

Efficient Monitoring of Untrusted Kernel-Mode Execution

Abhinav Srivastava and Jonathon Giffin

School of Computer Science, Georgia Institute of Technology

{abhinav, giffin}@cc.gatech.edu

Abstract

Recent malware instances execute completely in the kernel as drivers; they do not contain any user-level malicious processes. This design evades the system call monitoring used by many software security solutions, including malware analyzers and host-based intrusion detectors that track only user-level processes. To trace the behavior of kernel malware instances, we design and implement a hypervisor-based system called Gateway that monitors kernel APIs invoked by drivers. Gateway creates a hardened, non-bypassable monitoring interface by isolating drivers in an address space separate from the kernel. To overcome the performance degradation introduced by switches between these separate address spaces, our design rewrites binary kernel and driver code at runtime and generates new code on demand to optimize the address space transition speed. Our experimental measurements show performance overheads of 10% or better, with many overheads less than 1%. Our security evaluation shows that Gateway is able to monitor all kernel APIs invoked by malicious drivers across its non-bypassable interface.

1. Introduction

Recently developed malware instances push entire functionality out of malicious user-space processes down into kernel drivers or modules. For example, `srizbi` consists of a single kernel-mode driver implementing a full spam client with an HTTP based command and control infrastructure [21,45], and the `lvt.es` keylogger contains only a kernel-mode driver that intercepts users' keystrokes [7]. These malware instances are called *full-kernel malware* [23]. Existing host-based IDSs, honeypots, and other security utilities often monitor process' behavior at the system-call interface to detect malware and software attacks. To render existing

system call monitors ineffective, full kernel malware instances invoke kernel functionality with function calls from a driver rather than with system calls from a process. In order to observe the behavior of kernel malware, security software instead must monitor the function-call interface between drivers and the core kernel.

Previous research efforts focused on the interaction of drivers with the kernel through control and non-control data. SBCFI [31], Autoscopy [32], and Hook-Safe [49] protected control data (function and code pointers) allocated on heap, while RAD [10] and Stack-Ghost [15] protected control data (return addresses) present on the stack. Sentry [40], Gibraltar [3], and Semantic Integrity [30] protected a kernel's non-control data [8] from malicious modifications. Though these systems monitor and prevent illegal use of kernel data, they do not track use of kernel code by malicious drivers, which is equally important. A malicious kernel module can create illegitimate kernel control flows by directly invoking arbitrary kernel functionality via calls, jumps, or returns to arbitrary addresses inside the kernel code. To complement the extensive previous research on the protection of kernel data, we investigate the use of kernel code by malicious drivers.

We address the invisibility of full kernel malware with a new system that monitors all control-flow interactions of drivers with the core kernel. We design and implement a security monitor called *Gateway* that monitors all kernel APIs invoked by drivers, passes their invocations asynchronously to an arbitrary policy enforcement utility, and ensures complete mediation of direct accesses from drivers to kernel code. Note that Gateway's non-bypassable interface applies to code entry points, and attacks targeting kernel data require other techniques to monitor or detect. Gateway protects itself from kernel-mode malware by utilizing a higher privilege software layer created by a hypervisor. Gateway's monitoring provides the foundation on which security software can be built in the same manner that system call

monitoring provided the foundation for behavior-based application-level security software.

Security monitors, including Gateway, require the monitored interface to be non-circumventable. Commodity operating systems publish interfaces to be used by drivers and loadable modules to request services from the kernel. Operationally, this interface design is similar to the system call interface, which is used by userspace applications to request services from the core kernel. However, unlike the system-call interface boundary, which is explicitly trusted (all inputs are verified) and enforced by the hardware, there is no mechanism to enforce the implicitly trusted boundary between the core kernel and its drivers.

The lack of a memory barrier inside operating systems allows kernel-mode malware to directly invoke arbitrary kernel functionality by simply calling or jumping to arbitrary kernel code addresses. We thus harden the existing interface so that it becomes fully non-bypassable. Gateway creates distinct virtual memory regions for commodity monolithic kernels and their drivers in the same way that kernels manage distinct regions for higher-level application software. This design isolates drivers in a different memory region and requires them to invoke kernel code through published entry points. To keep driver code and core kernel code isolated throughout the execution of the system, Gateway prevents DMA-capable devices from corrupting the kernel's code via malicious DMA writes [5].

The creation of the non-bypassable interface via address space separation diminishes overall system performance. Control-flows between drivers and the core kernel result in page faults managed by our hypervisor-level software. The transition from execution in a guest virtual machine (VM) into the hypervisor is a *world switch* (*slow path*) with significant performance overhead. We address this challenge by reducing the number of world switches when crossing the barrier between the kernel and the drivers. Our solution establishes fast address space switching (*fast path*) by using on-demand runtime code generation and kernel and driver code rewriting. During execution, Gateway creates *transition pages*—read-only code pages shared by both the kernel and driver address spaces—that contain short instruction sequences that switch address spaces without invoking the hypervisor. We overwrite control-flow instructions at runtime in the kernel's and drivers' code pages to redirect control-flows spanning the memory barrier into the transition pages. As we only generate transition code for control flows correctly spanning the non-bypassable

interface, our design does not compromise security but provides significant improvements to performance.

Gateway monitors kernel APIs invoked by drivers on both the slow and fast paths. On the slow path, API invocations from drivers cause page faults that are intercepted by Gateway inside the hypervisor and logged for higher-level security tools. Since the guest system's execution does not reach the hypervisor on the fast path, the code on our transition pages logs API information in protected memory inside the guest VM. Gateway then periodically reads the logged information from the hypervisor.

To demonstrate the feasibility of our ideas, we developed a prototype security monitor. Gateway provides protection to a fully virtualized Linux 2.6 kernel running inside a guest VM created by the Xen hypervisor, which utilizes the virtualization extensions present in recent x86 hardware [4]. Gateway isolates all commodity Linux drivers present in the guest VM, performs on-demand dynamic binary rewriting and runtime code generation, and monitors kernel APIs invoked by drivers as control flows spanning the memory barrier occur. We tested Gateway's monitoring ability with two full-kernel malware instances: the `lvttes` keylogger and a synthetic kernel-mode bot. Our results show that Gateway is able to monitor and log all APIs invoked by malicious drivers. We empirically show the significant difference between the kernel APIs invoked by legitimate drivers and malware, information useful to full-kernel malware detection or analysis. We also verified Gateway's ability to enforce the non-bypassable interface by testing it with a synthetic malware instance that tries to execute kernel functionality not provided by the published interface. Gateway's overhead on various workloads varies between 0% and 10%, with many measurements below 1%.

