Published in ACSAC'08: Proceedings of the 24th Annual Computer Security Applications Conference, Anaheim, CA, December 2008

Automatic Inference and Enforcement of Kernel Data Structure Invariants

Arati Baliga, Vinod Ganapathy and Liviu Iftode
Department of Computer Science
Rutgers University
{aratib, vinodg, iftode}@cs.rutgers.edu

Abstract

Kernel-level rootkits affect system security by modifying key 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. Prior techniques for rootkit detection fail to identify such rootkits either because they focus solely on detecting control data modifications or because they require elaborate, manually-supplied specifications to detect modifications of non-control data.

This paper presents a novel rootkit detection technique that automatically detects rootkits that modify both control and non-control data. The key idea is to externally observe the execution of the kernel during a training period and hypothesize invariants on kernel data structures. These invariants are used as specifications of data structure integrity during an enforcement phase; violation of these invariants indicates the presence of a rootkit.

We present the design and implementation of Gibraltar, a tool that uses the above approach to infer and enforce invariants. In our experiments, we found that Gibraltar can detect rootkits that modify both control and non-control data structures, and that its false positive rate and monitoring overheads are negligible.

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 and trojan horses, logging keystrokes, disabling firewalls, and including the system into a botnet. Recent studies have shown a phenomenal increase in the number of malware that use stealth techniques commonly employed by rootkits. For example, a report by MacAfee Avert Labs [8] observes a 600% increase in the number of rootkits in the three year period from 2004-2006. Indeed, this trend continues even today; over 200 rootkits were discovered in the first quarter of 2008 alone (according to the forum antirootkit.com [1]).

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 and jump tables, 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.* [25] demonstrate a rootkit that hides a malicious user space process by manipulating linked lists used by the kernel for bookkeeping. Similarly, Baliga *et al.* [10] demonstrate 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. In summary, non-control data presents a much larger attack surface than control data, and these rootkits serve to demonstrate the ease with which attackers can subtly modify non-control data structures to subvert the kernel.

Prior techniques to detect rootkits that modify non-control data [25] have required elaborate specifications of kernel data structure integrity. These specifications are supplied by an expert who has a detailed understanding of kernel data structure semantics. Kernel data structures are continuously monitored during runtime against these specifications, and violations are used as indicators of rootkit behavior. While this approach has the advantage of detecting sophisticated rootkits, developing specifications is currently a manual procedure. Because the kernel maintains several hundred data structures, the specification writer could either fail to supply certain integrity specifications, *e.g.,* because he is unaware that they exist, or may fail to realize how a rootkit could exploit them.

We propose a novel approach based upon *automatic inference of data structure invariants* that can uniformly detect rootkits that modify both control and non-control data. The key idea is to monitor the values of kernel data structures during a training phase, and hypothesize invariants that are satisfied by these data structures. These invariants include properties of both control and non-control data structures, and serve as specifications of data structure integrity. 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 kernel for process scheduling) are also elements of the `all-tasks` linked list that is used by the kernel

for process accounting (an example of a non-control data invariant) [25]. These invariants are then checked during an enforcement phase; violation of an invariant indicates the presence of a rootkit. Because invariants are inferred *automatically* and *uniformly across both control and non-control data structures*, the approach presented in this paper overcomes the shortcomings of prior rootkit detection techniques.

To evaluate the viability of this 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. It uses these snapshots to reconstruct kernel data structures, and adapts Daikon [14], an invariant inference tool for application programs, to infer invariants on kernel data structures. In experiments with twenty rootkits, including those that modify non-control data, we found that Gibraltar detected *all* rootkits with a false positive rate of just 0.65%, and imposed a runtime monitoring overhead of under 0.5%.

In summary, this paper makes the following contributions:

- **Rootkit detection via invariant inference.** We propose 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 it detected in our experiments.
- **Design and implementation of Gibraltar.** We present Gibraltar, a prototype tool that uses the above approach for rootkit detection; Section 4 presents the design and implementation of Gibraltar.
- **Evaluation on real-world rootkits.** We present a comprehensive evaluation of Gibraltar on twenty rootkits that affect both control and non-control data structures and show that Gibraltar can detect all of them, with a low false positive rate and negligible monitoring overhead (Section 5).

