Kernel Data Integrity Protection via Memory Access Control

Abhinav Srivastava  Ikpeme Erete  Jonathon Giffin

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

Abstract

Operating system kernels isolate applications from other malicious software via protected memory created by virtual memory management. Even though modern kernels aggregate core kernel code with driver and module components of different provenance, kernel memory remains unified and without isolation. Kernel-level malicious software has full access to the data and operations of all kernel components. In this paper, we create kernel memory protection. We design an access control policy and enforcement system that prevents kernel components with low trust from altering security-critical data used by the kernel to manage its own execution. Our policies are at the granularity of kernel variables and structure elements, and they can protect data dynamically allocated at runtime. Our hypervisor-based design uses memory page protection bits as part of its policy enforcement; the granularity difference between page-level protection and variable-level policies challenges the system’s ability to remain performant. We develop kernel data-layout partitioning and reorganization to maintain kernel performance in the presence of our protections. We show that our system prevents illegitimate alteration of security-critical kernel data at a performance cost of 1–20%. By offering protection for critical kernel data, we guarantee that security utilities relying on the integrity of kernel-level state remain accurate.

1 Introduction

Commodity monolithic operating system kernels in execution contain a collection of code and data attributable to the core kernel and each of its dynamically-loaded components, such as drivers and modules. In contrast to the isolated virtual memory regions created by the kernel for each user-level application, all kernel components share a unified address space that allows any component to access the data and code of others. The core kernel manages private and security-critical data objects, such as a task list, a loaded module or driver list, and running process’ permissions, that describe fundamental properties of the computer system’s execution. Kernel data privacy is meant in intent only: the unified kernel memory space allows a malicious kernel component to alter the core kernel’s private data at will.

Widespread attacks regularly increase their potency by manipulating security-critical data in the kernel using a technique called direct kernel object manipulation (DKOM) [18, 36]. Rootkits often use this technique to hide the existence of malicious processes by eliding task structures for the processes from the kernel’s process accounting list. Malicious kernel-level components can hide their own presence by illicitly removing data structures identifying their presence from a kernel-managed list of loaded drivers or modules. In another attack, they escalate process’ privileges by overwriting the process’ user credentials with those for a root or administrative user. Security software relying upon the core kernel’s own management information will fail to identify the presence of malicious software. In many attack instances, security software that periodically verifies the consistency of core kernel data structures will be unable to locate any errors.

In this work, we present a system called Sentry that creates access control protections for security-critical kernel data. We partition kernel memory into separate regions having different access control policies or
restrictions detailing when the code of the core kernel and its loaded components can access data in a particular protected region. To avoid both direct and indirect attacks from kernel-level malware, our access control enforcement occurs within a hypervisor executing beneath the kernel running in a virtual machine (VM). The hypervisor mediates the execution of instructions attempting to write protected kernel data. Sentry enforces access control policies that specify what code regions of the kernel are allowed to write what data objects within kernel memory. To determine the origin of an attempted access to protected data, it examines the activation records on the kernel stack at every access.

Our access control protections are general, start from boot, and encompass both statically and dynamically allocated kernel data objects. Providing access control to static data objects is straightforward because they remain unaltered during the execution of operating systems. As evasive attacks often circumvent detection by manipulating heap-managed kernel data objects, we focus primarily on protections for dynamically-allocated data. In our design, the kernel communicates to the hypervisor the need for mediated access to a newly constructed data object at the time that it constructs the object.

Memory access control provides security for kernel data at the cost of diminished runtime performance. Our hypervisor-based enforcement mechanism verifies memory accesses at the granularity of high-level language variables in the kernel’s source code, including individual structure fields contained within a larger aggregate object. We implement mediated access at page-level granularity, the only granularity of memory protection offered by commodity hardware. This mismatch leads to inefficiencies in straightforward implementations of memory access mediation. A memory page containing even one protected variable would require the entire page’s data to be protected. While Sentry can easily distinguish between protected and unprotected regions within a single page, it would still be invoked and introduce performance cost for accesses to all unprotected variables on the page. In the worst case, every access to every kernel data object may be mediated; we expect the costs to be extremely high.

To address the performance challenge, we alter the kernel’s data structure design and runtime memory layout. Rather than allowing protected and unprotected data to freely intermix on memory pages, we segregate data based on its access policy. We cluster together data having the same access policy on the same memory pages, and separate data with different policies onto different pages. In this design, Sentry will be invoked only for accesses to data requiring mediation, as all unprotected data will be placed on unprotected memory pages. To show the effect of improved memory layout, Figure 1 presents measurements of context-switching time between two processes on a Linux system. It is clear that Sentry performs much better than the case where protection is offered to a kernel whose memory layout is not altered.

Critical data protection provides a variety of system and software security techniques with assurances in the integrity of kernel state. A rootkit that attempts to hide itself or a process by manipulating kernel data will trigger an access violation in Sentry; if the rootkit chooses not to attempt the alteration, then traditional rootkit detectors that search for suspicious modules, drivers, and processes will locate the attack. Security software using virtual machine introspection (VMI) [12] to understand the security state of an untrusted

Figure 1: Performance impact of Sentry’s access control protections for the Linux task_struct and module data structures. The graph shows context-switch time measured between two processes; smaller measurements are better. “Normal” measurements have no kernel memory protection, “protected” includes memory protection without performance optimizations, and “Sentry” includes both memory protection and optimization.
guest operating system can rely upon state information in the guest protected by Sentry.

We developed Sentry using the Linux operating system and Xen hypervisor. To demonstrate the feasibility of our ideas, we altered the memory layout of two Linux kernel data structures: task_struct and module. We chose these two data structures due to their complexity, widespread use in the kernel, and frequent alteration by DKOM rootkits. However, our framework is general and supports protection and memory layout optimization of any arbitrary kernel data structure.

Our system is effective at preventing and detecting rootkits without incurring significant performance overhead. We tested Sentry with a collection of rootkits that use DKOM to make malicious modifications to kernel data. It successfully prevented all rootkits that attempted to alter kernel data illegitimately and allowed only legitimate changes performed by the core kernel. We also carried out experiments to test Sentry’s performance using both micro and macro benchmarks. Sentry incurs 0–1% overhead on CPU and network bound processes, and it incurs 20% overhead on mixed loads. The results indicated that Sentry’s design significantly improves system’s performance when compared with the scenario where an unpartitioned kernel was protected (0–16% overhead on CPU and network bound processes and 45% overhead on mixed loads).