Gateway's API monitoring facilitates creation of other security tools that restrict the use of kernel code by drivers. For example: an intrusion detection system (IDS) can be built that enforces a policy restricting drivers' uses of the kernel's raw memory manipulation functions. In another example, kernel-level anomaly detectors similar to system-call based detectors can be developed to identify malicious drivers by differentiating the patterns of kernel functions invoked by legitimate drivers from those of the malware. `Ptrace` [22] allows user-mode security tools to monitor system calls invoked by processes for such anomaly detection. Like `ptrace`, Gateway itself does not attempt to distinguish between benign and malicious API invocations, but it rather provides the mechanism and information enabling higher-

level security software (anomaly and misuse detectors) to make such decisions.

In summary, our work contributes the following:

- We create a non-bypassable interface inside the kernel. Our hypervisor-level software imposes the non-bypassable kernel interface upon dynamically-loaded device drivers thereby preventing control flows from drivers into arbitrary kernel code (Section 3).
- We efficiently handle control flows spanning the kernel interface barrier via on-demand dynamic binary rewriting and runtime code generation. These actions reduce world switches into and out of the hypervisor without compromising the security of the system (Section 4).
- We design a kernel API monitoring tool that records all kernel APIs invoked by drivers via both the slow and fast paths. The monitor asynchronously passes the logged APIs to arbitrary security software monitoring driver execution (Section 5).
- We evaluate our system by demonstrating its ability to monitor the kernel APIs invoked by drivers (Section 6), the efficiency of our interface enforcement design, particularly with runtime rewriting and code generation, and the omission of false alarms on benign driver execution (Section 7).

2. Related Work

Gateway hardens the kernel API to drivers by isolating driver code in a different memory address space than the kernel. Previous researchers have studied the *extension isolation problem* from both the fault isolation and security perspectives, and they proposed solutions different than ours. Nooks [44] confined drivers to a separate address space using hardware page protection. The goal of Nooks was fault isolation; a malicious driver could easily bypass its protection. Since Nooks resided in the operating system, it was still susceptible to direct attacks by malicious kernel drivers. Further, it required assistance from both drivers and the kernel, and its overhead was high. In contrast, we designed Gateway to offer its protection from all drivers, including malware, and it protects itself by using the hypervisor. Though Gateway also isolates drivers using hardware protection, it has low overhead due to its fast address space switching. Vx32 [14] and NaCl [51] isolated applications in sandboxed environments to execute them

safely. They used segmentation and programming language techniques to prevent applications from breaking out of the sandbox. Gateway is different from Vx32 and NaCl as it isolates kernel extensions, protecting the core kernel. Further, it uses paging and binary rewriting to reduce the performance overhead.

Other solutions isolate drivers in user space. Ganapathy et al. [16] proposed the Microdrivers architecture in which drivers were broken into two components, one residing in kernel-space and the other in user-space. Though Microdrivers improved the reliability of the system, the drivers were not completely isolated from the kernel. Nexus [50] isolated drivers entirely in user-space using hardware protection mechanisms. However, it required additional device-specific safety specifications. Though these approaches have merits, they require extensive code changes and rewriting of all drivers.

Researchers have also explored protection domains implemented entirely in software. Software fault isolation (SFI) [48] used program rewriting techniques to modify the object code of untrusted modules to prevent them from writing or jumping to an address outside their access domain. Based on SFI, Seltzer et al. [35] presented a new operating system, VINO, that protected the core kernel from misbehaving kernel extensions. Unlike VINO, Gateway is designed to protect commodity operating systems.

The program rewriting technique was further used by XFI [13] and BGI [6]. XFI guarded all instructions to prevent control flow and data access violations. The control flow prevention included entry point protection similar to Gateway's protection. XFI's rewriter assumed either the availability of debugging information (PDB files) associated with drivers or specially-compiled drivers. Neither assumption is valid for malware: malware instances deliberately strip debugging information to make their analysis hard and use packing software to further hide the difference between code and data. It is not feasible to force malware authors to use special compilers. In contrast, Gateway does not assume any cooperation from drivers, hence it is suitable for protection from malware. BGI used byte-granularity memory protection to isolate kernel extensions. Their system did not modify drivers' source code directly, but modified the compiler to generate modified driver object code. Since BGI is designed for fault isolation, and it requires specially-compiled drivers, it is again not suitable for malicious drivers. Since Gateway does not require specially-compiled drivers, its protection confines all drivers, including full-kernel malware.

Gateway monitors control flows from drivers to the core kernel through the function-call interface provided to drivers. Previous work such as CFI [1] and Program Shepherding [25] monitored the control flow integrity of applications. SBCFI [31] detected persistent kernel control flow attacks by checking the addresses to which kernel function pointers point. Subsequently, Wang et al. [49] protected all kernel function pointers by redirecting them to a common memory region and making that region read-only. Autoscopy [32] discovered and protected kernel function pointers by scanning kernel memory. These systems complement Gateway’s ability to create a non-bypassable interface for drivers by protecting kernel code pointers. Gateway’s control flow monitoring implementation differs from previous tools, such as Dtrace [43] and Kprobes [28], that were designed for tracing and debugging rather than for adversarial environments.

Gateway uses hardware virtualization extensions and memory page protection bits in its creation of memory barriers. The use of virtualization has appeared in multiple research projects [12, 17, 20, 33, 41, 42]. Like those systems, Gateway also uses the isolation provided by virtualization to remain protected from malicious kernel-level code. Gateway manipulates page permissions bits so that it may interpose on control flow transfers between drivers and the kernel. Payne et al. [29] used page protection bits to protect trampoline code inserted into kernel memory. SecVisor [36] protected the core kernel code pages from modification by kernel-level malware. Litty et al. [27] identified covertly executing rootkit binaries present on an infected system by using page protection bits to intercept any code execution attempt involving protected pages. Sharif et al. [37] presented an in-VM design of a security monitor by isolating the security monitor driver in a separate memory region using hardware page protection. Chen et al. [9] proposed a system based on multi-shadowing that protected the privacy and integrity of an application, even if the underlying operating system was compromised. Similar to these systems, Gateway also uses page protection bits, here, to create a non-bypassable interface inside the kernel for drivers.