2. Related work

The evolution of rootkits and techniques to detect them has traditionally been an arms race between attackers and defenders. Early rootkits operated by replacing system binaries on disk with trojaned versions. These were countered with tools such as Tripwire [18], which checked the authenticity of system files. Tools such as Strider Ghostbuster [11] were developed to detect rootkits based on their hiding behavior by comparing kernel level view with the user level view.

Modern rootkits have evolved to modifying the kernel. Most rootkits either modify kernel code, or data structures that store control data, such as the system call table. Because these rootkits affect the kernel itself, runtime detection tools must execute on an entity that is outside the control of the kernel. Prior work has developed both virtual machine-based (*e.g.,* [16, 21]) and hardware co-processor based infrastructures (*e.g.,* [17, 36]), which allow rootkit detection tools to securely observe the runtime execution state of the kernel. In this paper, we use an external PCI-card to periodically fetch snapshots of kernel memory, which are then processed by Gibraltar. Other rootkits install themselves outside the control of the operating system such as virtual machine based rootkits, which install a virtual machine underneath the operating system [19] or exist independently by using hardware mechanisms to conceal themselves [13]. Such rootkits cannot be detected by monitoring the operating system and therefore need other techniques for detection.

Runtime rootkit detection tools themselves can be implemented in one of several ways. Livewire [16], CoPilot [17] and several commercial tools (*e.g.,* [34, 7, 6, 2]) periodically scan and check the authenticity of kernel code and key control data structures, such as the system call table and jump tables. They typically do so by comparing a cryptographic hash of the memory area containing these data structures against pre-specified values. SBCFI [26] enforces a more sophisticated policy that periodically scans function pointers in kernel memory and ensures that they point to pre-approved locations, *e.g.,* addresses of exported kernel functions.

An alternative to the above runtime techniques are tools that pro actively scan kernel modules and device drivers to determine whether they are malicious. These include both signature-matching techniques as employed by most commercial malware detection tools, and symbolic execution tools [35, 20], which statically approximate the behavior of a kernel module to determine whether it likely affects key kernel data structures. Another alternative is to detect rootkits using attestation-based techniques [32, 31, 30, 15, 29].

In this paper, our focus is on recently-proposed rootkits that affect non-control data structures in the kernel [25, 10]. In contrast to rootkits that hijack kernel control flow, these rootkits operate by modifying kernel data structures to hide user-space processes [25], affect application performance, or affect the output of the kernel's pseudo random number generator [10]. Existing rootkit detection tools (discussed above) do not monitor non-control data structures and therefore cannot detect such rootkits. We believe that as detection tools mature to identify attacks that hijack kernel control flow, rootkits will increasingly evolve to subverting the kernel by modifying non-control data. To detect such rootkits, Petroni *et al.* [25] have proposed a specification-based architecture. In this architecture, data structures in kernel memory are periodically checked against *integrity specifications*. These specifications describe key semantic properties of data structures, which must hold during normal execution of the kernel; violation of any of these specifications indicates the presence of a rootkit. While this technique has the advantage of being able to detect rootkits that modify both control and non-control data, it requires the integrity specifications to be developed manually.

3. Rootkit detection via invariant inference

This section motivates the use and effectiveness of data structure invariants at detecting rootkits by presenting six previously demonstrated attacks that employ stealth techniques [10, 33, 25]. These attacks either modify non-control kernel data (cf. Attacks 1-4) or modify kernel control data (cf. Attacks 5 and 6). Each of these attacks is successfully detected by Gibraltar; where applicable, we also

discuss existing tools that can detect each attack.

For each attack presented below, 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. 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 [10] 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 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 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.

[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.