The continuing evolution of hypervisors is following multiple paths and multiple philosophies. Our work exists in a vein that considers new functionalities, including security services, that may be provided from a hypervisor [13]. One view of this research direction is that hypervisors are gaining capabilities traditionally found in operating system kernels. The collection of components of varying provenance within a modern kernel’s address space echo collections of applications running at a computer system’s user-level. Sentry’s memory protections then mirror the isolated virtual memory address spaces provided by the OS kernels for such processes.

We believe that this work makes the following contributions:

- We create protected memory regions within the unified kernel data space. A kernel can then isolate its security-critical data from kernel components having low trust, creating assurance in the critical state.
- We protect dynamically-allocated kernel objects that are altered by illegitimate, untrusted kernel components. Sentry detects untrusted components that use DKOM to alter critical kernel data.
- We show how to optimize kernel memory space layout for the protection constraints created by our system. Our layout changes do not impact correctness of kernel execution, but they allow our access control enforcement to operate with higher performance.

2 Related Work

Our primary contribution is integrity protection for kernel data. Previous studies have examined this problem and arrived at solutions different than our own. Petroni et al. [24] proposed a system that detects semantic integrity violations in kernel objects, such as a process task structure reachable when traversing a linked list used by the scheduler but not reachable when traversing the process accounting list. Baliga et al. [1] developed a similar system that hypothesizes invariants on kernel data structures based on observations of the kernel’s execution. Periodic invariant verification attempts to discover the sort of data manipulation addressed by our work, but it has some limitations overcome by Sentry. These techniques succeed only when invariants can be stated for a data object. This is clearly possible for structures like a process, as the invariant can compare the reachable nodes along two different traversals: process accounting list and process scheduling list. It is not evident that invariants can be written for other structures, such as the list of loaded kernel modules, which do not offer multiple views. Sentry operates differently: it mediates all attempted data alterations and allows only those invoked by legitimate kernel functionality. While previous approaches detect malicious modifications, Sentry prevents the illegitimate changes of critical data from even occurring.
XFI [10] provides integrity protection for data (among other protections) via guarded write instructions in software components subject to access control policy constraints. XFI’s use of inlined verification imposes restrictions on software development that may prove difficult to satisfy in actual deployments. For example, it restricts use of the x86 instruction set, requires code that can be statically analyzed, and requires buy-in from all kernel drivers and modules (including rootkit modules). In contrast, Sentry operates with only cooperation from the core, static kernel; dynamically-loaded components are unconstrained. The designs of XFI and Sentry also highlight differences between inlined monitoring and external protection. XFI guards all computed writes to ensure that no write kills a protected value. When many of these writes target unprotected addresses, performance still degrades. Sentry, in comparison, mediates writes only when they actually attempt modification of protected data. XFI’s protections occur at the origin (the write instruction), whereas Sentry’s protections occur at the destination (the security-critical data).

Many kernel-level attacks hook a kernel’s control flow by overwriting a code pointer stored in a static kernel data table or variable. On the Windows platform, PatchGuard [21] protects read-only structures containing code pointers, such as the system service descriptor table (SSDT) and interrupt descriptor table (IDT). Xu et al. [40] implemented a framework that controls access to read-only kernel data structures. Petroni and Hicks [25] detected illicit control-flow transfers arising from hooked code pointers. These systems succeed at the protection or detection of alterations made to read-only dispatch tables or known potential hooks. The goals of Sentry are more general as it provides more extensive protection by additionally mediating access to dynamically-allocated data structures.

We partition kernel memory into protected and unprotected regions and partition data objects into matching components. The general concept of partitioning a single object into secure and insecure portions has appeared as a solution to other software security problems. Multics’ protection rings created different memory regions having different access permissions [7]. Mondrian [39], a fine-grained memory protection system, allows multiple protection domains to share and export services. Ta-min et al. [37] proposed a system that partitions the system-call interface into secure and insecure components. A hypervisor processes secure system calls while insecure calls are handled as usual. Payne et al. [28] split a single security application among multiple VMs and protected the components placed in any untrusted VM. Chong et al. [6] automatically partitioned web applications to ensure that the resulting placement of code and data are secure and efficient. Brumley et al. [3] automatically partitioned a single program into a privileged program that handles all the privileged operations, and an unprivileged program that does everything else. Unlike these examples of interface and application partitioning, where the goal is to offer better security, Sentry uses memory and object partitioning to provide protected kernels with better performance.

Sentry enforces a memory access policy at the hypervisor level and executes the protected operating system within a virtual machine. Wide-ranging security applications use virtualization technology to help build attack-resistant computer systems. Uses include file system protection [43], exfiltration detection [35], rootkit detection [30], malware analysis [8, 22, 42], and other security services [9, 11, 19, 29, 31–33, 42]. Garfinkel et al. [12] proposed the idea of virtual machine introspection (VMI) to detect rootkits; subsequent VMI research developed more comprehensive intrusion detection systems either by inspecting into the VM’s memory [16, 25, 35], disk [27], or other internal events such as system calls [15, 17]. VMI-based security solutions that read the runtime memory state of an untrusted guest system interpret that state to form conclusions about the security condition of the system. This interpretation introduces a trust inversion: it requires the high-privilege security software to trust the data read from the low-privilege guest VM. Sentry’s data protections can provide assurance in the security-relevant information read by a VMI-based utility, which restores the legitimacy of their analysis.

Our low-level memory access mediation uses memory page protection bits to cause deliberate faults when kernel code attempts to alter a protected variable. A hypervisor can provide protection to arbitrary memory pages allocated to a VM. Payne et al. [28] used memory protection to guard hooks inside the guest OS. SecVisor [34] protected static kernel code, though it provided no protection for kernel data. Chen et
Figure 2: Fragment of rootkit code that elevates privileges of non-superuser processes to superuser (ID 0).

```c
asmlinkage int give_root()
{
    if(current->uid != 0) {
        current->uid = 0;
        current->gid = 0;
        current->euid = 0;
        current->egid = 0;
    }
    return 0;
}
```

int init_module()
{
    task = find_task_by_pid(pid);
    if (task) {
        REMOVE_LINKS(task);
    }
}

Figure 3: Fragment of rootkit code that removes a malicious process identified by pid from the process accounting linked list.

al. [5] built a mechanism called multi-shadowing that presents different views of physical memory to protect the privacy and integrity of application data. Litty et al. [20] developed a system that neutralizes the ability of rootkits to hide executing binaries from the administrator by leveraging the non-executable bit of memory pages. Yang and Shin [41] used a hypervisor and cryptography to protect one application’s memory contents from other applications in the system. Sentry uses similar manipulation of page protections to control and monitor memory use within a guest OS.

