An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data

Nick L. Petroni, Jr.†
npetroni@cs.umd.edu
Department of Computer Science

Timothy Fraser†
tfraser@umiacs.umd.edu
Institute for Advanced Computer Studies

AAron Walters‡
arwalter@cs.purdue.edu
Department of Computer Science

William A. Arbaugh†
waa@cs.umd.edu
Department of Computer Science

† *University of Maryland, College Park, MD 20742, USA*
‡ *Purdue University, West Lafayette, IN 47907, USA*

Abstract

The ability of intruders to hide their presence in compromised systems has surpassed the ability of the current generation of integrity monitors to detect them. Once in control of a system, intruders modify the state of constantly-changing dynamic kernel data structures to hide their processes and elevate their privileges. Current monitoring tools are limited to detecting changes in nominally static kernel data and text and cannot distinguish a valid state change from tampering in these dynamic data structures. We introduce a novel general architecture for defining and monitoring *semantic integrity* constraints using a specification language-based approach. This approach will enable a new generation of integrity monitors to distinguish valid states from tampering.

1 Introduction

The foundation of the Trusted Computing Base (TCB) [26] on most currently deployed computer systems is an Operating System that is large, complex, and difficult to secure. Upon penetrating a system, sophisticated intruders often tamper with the Operating System's programs and data to hide their presence from legitimate administrators and to provide backdoors for easy re-entry. The Operating System kernel itself is a favored target, since a kernel modified to serve the attacker renders user-mode security programs ineffective. Many so-called "rootkits" are now available to automate this tampering.

Recent advances in defensive technologies, such as external kernel integrity monitors [17, 37, 13, 29] and code attestation/execution verification architectures [18, 34, 33], have demonstrated their ability to detect the kinds of tampering historically performed by rootkits. Unfortunately, rootkit technology has already moved to a more sophisticated level. While these defensive technologies have focused on the relatively straightforward task of detecting tampering in static and unchanging regions of kernel text and data structures—typical targets of the previous generation of rootkits—the new rootkit generation has evolved to more sophisticated tampering behavior that targets dynamic parts of the kernel. Seeking to avoid detection and subsequent removal from the system, clever intruders can hide their processes from legitimate administrators by modifying links in the Linux and Windows XP/2000 kernels' process tables. Because the state of the process table changes continuously during kernel runtime, identifying these modified links is difficult for the current generation of kernel integrity monitoring tools that focus only on static data. Although this targeting of dynamic data was not entirely unanticipated by researchers [37, 13], there has yet to be a general approach for dealing with this threat.

In response to a continually advancing threat, we introduce an architecture for the runtime detection of *semantic integrity* violations in objects dynamically allocated in the kernel heap or in static objects that change depending upon the kernel state. This new approach is the first to address the issue of dynamic kernel data in a comprehensive way. In order to be effective against the latest rootkit technology, defensive mechanisms must consider both static and dynamic kernel data, as changes in either can lead to the compromise of the whole. We believe our approach provides an excellent complement to state of the art binary integrity systems.

Our approach is characterized by the following properties:

Specification-based. The previous generation's detection methods, which can be characterized by calculating hashes of static kernel data and text and comparing the result to known-good values, is not applicable to the continuously changing dynamic data structures now being targeted by rootkits. Instead of characterizing a correct state using hashes, our architecture relies upon an expert to describe the correct operation of

the system via an abstract model for low-level data structures and the relationships between them. This model is a simplified description of security-relevant data structures and how they interoperate. Additionally, part of the specification is a set of constraints that must hold at runtime in order for the system to remain correct with regard to the *semantic integrity* of the kernel.

Automatic. The architecture includes a compiler that automatically translates the high-level specification language into low-level machine code to perform the checks. This automation allows experts to maximize the use of their time writing the specification and verifying its correctness, rather than writing low-level code.

Independent. Our architecture does not depend upon the correctness of the monitored kernel in order to detect that something is wrong. Instead, our approach relies on a trustworthy monitor that has direct access to kernel memory on the protected system and does not rely on the protected kernel's correctness.

Monitor agnostic. While our prototype implementation utilizes a PCI-based kernel monitor similar to Copilot [29] as the low-level mechanism for accessing system resources, our architecture allows for the use of any monitor with access to kernel memory that can also provide isolation. Other possibilities include software-based systems such as Pioneer [33] or a virtual machine introspection approach [13]. The focus of this work is on the *type* of checks performed, not the mechanism used to perform them. As such, our architecture is general enough to support different types of monitors, both software- and hardware-based.

Extensible response. The architecture is designed to allow specification writers to decide how the system should react to the violation of a particular constraint. At a minimum, most cases will require administrator notification. Currently, this is the only response we have implemented. However, the possibility for extension to other responses is apparent, particularly given the amount of forensic information available to our monitor.

We have demonstrated the feasibility of our approach by writing sample specifications for two different kernel subsystems in the Linux 2.6 kernel: the process (task) accounting system and the SELinux [22] mandatory access control (MAC) system's access vector cache (AVC). We have tested the system's effectiveness at detecting real-world attacks on dynamic kernel data in each subsystem, including a publicly available rootkit for the Linux kernel. Our results show that low-level code based on our initial specifications successfully detects the example attacks, which include data-only process hiding and modifications of SELinux access control results directly in memory.

2 Threats Against Dynamic Kernel Data

This section describes two examples of how intruders might, after gaining full administrative control of a GNU/Linux system, modify some of the kernel's dynamic data structures to their advantage. In the first example, an intruder removes tasks from the Linux kernel's all-tasks list in order to hide them from the system's legitimate administrators. In the second example, an intruder modifies an entry in the Linux kernel's SELinux access vector cache to temporarily elevate their privileges and disable auditing without making visible changes to the SELinux policy configuration. Note that neither of these examples expose flaws in the Linux kernel or its SELinux security module. These examples represent the potential acts of an intruder who has already gained full control of the system—perhaps by exploiting the trust or carelessness of the system's human operators in a manner entirely outside the scope of the system's technological safeguards.

2.1 Data-only Process Hiding

Rootkits have evolved beyond the historical methods of hiding processes, which included modifying the text of the `ps` program to lie to legitimate administrators or causing the kernel itself to lie by replacing the normally-static values of kernel text or function pointers, such as the system call vector or jump tables in the `/proc` filesystem, with the addresses of malicious functions. Even the most sophisticated threats became easy to detect by monitors that could compare the modified values against a known-good value—after all, in a healthy system, these values should never change [29].

Unfortunately, attackers do not need to modify any kernel code to hide processes within a running kernel. In fact, they do not need to rely on manipulating the control flow of the kernel at all. Instead, adversaries have found techniques to hide their processes even from correct, unmodified kernel code. By directly manipulating the underlying data structures used for process accounting, an attacker can quickly and effectively remove any desired process from the view of standard, unmodified administrator tools. While the process remains hidden for accounting purposes, it continues to execute as normal and will remain unaffected from the perspective of the scheduler. To understand how this state is achieved, we provide a brief overview of Linux 2.6 process management.