`poolinfo.tap1`	$\in$	{26, 103}
`poolinfo.tap2`	$\in$	{20, 76}
`poolinfo.tap3`	$\in$	{14, 51}
`poolinfo.tap4`	$\in$	{7, 25}
`poolinfo.tap5`	==	1

Figure 1. *The invariants satisfied by the coefficients of the polynomial used by the stirring function in the PRNG. The coefficients are fields of a* `struct poolinfo` *data structure, shown above as* `poolinfo`*. These invariants are violated by the entropy pool contamination attack (Section 3.1).*

`run-list` $\subseteq$ `all-tasks`

Figure 2. *The invariant that detects the process hiding attack (Section 3.2). In this attack, a task that is not in the* `all-tasks` *linked list appears in the* `run-list` *linked list, which is used by the kernel's task scheduler.*

3.2. Attack 2: Process hiding

The goal of this attack is to hide a (possibly 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 [25, 3].

• **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 linked 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.

• **Invariants.** Figure 2 presents the invariant automatically discovered by Gibraltar. When a rootkit attempts to remove a task from the `all-tasks` list, this invariant is violated, and is therefore detected by Gibraltar. We note that this attack was previously described by Petroni *et al.* [25] as an example of a non-control data attack. They also describe an invariant enforcement tool to detect such attacks; however, in contrast to Gibraltar, their enforcement tool requires data structure invariants, such as the one in Figure 2, to be supplied manually by a security expert.

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 [33]. 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.

```
length(formats) == 2
```

Figure 3. *Invariant inferred on the* `formats` *list; the attack discussed in Section 3.3 modifies the length of this list.*

```
zone_table[1].pages_min == 255
zone_table[1].pages_low == 510
zone_table[1].pages_high == 765
```

Figure 4. *Invariants inferred by Gibraltar for* `zone_table[1]`*, a data structure of type* `struct zone_struct` *(Gibraltar infers similar invariants for other elements of the* `zone_table` *array).*

• **Attack.** This attack operates by introducing a new binary format into the list of formats supported by the system. The handler provided to support this format is malicious in nature. 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.

A new process is created on the system, the kernel creates the process address space, sets up credentials and in 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. Because 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 3 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. Because this attack inserts a new binary format it changes the length of the `formats` list violating the invariant in Figure 3; consequently, Gibraltar detects this attack.

3.4. Attack 4: Resource wastage

This attack creates artificial pressure on the memory subsystem [10], 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 fulfil allocation requests made both from the kernel and user-space applications. To do so, it employs *memory balancing algorithms* that use three watermarks to gauge memory pressure, namely, the fields `pages_min`, `pages_low` and `pages_high`, of a `struct zone_struct` data structure. 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 4 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 is 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 4 are violated.

```
nf_hooks[2][1].next.hook == 0xc03295b0
```

Figure 5. *An invariant inferred for the* `netfilter` *hook. Firewalls are disabled by modifying the function pointer, thereby violating the invariant.*

3.5. Attack 5: Disabling firewalls

The goal of this attack is to stealthily disable firewalls installed on the system [10]; 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 5 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.

Because this attack modifies kernel function pointers, it can also be detected by SBCFI [26], which automatically extracts and enforces kernel control flow integrity. In fact, function pointer invariants inferred by Gibraltar implicitly

```
random_fops.read == 0xc028bd48
urandom_fops.read == 0xc028bda8
```

Figure 6. *Invariants inferred for the PRNG function pointers. These are replaced to point to attacker-specified code, thereby disabling the PRNG.*

determine a control flow integrity policy that is equivalent to SBCFI. However, in contrast to SBCFI, Gibraltar can also detect non-control attacks, such as Attacks 1-4, discussed above.

3.6. Attack 6: Disabling the PRNG

This attack overwrites the addresses of the functions registered with the virtual file system layer by the PRNG [10]. 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 6. The attack code changes the values of the above two function pointers, violating the invariants. As with Attack 5, this attack can also be detected using SBCFI.

4. Design and implementation of Gibraltar

Because Gibraltar aims to detect rootkits, it must execute on an entity that is outside the control of the monitored kernel, such as a coprocessor [17, 36] or inside a separate virtual machine [16]. In our architecture, Gibraltar executes on a separate machine (the *observer*) and monitors the execution of the target machine (the *target*). Both the observer and the target are interconnected via a secure back-end network using the Myrinet PCI intelligent network cards [4] [2]. The back end network allows Gibraltar to remotely access the target kernel's physical memory. Gibraltar is built to infer data structure invariants when supplied with raw kernel memory as input. Since coprocessor and VMM based external monitors use a similar asynchronous monitoring technique to read the target memory, Gibraltar can be easily adapted to work with these infrastructures.