3 Overview

A running kernel aggregates code and data from the core kernel and from a collection of dynamically-loaded modules and drivers. These different components may engender varying levels of trust in regions of the kernel. We assume that the core kernel implemented by the system’s kernel developers receives full trust. Many modules and drivers, however, have unknown provenance and hold only limited trust. Sentry monitors the interactions between code with low trust and critical data with high trust.

The core kernel includes operations and data exported to modules for their use as well as internal functionality and objects meant to be managed only by the core kernel itself. For example, the list of loaded modules, the process accounting list, the scheduler list, and the run queue of the Linux kernel exist in the core kernel’s data and heap space and are altered by internal functionality of the kernel. However, the lack of memory barriers between the core kernel and its dynamically-loaded components prevents the kernel from disallowing illicit alterations of its internal data structures by malicious modules or drivers.

Consider as an example the Linux kernel’s task management. The kernel stores per-process data, such as user IDs and group IDs that determine a process’ privilege level and allowed access, in an aggregate data structure called the `task_struct`. Each `task_struct` also contains references to the next and previous task structures and thus forms a node in a doubly-linked list. Security tools in the kernel or in a monitoring virtual machine find the set of running processes by traversing the doubly-linked list.

Suppose that a malicious application wants to execute undesirable functionality as a high-privilege user while remaining elusive from security software that searches for unexpected processes. The application may include a kernel-mode rootkit that elevates the malicious process’ privilege by directly writing to its `task_struct` and hides the process by altering the kernel’s process accounting list. Figure 2 shows a C code snippet from a rootkit [26] that assigns superuser privileges to its malicious process. In the code shown in Figure 3, a rootkit [38] uses the kernel macro `REMOVE_LINKS` to remove its malicious process’ `task_struct` from the doubly-linked process accounting list. This macro alters the `next` and `previous` pointers within the list and allows the rootkit to hide its malicious process.
3.1 Threat Model

We assume that a powerful adversary is able to execute malicious code both at the kernel and user level, and is able to alter kernel data structures at will. Kernel-level access can be achieved via kernel-level exploits or social engineering that entices a user to install a malicious kernel module, or by some other security flaws present in the system. This threat model reflects the current generation of rootkit attacks.

Due to kernel-level attacks, our defense is implanted in a hypervisor to provide tamper resistance. Virtualized environments create a security virtual machine (VM) and multiple user VMs. Our threat model considers the hypervisor and security VM as part of the trusted computing base (TCB). User VMs are untrusted and are open to both direct and indirect attacks. We do assume that the core kernel code of a user VM is protected and cannot be subverted by any malware running in the user VM. This requirement can be realized by making the core kernel’s code pages read-only [34].

3.2 Kernel Data Integrity Model

Potentially malicious kernel modules should not be able to alter security-critical data intended to be managed only by the core kernel. Sentry protects such data and enforces data access restrictions based upon the origin of the access within the code of the kernel and its modules or drivers. The data integrity model is straightforward and matches that of the Biba ring policy [2]:

- Trusted core kernel code may write to any kernel data.
- Loaded modules and drivers can write to any low-integrity data, which includes all driver data and portions of the core kernel data.
- Loaded modules and drivers cannot write to high-integrity data of the core kernel either through a direct write or by calling into existing code of the core kernel that will execute the write on behalf of the module, unless the control-flow transfer into the core kernel targets an exported function. Exported functions act as trusted upgraders. The intent of the kernel developers was to provide an API through which modules and drivers can legitimately make changes to critical data, and the changes leave the kernel in a consistent state.

To determine if a loadable kernel component called into a core kernel function to induce alteration of high-integrity data, Sentry performs a stackwalk of the kernel stack to identify activation records and the call chain that led to an attempted write.

Consider again the rootkit behaviors of Figures 2 and 3. When the rootkit attempts to directly modify the privileges of a process, Sentry will mediate the write to security-critical data of the core kernel. The malicious code that modifies privileges by directly writing to memory is in a loaded module and not in the core kernel code, so Sentry will prevent the write (and optionally alert the system’s user or administrator of a likely malware infection). Should the macro expansion of the rootkit’s attempted unlisting of the task_struct from the linked list include function calls into internal list management functions of the kernel, then Sentry’s stackwalk will step into an activation record for the code of the loaded module. Again, Sentry will deny the data write.

4 Kernel Memory Access Control

We designed a hypervisor-based kernel data protection system, Sentry, that protects sensitive kernel data from unauthorized modification. Its design reflects these goals:
• **Resist direct attack:** Sentry protects a kernel’s dynamic data from rootkits. Since a rootkit executes in kernel space, any kernel-level tools can easily be subverted. In order to be tamper-resistant, Sentry’s design uses a hypervisor to remain isolated from an untrusted kernel.

• **Performance:** Mediation of memory accesses may incur high performance cost if designed na"ıvely. To keep the overhead low, Sentry uses a novel memory partitioning based approach that lays out sensitive members on separate memory pages and protects those pages using the hypervisor.

Sentry’s architecture consists of three components: a hypervisor, a user virtual machine (VM), and a trusted security VM (Figure 4). The hypervisor is the heart of Sentry’s protections, comprised of the memory access policy description, a memory protection module, and a policy enforcement module. The policy statement includes a description of trusted code regions allowed to modify secure data. The memory protection module protects all kernel memory pages containing security-critical variables. The policy enforcer mediates attempted writes to protected data and uses the policy to determine when writes should be permitted. If permitted, the enforcer’s instruction emulator emulates the write operation in a manner transparent to the user VM. The enforcer includes functionality to extract execution history in the form of activation records present on the guest kernel’s call stack. The user VM runs the operating system that may fall victim to kernel-level attacks. Finally, the security VM runs the controller software that starts and stops Sentry’s protections and receives alerts from the hypervisor when a malicious modification is attempted.

Sentry currently supports the Xen hypervisor and Linux guest kernels, and our continuing presentation of Sentry will use Xen and Linux terminology. Sentry’s techniques are general and apply to many hypervisors and operating systems.