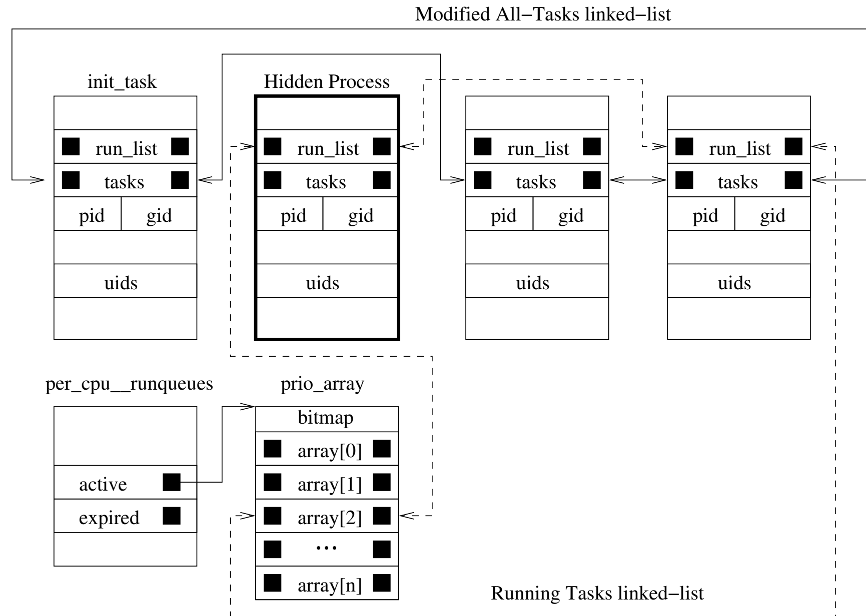

Figure 1: Data-only process hiding in Linux.

The primary data structure for process management in the Linux kernel is the `task_struct` structure [23]. All threads are represented by a `task_struct` instance within the kernel. A single-threaded process will therefore be represented internally by exactly one `task_struct`. Since scheduling occurs on a per-thread basis, a multi-threaded processes is simply a set of `task_struct` objects that share certain resources such as memory regions and open files, as well as a few other properties including a common process identifier (PID), the unique number given to each running process on the system.

In a correctly-running system, all `task_struct` objects are connected in a complex set of linked lists that represent various groupings relevant to that task at a particular time [23]. For accounting purposes, all tasks are members of a single doubly-linked list, identified by the `task_struct.tasks` member. This list, which we refer to as the all-tasks list, insures that any kernel function needing access to all tasks can easily traverse the list and be sure to encounter each task exactly once. The head of the task list is the `swapper` process (PID 0), identified by the static symbol `init_task`. In order to support efficient lookup based on PID, the kernel also maintains a hash table that is keyed by PID and whose members are hash-list nodes located in the `task_struct.pid` structure. Only one thread per matching hash of the PID is a member of the hash table; the rest are linked in a list as part of `task_struct.pid` member. Other list memberships include parent/child and sibling relationships and a set of scheduler-related lists discussed next.

Scheduling in the Linux kernel is also governed by a set of lists [23]. Each task exists in exactly one state. For example, a task may be actively running on the processor, waiting to be run on the processor, waiting for some other event to occur (such as I/O), or waiting to be cleaned up by a parent process. Depending on the state of a task, that task will be a member of at least one scheduling list somewhere in the kernel. At any given time, a typical active task will either be a member of one of the many wait queues spread throughout the kernel or a member of a per-processor run queue. Tasks cannot be on both a wait queue and a run queue at the same time.

Primed with this knowledge of the internals of Linux process management, we now describe the trivial technique by which an attacker can gain the ultimate stealth for a running process. Figure 1 depicts the primary step of the attack: removing the process from the doubly-linked all-tasks list (indicated by the solid line between tasks). Since this list is used for all process accounting functions, such as the `readdir()` call in the `/proc` filesystem, removal from this list provides all of the stealth needed by an adversary. For an attacker who has already gained access to kernel memory, making this modification is as simple as modifying two pointers per hidden process. As a secondary step to the attack, adversaries might also choose to remove their processes from the PID hash table (not pictured) in order to prevent the receipt of unwanted signals.

As shown in Figure 1, a task not present in the all-tasks list can continue to function because the set of lists used for scheduling is disjoint from the set used for accounting. The dashed line shows the relationship between objects relevant to a particular processor's run queue, including tasks that are waiting to be run (or are currently running) on that processor. Even though the second depicted task is no longer present in the all-tasks list, it continues to be scheduled by the kernel. Two simple changes to dynamic data therefore result in perfect stealth for the attacker, without any modifications to static data or kernel text.

2.2 Modification of System Capabilities

When most actions occur in the kernel, some form of a capability is used to identify whether or not a principal should be given (or already has been given) access to a resource. These capabilities therefore represent a prime target for attackers wishing to elevate privilege. Changing process user identifiers (UIDs) has long been a favorite technique of attackers. Other examples include file descriptors and sockets (both implemented in the same abstraction in the kernel).

The SELinux access vector cache provides a good example of this kind of capability and represents a potential target for an adversary seeking privilege escalation. This section describes the structure and purpose of the AVC and how an adversary might tamper with its state. Section 4 describes an experiment that demonstrates such tampering and the effectiveness of a prototype monitor for detecting this tampering.

SELinux [22] is a security module for Linux kernels that implements a combination of Type Enforcement [3] and Role-based [11] mandatory access control, now included in some popular GNU/Linux distributions. During runtime, SELinux is responsible for enforcing numerous rules governing the behavior of processes. For example, one rule might state that the DHCP [10] client daemon can only write to those system configuration files needed to configure the network and the Domain Name Service [24], but no others. By enforcing this rule, SELinux can limit the damage that a misbehaving DHCP client daemon might cause to the system's configuration files should it be compromised by an adversary (perhaps due to a buffer overflow or other flaw).

To enforce its rules, SELinux must make numerous decisions during runtime such as "Does the SELinux configuration permit this process to write this file?" or "Does it permit process A to execute program B?" Answering these questions involves some overhead, so SELinux includes a component called the access vector cache to save these answers. Whenever possible, SELinux rapidly retrieves answers from the AVC, resorting to the slower method of consulting the policy configuration only on AVC misses.

On our experimental system, the AVC is configured to begin evicting least frequently used entries after reaching a threshold of 512 entries. Our single-user system never loaded the AVC much beyond half of this threshold—although it was occasionally busy performing builds, these builds tended to pose the same small number of access control questions again and again. However, one could imagine a more complex multi-user system that might cause particular AVC entries to appear and disappear over time. Installations that permit SELinux configuration changes during runtime might also see AVC entries evicted due to revocation of privileges.

SELinux divides all resources on a system (such as processes and files) into distinct classes and gives each class a numeric Security Identifier or "SID." It expresses its mandatory access rules in terms of what processes with a particular SID may and may not do to resources with another SID. Consequently, at a somewhat simplified abstract level, AVC entries take the form of tuples:

```
<ssid, tsid, class, allowed, decided,
       audit-allow, audit-deny>
```

The `ssid` field is the SID of the process taking action, the `tsid` field is the SID of the resource the process wishes to act upon, and the `class` field indicates the kind of resource (file, socket, and so on). The `allowed` field is a bit vector indicating which actions (read, write, and so on) should be allowed and which should be denied. Only some of the `allowed` field bits may be valid—for example, if the questions answered by SELinux so far have involved only the lowest-order bit, then that may be the only bit that contains a meaningful 0 or 1. SELinux may or may not fill in the other `allowed` field bits until a question concerning those bits comes up. To distinguish a 0 bit indicating "deny" from a 0 bit indicating "invalid," the `decided` field contains a bit vector with 1 bits for all valid positions in the `allowed` field. The `audit-allow` and `audit-deny` fields are also bit vectors; they contain 1 bits for operations that should be logged to the system logger when allowed or denied, respectively.