Figure 7 presents the architecture of Gibraltar. It operates in two modes, namely, a *training* mode and an *enforcement* mode. In the training mode, Gibraltar infers invariants on data structures of the target's kernel. Training

[2]Prior work [28] shows that PCI- and co-processor-based monitoring techniques are bypassable. However, Gibraltar operates with any technique that can securely fetch memory pages from the target machine, e.g., VMM-based monitors.

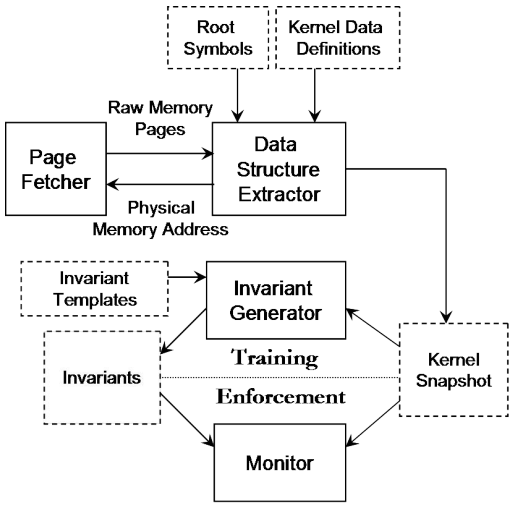

Figure 7. *Boxes with solid lines show components of Gibraltar. Boxes with dashed lines show data used as input or output by the different components.*

happens in a controlled environment on an uncompromised target (*e.g.,* a fresh installation of the kernel on the target machine). In the enforcement mode, Gibraltar ensures that the data structures on the target's kernel satisfy the invariants inferred during the training mode.

As shown in Figure 7, Gibraltar consists of four key components (shown in the boxes with solid lines). The *page fetcher* responds to requests by the *data structure extractor* to fetch kernel memory pages from the target. The data structure extractor, in turn, extracts values of data structures on the target's kernel by analyzing raw physical memory pages. The data structure extractor also accepts as input the data type definitions of the kernel running on the target machine and a set of root symbols that it uses to traverse the target's kernel memory pages. Both these inputs are obtained via an off line analysis of the source code of the kernel version executing on the target machine. The output of the data structure extractor is the set of kernel data structures on the target. The *invariant generator* processes these data structures and infers invariants. These invariants represent properties of both individual data structures, also called *objects*, (*e.g.,* scalars, such as integer variables and arrays and aggregate data structures, such as structs) as well as collections of data structures (*e.g.,* linked lists). During enforcement, the *monitor* uses the invariants as specifications of kernel data structure integrity, which raises an alert when an invariant is violated by a kernel data structure. 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 to which the page fetcher issues a request for a physical memory page. Upon receiving a request, the firmware on the target initiates a DMA

request for the requested page. It sends the contents of the physical page to the observer upon completion of the DMA. The Myrinet card on the target system runs an enhanced version of the original firmware. Our enhancement ensures that when the card receives a request from the page fetcher, the request is directly interpreted by the firmware and serviced.

4.2. The data structure extractor

This component reconstructs snapshots of the target kernel's data structures from raw physical memory pages. The data structure extractor processes raw physical memory pages using two inputs to locate data structures within these pages. First, it uses a set of *root symbols*, which denote kernel data structures whose physical memory locations are fixed, and from which all data structures on the target's heap are reachable. In our implementation, we use 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 1292 type definitions by analyzing the source code of the target Linux-2.4.20 kernel using a CIL module [24].

The data structure extractor uses the roots and type definitions to recursively identify data structures in physical memory using a standard worklist algorithm. The extractor first adds the addresses of the roots to a worklist; it then issues a request 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 more (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 all the data structures in the target kernel's memory (reachable from the roots) have been identified. 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.