### 4.1 Policy

Sentry enforces integrity protection policies based on the trust level of code attempting to alter critical data. Sentry’s policies describe code regions or function call chains that are allowed to modify security-critical kernel data. Any access request that does not fall into pre-defined trusted code regions will be denied. We identify the following three types of code regions that can legitimately modify protected data:

1. **All core kernel code** (that not in loadable modules or drivers) is trusted to correctly manage its own private data. This code spans memory from the Linux kernel symbol _text to _etext.

2. **Kernel code from _init_begin to _init_end** contains code required for the kernel to successfully boot and is likewise trusted.
3. Alterations reachable from an exported function’s entry point reflect valid management of private kernel data even when a loaded component calls into the function. Exported functions are deliberate APIs created by the core developers specifically for loadable modules and drivers; these functions leave an operating system’s kernel in a consistent state.

4.2 Activation of Mediated Access

To enforce integrity protections, Sentry must mediate all attempts to overwrite security-critical data. Sentry uses Xen’s management of memory page access protections to disable write permissions on any page holding protected kernel data. This protection provides Sentry with the ability to interpose on write accesses to protected memory pages; any write operation to a protected page causes a page fault, and on each fault the hypervisor gains the control of execution.

Sentry’s memory protection relies on knowledge of the location of dynamically-allocated kernel objects. It uses code instrumentation within the user VM’s kernel to activate and deactivate protections in concert with object construction and destruction. The instrumentation uses the Intel virtualization instruction \texttt{VMCALL} [14], which transfers control to the hypervisor when executed. Each \texttt{VMCALL} passes the page frame number (PFN) and virtual address of the newly allocated memory page requiring protection. We assume that existing techniques can protect the core kernel code’s integrity [34], so an attacker will not be able to remove our instrumentation. Sentry prevents attackers from making a \texttt{VMCALL} to deactivate protection by verifying that the request was generated within the core kernel at an expected execution point.

Sentry receives the \texttt{VMCALL} within the hypervisor and handles the request from the guest. When the memory protection module receives a request to add protection for a guest’s page, it adds the PFN to a list of protected pages and removes the page’s write permission. Sentry also flushes the translation lookaside buffer (TLB) cache to remove any previous mappings that may exist for the protected PFN. When a request to remove protection on a page comes to the memory protector, it removes the PFN of the page from the list of protected pages and restores write permission. Figure 5 presents a flowchart describing the entire process of adding and removing protection.

4.3 Policy Enforcement

Sentry’s policy enforcer also resides in the hypervisor, and its duty is to enforce the pre-defined security policies. Both legitimate and malicious writes to protected pages will cause a page fault received by Xen, providing opportunities for mediation. At each fault, the enforcer determines if it is due to Sentry’s protection by verifying the PFN of the faulting page. If the PFN belongs to the list of protected PFNs, then it performs further actions, otherwise, it directs Xen to resume normal operation. On a page fault caused by Sentry’s protection, the enforcer first uses the user VM’s instruction pointer (\texttt{eip}) to know which code is
directly attempting to write to the protected page. If the instruction pointer belongs to an untrusted code region, then the access must be denied. If the instruction pointer belongs to a trusted code region, then the enforcer must ensure that the trusted code was invoked legitimately. It extracts the execution history of the kernel associated with the memory write by executing a stackwalk of the user kernel’s stack. When encountering a stack frame for a core kernel function with a return address pointing back to untrusted code, Sentry checks to see if the core function is part of the kernel’s exported API. If not, then untrusted code invoked an unsafe call into the kernel, and the memory alteration must be prevented. Otherwise, Sentry allows the write operation.

Once Sentry determines that a write is permitted by the integrity protection policy, it emulates the write in the same way that the guest system would have done had the protection been absent. When the emulation completes, Sentry updates the guest context registers so that the mediation of writes remains completely transparent to the guest. The complete flowchart of resolving a write page fault is shown in Figure 6.

Figure 6: Steps used by Sentry to resolve a write fault on a protected page.

4.4 Memory Layout Optimization

The normal layout of kernel objects in memory challenges Sentry’s ability to achieve the performance goal. Sentry’s policies protect individual kernel variables and fields of aggregate structures, but it mediates writes at the granularity of complete memory pages. Structures, like `task_struct` and `module`, contain a mix of security-critical and non-critical fields laid out contiguously in memory, thereby making it difficult to provide efficient and fine-grained protection. If any variable on a page is protected, the entire page suffers the overhead of mediated writes. We would prefer that only write operations to security-sensitive members invoke a fault, thereby eliminating the performance overhead resulting from faults generated by writes to non-critical data.

To address this problem, we developed two approaches to partition a data structure into secure and insecure pieces. All secure structures should be located together on protected pages, and all insecure structures can reside on unprotected pages. Once we identify all sensitive members (described in Section 4.5) in a kernel data structure, say `Obj`, we partition it. The first approach, *structure division*, partitions `Obj` by creating a new data structure `insecure_Obj` and moving all non-sensitive members into this new structure. This procedure leaves the original `Obj` structure with security-critical members only. For example, Figure 7 shows the partial division of the Linux `task_struct`. A kernel using partitioned structures must allocate memory separately for the secure and insecure pieces to create different memory regions for protected and unprotected members.

All existing kernel code must be updated to use partitioned structures. All members that belong to `Obj` can still be accessed as before, however, code that accesses non-critical members must access them through `insecure_Obj`. To solve this problem, we first linked `Obj` and `insecure_Obj` by adding a new pointer field in `Obj` called `insecure` that points to the `insecure_Obj` structure. Second, we modified all affected kernel code by adding another level of indirection through the `insecure` pointer. For example, if code in the kernel was accessing the field `journal_info` as `current->journal_info`, it is modified to become `current->insecure->journal_info`, where `current` points to the `task_struct` of
the currently executing process. Although structure division requires code revision, it is largely a one-time design cost borne by kernel developers offering long-term improvements to a kernel’s security. Subsequent, unrelated kernel development may incur small maintenance costs as developers choose to shift variables to or from the protected region.

A variation of the structure division approach keeps insecure members in Obj and moves secure members to a new data structure, say secure_Obj. It links the two structures with a new pointer from Obj to secure_Obj. This design creates a security flaw in the system. An attacker who wants to hide her malicious process can easily remove the link from Obj that points to secure_Obj because Obj contains non-critical members and is not protected by Sentry. Once the pointer to secure_Obj is removed, she can easily add a new pointer that points to her own secure_Obj, which is not protected by Sentry. Our previously described partitioning approach that contains pointers from secure to insecure does not suffer from this problem. Since all secure members—including the pointer to insecure members—are protected, no such modification can occur.