It is conceivable that adversaries who have already gained administrative control over a system might wish to modify the SELinux configuration to give their processes elevated privileges. Certainly, they could accomplish this most directly by modifying the SELinux configuration files, but such modifications would be easily detected by filesystem integrity monitors like Tripwire [19]. Alternately, they might modify the in-kernel data structures representing the SELinux configuration—the same data structures SELinux consults to service an AVC miss. However, these data structures change in-

frequently, when administrators decide to modify their SELinux configuration during runtime. Consequently, any tampering might be discovered by a traditional kernel integrity monitor that performs hashing or makes comparisons with correct, known-good values.

The state of the AVC, on the other hand, is dynamic and difficult to predict at system configuration time. Entries come and go with the changing behavior of processes. An adversary might insert a new AVC entry or modify an old one to effectively add a new rule to the SELinux configuration. Such an entry might add extra `allowed` and `decided` field bits to grant additional privileges, or remove existing `audit-allow` and `audit-deny` field bits to turn off troublesome logging. Such an entry would override the proper in-memory and on-disk SELinux configuration for as long as it remained in the cache. On a single-user installation like our experimental system, it would face little danger of eviction. On a busier system, frequent use might keep it cached for as long as needed.

3 The Specification Architecture

Our approach for detecting semantic integrity violations in dynamic kernel data structures is to define a high-level security *specification* [20] for kernel data that provides a simplified but accurate representation of how kernel objects in memory relate to one another, as well as a set of constraints that must hold on those data objects for the integrity of the kernel to remain intact. The result is a methodology that allows experts to concentrate on high-level concepts such as identifying security-relevant constraints, rather than writing low-level code to parse kernel data structures. The architecture we propose is composed of the following five components:

- *A low-level monitor*. The monitor is the entity that provides access to kernel memory at runtime. While there are a number of possible implementations, the primary requirement is consistent access to all of kernel virtual memory without reliance on the correctness of the protected kernel. Monitors that provide synchronous access to kernel memory, such as virtual machine monitors [13] or verifiable code execution [33], provide consistent views of kernel data, but run on the same host as the protected system and must contend with local applications for processor time. Asynchronous monitors typically have their own dedicated processor [29, 37, 17], but must make sense of snapshots of kernel memory that catch data structures in a temporarily-inconsistent mid-update state. In addition, monitors with access to system registers can protect themselves against attempts to bypass the monitor via malicious register changes [33].

- *A model builder*. The model builder is responsible for taking raw data from the low-level monitor and turning that data into the model abstraction defined by the specification, which is an input to the model builder. Effectively, the model builder is the bridge between the "bits" in kernel memory and the abstract objects defined by the user.

- *A constraint verifier*. As described above, the goal of the system is to apply high-level constraints to an abstract model of kernel data. The constraint verifier operates on objects provided by the model builder to determine if the constraints identified by the specification are met.

- *Response mechanisms*. When a constraint is violated, there is a security concern within the system. Depending on the nature of the violated constraint, an administrator may wish to take actions varying from logging an error to notifying an administrator or even shutting down the system. The constraint verifier is aware of the available response mechanisms and initiates those mechanisms according to the response determined by the specification.

- *A specification compiler*. Specifications are written in a high-level specification language (or languages) that describes the model, the constraints, and the responses to violated constraints. The specification compiler is responsible for turning the high-level language into a form that can be used by the model builder and the constraint verifier.

As shown in Figure 2, the first four of these are runtime components that work together to assess the integrity of a running kernel based on the input specification. The specification compiler is an offline component used only at the time of system setup or when specification updates are required. The primary logic of the monitor is driven by the constraint verifier, which iterates through all constraints to verify each in order. To facilitate the verification of each constraint, the verifier requests a consistent subset of the model from the model builder, which either has the information readily available or uses the low-level monitor to re-build that portion of the model. If a constraint passes, the verifier simply continues to the next. Failed constraints cause the verifier to dispatch a response mechanism according to the specification.

We now describe several aspects of the system in more detail, focusing primarily on the requirements for each component.

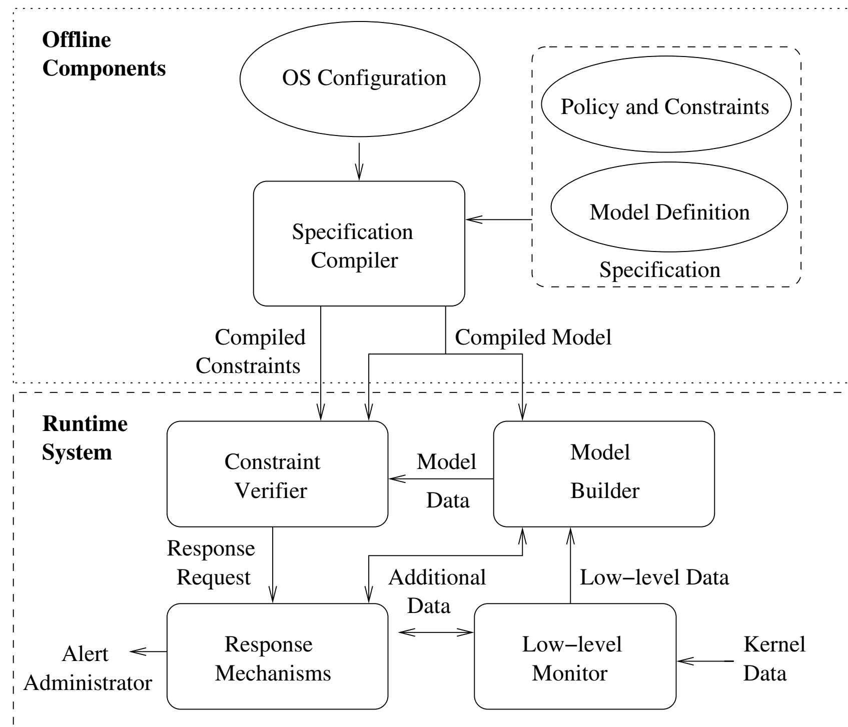

Figure 2: The semantic integrity monitor architecture.

3.1 Modeling Kernel Data

The concept of modeling low-level kernel data structures can be loosely thought of as a form of "inverted software design." A software designer's job is typically to take a high-level abstraction or set of real-world objects and represent those objects digitally in the system. One of the complex tasks for the programmer is efficiently and correctly representing real-world relationships among digital objects. Object modeling languages like the Unified Modeling Language (UML) [1] seek to aid the designer by providing formal constructs with which to define objects. In our system, the goal of the model specification writer is to abstract on the programmer's choice of data structures in order to describe the relevant relationships among digital objects. The resulting model allows an expert to write constraints at a high enough level of abstraction to express relevant object relationships without getting caught up in low-level details. To this end, the choice of modeling language directly affects the types of constraints that can and cannot be expressed on the model. Modeling languages that fail to capture relevant details of the underlying system will not allow potentially important constraints to be expressed. Similarly, modeling languages that provide too much expressive power on the underlying data will make the job of constraint writing overly complex. As a convenience, rather than inventing our own modeling language, we have chosen to reuse the data structure specification language created by Demsky and Rinard [7]. After redefining the language's semantics for our domain, the syntax proved effective for our example kernel data specifications with only minor modifications. We discuss these example kernel data specifications in Section 4.