When the data structure extractor finds a pointer-valued field, it may require assistance in the form of code annotations to clarify the semantics of the pointer. In particular, the data structure extractor requires assistance when it encounters linked lists, implemented in the Linux kernel using the `list_head` structure. In Linux, other kernel data structures (called *containers*) that must be organized as a linked list simply include the `list_head` data structure. The kernel provides functions to add, delete, and traverse `list_head` data structures. Such linked lists are problematic for the data structure extractor. In particular, when it encounters a `list_head` structure, it will be unable to identify the container data structure. To handle such linked lists, we use the `CONTAINER` annotation. The annotation explicitly specifies the type of the container data structure and the field within this type, to which the `list_head` pointers refer. The extractor uses this annotation and locates the container data structure. In our experiments, we annotated all 163 annotations of the `list_head` data structure in the Linux-2.4.20 kernel.

In addition to linked lists, 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 arrays in order to traverse and locate objects in the array. We plan to add support for dynamic arrays, opaque pointers and untagged unions in future work.

Because the page fetcher obtains pages from the target asynchronously (without halting the target), it is likely that the data structure extractor will encounter inconsistencies, such as pointers to non-existent objects. Such invalid pointers are problematic because the data structure extractor will incorrectly fetch and parse the memory region referenced by the pointer (which will result in more invalid pointers being added to the worklist of the traversal algorithm). To remedy this problem, we currently place an upper bound on the number of objects traversed by the extractor. 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 place 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 rarely reached, and even so, only when the system was under heavy load.

4.3. The invariant generator

In the training mode, the output of the data structure extractor is used by the invariant generator, which infers likely data structure invariants. These invariants are used as specifications of data structure integrity.

To extract data structure invariants, we adapted Daikon [14], a state of the art invariant inference tool. Daikon attempts to infer likely program invariants by observing the values of variables during multiple executions of a program. Daikon first instruments the program to emit a trace that contains the values of variables at selected *program points*, such as the entry points and exits of functions. It then executes the program on a test suite, and collects the traces generated by the program. Finally, Daikon analyzes these traces and hypothesizes invariants—properties of variables that hold across all the executions of the program. The invariants produced by Daikon conform to one of several *invariant 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; if `x` has the constant value 5, Daikon will infer `x == 5` as the invariant). Daikon also infers invariants over *collections* of objects. For example, if it observes that the field `bar` of all objects of type `struct foo` at a program point have the value `5`, it will infer the invariant "The fields `bar` of all objects of type `struct foo` have value 5."

We had to make three key changes to adapt Daikon to infer invariants over kernel data structures.

(1) Inference over snapshots. Daikon is designed to analyze multiple execution traces obtained from instrumented programs and extract invariants that hold across these traces. We cannot use Daikon directly in this mode because the target's kernel is not instrumented to collect execution traces. Rather, we obtain values of data structures by

asynchronously observing the memory of the target kernel. To adapt Daikon to infer invariants over these data structures, we represent all the data structures in one snapshot of the target's memory as a *single* Daikon trace. As described in Section 4.2, the data structure extractor periodically reconstructs snapshots of the target's memory. Multiple snapshots therefore yield multiple traces. Daikon processes all these traces and hypothesizes properties that hold across all traces, thereby yielding invariants over kernel data structures

(**2**) **Naming data structures.** Because Daikon analyzes instrumented programs, it represents invariants using global variables and the local variables and formal parameters of functions in the program. However, because Gibraltar aims to infer invariants on data structures reconstructed from snapshots, the invariants output by Gibraltar must be represented using the root symbols. Gibraltar represents each data structure in a snapshot using its name relative to one of the root symbols. 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 extractor names each data structure as it is visited for the first time.

In addition, Gibraltar also associates each name with the virtual memory address of the data structure that it represents in the snapshot. These addresses are used during invariant inference, where they help identify cases where the same name may represent different data structures in multiple snapshots. This may happen 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. Because the name `init_tasks->next_task` will be associated with different virtual memory addresses before and after allocation, it represents different data structures; Gibraltar ignores such objects during invariant inference.

(**3**) **Linked data structures.** Linked lists are ubiquitous in the kernel and, as demonstrated in Section 3, can be exploited subtly by rootkits. It is therefore important to preserve the integrity of kernel linked lists. Daikon, however, does not infer invariants over linked lists. To overcome this shortcoming, we represented kernel linked lists as arrays in Daikon trace files, and leveraged Daikon's ability to infer invariants over arrays. We then converted the invariants that Daikon inferred over these arrays to invariants over linked lists.