Our second partitioning approach, structure alignment, places protected and unprotected variables of a unified data structure on separate memory pages by aligning the structure in memory so that it lays across a page boundary. Only one of the two pages is protected by Sentry, and the structure’s security-critical fields lay on that page. To accomplish the structure alignment, we group all security-critical members in Obj together at the start of the structure, group all non-critical fields at the end, and add a new alignment buffer to the structure between the two fields. This buffer is simply padding that forces all non-critical members onto the second, unprotected memory page.

These partitioning strategies have trade-offs. Structure division provides freedom in laying out the protected and unprotected structures in memory at runtime, though its need for kernel source code revision adds complexity. Structure alignment has only a minimal source code change to the definition of the structure, and it could easily be integrated into an existing kernel by creating a compile-time option that inserts or removes the alignment buffer. However, it may not be an effective use of kernel memory, and it imposes constraints on runtime memory layout. We demonstrate in Section 5 that both designs are feasible, even on pervasive kernel data structures.

### 4.5 Identification of Security-Critical Members

Our memory layout optimization design depends upon the identification of security-critical and non-critical members in a data structure. We define a member in a data structure to be security-critical if:

- It is manipulated by attackers to carry out malicious activities. This approach provided a reasonable idea of those data structure fields commonly modified by attackers and needing immediate protection.
For example, many Linux-based rootkits modify uid and gid fields in a task_struct to elevate privileges of their malicious process. In another example, they hide the presence of their malicious modules by modifying the next and prev pointers of a module structure. Based on this notion, we collected rootkits to identify kernel variables that they alter.

- Subject-matter experts, such as core kernel developers, identify the variable as security critical. They can identify important members in a kernel data structure during a development phase before they are misused by attackers.
- Defensive systems rely on the variable’s integrity in order to understand the security state of the user VM. For example, VMI applications [16] often rely on the integrity of process accounting lists and filesystem structures when constructing a trusted view of the system.

We believe that these criteria offer sufficient ability to identify security-sensitive members present in kernel data structures.

### 4.6 Sentry Controller

The security VM runs an application that controls Sentry’s operation. The controller can enable or disable Sentry’s protection inside the hypervisor. It interacts with the hypervisor using the hypercall interface provided by Xen. Whenever the hypervisor component of Sentry records an attack, it informs the controller so that necessary actions can be taken, as specified by the policy or by an administrator.

### 5 Implementation

We developed Sentry using Linux 2.6.16 guest operating systems and the Xen hypervisor version 3.2. We describe low-level implementation details of our system in this section.

#### 5.1 Data Structure Layout

We applied our partitioning techniques to Linux’s task_struct and module data structures. We chose these two data structures due to their complexity, their relevance to current kernel-level attacks, and their pervasiveness in the kernel. The process data structure is important to the kernel because it is the fundamental unit of execution, and its complexity is based on the fact that it contains 122 members. The module data structure has 29 members and is used when any driver or module is loaded or unloaded, and whenever any subsystem of the kernel implemented as a driver, such as a filesystem, disk, or network device, is accessed.

To demonstrate the feasibility of different partitioning strategies, we partitioned each of the two structure types with different techniques. We applied structure division to the widely-used task_struct (Section 5.1.1) and structure alignment to the module structure (Section 5.1.2).

##### 5.1.1 Partitioning of task_struct

To apply our partitioning strategy on the task_struct structure, we first identify its security-critical members. As described in Section 4.5, we identified critical members by analyzing rootkits and the Linux kernel source code. We categorized 28 of 122 members as critical and chose those for protection and partitioning. We divided task_struct into two parts: task_struct containing security sensitive members, and insecure_task_struct containing all non-sensitive fields.

Before partitioning, memory is allocated to an instance of the task_struct via kmem_cache_alloc. This function allocates the requested memory bytes by retrieving them from a cache of available memory.
The retrieved memory size might cross page boundaries, consequently making it difficult to provide protection for only those members that need it. Hence, we changed the memory allocation to instead use `get_free_pages`, which allocates complete pages, and `kmalloc`. Using `get_free_pages`, we allocated each task_struct on a complete page, thereby separating the critical members from non-critical fields in the kernel memory; we allocated each instance of `insecure_task_struct` using `kmalloc`. As described previously, we connect the two substructures by maintaining a reference from task_struct to insecure_task_struct. We finally modified Linux’s free_task function to deallocate the memory pages allocated to task_struct and insecure_task_struct separately.

The first process created on a Linux system during boot, init_task, is defined specially in the kernel code and requires its own partitioning. Its creation is different from other processes—other processes are created and initialized using the kernel function `do_fork`, which copies the current process’ parent’s task_struct into the current process’ memory and then modifies the necessary members, but init_task has no parent process. The init_task object is created separately by the kernel and initialized using the INIT_TASK macro, a static initializer of the init_task process. To apply our partitioning and protection on the init_task object, we modified the existing INIT_TASK macro and created a new macro called INIT_UNSCR_TASK. We removed the initialization of all unprotected members from INIT_TASK and inserted them into the new macro. In this way, all security-critical members of init_task are initialized using the INIT_TASK macro, and the remaining non-critical members are initialized using INIT_UNSCR_TASK.

After the structure partitioning, we modified the Linux kernel source code in order to work with the partitioned structure. As part of this process, we also updated driver code since some drivers also rely on the process structure for portions of their functionality.

### 5.1.2 Partitioning of module

We partitioned Linux’s module structure using our page alignment technique. We categorized 2 of the structure’s 29 members as critical and separated them from the non-critical members with an alignment buffer that places the critical fields on a different page than non-critical fields at runtime. We first grouped all the security-critical members in the module data structure together by reorganizing the data structure. Our new alignment field provided padding that filled the rest of the memory page and caused the non-critical members to cross the page boundary to a new, unprotected page. This alternative partitioning technique did not require creation of a new insecure structure as done for the task_struct. The approach required no source code modification beyond the addition of the alignment buffer and required only a straightforward recompilation of the kernel due to the changed field offsets within the module structure.