It should be noted that model specifications correspond to a particular version (or versions) of the kernel. Therefore, as updates are made to kernel subsystems, so must the specification be updated. However, once a specification is written for a given kernel version, it can be shared and used at any deployed location. Furthermore, the specification compiler takes into account site-specific kernel configuration and symbol information to allow more widespread use of the specification. Finally, the relationships described in the specification will not change frequently and, even when they do change, will rarely change significantly enough to invalidate the entire specification. Tools for automating and improving the specification process are an area for future work.

3.2 Writing Model Constraints

At a high level, constraints are the underlying logic that determine whether or not the kernel is secure with regard to integrity. Constraints are therefore expressions of

predicates reflecting invariant relationships among kernel objects represented in the model. Conceptually, constraints can be divided into two classes: those that are inherent to the correct operation of the system and those that represent site-specific policy. For example, the hidden process example described previously is clearly a violation of kernel integrity in any running kernel. However, one can envision a set of external constraints on kernel data objects that do not relate explicitly to the "correct" operation of the kernel as it was designed by kernel developers, but rather to conditions that an administrator has deemed should never occur on that machine. One example of such a constraint would be a requirement that no shell processes have user id zero (root). The requirements for a good constraint language include easy expression of properties of the underlying model, an ability to specify conditions under which a constraint must hold, and a mechanism for assigning a response to any violated constraint. To match our choice of initial modeling language, we have adapted Demsky and Rinard's constraint language to meet the needs described here [7].

Similar to the model specification, the constraints that must hold for a system may change when kernel developers make changes. However, like model specifications, constraints can be distributed for use at any deployment where a given model is valid.

3.3 Automating the System

One of the fundamental goals of our architecture is to relieve the engineering difficulties related to dynamic kernel data constraint checking and allow the expert to focus on security-relevant relationships among kernel objects. To this end, automating the model builder is a critical step. The primary responsibility of the specification compiler is to provide the model builder with a description of how digital objects should be turned into abstract objects and how those abstract objects are related. As in Demsky and Rinard's work [7], we propose that the specification compiler utilize automatic code generation to automate the model building and constraint checking processes. However, unlike the environment in which Demsky and Rinard's system functioned, the likely response for our system when a constraint fails is not repair. In fact, there may be reasons *not* to immediately fix the integrity violation so that more forensic information can be obtained without the attacker becoming aware that he or she has been detected. Furthermore, unlike Demsky and Rinard, in our system we do not have the benefit of executing within the running program that we are checking. Memory accesses are not free and pointer values are not local. In our system, every pointer dereference requires read operations by the low-level monitor. For these reasons, optimizing for repair is not the best approach for our environment. Rather, optimizing for efficient object accesses is more appropriate. Finally, performing checks asynchronously with the running kernel adds some additional challenges.

For a system that is externally analyzing a running kernel, the design of the model builder is non-trivial due to the complications of constantly changing data within the kernel. The assumptions that can be made by the model builder are closely tied to the properties of the low-level monitor. However, assuming a monitor that is running asynchronously relative to the protected kernel, the following are a minimal set of design considerations for the model builder and specification compiler components:

- How will the system distinguish inconsistent data resulting from a read that occurs while the kernel is in the middle of a data structure update from an invalid kernel state?
- How can the system schedule data reads such that relationships to be tested among digital objects are tested on a set of objects that were read at or about the same time?
- How can the system schedule data reads to minimize the total number of reads necessary to check a particular constraint or set of constraints?

In Section 4, we discuss how our initial implementation handles these issues. To summarize our results, we postulate that simple extensions to the modeling language can help the specification compiler reason about the nature of underlying data, including how likely it is to change over time and the best order in which to process it. As a promising indication of our success, the resulting system experienced no false positives in any of our tests. However, in a specification-based system the possibility for false positives or false negatives is more a reflection of the specification than of the system. An expert with better knowledge of the system will have more success in this regard.

4 Implementation

In this section, we describe our implementation of the above architecture and the testing we performed on a system running the Fedora Core 4 GNU/Linux distribution. Using our system, we have implemented (in C) two specifications designed to protect the Linux 2.6 process accounting and SELinux AVC subsystems respectively. We then tested our specifications against implementations of the two attacks described in Section 2. We successfully detected both of these attacks with zero false positives when our detection code was running on a PCI-based monitor similar to Copilot [29]. Table 1 provides more detailed information about our test environment. These

	Protected Host	PCI-based Monitor
Machine Type	Dell Dimension 4700	Bus-mastering PCI add-in card
RAM	1GB	32MB
Processor	Single 2.8GHz Pentium 4	200MHz Motorola PowerPC 405GP
Storage	40GB IDE Hard Disk	4MB Flash memory
Operating System	Redhat Fedora Core 4 full installation	Embedded Linux 2.4 kernel
Networking	10/100 PCI NIC	10/100 on-board NIC

Table 1: Semantic integrity test platform summary.

tests demonstrate that it is possible to write useful specifications using our technique, and that these specifications can be coupled with an existing integrity monitor to provide an effective defense against real attacks.

We begin our discussion by describing our specification language, an adaptation of that presented by Demsky and Rinard [7], in the context of our Linux process accounting example.

4.1 Writing Specifications: a Linux Hidden Process Example

Demsky and Rinard introduced a system for automatically repairing data structure errors based on model and constraint specifications [7]. The goal of their system was to produce optimized data structure error detection and repair algorithms [9] that were guaranteed to terminate [8]. Because of the differences explained in Section 3, we have adapted Demsky and Rinard's specification languages and the corresponding parser and discarded all of the automatic code generation portions. Our intention is to replace them with a code generation algorithm better suited to our environment. This section provides a brief overview of their specification language syntax and identifies the changes necessary to support our kernel integrity system. It also introduces our first example specification for detecting hidden processes in the Linux kernel. Demsky and Rinard's specification system is actually composed of four separate languages:

Low-level Structure Definition: The structure definition language provides C-like constructs for describing the layout of objects in memory. Demsky and Rinard provide a few additions to the normal C language syntax. First, fields may be marked "reserved," indicating that they exist but are not used. Second, array lengths may be variable and determined at runtime through expression evaluation. Third, a form of structure "inheritance" is provided for notational simplicity whereby structures can be defined based on other structures and then expanded with additional fields. We found no need to change the structure definition language syntax developed by Demsky and Rinard. However, it was necessary to adapt the language's semantics in two important ways because of the "external" nature of our monitor.

First, named structure instances, which are also declared in the structure definition language, cannot be resolved because our monitor is not part of the normal software linking process. Instead, we must use an external source for locating variables. Our current implementation allows the user to provide these locations manually or to have them extracted automatically from a Linux `System.map` symbol table file. The second semantic modification necessary for the structure definition language is the handling of pointer values, which are not "local" to our monitor. Instead, pointers must be treated as foreign addresses accessed through the monitor's memory access mechanism.

Figure 3(a) contains our specification of the Linux kernel's process accounting data structures written in the structure definition language. Figure 3(b) contains the result of a manual translation from this specification into the corresponding C declarations that will become part of the monitoring code. Note the use of the `host_addr_t` to represent host addresses after byte-order conversion on the monitor. As described above, the appropriate value for the `LINUX_SYMBOL_init_task` constant (and other required symbols) is automatically extracted from the Linux `System.map` symbol table file by our configuration tool.