3. Non-Bypassable Kernel Interface to Drivers

We designed and developed Gateway, shown in Figure 1, to fulfill the following goals:

- **Kernel API Monitoring:** As full-kernel malware instances contain all malicious functionalities in

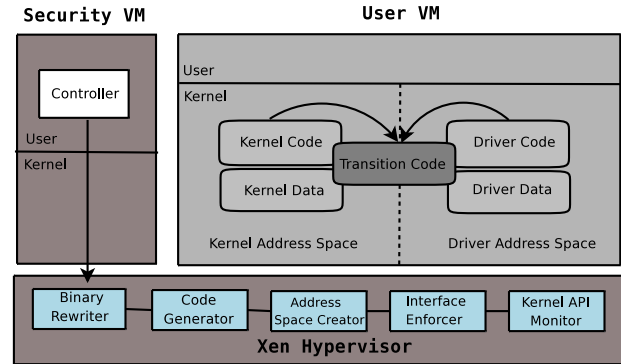


Figure 1. High-level architecture of Gateway.

drivers, we need a security monitor that can monitor kernel interfaces provided to drivers.

- **Kernel Interface Enforcement:** Gateway requires a non-circumventable kernel interface to monitor drivers’ interactions with the kernel. Gateway creates a non-bypassable interface by isolating drivers in a separate address space and allowing kernel invocations only via predefined entry points.
- **Efficiency:** The techniques used to enforce a non-bypassable interface to the kernel must not introduce serious performance impediments to normal system usage. Gateway performs on-demand dynamic binary rewriting and code generation, a design that keeps overhead low.
- **Tamper Resistance:** Gateway constrains kernel-mode malware, hence any security tools deployed at the same privilege level in the kernel may be easily compromised. Gateway avoids this problem with a hypervisor-based design.

3.1. Threat Model

We designed Gateway for an environment that expects installed attacks to include kernel-mode components. These malicious drivers may include functionality traditionally implemented in user-space processes so that the malware eludes monitoring or detection by application-level security software, including system-call monitors. We expect that the drivers will be installed successfully, whether through technical means or through user naïvety. We also expect users to install benign drivers and to update existing drivers with benign

patches, though neither we nor they can differentiate malicious drivers from benign drivers when installed.

To provide security in an environment with kernel-level malicious software, we implant Gateway in an isolated hypervisor layer beneath the vulnerable system. Our trusted computing base includes the hypervisor and a trusted management console called the security VM. We assume that the trusted hypervisor is already installed on the system, hence we do not consider attacks such as Bluepill [34] and Subvirt [24] due to their inability to handle nested virtualization. Since our goals are to constrain drivers' invocation of kernel functionality to the kernel's standard API and to monitor those invocations, we do not consider data attacks and rely on previous research to protect against these attacks [8, 40, 49]. Correct functioning of Gateway does not require that the guest kernel in the low privilege user VM be trusted or free from exploit, however, an exploited or overtly malicious kernel will clearly negate the usefulness of the driver-to-kernel API monitoring provided by our work.

3.2. Driver Isolation

Commodity operating systems such as Windows and Linux rely on paging to provide address space isolation between user applications and the kernel. Page tables, a data structure defined by the hardware, map virtual addresses to physical addresses. Each process has its own page tables (virtual address space), each of which contains mappings for all kernel virtual memory addresses at a fixed location. The kernel page mappings include all kernel-mode components. In this work, we are concerned with the kernel-level security, so we describe only the kernel address space in the rest of the paper. Our approach creates separate page tables for the kernel and for drivers, much in the same way that an OS kernel creates distinct page tables for each running process. Separate page tables force all control flows spanning the kernel-driver interface to induce page faults handled by code in the hypervisor that verifies the legitimacy of the control flow.

In virtualized environments, the hypervisor controls the machine's memory by creating its own page tables to be used by the memory management hardware. These hypervisor-level tables are called shadow page tables in virtualization literature and active page tables (APTs) by hardware vendors [19]. We refer to the portion of the shadow page tables that translates kernel virtual addresses to kernel physical addresses as the kernel page table (KPT). The page tables present in the guest VM—normally used in the absence of virtualization—are re-

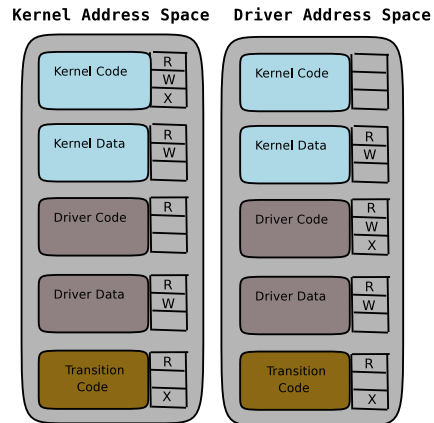


Figure 2. Layout of kernel and driver address spaces with permissions set on memory pages.

named as virtual page tables and provide the illusion to the guest kernel that it controls its own memory.

Gateway creates a separate *driver* address space inside the Xen hypervisor analogous to the existing kernel address space (Figure 2). It maps all driver code pages from all drivers loaded by the guest system into the driver page table (DPT) in a manner transparent to the guest. To maintain consistency between the DPT and the KPT, we map all memory pages of the kernel address range into both page tables, though we set permissions differently. In the KPT, driver code pages are marked non-executable and non-writable. In the DPT, kernel code pages are marked non-executable, non-readable, and non-writable. We also mark all data pages non-executable in both the KPT and DPT to prevent drivers from using data pages for code execution. This configuration of execution permissions across the page tables sets up our subsequent monitoring of a driver's invocation of kernel functionality. Both tables contain executable and non-writable transition code pages, described subsequently in Section 4. Both the KPT and DPT are stored in the hypervisor's memory space, and hence remain isolated from attack by malicious software executing in a guest VM.