### 5.2 Access Mediation and Policy Enforcement

The memory protection system of Sentry operates in two phases, both occurring concurrently: a management phase when the kernel adds or removes a page frame number (PFN) to or from the list of protected PFNs, and the mediation phase providing the actual memory protection. To perform addition, removal, and lookup operations on a PFN, we created new APIs inside the Xen hypervisor. The API includes the functions `addPFNToDB`, `removePFNFromDB`, and `checkPFNInDB`, providing Sentry with the ability to add, remove, and find a protected PFN, respectively.

The second phase works by modifying the shadow page table (SPT) code of Xen. The SPT is the native page table used by the hardware and managed by Xen. To provide memory protection, we modify the `sh_propagate` function. When a new memory page is added to the guest page table, `sh_propagate` propagates this entry to the SPT to keep both the tables in sync. While propagating this update, the memory protection system of Sentry intercepts the update and checks whether the propagation involves a protected
page. If a page belongs to the list of protected pages, it removes the write permission from the page by setting the page write bit to zero in the SPT.

To intercept subsequent write faults on a protected page, we modified the shadow page fault handler function called *sh_page_fault*. When a fault occurs, Sentry’s code inside *sh_page_fault* verifies whether or not the fault is a result of its protection mechanism. The verification dictates how the fault is to be processed by Sentry. If the fault is due to some other activity in the guest, then Sentry ignores it and resumes the normal operation. Otherwise, Sentry looks into the fault to determine what code region is attempting to write to the protected page, as described in Section 4.3.

5.3 Instruction Emulation

When the kernel memory access control policy permits a mediated write operation, Sentry must reproduce the effects of the operation in a guest operating system’s memory. This capability resides in Sentry because the guest operating system cannot execute the write operation itself due to the write protection bit set on the faulted page. We implemented an instruction emulator inside Xen to perform the emulation of memory writes. With this emulator, Sentry can replay attempted writes subsequently when determined legitimate. To emulate an instruction, Sentry needs to first locate the instruction and then fetch it from guest memory. To achieve this, Sentry uses Xen’s function called *hvm_copy_from_guest_virt* that reads and writes to arbitrary guest locations.

When a faulting instruction is fetched, the emulator decodes and executes the instruction inside Xen. Depending upon the instruction type, the decoding process may identify source and destination operands. The emulator executes the instruction by reading and writing the memory locations directly from the hypervisor. To ensure transparency to the guest operating system, it updates all context registers including the instruction pointer to point to the next instruction.

5.4 Execution History Extraction

The execution history extractor of Sentry performs a walk on the kernel stack of the guest operating system by mapping the guest kernel’s stack pages into the hypervisor’s memory and then traversing stack frames present on the pages. Performing a walk on the kernel stack is challenging due to the unstructured layout of call stacks on x86 processors and the presence of interrupt stack frames on a kernel stack. To successfully extract stack frames, we complied our guest kernel with a compile-time option that produces kernel code maintaining a stack frame base pointer (*ebp* register) throughout the kernel’s execution. Fortunately, interrupt stack frames do not pose problems for Sentry because Sentry only performs stackwalks following a page fault that occurred due to its protection; it does not perform a stackwalk on the kernel stack at arbitrary points of the kernel’s execution. Sentry additionally pauses the guest kernel while executing the stackwalk to ensure that no modifications occur while reading the guest’s memory state.

When a page fault occurs, the extractor finds the location of the current stack frame from the *ebp* register. It subsequently determines the location of the return address by adding 4 bytes to the value of *ebp*. To get the actual return address, the extractor reads the value present on the computed return address location. To get the previous frame, it extracts the address stored at the location pointed to by the *ebp* register’s contents. The extractor repeats this process until it reaches the end of the stackwalk.

6 Evaluation

In this section, we evaluate both the attack detection capability and the performance of Sentry. We also perform detailed analysis of false positives associated with our system.
Table 1: Sentry’s attack detection results against Linux based rootkits that modify the kernel’s process and module data structures.

<table>
<thead>
<tr>
<th>Name</th>
<th>Hidden Process</th>
<th>Hidden Module</th>
<th>Privilege Escalation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>hp</td>
<td>√</td>
<td></td>
<td></td>
<td>Detected</td>
</tr>
<tr>
<td>all-root</td>
<td></td>
<td>√</td>
<td></td>
<td>Detected</td>
</tr>
<tr>
<td>kbd-version2</td>
<td></td>
<td>√</td>
<td></td>
<td>Detected</td>
</tr>
<tr>
<td>kbd-version3</td>
<td></td>
<td>√</td>
<td></td>
<td>Detected</td>
</tr>
<tr>
<td>override</td>
<td></td>
<td>√</td>
<td></td>
<td>Detected</td>
</tr>
<tr>
<td>synapsys</td>
<td></td>
<td>√</td>
<td></td>
<td>Detected</td>
</tr>
<tr>
<td>rkit</td>
<td></td>
<td></td>
<td></td>
<td>Detected</td>
</tr>
<tr>
<td>lvtes</td>
<td>√</td>
<td>√</td>
<td></td>
<td>Detected</td>
</tr>
<tr>
<td>adore-ng</td>
<td></td>
<td>√</td>
<td></td>
<td>Detected</td>
</tr>
</tbody>
</table>

6.1 Security

Sentry prevents unauthorized modification of security-critical kernel data structures. Note that Sentry is able to protect both static and dynamic kernel data structures. However, our experiments focus on rootkit identification based upon their attempted alteration of dynamic kernel data, as this represents a significant new advance in defensive capabilities.

We tested Sentry against a collection of DKOM rootkits present in the wild. Table 1 shows our Linux rootkit samples and the malicious behaviors that they exhibit. During our testing, we ran each rootkit sample in the user VM, which was running our Linux 2.6.16 kernel with partitioned task struct and module data structures. Sentry’s protections started at kernel boot so that all processes beginning with the init process and all modules can be protected. Below, we provide a detailed description of Sentry’s detection of the lvtes keylogger and the all-root rootkit.

The lvtes keylogger [4] hides its malicious module by removing it from the doubly-linked module list. It uses the kernel macro list del, which directly deletes a module from the list by modifying a member called list in the module structure. This member stores next and previous pointers of the list. To test Sentry against lvtes, we inserted the keylogger module in the user VM’s kernel. When the rootkit tried manipulating list.next and list.prev to remove the module from the doubly-linked list, it caused a page fault because list is considered to be a sensitive member and is protected by Sentry. The enforcement mechanism of Sentry verifies whether the access should be allowed or denied by checking which code attempted to modify protected members. Sentry looks at the instruction pointer, which in this case belongs to an untrusted code region in a module. Consequently, Sentry denies the access to the rootkit.