Model Space Definition: The second language, shown in Figure 3(c) for our process accounting example, defines a group of sets or relations (there are no relations in our first example) that exist in the model [7]. There are two sets in our specification: one corresponding to all processes in the all-tasks list (the `AllTasks` set) and one corresponding to all processes in the run queue (the `RunningTasks` set). Both are of type `Task` in the model. We made no modifications to this simple language, as all of our example specifications were able to be expressed in the context of sets and relations. The model space definition language provided by Demsky and Rinard also provides support for set partitions and subsets.

```
Task init_task;

structure Task {
    reserved byte[32];
    ListHead run_list;
    reserved byte[52];
    ListHead tasks;
    reserved byte[52];
    int pid;
    reserved byte[200];
    int uid;
    reserved byte[60];
    byte comm[16];
}

structure ListHead {
    ListHead *next;
    ListHead *prev;
}

sructure Runqueue {
    reserved byte[52];
    Task *curr;
}
```

(a) Low−Level Structure Definiton

```
host_addr_t init_task =
    LINUX_SYMBOL_init_task;
struct Task {
    unsigned char reserved_1[32];
    ListHead run_list;
    unsigned char reserved_2[52];
    ListHead tasks;
    unsigned char reserved_3[52];
    int pid;
    unsigned char reserved_4[200];
    int uid;
    unsigned char reserved_5[60];
    unsigned char comm[16];
};

struct ListHead {
    host_addr_t next;
    host_addr_t prev;
};

struct Runqueue {
    unsigned char reserved_1[52];
    host_addr_t curr;
};
```

(b) Translated Structure Definiton

```
set AllTasks(Task);
set RunningTasks(Task);
```

(c) Model Space Definition

```
[ for_circular_list i as ListHead.next starting init_task.tasks.next ], true => container(i, Task,tasks.next) in AllTasks;
[ ], true => runqueue.curr in RunningTasks;
```

(d) Model Building Rules

```
[ for t in RunningTasks ],  t  in  AllTasks
    : notify_admin("Hidden task " + t.comm + " with PID " + t.pid + " detected at kernel virtual address " + t);
```

(e) Constraints

Figure 3: Process accounting subsystem specification.

Model Building Rules: Thus far we have discussed languages for describing the low-level format and organization of data in kernel memory and for declaring the types of high-level entities we will use in our model. The model building rules bridge the gap between these by identifying which low-level objects should be used within the abstract model. These rules take the form

```
[<quantifiers>], <guard> ->
    <inclusion rule>;
```

For each rule, there is a set of quantifiers that enumerates the objects to be processed by the rule, a guard that is evaluated for each object to determine if it should be subject to the rule, and an inclusion that determines how that object should be classified in the abstract model. We have made the following (syntactic and semantic) modifications to Demsky and Rinard's model building language:

1. *User-defined rule order.* In Demsky and Rinard's system, the specification compiler could identify the dependencies among rules and execute them in the most appropriate order. Furthermore, their denotational semantics required execution of the rule function until a least fixed point was reached. This approach is not suited for external monitors for two reasons. First, because memory accesses are of a much higher performance penalty in our system, the expert benefits from the ability to describe which objects should be read in which order to build a complete model. Second, unlike in Demsky and Rinard's environment, the low-level monitor may be performing its reads asynchronously with the monitored system's execution. Model building accesses that have not been optimized are more likely to encounter inconsistent data as the system state changes.

2. *Pointer handling.* As previously mentioned, pointer references are not local in our environment and must go through the low-level monitor's memory access system. To detect invalid pointers, Demsky and Rinard developed a runtime system that instruments the heap allocation and deallocation (`malloc()`, `free()`, etc.) functions to keep

track of valid memory regions. This approach is clearly not an option for external monitors, which are not integrated with the system's runtime environment. Currently, invalid pointers are handled by restarting the model build process. If the same invalid pointer is encountered during two consecutive model build operations, an error is generated. If the invalid pointer is not encountered again, it is assumed the first error was an inconsistency stemming from the asynchronous nature of the monitor.

3. *The* `contains()` *expression.* A common programming paradigm (especially in the Linux kernel) is to embed generic list pointer structures as members within another data structure. Our added expression gives specification writers an easy way to identify the object of which a particular field is a member.

4. *The* `for_list` *quantification.* Linked lists are a common programming paradigm. This expression gives specification writers a straightforward way to indicate they intend to traverse a list up to the provided stop address (or NULL if not indicated).

5. *The* `for_circular_list` *quantification.* This is syntactic sugar for the `for_list` construct where the end address is set equal to the first object's address. The Linux kernel makes heavy use of circular lists.

Figure 3(d) shows the model rules for our process accounting example. The first rule indicates that a circular list starting (and ending) at `init_task.tasks.next` will be processed. The keyword `true` in the guard indicates that all members of this list should be subject to the inclusion. The inclusion itself uses our `container()` expression to locate the `Task` that contains the list pointer and to include that `Task` in AllTasks. The second rule is very simple; it creates a singleton set `RunningTasks` with the current task running on the run queue.

Constraints: The final part of the specification defines the set of constraints under which the model is to be evaluated. The basic form of a rule in Demsky and Rinard's constraint language is as follows [7]:

```
[ <quantifiers> ], <predicate>;
```

In the constraint language, the set of quantifiers may include only sets defined in the model. The predicate is evaluated on each quantified member and may include set operations and evaluations of any relations defined in the model. If the predicate fails for any quantified member, Demsky and Rinard's system would seek to repair the model (and the underlying data structures accordingly). In our system, however, we have added a "response" clause to the end of the constraint rule as follows:

```
[ <quantifiers> ], <predicate> :
    <[consistency,] response>;
```

This critical extension allows the specification writer to dictate how failures are to be handled for a particular rule. In addition to identifying which action to take, the response portion allows for an optional "consistency parameter." This parameter allows the specification writer to identify a "safe" number of failures before taking action and helps prevent false positives that might occur due to data inconsistencies. If no such parameter is provided, the default value of two consecutive failures is used. Of course, a secondary result is that actual rule violations will be given an opportunity to occur once without detection. The specification writer will need to balance the advantages and the disadvantages for each constraint rule and can always disable this feature by setting the value to zero. For the threat considered in our Linux process accounting example, the default value is acceptable because of the nature of the targeted threat. A process that is short-lived has no reason to hide, since an administrator is unlikely to notice the process. Finally, the consistency value has no meaning for synchronous monitors, which do not suffer from the same consistency problems.

Figure 3(e) shows the single constraint rule for our hidden process example. The rule states that if any process is ever seen running on the processor that is not in the all-tasks list, we have a security problem and need to alert the administrator. This example describes a relatively simple method of detecting hidden processes. In order to detect a hidden process, the monitor must catch the process while it has the host CPU—a probabilistic strategy that is likely to require the taking of many snapshots of the host's state over time before the hidden process's luck runs out. A more deterministic approach might be to compare the population of the kernel's numerous wait and run queues with the population of the all-tasks list. In order to be eligible for scheduling, a process must be on one of these wait or run queues; a process on a wait or run queue but not in the all-tasks list is hiding. This strategy would require a more complex model specification.