Daikon infers invariants that conform to 75 different templates [14], and infers several thousand invariants over kernel data structures using these templates. 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).

(**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 ($\texttt{coll}_1 \subset \texttt{coll}_2$).** This template represents invariants that describe that the collection $\texttt{coll}_1$ is a subset of collection $\texttt{coll}_2$. 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. Fortunately, Daikon supports an extensible architecture. Newer invariant templates can be supplied to Daikon, thereby allowing Gibraltar to detect more attacks.

4.4. The monitor

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

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. All the examples in Figures 1-6 are persistent invariants.

An invariant is persistent if and only if the names of the variables in the invariant persist across reboots *and* the property represented by the invariant holds across reboots. Thus, a transient invariant either expresses a property of a variable whose name does not persist across reboots or represents a property that does not hold across reboots. For example, consider the invariant in Figure 8, which expresses a property of a `struct file_operations` object. This invariant is transient because it does not persist across reboots. The name of this object changes across reboots as it appears at different locations in kernel linked lists; consequently, the

Attack Name	Data Structures Affected
Rootkits from Packet Storm [5].	
Adore-0.42, All-root, Kbd, Kis 0.9, Linspy2, Modhide, Phide, Rial, Rkit 1.01, Shtroj2, Synapsys-0.4, THC Backdoor	System call table
Adore-ng	Vfs hooks,udp recvmsg
Knark 2.4.3	System call table, proc hooks
Rootkits from research literature [10].	
Disable Firewall	Netfilter hooks
Disable PRNG	Vfs hooks

Table 1. *Rootkits for Linux 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.*

number of `next` and `prev`s that appear in the name of the variable differ across reboots.

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

Figure 8. *Example of a transient invariant. The name of the variable changes 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. In our experiments, we found that out of a total of approximately 718,000 invariants extracted by Gibraltar, approximately 40,600 invariants persist across reboots of the target system.

Although it is evident that the number of persistent invariants is much smaller than the total number of invariants inferred by Gibraltar (thus necessitating a training each time the target is rebooted), we note that this does not reflect poorly on our approach. In particular, the persistent invariants can be enforced as Gibraltar infers transient invariants after a reboot of the target machine, thus providing protection during the training phase as well. The cost of retraining to obtain transient invariants can potentially be ameliorated with techniques such as live-patching [9, 12], which can be used to apply patches to a running system.

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:

• **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 [25, 10, 33]. Gibraltar detected all these rootkits (Section 5.2).

• **False positives.** During enforcement Gibraltar raises an alert when it detects an invariant violation; if the violation was not because of a malicious modification, the alert is a false positive. Our experiments showed that Gibraltar has a false positive rate of 0.65% (Section 5.3).

• **Performance.** We measured three aspects of Gibraltar's performance and found that it imposes a negligible monitoring overhead (Section 5.4).

All our experiments are 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 has an identical configuration.

5.1. Experimental methodology

Our experiments with Gibraltar proceeded as follows. We first ran Gibraltar in training mode and executed a workload that emulated user behavior (described below) on the target system. We configured Gibraltar to collect fifteen snapshots during training. Gibraltar analyzes these snapshots and infers invariants. We then configured Gibraltar to run in enforcement mode using the invariants obtained from training. During enforcement, we installed rootkits on the target system, and observed the alerts generated by Gibraltar. Finally, we studied the false positive rate of Gibraltar by executing a workload consisting of benign applications.

Workload. We chose the Lmbench [23] benchmark as the workload that runs on the target system. This workload consists of a micro benchmark suite that is used to measure operating system performance. These micro benchmarks measure 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 therefore exercises several kernel subsystems and modifies several kernel data structures as it executes.

5.2. Detection accuracy

We report the results obtained in the use of the inferred invariants to detect control and non-control data attacks.