The all-root rootkit [26] directly modifies the uid, gid, euid, and egid members of a task struct structure. These members determine the privilege level of a process, which in turn restricts the types of execution that a process can perform. When the rootkit loads, it hooks into the system-call table and replaces the function pointer associated with the getuid system call with its malicious function mal_getuid. When a malicious process cooperating with the rootkit executes getuid, the rootkit’s mal_getuid function will execute instead and will set the uid, gid, euid, egid of the process to 0. This escalates the privileges of the invoking process to a superuser. We tested this attack with Sentry, and Sentry detected this modification because these fields were protected. Modifications to these fields caused a page fault, and Sentry found that the modifying code was in a module and denied the access.

We tested Sentry with other rootkits as well, and our results, shown in Table 1, indicate that Sentry provided a 100% detection rate for DKOM rootkits.
<table>
<thead>
<tr>
<th>Operations</th>
<th>Normal VM (µs)</th>
<th>Protected (µs)</th>
<th>Overhead (µs)</th>
<th>Sentry (µs)</th>
<th>Overhead (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process fork+exit</td>
<td>342.89</td>
<td>1654.45</td>
<td>1311.56</td>
<td>1361.27</td>
<td>1018.38</td>
</tr>
<tr>
<td>Process fork+execve</td>
<td>355.69</td>
<td>1774.66</td>
<td>1418.97</td>
<td>1458.9761</td>
<td>1103.28</td>
</tr>
<tr>
<td>Process fork+/bin/sh -c</td>
<td>2482.30</td>
<td>8152.05</td>
<td>5669.75</td>
<td>5811.0323</td>
<td>3328.73</td>
</tr>
<tr>
<td>Context switch</td>
<td>1.85</td>
<td>41.35</td>
<td>39.50</td>
<td>6.46</td>
<td>4.61</td>
</tr>
</tbody>
</table>

Table 2: Process creation and context-switch time measured with lmbench; smaller measurements are better. The CPU time-slice on the system was 100 ms.

6.2 Performance

Sentry is designed to provide both security and performance. It provides security by protecting kernel dynamic data, and it partitions kernel data structures to achieve usable performance. We carried out several experiments to measure the performance overhead incurred by the protection offered by Sentry as well as to show the effect of partitioning. Our premise is that if the same protection is offered to both a partitioned and an unpartitioned kernel, the partitioned kernel outperforms the latter. To determine the efficacy of Sentry, we tested its performance with both micro and macro benchmarks. Our results were measured on an Intel Core 2 Quad 2.66 GHz system with Fedora 8 in the security VM and our partitioned Linux 2.6.16 kernel in the user VM. We assigned 1GB of memory to the user VM and 3GB total to the security VM and Xen. Unless otherwise specified, we performed the experiments reported in tables five times and present the median value. We also show the results of some experiments using boxplots due to measurement variations common to virtualized environments. In all the experiments, “normal” refers to measurements that have no kernel memory protection, “protected” includes memory protection without partitioning, and “Sentry” includes both memory protection and partitioning.

In our micro benchmark experiments, we first measured the effect of protection and partitioning of a process data structure. We measured the effect with two tests that exercise heavy legitimate use of the protected structures—process creation and context switch time—using the lmbench\footnote{http://www.bitmover.com/lmbench/} Linux benchmark tool. Lmbench performs three different experiments to measure the cost of process creation. Our results, shown in Table 2, indicate that Sentry incurs low overhead on a partitioned kernel as compared to the overhead on an unpartitioned kernel.

In another micro benchmark experiment, we evaluated the effect of protection and partitioning of module structures on Linux modules by measuring the time taken by module load (\texttt{insmod}) and unload (\texttt{rmmod}) operations. We wrote a sample module (that traverses the list of loaded modules) and inserted it using the \texttt{insmod} program. The same module is then unloaded using the \texttt{rmmod} program. Figures 8 and 9 present the results, and it can be seen that the loading and unloading time is higher for the unpartitioned kernel when compared with Sentry’s time.

In the next set of experiments, we measured the effect of Sentry on filesystem cache performance. To test this, we used a filesystem benchmark called IOzone\footnote{http://www.iozone.org}, which measures the throughput of read, write, re-read and re-write operations. Figure 10 shows quartile measurements of ten repetitions of each test. Note that these results show use of the kernel’s filesystem cache—rather than of disk operations—due to the instability of disk performance measurements in a virtualized environment. Sentry’s protection incurs less overhead on file cache operations when performed using the partitioned kernel as compared to its effect on unpartitioned kernel, and in many cases, its performance nears that of the unprotected system.

To measure Sentry’s performance on real world software, we tested it against full applications. In our first two experiments, we compiled a Linux kernel and Apache source code in the user VM. Source code compilation invokes many \texttt{gcc} processes and disk reads & writes, providing a mixed load to measure Sen-
Figure 8: Module loading operation performed via `insmod`; smaller measurements are better.

Figure 9: Module unloading operation performed via `rmmod`; smaller measurements are better.

Figure 10: Performance impact of kernel memory protection on use of the kernel’s file cache. All measurements show throughput in MB/s; higher measurements are better. Boxes show medians and first and third quartiles. Outliers appear as dashes. Groupings show performance of (a) file-cache cold reads, (b) cache-warm reads, (c) cache-cold writes, and (d) cache-warm writes.

try’s performance. Table 3 shows the results of mixed loads, and it is evident that Sentry’s partitioning has reduced the overhead caused by its protection by 50% when compared with the overhead on an unpartitioned kernel.

The next two experiments measure Sentry’s performance on CPU-bound applications. We performed these experiments with the Linux compression and decompression utilities `bzip2` and `bunzip2`. We performed the compression experiment by providing a 225 MB tar file to `bzip2` for compression. Later, we performed the reverse operation by passing the 40 MB compressed file to `bunzip2` to decompress. The results show that Sentry’s overall effect on CPU-bound applications is negligible; a system with this protection is certainly usable.

Our last experiment measures Sentry’s effect on network processes by testing it against network loads. We copied a file of size 174 MB over the physical network using a `thttpd` daemon accessed by the `wget` command. The results, shown in Table 3, demonstrate that Sentry’s overhead is negligible for network workloads.