4.2 A Second Example: the SELinux AVC

In Section 2, we described an attack against the SELinux AVC whereby an attacker with the ability to write to memory could modify the permissions of an entry in

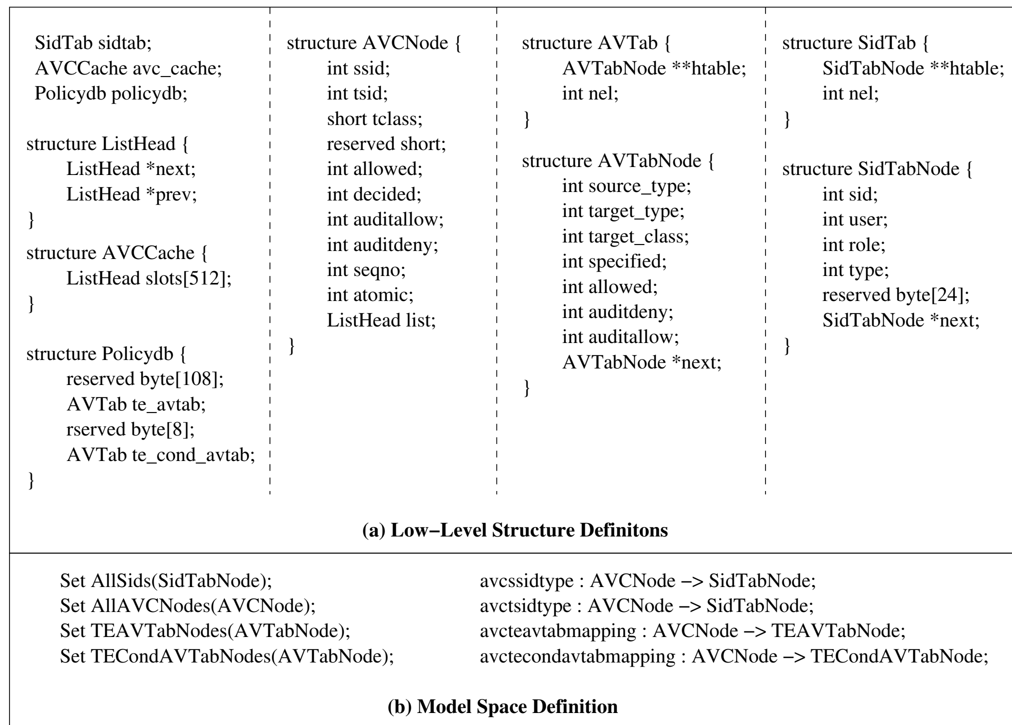

Figure 4: SELinux access vector cache structure and model definitions.

the cache to give a particular process access not permitted by the SELinux policy. We further explained that existing hashing techniques can be used to protect the memory-resident full policy, but not the AVC because of its dynamic nature. Our approach for protecting the AVC therefore begins with the assumption that a simple "binary" integrity system is protecting the static data structures that represent the full policy. We then use our semantic integrity monitor to implement a specification whose goal is to compare all AVC entries with their protected entries in the full policy. Figures 4 and 5 display the full specification we used to protect the SELinux AVC. This specification is more complex than the previous example largely due to the complexities of the SELinux system and its data structures. However, the complexity of the specification is minimal as compared with the number of lines of code that would be required to implement the equivalent checks in low-level code (eight model definition rules and one constraint rule versus the 709 lines of C code in our example implementation).

There are four primary entities in our SELinux specification: the security identifier table (of type SIDTab), the access vector cache (an AVCCache), the Type Enforcement access vector table (an AVTab), and its counterpart the Type Enforcement conditional access vector table (also an AVTab). The model definition rules first create a set of SIDs by walking through the SID table and then, similarly, create a set of all AVC nodes from the AVC. The third and fourth rules are used to create mappings between the AVC nodes and their source and target SIDs. Rules five and six look-up each AVC node in the full Type Enforcement policy for both conditional and non-conditional access vector tables. The final two model definition rules create a mapping between AVC nodes and their corresponding entries in the Type Enforcement access vector tables. The single constraint rule simply walks through all AVC nodes and checks that the allowable field matches the combined (bitwise OR) value of the two corresponding Type Enforcement access vector entries for that AVC node. As with the last example, an administrator is notified if the data structures are found to be inconsistent.

We have tested our code against an attacking loadable kernel module that modifies the permissions for a particular AVC entry. A rootkit might make such a modification to temporarily elevate the privileges of one or more processes in a manner that could not be detected by an integrity monitor that observed only static data structures. Our specification successfully detects the attack against our Fedora Core 4 system configured with the default SELinux "targeted" policy operating in "enforcing" mode.

```
[ for i = 0 to 128, for_list j as SidTabNode.next starting sidtab.htable[i] ], true => j in AllSids ;
[ for i = 0 to 512, for_circular_list j as ListHead.next starting avc_cache.slots[i] ], true =>
      true => container (j, AVCNode, list.next ) in AllAVCNodes ;
[ for a in AllAVCNodes, for s in AllSids ], (a.ssid = s.sid) => <a,s> in avcssidtype ;
[ for a in AllAVCNodes, for s in AllSids ], (a.tsid  = s.sid) => <a,s> in avctsidtype ;
[ for a in AllAVCNodes, for_list j as AVTabNode.next starting
      policydb.te_avtab.htable[a.tclass + a.avctsidtype.type * 4 + a.avcssidtype.type * 512].next ],
      (j.source_type = a.avcssidtype.type AND j.target_type = a.avctsidtype.type)
            => j in TEAVTabNodes;
[ for a in AllAVCNodes, for_list j as AVTabNode.next starting
      policydb.te_cond_avtab.htable[a.tclass + a.avctsidtype.type * 4 + a.avcssidtype.type * 512].next ],
      (j.source_type = a.avcssidtype.type AND j.target_type = a.avctsidtype.type)
            => j in TECondAVTabNodes;
[ for c in AllACNodes, for a in TEAVTabNodes ],
      (c.avcssidtype.sid = a.source_type AND
       c.avctsidtype.sid = a.target_type AND
       c.tclass = a.target_class) =>
         <c,a> in avcteavtabmapping;
[ for c in AllACNodes, for a in TECondAVTabNodes ],
      (c.avcssidtype.sid = a.source_type AND
       c.avctsidtype.sid = a.target_type AND
       c.tclass = a.target_class) =>
         <c,a> in avctecondavtabmapping;
```

(a) Model Building Rules

```
[ for c in AllAVCNodes ], c.allowed = (c.avcteavtabmapping.allowed | c.avctecondavtabmapping.allwed)
        : notify_admin ("AVC Cache entry has improper privileges " + c.callowed + " at virtual address " + c);
```

(b) Constraints

Figure 5: SELinux access vector cache specification rules.

5 Discussion

The approach proposed in this paper is to detect malicious modifications of kernel memory by comparing actual observed kernel state with a specification of correct kernel state. The specification describes possible correct kernel states, not signatures of known attacks. In this way, our approach is a type of specification-based intrusion detection. We do not follow the approach of traditional signature-based virus scanners. Thus far, we have provided two example specifications for our system and identified the types of modifications that these specifications can detect. While our examples are useful for demonstrating how the proposed system works, they provide little intuition about how specifications would be developed in a real deployment. In this section, we provide a high-level methodology for identifying system properties of interest and describe three classes of threats we have identified.

Currently, there are two methods for identifying data properties and writing their corresponding specifications: (1) analyzing and abstracting on known threats and (2) deriving data properties and specifications from a high-level English-language security policy. In the analysis of known threats, the goal is to classify the techniques used by adversaries in previous attacks in order to abstract on these methodologies. The result is the identification of a set of data invariants that may be violated by future attacks. Of course, this approach permits the possibility that new attacks may avoid detection by exploiting only those details of the kernel abstracted out of the specification, leading to an interminable ”arms race” between attackers and specification-writers. Nevertheless, this approach is still better than the traditional signature-based virus-scanning approach in that each specification has the potential to detect an entire class of similar attacks, rather than only a single instance.