To map driver code pages into the DPT, Gateway must know the virtual addresses of driver code pages in the guest kernel's memory space. Since the operating system loads drivers and modules dynamically, they are relocatable and do not reside at a fixed location at every load. To acquire the addresses at which drivers are loaded, we interpose on the driver loading process. At

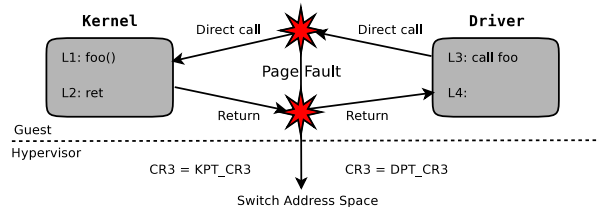


Figure 3. Address space switching between drivers and the kernel on the invocation of a direct call from the driver (slow path).

runtime, we automatically rewrite the target of a direct call instruction along the driver loading path inside the kernel to point to a memory location that is not mapped to the guest VM; we store the correct target location inside the hypervisor. This design creates a page fault during driver loading, allowing Gateway to gain control. Since this fault happens after the OS decides where to load a driver, Gateway extracts this address information from the guest memory in a secure way and then resumes the guest’s execution.

We include the (non-executable) driver code pages in the kernel’s KPT for simple efficiency purposes. Some drivers contain read-only data, such as the import table for the kernel, and executable code on the same memory page. By including the drivers’ code pages in the KPT, the kernel can still read the drivers’ read-only data without introducing extra page faults. In contrast, kernel code pages must be unreadable in the DPT to prevent introduction of a code-pullout attack [2, 21] and to inhibit return-oriented rootkits. A return-oriented rootkit [18] requires gadgets, many taken from the core kernel, to perform arbitrary computation. Gateway limits return-oriented programming by forcing drivers to enter the kernel at legitimate API entry points; attackers are unable to construct kernel gadgets different than the kernel functions themselves. Though attackers could construct gadgets from the code of other drivers in the DPT, we discuss in Section 8 an extension to Gateway that hardens and monitors the interface between drivers to remove even that opportunity.

3.3. Address Space Switching

Gateway changes address spaces by switching between the kernel and driver page tables. The root of the page tables is called the *page directory*, and in x86 architectures, the hardware *CR3* register stores the phys-

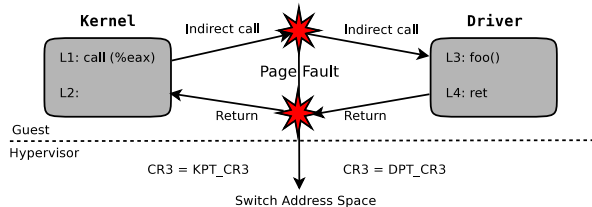


Figure 4. Address space switching between the kernel and drivers on the invocation of an indirect call from the kernel (slow path).

ical address of the current active page directory. In a virtualized environment, read and write instructions involving the *CR3* register are privileged operations. A guest operating system is not allowed to write into the *CR3* register, so any write operation by guests to the *CR3* register causes a world switch that passes control to the hypervisor. This feature thwarts attacks in which malware running in the kernel attempts to modify the *CR3* register to point to new page tables. The hardware *CR3* register points to the root of the shadow page tables managed by the hypervisor, and it is used by the memory management hardware.

Gateway switches between two address spaces by changing the value stored in the *CR3* register. During driver code execution, the *CR3* register stores the DPT’s root address, *DPT_CR3*. When the driver code calls or jumps into any kernel code, the execution faults into the hypervisor due to the page protection bits set on the kernel code pages in the DPT. Inside the hypervisor, Gateway intercepts the fault, and if the fault is for the kernel code pages, it changes the *CR3* value stored in the register by using the KPT’s root address, *KPT_CR3*. After changing the *CR3* value in the hypervisor, Gateway returns to the guest OS to re-execute the faulted instruction. On the return path from the core kernel to drivers, execution faults again due to the page permission bits set on the driver code pages in the KPT. In this case, Gateway replaces the *CR3* register’s value with the value of *DPT_CR3*. Figure 3 describes the address space switching process on a direct call to and *ret* from the kernel. Gateway performs similar switching on an indirect call to and *ret* from drivers code (Figure 4). Indirect calls from the driver are rare and direct calls from the kernel are non-existent, but they would be handled analogously should they ever be encountered.

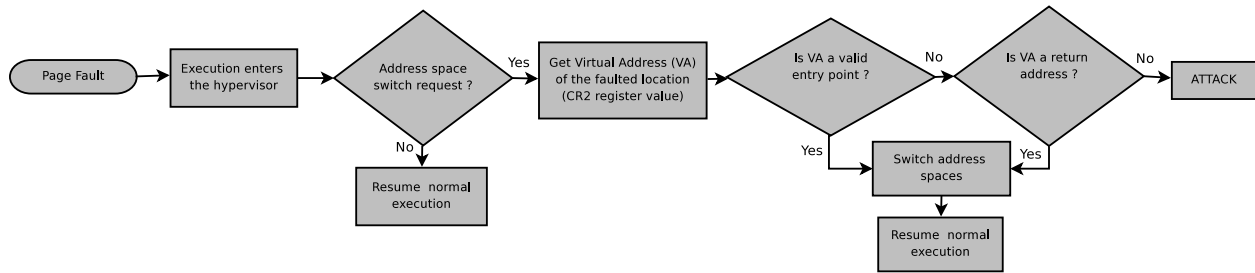


Figure 5. Steps involved in the verification of interface invocation from drivers to the core kernel.

3.4. Non-Bypassable Interface Enforcement

The creation of a separate KPT and DPT with distinct page access permissions allows Gateway to intercept control flows from any driver to the kernel, and hence to limit allowed control flows to only those targeting valid API entry points in the kernel. Commodity operating systems publish interfaces meant to be used by third party developers creating drivers and loadable modules. Our expectation is that any interaction with the core kernel through these interfaces is legitimate and should be allowed, but attempts by a driver to jump or call other kernel addresses represents illicit behavior attempting to bypass the kernel interface. Gateway enforces the use of these interfaces upon drivers.