The above results provide a strong evidence that Sentry can provide a balance between security and performance. Its performance on several applications is encouraging and suggests it may be suitable in real world environments.
<table>
<thead>
<tr>
<th>Operations</th>
<th>Normal VM (sec)</th>
<th>Protected (sec)</th>
<th>% Overhead</th>
<th>Sentry (sec)</th>
<th>% Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Compilation (make)</td>
<td>56.394</td>
<td>75.246</td>
<td>33.42</td>
<td>66.680</td>
<td>18.23</td>
</tr>
<tr>
<td>Apache Compilation (make)</td>
<td>5.876</td>
<td>8.526</td>
<td>45.10</td>
<td>7.147</td>
<td>21.63</td>
</tr>
<tr>
<td>Compression (bzip2)</td>
<td>41.905</td>
<td>43.435</td>
<td>3.65</td>
<td>42.245</td>
<td>0.80</td>
</tr>
<tr>
<td>Decompression (bunzip2)</td>
<td>10.088</td>
<td>11.690</td>
<td>15.88</td>
<td>10.113</td>
<td>0.24</td>
</tr>
<tr>
<td>Network File Transfer (wget)</td>
<td>18.455</td>
<td>19.092</td>
<td>3.45</td>
<td>18.587</td>
<td>0.71</td>
</tr>
</tbody>
</table>

Table 3: Performance impact of Sentry on real-world applications; smaller measurements are better.

6.3 Potential Performance Improvements

Although the previous section shows the overhead of Sentry on real world applications to be low, its performance can further be improved by providing efficient ways to do common operations. We have identified the following improvement opportunities:

**New memory allocator.** When using structure division partitioning, Sentry allocates a security-critical structure on a protected page using the kernel function `get_free_page`, which returns a new page from the kernel memory pool. Since the security-critical portion of each data structure is allocated on a new page, this allocation wastes system memory because these pages are not used further.

An improved memory allocator can better utilize a protected page by allocating more secure structures on the same page until the page gets filled. When one page gets filled, the kernel allocates a new page for secure structures. This approach provides two advantages: first, it reduces the TLB flushing that happens inside Xen when Sentry adds a new page to the list of protected pages. Second, this approach reduces the number of VMCALLs from the guest to Xen requesting protection on a new page because the page where a new structure will be located may already be protected. The downside of reusing a page is that it leads to increased page faults during the initialization of a new structure.

**Mapping memory pages.** Sentry uses the memory mapping and unmapping functions of Xen, such as `hvm_copy_to_guest_virt` and `hvm_copy_from_guest_virt` during the process of stackwalk and instruction emulation. These functions access the guest VM’s memory from the hypervisor, however, Xen provides a very inefficient implementation. On each invocation of these functions, Xen maps a requested page from the guest kernel’s memory into the hypervisor memory, performs the operation, and then unmaps the page.

An improvement can be made in the implementation of these functions by keeping a memory page mapped inside the hypervisor memory to provide locality of reference. For example, during a stackwalk, Sentry maps and unmaps a stack page everytime when it accesses a return address and frame pointer. If the page were mapped once during a single walk, then Xen would have avoided multiple mapping and unmapping operations.

6.4 False Positive Analysis

A false positive in our approach occurs when a security-critical member is modified by a benign module or driver that violates our integrity policies described in Section 4.1. Our analysis revealed the following:

- There were no instances when security-critical kernel data protected by Sentry is directly modified by a benign driver. Hence, any such attempts must be detected as attacks.

- Whenever security-critical data protected by Sentry was altered by a benign driver, it was done using exported functions designed by kernel developers for that purpose, and they left the kernel in a consistent state. We illustrate this point with an example: a `task_struct` contains a member
called run_list, which is similar to tasks (pointer to accounting list), but contains next and previous pointers for scheduling list; Sentry protects the run_list member. These pointers are modified by functions such as enqueue_task and dequeue_task, which in turn are called from the schedule function, which is exported by the kernel. The schedule function is invoked on each context-switch and modifies the run list; it is also invoked from all modules. Since our policy allows changes made to kernel data via exported functions as they act as trust upgraders, whenever members such as run_list were modified, we verified the call-chain and allowed the modification.

With the above design in place, our system did not show any false positives and detected all attacks.

7 Discussion

Sentry is designed to protect security-critical kernel data. It partitions a data structure into secure and insecure structures and protects the secure structure using the hypervisor. In this section, we discuss some of the possible ways in which an attacker may try to bypass our system to manipulate secure data. We also comment on possible ways to overcome the limitations of Sentry, and on its other applications.

**Stack Frame Farming Attack.** An attacker cannot bypass our protection system as any write operation on a protected page invokes Sentry. However, a motivated attacker who wants to bypass Sentry can launch a stack frame farming attack. In this attack, an attacker creates fake frames on the kernel stack and jumps into the kernel code to modify protected data. During a stackwalk, Sentry concludes that the request has originated from trusted code regions because of fake frames. Since a trusted code region is allowed to make modifications to security-critical kernel data, the access would be granted. Though this kind of attack is possible, it is challenging to launch on a kernel stack. A kernel stack is more complicated than a process stack as it keeps both function call frames and interrupt frames, and any corruption of an interrupt frame may crash the system.

**Windows Operating System Support.** Sentry requires the source code of an operating system in order to partition a structure into secure and insecure parts. This kind of protection is difficult to design for a closed source operating system such as Windows. However, the solution presented in this paper to protect kernel dynamic data can be adopted by Windows developers by creating a partitioned kernel during an OS development cycle to support such protection mechanisms.

**Automatic Partitioning.** Our two proposed partitioning strategies require modifications to an existing kernel. We currently perform this task manually. Though manual partitioning is difficult to perform, it demonstrates the feasibility of our ideas. However, in the future, automated techniques and tools such as source-to-source transformations performed by CIL [23] would be helpful.

**Automatic Data Access Profile Extraction.** Our framework monitors data accesses and checks whether the accessed data is modified legitimately or not. This feature can be used to extract data access profiles for any kernel data structures and their members. The extracted information can be used to create fine-grained policies specific to monitored data.

8 Conclusions

We developed Sentry to provide partitioned kernel memory in a manner similar to memory isolation provided by the kernel for its applications. We protect security-critical data by protecting memory pages containing that data. To provide balance between security and performance, we altered kernel memory layouts to aggregate data needing the same policy enforcement on the same memory pages. Sentry’s security evaluation shows that the system is capable of detecting attacks against dynamic kernel data, and its performance evaluation shows low overheads on microbenchmarks and real-world applications.
References