Detecting control data modifications. We used fourteen publicly-available rootkits [5] that modify kernel data structures to test the effectiveness of Gibraltar. We also include two rootkits that have been proposed in the research literature [10] (Attacks 5 and 6 from Section 3); these rootkits also modify kernel function pointers. Table 1 summarizes the list of rootkits that modify kernel control data that we used in our experiments.

Gibraltar successfully detects 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.

Detecting non-control data modifications. We used four non-control data attacks discussed in prior work [10, 25, 33] to test Gibraltar. These attacks, and the invariants that they violate were discussed in detail in Section 3. Table 2 summarizes these attacks, and shows the data structures modified by the attack, the invariant type (collection/object) violated, and the template that classifies the invariant. Each of

Attack Name	Data Structures Affected	Invariant Type	Template
Entropy Pool Contamination	struct poolinfo	Collection	Membership
Hidden Process	all-tasks list	Collection	Subset
Linux Binfmt	formats list	Collection	Length
Resource Wastage	struct zone_struct	Object	Membership (constant)

Table 2. *Rootkits that modify non-control data [10, 25, 33]. 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.*

the invariants that was violated was a persistent invariant, which survives a reboot of the target machine.

5.3. Invariants and false positives

Invariants. As discussed in Section 4, Gibraltar uses Daikon to infer invariants; these invariants express properties of both individual objects, as well as collections of objects (*e.g.*,, all objects of the same type; invariants inferred over linked lists are also classified as invariants over collections). Table 3 reports the number of invariants inferred by Gibraltar on individual objects as well as on collections of objects. Table 3 also presents a classification of invariants by templates; the length and subset invariants apply only to linked lists. As this table shows, Gibraltar *automatically* infers several thousand invariants on kernel data structures.

False Positives. To evaluate the false positive rate of Gibraltar, we designed a test suite consisting of several benign applications, that performed the following tasks: **(a)** copying the Linux kernel source code from one directory to another; **(b)** editing a text document (an interactive task); **(c)** compiling the Linux kernel; **(d)** downloading eight video files from the Internet; and **(e)** perform file system read/write and meta data operations using the IOZone benchmark [27]. This test suite ran for 42 minutes on the target. We enforced the invariants inferred using the workload described in Section 5.1.

The false positive rate is measured as the ratio of the number of invariants for which violations are reported and the total number of invariants inferred by Gibraltar. Table 3 presents the false positive rate, further classified by the type of invariant (object/collection) that was erroneously violated by the benign workload, and the template that classifies the invariant. As this table shows, the overall false positive rate of Gibraltar was 0.65%.

	Invariants		False Positives	
Template	**Object**	**Collection**	**Object**	**Collection**
Membership	643,622	422	0.71%	1.18%
Non-zero	49,058	266	0.17%	2.25%
Bounds	16,696	600	0%	0%
Length	NA	4,696	NA	0.66%
Subset	NA	3,580	NA	0%

Table 3. *Invariants and false positives classified by the type of invariant and the template used to mine the invariant. Gibraltar infers a total of* 718,940 *invariants. Average false positive rate:* 0.65%.

5.4. Performance

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

Training time. The training time is calculated as the cumulative time taken by Gibraltar to gather kernel data structure snapshots and infer invariants when executing in training mode. Overall, the process of gathering 15 snapshots of the target kernel's memory requires approximately 25 minutes, followed by 31 minutes to infer invariants, resulting in a total of 56 minutes for training.

Training is currently a time-consuming process because our current prototype invokes Daikon to infer invariants after collecting all the kernel snapshots. Training time can potentially be reduced by adapting Daikon to use an incremental approach to infer invariants. In this approach, Daikon would hypothesize invariants using the first snapshot, in parallel with the execution of the workload to produce more snapshots. As more snapshots are produced, Daikon can incrementally refine the set of invariants. We leave this enhancement for future work.

Detection time. We measure the detection time as the interval between the installation of the rootkit and Gibraltar detecting that an invariant has been violated. Because 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.

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 [22], 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 are chosen so that they clear the last-level cache in the system, forcing data to be fetched from main memory. Gibraltar imposes a negligible

overhead of 0.49% on the operation of the target system.