Gateway verifies calls and jumps from driver code to kernel code, returns from the driver to the kernel following the kernel’s invocation of driver functionality, and interrupts (Figure 5). When driver code executing from the DPT invokes a kernel function, the system’s execution will fault into the hypervisor because the kernel code in the DPT does not have execute permission. In the hypervisor, Gateway extracts the guest VM’s execution context information, such as the virtual address of the faulted instruction. It verifies whether the faulted address corresponds to a predefined valid entry point into the kernel code—these include entry points of exported functions, interrupt handlers, and exception handlers. If the entry point is legitimate, Gateway alters the CR3 to specify the KPT as the current page table. If the faulted address is not a valid entry point into the kernel, then either the driver is attempting to invoke a kernel function which is not meant to be used by drivers, or it is trying to jump into the middle of a block of code. Gateway prevents such illicit control flow.

Gateway similarly verifies control flows that return back to the core kernel code upon execution of the `ret` instruction inside a driver. This driver-to-kernel transi-

tion is valid provided that the return address on the kernel’s call stack has not been altered. At the original call from the kernel to the driver, Gateway records the return address at the top of the stack prior to switching the page tables from KPT to DPT. When the subsequent return instruction faults, Gateway then verifies whether the fault location matches the previously stored return address. This design defeats attacks in which attackers modify the return address to return to an arbitrary location in the kernel code.

A malicious driver may attempt to use DMA to write memory mapped into the KPT and then to execute that code in the kernel’s context. Gateway prevents malicious DMA writes by verifying the targeted memory regions in the requested DMA operations. Xen emulates all guest DMA operations using a software IOMMU. Gateway intercepts all DMA requests and rejects any request that contains an address not writable in the DPTs such as the kernel code and transition code pages. Xen also virtualizes recent hardware IOMMUs, such as Intel’s VT-d and AMD’s DEV. Protection from these DMA requests requires address verification at the virtualized hardware [11, 36].

Gateway must be aware of legitimate entry points into the kernel. We extract the virtual address of all kernel functions available to drivers for legitimate use from the symbol file (*System.map* or *kallsyms* in Linux) maintained by the guest kernel and keep this information with Gateway in the hypervisor.

3.5. Controller

The security VM is a management console and runs an application that controls Gateway’s operation. The controller can enable or disable Gateway’s protection inside the hypervisor. It interacts with the hypervisor using the hypercall interface provided by the hypervisor. Whenever the hypervisor component of Gateway iden-

tifies an attempted control flow that bypasses the kernel interface, it informs the controller so that necessary actions can be taken, as specified by policy or an administrator.

3.6. Driver Page Table Implementation

We developed the driver address space as new shadow page tables created in the Xen hypervisor. Equally suitable alternative implementations could use other hypervisors, such as KVM or VMware, and hardware-supported nested/extended page tables. Although the guest Linux system used for our prototype development is a 32-bit system with 2-level page tables, we used Xen in its physical address extension (PAE) mode. In PAE mode, the x86 memory management unit expects 3-level page tables and offers the non-execute (NX) memory page permission absent from 2-level page tables. In PAE mode, Xen automatically maps 2-level guest page tables to its 3-level shadow page tables. Gateway then creates the DPT by allocating memory for a 3-level page table separate from Xen’s original table. After allocating memory, it sets up the DPT as a copy of the shadow page table with kernel code marked non-executable, non-writable, and non-readable. Finally, it edits the KPT so that driver code pages are marked non-executable and non-writable.

Gateway ensures that address spaces remain isolated throughout the guest system’s execution. It thwarts attacks that attempt virtual memory remapping or creation of new memory mappings that reintroduce executable kernel code into the DPT and vice versa. Gateway ignores such requests and injects a page fault into the guest, indicating that the region is not for mapping. Gateway utilizes the hypervisor’s ability to interpose on the guest VM’s virtual page table updates. It prevents the guest OS from mapping or changing protections on protected memory pages by hooking inside Xen’s page table propagation `sh_propagate` and page fault handler `sh_page_fault` code. On each page fault, it verifies that the page protection bits have not been altered.

4. Fast Address Space Switching

The isolation of driver code pages in an address space separate from kernel code pages comes at a price. Each address space transition causes page faults, which in turn cause hypervisor world switches. Since the interaction between the kernel and drivers happens at a high rate, we expect the performance cost to be high. To this end, we propose a novel approach that reduces the transition

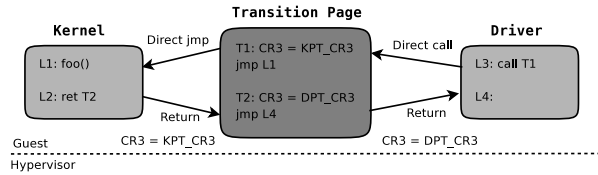


Figure 6. Address space switching between the kernel and drivers on the invocation of a direct call from a driver (fast path).

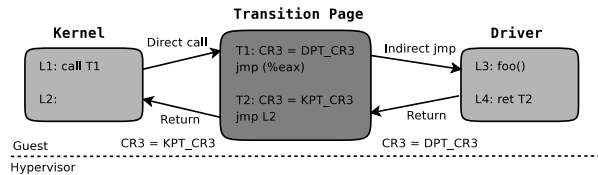


Figure 7. Address space switching between the kernel and drivers on the invocation of an indirect call from the kernel (fast path).

overhead by establishing a fast path for address space switching. In this section, we describe the design and implementation of the fast path.

Our performance improvement comes by dramatically reducing the number of world switches that occur during guest system execution. In the design as presented in the previous section, every call and return spanning the barrier between the kernel and drivers induces a world switch to the hypervisor at the page fault. In our fast path design, only the first call at a particular call site faults to the hypervisor. All subsequent calls from the same location and all corresponding returns execute at full speed. This design is similar to lazy linking of library functions in dynamically-linked applications: the first invocation executes functionality that fixes up the code so that all subsequent calls execute with no delay. Our fixups include runtime code generation and selective rewriting of guest kernel and driver code. A further optimization to prevent execution faults on even the first call instruction would require altered compilation or pre-execution offline code rewriting at all control-flow transfers, and we have not pursued such changes.

We leverage a hardware feature present in Intel and AMD processors called *CR3-Target Controls* [19]. This feature allows a guest kernel to change the CR3 value without causing a world switch to the hypervisor, provided that the value written into the CR3 register was

previously specified by the hypervisor in the CR3-Target registers. Gateway adds the KPT_CR3 and DPT_CR3 values into the registers.