It may be possible to avoid such an arms race by using an alternate approach: deriving specifications from a high-level English-language security policy rather than from an analysis of known attacks. In this approach, an analyst might begin with a policy such as ”no runnable processes shall be hidden” or ”my reference monitor enforces my particular mandatory access control policy” and then examine the kernel source to determine which data structures have relevant properties and what those properties should be in order for the high-level policy to hold. The analyst’s task is similar to constructing a formal argument for correctness, except that the end result is a configuration for a runtime monitor.

Section 4 presents two examples of the types of specifications one might obtain as a result of the

methodologies just described. Using these techniques, we have identified three classes of attacks against dynamic kernel data. While it is likely there are other classes of attacks, we believe the three identified thus far provide evidence of the potential success of our approach. The following are the attack classes we have identified:

Data hiding attacks. This class of attacks was demonstrated in Section 2.1 with the Linux process hiding example. The distinguishing characteristic of this class is the removal of objects from data structures used by important kernel subsystems for accounting and reporting. Writing specifications capable of detecting these attacks requires identifying data structures that are used by kernel resource reporting procedures such as system calls and, in the case of Linux, the `/proc` filesystem.

Capability/access control modification attacks. One of the fundamental goals of kernel attackers is to provide their processes with privileges and access to resources. To this end, process capabilities in the form of tokens, flags, and descriptors are likely targets of an attacker with kernel memory access. In addition to the SELinux AVC example, described in Section 2.2, we have identified user/group identifiers, scheduler parameters (e.g., nice value), and POSIX capabilities as potential targets. We are actively writing specifications to protect this data.

Control flow modification attacks. One popular technique for gaining control of kernel functionality is the modification of function pointers in dynamic data structures such as those associated with the virtual filesystem (VFS) and `/proc` filesystem. As demonstrated by popular rootkits like `adore-ng`, manipulating these pointers provides attackers with a "hook" to execute their inserted code. While previous generations of kernel integrity monitors have demonstrated effective detection of hooks placed in static data (e.g., the system call table), dynamic function pointers have remained an elusive target. We are actively writing a large number of simple specification rules to test the validity of kernel pointers throughout dynamic data. Additionally, we intend to investigate the use of automated tools to make this process easier and more complete.

Unlike misuse detection systems, our specification-based approach allows for the identification and detection of *classes* of attacks without a priori knowledge of particular instances of threats.

6 Related Work

The architecture we have proposed was inspired by the work of four separate areas: external kernel monitors [17, 37, 13, 29], specification-based intrusion detection [20, 32], specification-based data structure repair [7], and semantic integrity in database systems [15]. Work in the areas of software attestation and verifiable code generation is also closely related. We briefly describe this body of work here.

6.1 Kernel Integrity Monitors

We broadly categorize external kernel monitors as any system that operates outside of the protected kernel in order to provide independent, trustworthy analysis of the state of the protected host. Examples of such systems in the recent literature include coprocessor-based monitors [37, 29], SMP-based monitors [17], and virtual machine introspection [13]. While each of these systems has introduced its own mechanism for inspecting the internal state of the protected host, all have at least one common goal: to monitor a running host for unauthorized modifications of kernel memory. While some ad-hoc techniques for limited protection of dynamic data have been demonstrated (although not described in detail) on a couple of these systems [13, 37], the predominant detection technique remains binary or checksum comparison of known static objects in memory.

The types of checks performed by these systems are not incorrect or without value. These systems provide a foundation on which our approach aims to extend to broaden the set of kernel attacks detectable from such platforms.

6.2 Attestation and Verifiable Execution

Code attestation [18, 12, 34, 30, 31, 35] is a technique by which a remote party, the "challenger" or "verifier," can verify the authenticity of code running on a particular machine, the "attestor." Attestation is typically achieved via a set of measurements performed on the attestor that are subsequently sent to the challenger, who identifies the validity of the measurements as well as the state of the system indicated by those measurements [30]. Both hardware-based [12, 30, 31] and software-based [18, 34] attestation systems have been developed. Measurement typically occurs just before a particular piece of code is loaded, such as between two stages of the boot process, before a kernel loads an new kernel module, or when a kernel loads a program to be executed in userspace [30]. All of the hardware-based systems referenced in this paper utilize the Trusted Computing Group's (TCG) [2] Trusted Platform Module (TPM), or a device with sim-

ilar properties, as a hardware root of trust that validates measurements prior to software being loaded at runtime. Software-based attestation systems attempt to provide similar guarantees to those that utilize trusted hardware and typically rely on well-engineered verification functions that, when modified by an attacker, will necessarily produce incorrect output or take noticeably longer to run. This deviation of output or running time is designed to be significant enough to alert the verifier of foul play.

Traditional attestation systems verify only binary properties of static code and data. In such systems, the only runtime benefit provided is the detection of illegal modifications that utilize well-documented transitions or interfaces where a measurement has explicitly been inserted before the malicious software was loaded. Unfortunately, attackers are frequently not limited to using only these interfaces [33].

Haldar et al. have proposed a system known as "semantic remote attestation" [14] in an attempt to extend the types of information the verifying party can learn about the attesting system. Their approach is to use a language-based trusted virtual machine that allows the measurement agent to perform detailed analysis of the application rather than simple binary checksums. The basic principle is that language-based analysis can provide much more semantic information about the properties of an application. Their approach does not extend to semantic properties of the kernel and, since their VM runs on top of a standard kernel, there is a requirement for traditional attestation to bootstrap the system.

Verifiable code execution is a stronger property than attestation whereby a verifier can guarantee that a particular piece of code actually runs on a target platform [33]. This contrasts traditional attestation, where only the loading of a particular piece of software can be guaranteed. Once that software is loaded however, it could theoretically be compromised by an advanced adversary. With verifiable code execution, such a modification should not be possible without detection by the verifier. Both hardware-based [5, 35] and, more recently, software-based [33] systems have been proposed.

Verifiable code execution is a promising direction for ensuring that the correct code is run on a potentially untrusted platform. As shown by Seshadri et al. [33], such a system could be used as the foundation for a kernel integrity monitor. We therefore view verifiable code execution as a potential monitor extension for our architecture.

6.3 Specification-Based Detection

Specification-based intrusion detection is a technique whereby the system policy is based on a specification that describes the correct operation of the monitored entity [20]. This approach contrasts signature-based approaches that look for known threats and statistical approaches for modeling normalcy in an operational system. Typically, specification-based intrusion detection has been used to describe program *behavior* [20, 21, 32] rather than correct state, as we have used it [20, 21, 32]. More recently, specifications have been used for network-based intrusion detection as well [36].

6.4 Data Structure Detection and Repair

We have already described Demsky and Rinard's [7] work towards data structure error detection and repair. This work places one level of abstraction on top of the historical 5ESS [16] and MVS [25] work: in those systems, the inconsistency detection and repair procedures were coded manually. We have utilized the basic techniques of Demsky and Rinard's specification system with the necessary adaptations for operating system semantic integrity. The environments are sufficiently different so as to require significant modifications. These differences were discussed in Section 3.3.