6. Conclusions

Kernel-level rootkits pose a significant and growing threat to computer systems. Previously-proposed rootkit-detection techniques largely detect attacks that modify kernel control data; techniques that detect non-control data attacks, especially on dynamically-allocated data structures, require specifications of data structure integrity to be supplied manually.

This paper presented Gibraltar, a tool that automatically infers and enforces specifications of kernel data structure integrity. Gibraltar infers invariants uniformly across control and non-control kernel data, and enforces these invariants as specifications of data structure integrity. Our experiments showed that Gibraltar successfully detects rootkits that modify both control and non-control data structures, and does so with a low false positive rate and negligible performance overhead.

Acknowledgements. We would like to thank Joe Kilian for his insightful discussions on this work. This work has been supported in part by the NSF grants CCF-0728937 and CNS-0831268 and award from the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS).

References

[1] Anti rootkit software, news, articles and forums. http://antirootkit.com/.

[2] Chkrootkit - rootkit detection tool. http://www.chkrootkit.org/.

[3] Fu rootkit. http://www.rootkit.com/project.php?id=12.

[4] Myricom: Pioneering high performance computing. http://www.myri.com.

[5] Packet storm. http://packetstormsecurity.org/UNIX/penetration/rootkits/.

[6] Rootkit revealer, rootkit detection tool by microsoft. http://technet.microsoft.com/en-us/sysinternals/bb897445.aspx.

[7] Sophos anti-rootkit. http://www.sophos.com/products/free-tools/sophos-anti-rootkit.html.

[8] Rootkits, part 1 of 3: A growing threat, April 2006. MacAfee AVERT Labs Whitepaper.

[9] Jeffrey Brian Arnold. Ksplice: An automatic system for rebootless linux kernel security updates. http://web.mit.edu/ksplice/doc/ksplice.pdf.

[10] Arati Baliga, Pandurang Kamat, and Liviu Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007.

[11] Doug Beck, Binh Vo, and Chad Verbowski. Detecting stealth software with strider ghostbuster. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, 2005.

[12] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, 2006.

[13] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. Cloaker: Hardware Supported Rootkit Concealment. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[14] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

[15] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP03: ACM Symposium on Operating System Principles*, October 2003.

[16] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, 2003.

[17] Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Security '04: Proceedings of the USENIX Security Symposium*, 2004.

[18] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, 1994.

[19] Samuel King, Peter Chen, Yi-Min Wang, Chad Verblowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing malware with virtual machines. In *SP06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.

[20] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference*, 2004.

[21] Lionel Litty and David Lie. Manitou: a layer-below approach to fighting malware. In *ASID*, 2006.

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

[23] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *ATEC '96: Proceedings of the USENIX Annual Technical Conference*, May 1996.

[24] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, 2002.

[25] Jr. Nick L. Petroni, Timothy Fraser, AAron Walters, and William A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Security '06: Proceedings of the USENIX Security Symposium*, 2006.

[26] Jr. Nick L. Petroni and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[27] W. Norcott. Iozone benchmark. http://www.iozone.org, 2001.

[28] Joanna Rutkowska. Defeating hardware based ram acquisition. In *Blackhat Conference*, 2007.

[29] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *ACM Conference on Computer and Communications Security*, October 2004.

[30] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Security04: Proceedings of the 2004 USENIX Security Symposium*, August 2004.

[31] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP '05: 20th ACM Symposium on Operating System Principles*, 2005.

[32] Elaine Shi, Adrian Perrig, and Leendert van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *SP '05: IEEE Symposium on Security and Privacy*, 2005.

[33] Shellcode Security Research Team. Registration weakness in linux kernel's binary formats. http://goodfellas.shellcode.com.ar/own/binfmt-en.pdf, September 2006.

[34] Yi-Min Wang, Roussi Roussev, Chad Verbowski, Aaron Johnson, Ming-Wei Wu, Yennun Huang, and Sy-Yen Kuo. Gatekeeper: Monitoring auto-start extensibility points (aseps) for spyware management. In *LISA '04: Proceedings of the 18th USENIX conference on System administration*, 2004.

[35] Jeffrey Wilhelm and Tzi cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *RAID*, 2007.

[36] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. Secure coprocessor-based intrusion detection. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC*, 2002.