It is then the responsibility of guest kernel code to switch the CR3 value when transitioning the memory barrier between the kernel and the drivers. The instructions to execute the switch are not present in the stock Linux kernel. Gateway thus generates short sequences of instructions that correctly change the CR3 value, writes those sequences into guest OS memory pages that we term *transition pages*, and overwrites call instructions in the kernel code and driver code to redirect the control flows spanning the memory barrier through the transition pages. The transition pages are guest memory pages, and they are mapped into both the KPT and DPT as read-only and executable pages. We call the short sequence of instructions on transition pages *transition code*.

In the hypervisor, Gateway generates transition code and rewrites call instructions on-demand at runtime every time a call instruction executes for the first time and faults to the hypervisor. Subsequent to the code alteration, execution of the same call instruction will pass through the transition code and avoid a world switch. We only redirect direct call instructions from drivers to the core kernel, indirect call instructions from the kernel to drivers, and their corresponding returns; indirect call instructions from drivers to the kernel and direct call instructions from the kernel to drivers still use the slow path for switching between address spaces (see Section 4.2 for explanation). By adding the KPT_CR3 and DPT_CR3 values to the CR3-Target registers, performing runtime code generation on transition pages, and rewriting control transfer instructions, any CR3 switch between the KPT and the DPT happens at native speed without invoking the hypervisor. Figures 6 and 7 show the effect of the fast path on direct and indirect call instructions occurring through the interface.

Our fast path design is secure because Gateway only overwrites those `call` instructions that bring legitimate control flows into the kernel at a valid API entry point. All transitions that have not been overwritten still fault, and Gateway verifies those transitions. This verification is sufficient to guarantee the non-bypassable interface enforcement. Since transition pages are read-only, attackers cannot modify the generated code on transition pages to enter into arbitrary locations inside the kernel. The transition code is the only code that is executable in both the KPT and DPT, and it is the only way of switching the address spaces without invoking the hypervisor. Malicious drivers executing from the DPT cannot exe-

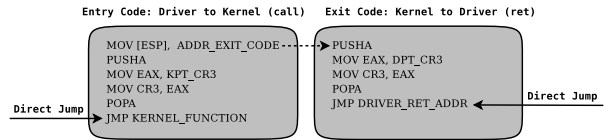


Figure 8. Runtime transition code generated by Gateway to enter in and exit from the kernel code on direct call and ret instructions, respectively.

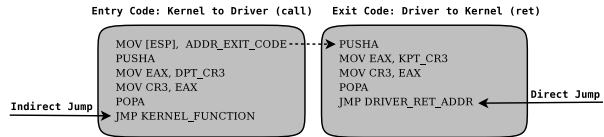


Figure 9. Runtime transition code generated by Gateway to enter in and exit from the driver code on indirect call and ret instructions, respectively.

ecute kernel code in the KPT by changing the CR3 to KPT_CR3 themselves without using the transition code. Though the CR3 switch will not fault as KPT_CR3 is in the CR3-Target registers, the execution will fault on the driver's next instruction as that instruction is not executable from the KPT.

4.1. Runtime Transition Code Generation

Gateway generates transition code on transition pages at runtime to switch the CR3 register value. The transition code can be divided into two parts: the entry code and the exit code. The entry code corresponds to a `call` instruction while the exit code corresponds to the paired `ret` instruction. Gateway generates the entry and exit code customized for each `call` and corresponding `ret`. We describe entry and exit code for both the direct and indirect call instructions that Gateway overwrites.

The entry code for a direct call instruction from a driver to the kernel has three sequential components: (a) code that overwrites the return address on the stack with the address of the start of the paired exit code, (b) CR3 switch code, and (c) a jump to the original target address in the kernel. In a single instruction, the transition code overwrites the return address to redirect the subsequent return from the kernel back to the driver through the exit code on the transition page. Note that Gateway records the original return address before overwriting its value; the original value will be used when generating the exit

code. Gateway then generates the code that switches the address spaces without invoking the hypervisor, using a sequence of four instructions. When executed, this will not cause a page fault because the transition code is present in both the DPT and KPT. Finally, it adds a direct `jmp` instruction to the original kernel function. Due to the address space switch that happens before the jump, this jump will not cause a page fault.

The paired exit code is similar to the entry code in that the generated code (a) switches the CR3 value back to `DPT_CR3` and then (b) jumps to the original return address in the driver. Since both the entry code and exit code are customized for each call and return, the direct `jmp` instructions use hardcoded values taken from the return address on the stack prior to its overwrite. Gateway writes these hardcoded values on the transition page at the time of the code generation. Figure 8 shows the transition code that Gateway generates for a direct call instruction from a driver to the kernel and for its return.

In a similar way, Gateway generates entry and exit code for indirect calls from the kernel to drivers. When producing entry code, Gateway first saves the original return address and generates code to replace it with the start of the exit code. Then, it generates code to switch the CR3 to `DPT_CR3`. In the next instruction, however, the jump target cannot be hardcoded because the indirect target may change later in execution. The address of the targeted driver function is either in a register or in a memory location specified as the operand of the call instruction. In order to ensure that the transition code targets the correct address, Gateway copies the operand of the indirect call instruction from the kernel code over to the indirect jump instruction that it is generating on the transition page. For example, if an indirect call instruction is `ff d1`, Gateway generates the code `ff e1` on the transition page to jump to the driver function; that binary code is the equivalent indirect jump with the same operand. Gateway also generates the exit code to return control from the driver back to the kernel. The exit code for an indirect call is identical to the exit code for a direct call instruction with a hardcoded jump location. Figure 9 shows the transition code that Gateway generates for an indirect call instruction from the kernel to a driver and for its return.

4.2. Dynamic In-Memory Code Rewriting

Gateway redirects calls through the transition pages by dynamically rewriting the call instructions on the code pages of the kernel and drivers when those calls cause transitions between the DPT and KPT. In combination with the runtime code generation, this redirection

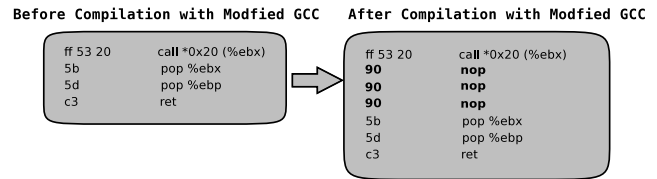


Figure 10. Effect of compilation of the kernel with the modified GCC that adds `nop` instructions after each indirect calls.

allows address space switching to occur at native speed. The on-demand rewriting allows a `call` instruction to fault once. During the processing of the fault, Gateway first validates the control flow transfer. If it finds the transition valid, it generates transition code and rewrites the faulted call instruction to point to the entry code. Gateway also removes the fault caused due to the `ret` instruction. This design does not let the execution fault on the verified re-written instructions for every future invocation: this call and its return are now on the fast path.