In similar work, Nentwich and others [27] have developed xlinkit, a tool that detects inconsistencies between distributed versions of collaboratively-developed documents structured in XML [4]. It does so based on consistency constraints written manually in a specification language based on first order logic and XPath [6] expressions. These constraints deal with XML tags and values, such as "every item in this container should have a unique name value." In later work [28], they describe a tool which analyzes these constraints and generates a set of repair actions. Actions for the above example might include deleting or renaming items with non-unique names. Human intervention is required to prune repair actions from the list and to pick the most appropriate action from the list at repair time.

6.5 Semantic Integrity in Databases

There is a long history of concern for the correct and consistent representation of data within databases. Hammer and McLeod addressed the issue in the mid 1970's as it applies to data stored in a relational database [15]. The concept of insuring transactional consistency on modifications to a database is analogous to that of doing process accounting within the operating system. The database, like the operating system, assumes that data will be modified only by authorized parties through pre-defined interfaces. While the environments are very different, Hammer and McLeod's work provided excellent insight to us regarding constraint verification. Their system includes a set of constraints over database relations that include an assertion (a predicate like in our system), a validity requirement (analogous to the guard in Demsky and Ri-

nard's model language), and a violation action, similar to our response mechanism but which only applies to updating the database. Hammer and McLeod argue that assertions should not be general purpose predicates (like first-order logic), but should instead be well-defined.

7 Future Work

Each part of the architecture described above provides avenues for significant impact and advancement in the system. The three most promising areas are the extension to other monitors, advancement in system responses, and the analysis and automation of specifications.

We have designed the semantic integrity architecture to be easily extended to other monitor platforms. Two of the most promising such platforms include virtual machine monitors [13, 12] and software-based monitors achieved via verifiable code execution [33]. These systems provide the possibility for unique extensions such as the inclusion of register state into specifications and the benefit of added assurance without the need for extra hardware. It is our intention to extend our work to at least one such software-based monitor.

A second avenue of work we intend to pursue is that of additional response vectors. Having an independent monitor with access to system memory and a system for easily interpreting that memory can provide a huge amount of leverage for advanced response. Perhaps the most significant potential for work is the advancement of automated runtime memory forensics.

Finally, as with all security systems, having a good policy is very important for the success of the system. Our current architecture requires experts with advanced knowledge of kernel internals to write and verify specifications. Developing tools to help automate the process, including a number of kernel static analysis tools, could significantly improve this process. We intend to investigate techniques for analyzing kernel properties automatically, both statically and at runtime.

8 Conclusion

We have introduced a novel and general architecture for defining and monitoring *semantic integrity* constraints—functionality required to defeat the latest generation of kernel-tampering rootkit technology. For our initial prototype implementation, we have adapted Demsky and Rinard's specification languages for implementing internal monitors for application data structures [7] to the task of implementing external monitors for operating system kernel data structures. This adaptation required adding features to their specification languages to overcome a number of issues not present in the original application problem domain, including managing memory transfer overhead and providing for flexible responses to detected compromises.

Our general architecture is applicable to a variety of low-level monitoring technologies, including external hardware monitors [29], software-based monitors [33] and virtual machine introspection [13]. We believe our approach is the first to address the issue of monitoring the integrity of dynamic kernel data in a comprehensive way, and we believe it will provide an excellent complement to present state of the art binary integrity systems.

Acknowledgments

We would like to thank Trent Jaeger for his time and feedback during the final preparation of this work. We would also like to thank the anonymous reviewers for their helpful comments. This work was supported by the National Science Foundation (NSF) under CAREER award 0133092.

References

[1] The Unified Modeling Language (UML). `http://www.uml.org`, 2005.

[2] Trusted Computing Group (TCG). `http://www.trustedcomputinggroup.org`, 2005.

[3] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, pages 18–27, Gaithersburg, Maryland, September 1985.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language. Recommendation REC-xml-20001006, World Wide Web Consortium, October 2000.

[5] B. Chen and R. Morris. Certifying Program Execution with Secure Processors. In *9th Workshop on Hot Topics in Operating Systems (HotOS)*, Lihue, Hawaii, May 2003.

[6] J. Clark and S. Derose. XML Path Language (XPath) Version 1.0. Recommendation REC-xpath-19991116, World Wide Web Consortium, November 1999.

[7] B. Demsky and M. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Anaheim, CA, October 2003.

[8] B. Demsky and M. Rinard. Static Specification Analysis for Termination of Specification-Based Data Structure Repair. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, November 2003.

[9] B. Demsky and M. Rinard. Data Structure Repair Using Goal-Directed Reasoning. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, St. Louis, MO, May 2005.

[10] R. Droms. Dynamic host configuration protocol. Technical Report RFC 2131, Bucknell University, March 1997.

[11] D. Ferraiolo and R. Kuhn. Role-Based Access Controls. In *Proceedings of the 15th National Computer Security Conference*, pages 554–563, Baltimore, Maryland, October 1992.

[12] T. Garfink el, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual-Machine Based Platform for Trusted Computing. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, Sagamore, NY, October 2003.

[13] T. Garfink el and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *The 10th Annual Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, February 2003.

[14] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation – a virtual machine directed approach to trusted computing. In *Proceedings of the 3rd USENIX Virtual Machine Research & Technology Symposium*, May 2004.

[15] M. Hammer and D. McLeod. A Framework For Data Base Semantic Integrity. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, San Francisco, CA, October 1976.

[16] G. Haugk, F. Lax, R. Royer, and J. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT & T Technical Journal*, 64 part 2(6):1385 – 1416, July-August 1985.

[17] D. Hollingworth and T. Redmond. Enhancing operating system resistance to information warfare. In *MILCOM 2000. 21st Century Military Communications Conference Proceedings*, volume 2, pages 1037–1041, Los Angeles, CA, USA, October 2000.

[18] R. Kennell and L. H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *Proceedings of the 12th USENIX Security Symposium*, pages 295–310, Washington, D.C., August 2003.

[19] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29, Fairfax, Virgina, November 1994.

[20] C. Ko, G. Fink, and K. Levitt. Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference (ACSAC)*, Orlando, FL, 1994.

[21] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach . In *1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[22] P. A. Loscocco and S. D. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, Boston, Massachusetts, June 2001.

[23] R. Love. *Linux Kernel Development*. Novell Press, second edition, 2005.

[24] P. Mockapetris. Domain names—conceptsand facilities. Technical Report RFC 1034, ISI, November 1987.

[25] S. Mourad and D. Andrews. On the Reliability of the IBM MVS/XA Operating System. *IEEE Transactions on Software Engineering*, 13(10):1135–1139, 1987.

[26] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria*, December 1985.

[27] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151 – 185, May 2002.

[28] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proceedings fo the 25th International Conference on Software Engineering*, May 2003.

[29] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *13th USENIX Security Symposium*, San Diego, CA, August 2004.

[30] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based Policy Enforcement for Remote Access. In *11th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, November 2004.

[31] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *13th USENIX Security Symposium*, San Diego, CA, August 2004.

[32] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *8th USENIX Security Symposium*, pages 63–78, Washington, D.C., August 1999.

[33] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, United Kingdom, October 2005.

[34] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[35] E. Shi, A. Perrig, and L. V. Doorn. BIND: A Fine-grained Attestation Service for Secure Distributed Systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[36] C. Tseng, P. Balasubramanyam, C. Ko, R. Limprasittiporn, J. Rowe, and K. Levitt. A specification-based instrusion detection system for aodv. In *2003 ACM Workshop on security of Ad Hoc and Sensor Networks (SASN '03)*, Fairfax, VA, October 2003.

[37] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure Coprocessor-based Intrusion Detection. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.