Gibraltar has acquired a complete snapshot of the target's memory.

- **Portability of transient invariants.** Transient invariants inferred by Gibraltar are not portable, *i.e.,* invariants must be inferred for each target system to be protected by Gibraltar, and must also be inferred each time the system is booted. Two reasons contribute to this limitation. First, the names used to identify objects are not portable. As discussed using the example in Figure 7, several data structures in dynamic regions do not have portable names because the path to these data structures constantly changes as a result of allocations and deallocations. Second, invariants may contain absolute values, such as memory addresses, that may not be the same across different machines or after a reboot. This limitation can possibly be overcome by naming kernel data structures differently and improving the invariant specification language. For example, kernel allocators can possibly be annotated to assign object names, which in turn can be used to uniquely identify objects. Similarly, constant values, such as memory addresses, can be encoded symbolically rather than using their actual values, thereby ensuring that invariants inferred on one system can be applied to other systems.

• **Soundness and completeness of invariants.** As Gibraltar uses a dynamic approach to infer invariants, the resulting invariants are neither sound nor complete. That is, false invariants may potentially be inferred, *e.g.,* because the inference workloads fail to discover all possible values of a data member, leading to spurious alerts during detection. Similarly, it is challenging to infer all possible invariants. This is because the invariant templates used by Daikon may not suffice to capture an important data structure property. While it is challenging to overcome these limitations, prior work has developed techniques to produce inference workloads that improve the accuracy of invariants [23]. Future work can possibly adapt these techniques to improve the kernel data structure invariants used by Gibraltar.

• **Need for source code.** Gibraltar's data structure extractor critically relies on knowing the layout of kernel data structures. We currently obtain this information using static analysis of the kernel's source code. Consequently, Gibraltar cannot be used with closed-source operating systems that lack debug symbols. This limitation can possibly be overcome using static analysis of binary executables, or using recent techniques to infer data structures from memory layouts [15].

• **Inability to detect transient attacks.** Gibraltar's approach of inferring and enforcing invariants restricts it to detecting persistent modifications to kernel data. A rootkit might violate an invariant for a short period of time and revert it between two consecutive snapshots that Gibraltar collects [48]. Such rootkits can possibly be detected by reducing the time required to collect a single snapshot and increasing the frequency at which Gibraltar samples the target's memory.

7 Conclusions and Future Work

Several recent rootkits and research papers have demonstrated that attacks against control and non-control data in dynamically-allocated kernel memory are a realistic and growing threat. Such rootkits are challenging to detect because the vast number and heterogeneity of kernel data structures makes writing correctness specifications for these data structures a challenging exercise. Motivated by this challenge, we sought to develop an automated approach to detect such rootkits.

In this quest, we developed Gibraltar, a tool that detects kernel-level rootkits using data structure invariants. Gibraltar uses an anomaly detection-based approach: it uses an automatic technique to infer invariants on kernel data structures, including both control and non-control data structures, and uses these invariants to detect rootkits. Gibraltar was able to detect all 23 rootkits that we tested it against, while imposing a runtime monitoring overhead of under 0.5%. We also found that the automatically-inferred invariants were quite precise. For example, we observed spurious warnings on only 0.035% of the persistent invariants during a 42 minute testing phase. The approach described in this article therefore demonstrates the feasibility of automatic invariant inference for rootkit detection.

There are a number of avenues to improve Gibraltar. Foremost among these are techniques to address the limitations outlined in Section 6. Additional directions include:

• **Attack analysis.** While Gibraltar successfully detected all 23 rootkits that we tested it against, it may be possible to design a rootkit that compromises the kernel without violating any of Gibraltar's automatically-inferred data structure invariants. Such a rootkit could exploit deficiencies in the coverage provided by Gibraltar's invariants. Future work could develop techniques to estimate the difficulty of creating such rootkits by measuring how much of the kernel's attack surface is covered by Gibraltar's invariants. An alternative approach could be to employ a "red team" to design rootkits that bypass Gibraltar's invariants, *e.g.,* as done by Perkins *et al.* [32].