We first describe the on-demand dynamic binary rewriting of direct `call` instructions that transfer control from drivers to the kernel. A direct `call` instruction contains one byte of opcode and four bytes of operand specifying the location of the invoked kernel function. On a fault, Gateway rewrites this call instruction by replacing its operand with the address of the entry code generated on the transition page; the original target operand is inserted on the transition page as the jump target of the entry code. With this rewriting, the existing direct call instruction to the kernel function becomes the direct call instruction to a transition code present on the transition page.

Gateway also rewrites indirect call instructions in the core kernel targeting drivers. (Note that the kernel never targets a loadable driver with a direct call as such kernel code would fail to statically link.) Overwriting indirect call instructions with direct calls is complicated because most x86 indirect calls are 2 bytes, 3 bytes, or 6 bytes in length. Gateway needs 5 bytes to rewrite an indirect call instruction with a direct call instruction targeting transition code.

To perform the rewriting of short indirect call instructions, we insert `NOP` instructions in the kernel binary after each indirect call instruction. An indirect call instruction followed by `NOP` padding provides sufficient width to replace the indirect instruction with the direct call. To insert these padding instructions, we use a compiler-

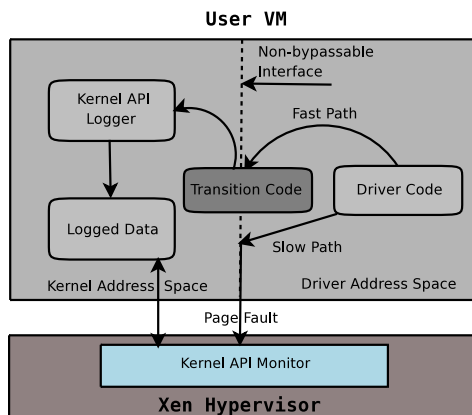


Figure 11. Low-level architecture of the Kernel API monitor that records kernel APIs on both the slow and the fast path.

based approach. We modify `gcc` so that it generates new binaries containing `NOP` instructions after each indirect call instruction. Figure 10 shows a snippet of the kernel’s binary code and its transformation after compiling it with the modified `gcc`. With the new kernel binary, Gateway is able to overwrite indirect call instructions in the kernel code with direct call instructions pointing at entry code on the transition page.

Importantly, note that *our design does not require drivers to be recompiled* with the modified compiler. We specifically chose this design because it does not force third party vendors (or full-kernel malware authors) to compile their drivers with our compiler; it is also one of the reasons why we do not rewrite indirect calls from drivers to the core kernel. A second reason to not rewrite indirect calls from drivers to the kernel is security. Recall that during the code generation of indirect call instructions, we copy the operand of the call instruction to our transition page. An attacker could easily change the value stored in the registers that are part of the operand and could invoke unchecked arbitrary kernel functionality. Our design does not allow such transitions into the core kernel and strictly enforces the non-bypassable interface to drivers.

5. Kernel API Monitoring

Gateway functions as kernel API monitoring software that records all kernel functions invoked by drivers through the non-bypassable interface. Our optimized design creates two different paths by which drivers may invoke kernel functions: a slow path that causes a world

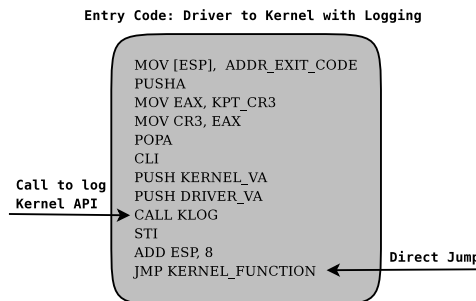


Figure 12. Runtime entry code generated by Gateway to enter into the kernel code from drivers. This code includes the API logging logic also.

switch and a fast path that uses transition pages. To be able to record all kernel functions invoked by drivers, Gateway must monitor both the slow and the fast path.

To monitor kernel API invocations on the slow path, Gateway records the virtual address of the invoked kernel function at each page fault from drivers to the core kernel. It also finds the virtual address of the callee by using the return address present on the stack. Since Gateway protects the return address, it can securely identify the driver that invoked the kernel function. Once Gateway identifies the source and destination information, it passes this data to the controller for use by higher-level security software.

Monitoring API invocations via the fast path requires a different strategy. Since fast path memory barrier transitions do not reach the hypervisor, Gateway will not be able to record these APIs invocations. To be able to monitor the kernel APIs on the fast path, Gateway augments the code generated on transition pages so that it additionally logs the kernel API invocation information in protected guest kernel memory. Gateway allocates guest memory pages called *log memory* from the hypervisor and uses this memory pool to store the API invocation information occurring through the fast path. In a standard producer-consumer model, the logs are written inside the guest by the transition code, and Gateway reads the logged data asynchronously from the hypervisor. To protect the logged data from malicious drivers, Gateway marks the allocated memory pages as non-readable, non-writable, and non-executable in the DPT. These pages have read-write permissions inside the KPT. With this design, Gateway is able to log all kernel API invoked by drivers both on the slow and fast

<i>Malicious Driver</i>	<i>Kernel API Invoked</i>
Lvtes	sys_open, sys_read, sys_write, sys_close sys_getdents, _spin_unlock snprintf, __wake_up, __kmalloc copy_from_user, copy_to_user, _spin_lock, kfree, memmove
Full Kernel Bot	kmem_cache_alloc, sock_create, inet_stream_connect sock_recvmsg, sys_sendmsg

Table 1. Kernel APIs invoked by the Lvtes keylogger and the kernel-level bot.

paths. Figure 11 shows the architecture of the kernel API monitor.