• **Studying the quality of invariants.** Our study left unexplored the semantic quality of automatically-inferred invariants. For example, it may be possible for a kernel expert to craft a single, high-quality invariant that subsumes thousands of automatically-inferred invariants. Future work could compare the merits of a manual approach against an automated approach to infer kernel data structure invariants.

• **Incremental invariant updates.** As presented, the design of our system requires a new set of invariants to be inferred upon each update to the operating system, *e.g.,* via security patches. However, future enhancements to Gibraltar may make it possible to incrementally update the set of invariants by performing a change-impact analysis on how the updates affects kernel data structures.

Acknowledgments. We thank Joe Kilian for insightful discussions about this work. We also thank the anonymous reviewers of this article for their insightful comments. This work has been supported in part by the NSF under grants 0728937, 0831268, 0915394 and 0931992 and award from the Rutgers DIMACS center.

References

[1] Chkrootkit: Locally checks for rootkits. http://www.chkrootkit.org.
[2] F-Secure rootkit information pages: Fu rootkit. http://www.f-secure.com/v-descs/fu.shtml.
[3] Myricom: Pioneering high performance computing. http://www.myri.com.
[4] Packet storm. http://packetstormsecurity.org/UNIX/penetration/rootkits.
[5] Rootkits, part 1 of 3: A growing threat, April 2006. MacAfee AVERT Labs Whitepaper.
[6] 2010 threat predictions, December 2009. MacAfee AVERT Labs Whitepaper.
[7] A. Baliga, X. Chen, and L. Iftode. Automated containment of rootkit attacks. *Elsevier Computers and Security Journal*, 27:323–334, 2008.
[8] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Annual Computer Security Appl. Conf.*, 2008.
[9] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *IEEE Symp. on Security and Privacy*, 2007.
[10] D. Beck, B. Vo, and C. Verbowski. Detecting stealth software with Strider GhostBuster. In *Intl. Conf. on Dependable Systems and Networks*, 2005.
[11] K. Butler, S. McLaughlin, and P. McDaniel. Rootkit-resistant disks. In *ACM Conf. on Computer and Communications Security*, 2008.
[12] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *ACM Conf. on Computer and Communications Security*, 2009.

[13] T. Chilimbi and V. Ganapathy. HeapMD: Identifying heap-based bugs using anomaly detection. In *Intl. Conf. on Arch. Support for Prog. Lang. and Operating Systems*, 2006.

[14] B. Cogswell and M. Russinovich. RootkitRevealer v1.71: Rootkit detection tool by Microsoft.

[15] A. Cozzie, F. Stratton, H. Xue, and S. King. Digging for data structures. In *ACM/USENIX Symp. Operating System Design and Impl.*, 2008.

[16] C. Csallner and Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *Intl. Symp. on Software Testing and Analysis*, 2006.

[17] B. Demsky, M. Ernst, P. Guo, S. McCamant, J. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *Intl. Symp. on Software Testing and Analysis*, 2006.

[18] M. D. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin. Dynamically discovering program invariants involving collections. Technical Report UW-CSE-99-11-02, University of Washington, 2000.

[19] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. of Comp. Prog.*, 2006.

[20] T. Fraser, M. R. Evenson, and W. A. Arbaugh. VICI: Virtual machine introspection for cognitive immunity. In *Annual Computer Security Appl. Conf.*, 2008.

[21] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM Symp. on Operating System Principles*, 2003.

[22] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distr. Systems Security Symp.*, 2003.

[23] N. Gupta. Generating test data for dynamically discovering likely program invariants. In *Intl. Workshop on Dynamic Analysis*, 2003.

[24] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Intl. Conf. on Software Engineering*, 2002.

[25] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *ACM Conf. on Computer and Communications Security*, 1994.

[26] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Annual Computer Security Appl. Conf.*, 2004.

[27] L. Litty and D. Lie. Manitou: a layer-below approach to fighting malware. In *Arch. and System Support for Improving Software Dependability*, 2006.

[28] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. In *IEEE Technical Committee on Computer Architecture*, 1995.

[29] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conf.*, 1996.

[30] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of C programs. In *Intl. Conf. on Compiler Construction*, 2002.

[31] W. Norcott. IOzone filesystem benchmark. http://www.iozone.org, 2001.

[32] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *ACM Symp. on Operating System Principles*, 2009.

[33] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot: a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symp.*, 2004.

[34] N. L. Petroni, 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 Symp.*, 2006.

[35] N. L. Petroni and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *ACM Conf. on Computer and Communications Security*, 2007.

[36] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Intl. Symp. on Recent Advances in Intrusion Detection*, 2008.

[37] J. Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition, part I: AMD case. In *Blackhat Conf.*, 2007.

[38] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based policy enforcement for remote access. In *ACM Conf. on Computer and Communications Security*, 2004.

[39] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symp.*, 2004.

[40] “sd” and “devik”. Linux on-the-fly kernel patching without LKM: SucKIT rootkit. *Phrack Magazine*, #58, Dec 2001.

[41] E. Shi, A. Perrig, and L. van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *IEEE Symp. on Security and Privacy*, 2005.

[42] S. Sidiroglou, O. Laadan, N. Viennot, C. Perez, A. Keromytis, and J. Neih. ASSURE: Automatic software self-healing using rescue points. In *Intl. Conf. on Arch. Support for Prog. Lang. and Operating Systems*, 2009.

[43] S. Sparks and J. Butler. Shadow walker. *Phrack Magazine*, #63, Jan 2005.

[44] Shellcode Security Research Team. Registration weakness in Linux kernel’s binary formats: Polluting `sys_execve` in kernel space without depending on the `sys_call_table`. http://goodfellas.shellcode.com.ar/own/binfmt-en.pdf, 2006.

[45] Y. Wang, R. Roussev, C. Verbowski, A. Johnson, M. Wu, Y. Huang, and S. Kuo. Gatekeeper: Monitoring auto-start extensibility points (ASEPs) for spyware management. In *USENIX Conf. on System Administration*, 2004.

[46] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *ACM Conf. on Computer and Communications Security*, 2009.

[47] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering persistent kernel rootkits through systematic hook discovery. In *Intl. Symp. on Recent Advances in Intrusion Detection*, 2008.

[48] J. Wei, B. Payne, J. Giffin, and C. Pu. Soft-timer driven transient kernel control flow attacks and defense. In *Annual Computer Security Appl. Conf.*, 2008.

[49] J. Wilhelm and T. Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Intl. Symp. Recent Advances in Intrusion Detection*, 2007.

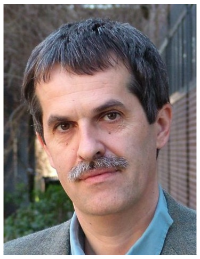
[50] H. Yin, Z. Liang, and D. Song. Hookfinder: Identifying and understanding malware hooking behaviors. In *Network and Distr. System Security Symp.*, 2008.

[51] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure coprocessor-based intrusion detection. In *10th ACM SIGOPS European workshop: beyond the PC*, 2002.

All URLs were last verified on February 3, 2010.

Arati Baliga is a research scientist at the wireless information network laboratory (WINLAB) at Rutgers University. She obtained her Ph.D. degree in Computer Science from Rutgers University in 2009. Her research interests broadly lie in systems security and security in emerging wireless networks.

Vinod Ganapathy is an assistant professor of Computer Science at Rutgers University. He earned a B.Tech. degree in Computer Science & Engineering from IIT Bombay in 2001 and a Ph.D. degree in Computer Science from the University of Wisconsin-Madison in 2007. His research interests are in computer security and reliability.



Liviu Iftode is a professor of Computer Science at Rutgers University. He earned a Ph.D. degree in Computer Science from Princeton University in 1998. His research interests are in operating systems, distributed systems, systems security, mobile and pervasive computing and vehicular computing and networking.