To record the API information in the guest memory, Gateway first generates a logging function called `Klog` on a separate guest kernel memory page mapped as non-readable, non-writable, and non-executable in the DPT. It is read-only and executable inside the KPT. Then, during the generation of the entry code on a transition page, Gateway adds extra instructions that invoke `Klog` from the transition page after switching CR3 to the KPT. When a direct call from drivers to the kernel goes through the transition page, the transition code invokes the `Klog` code. The transition code also passes the virtual addresses of the called and callee functions using the kernel stack. Figure 12 describes the transition code augmented with the logging code. To avoid attacks in which untrusted drivers tamper with the information present on the stack, we first switch the CR3 from the `DPT_CR3` to the `KPT_CR3` and then push the information on the stack. In this design, driver code becomes non-executable and `Klog` extracts the information in a secure way. To avoid security issues due to interrupts during logging, we disable interrupts before parameters are pushed on the stack and enable them after `Klog` completes. `Klog` writes invocation information in the log memory to be consumed by the hypervisor, which subsequently passes the data to the controller and any high-level security software.

6. Security Evaluation

We evaluated Gateway’s enforcement of the non-bypassable interface and monitoring of the kernel APIs invoked by drivers.

6.1. Non-Bypassable Interface Evaluation

We tested Gateway’s ability to enforce the boundary between drivers and the kernel code with a synthetic malware instance that jumps into the middle of the kernel code to execute an operation. When we ran our malicious driver inside the guest VM, Gateway loaded it into the DPT and marked its code pages non-executable and non-writable in the KPT. When the malicious driver tried running its malicious code from its initialization function, the `jmp` instruction caused a fault into the hypervisor as the kernel code was not present in the DPT. On verification, Gateway correctly found that the target address was not a valid kernel entry point and raised an alarm.

6.2. Kernel API Monitoring Evaluation

We evaluated Gateway’s ability to monitor all kernel APIs invoked by malicious drivers. We ran Gateway with two malicious drivers: the `lvtes` keylogger and a synthetic kernel-mode bot. The keylogger installs a kernel driver, receives user keystrokes, and logs them to a file. It performs all operations inside the kernel, and it does not contain any user-space process. When we loaded `lvtes` in the guest VM, Gateway set up the appropriate permissions for the code and data pages of `lvtes` both in the KPT and DPT. During its execution, `lvtes` invoked several kernel APIs to read data, to write data to the log file, to hide the file, to allocate memory, and to perform some other functions. Gateway was able to log all APIs invoked by `lvtes`, shown in Table 1.

In our second test, we ran Gateway with a synthetic kernel-level bot having basic functionalities, such as socket creation, network connection, data transmission, and packet receipt. This bot again completely resided in the kernel as a driver, and it did not have any user-level component. We ran one server on a separate test machine so that bot could communicate with it. After loading the bot and isolating its code pages in the DPT, when it executed its functionality, Gateway successfully detected all API functions invoked by the bot, listed in Table 1.

The above results show that Gateway is effective in enforcing the non-bypassable interface. Given this interface, Gateway then logs all interaction of drivers with the core kernel. The information provided by Gateway can be used by high-level security software that could, for instance, identify malicious software based on their unusual use of the kernel interface. To demonstrate the usefulness of the kernel API monitoring, we empirically evaluated the difference between the kernel APIs

<i>Benign Drivers</i>	<i>Kernel API Invoked</i>
File system driver	kmem_cache_alloc, clear_inode, new_inode, generic_commit_write, block_prepare_write, block_write_full_page, generic_file_aio_write, rb_erase, _spin_lock, _spin_unlock, truncate_inode_pages, submit_bh, rb_first
Network driver	_spin_unlock, _alloc_skb, eth_type_trans, _spin_lock, netpoll_trap, raise_softirq_ireoff, _spin_lock_irq, _spin_unlock_irq, netif_receive_skb

Table 2. Kernel APIs invoked by the benign drivers and logged by Gateway.

<i>Task</i>	<i>Count</i>
Lines of gcc source code modified	5
Drivers isolated	36
Approx. direct instruction overwritten	500
Approx. indirect instruction overwritten	65
Approx. transition pages used	8

Table 3. Statistics related to Gateway’s implementation and impact on a running system.

invoked by malicious and legitimate drivers. Table 2 shows the key kernel APIs invoked by file system and networking drivers on our test system. A comparison of the two tables shows a clear distinction between the set of APIs invoked by malware and legitimate drivers. These anomalies can be used by high-level security software to detect attacks.

7. Performance Evaluation

Compatibility Evaluation: We designed and developed Gateway to offer its protection to the kernel from drivers. We conducted a compatibility test to show that Gateway did not make any assumption on driver code. We tested Gateway with 36 commodity Linux drivers, and Gateway was able to isolate all of them. Further, Gateway was able to perform binary rewriting and runtime code generation for all these drivers and the core kernel. Table 3 presents detailed statistics related to Gateway’s implementation and basic impact on the guest Linux kernel. Our compatibility evaluation shows that Gateway’s design is effective, and it can be used to protect operating systems from drivers, including kernel-malware. Appendix A shows the list of all isolated commodity drivers along with their sizes.

Experimental Evaluation: We evaluated Gateway’s impact on a system’s performance with extensive benchmark-driven evaluation. Our testbed contained an Intel 2.8 GHz Core 2 Quad processor, 4 GB of RAM, and a 100Mbps ethernet card. We used the Xen hypervisor in PAE mode, and our guest user VM used the 32-bit Linux 2.6 kernel. For our experiments, we assigned 1GB of memory to the guest VM, and 3 GB of memory was shared between the security VM and the hypervisor.

We tested Gateway with a collection of benchmarks exercising the CPU, disk I/O, and network I/O: Lmbench [26], BYTEmark [47], Iperf [39], and Bonnie [46]. We performed all experiments five times and, due to occasional large outliers common to virtualized environments, report median values together with the median absolute deviation. We present the results of file-system benchmarks in boxplots due to high variance in disk I/O measurements. In our results, “Normal” refers to measurements that do not have Gateway’s protection and “Gateway” includes our protection.

In our micro-benchmark experiments, we first measured the effect of Gateway on operations that happen very frequently. Using Imbench, we measured the cost of a context-switch, procedure call, and system call. Table 4 shows our results. It can be seen from the table that Gateway’s overhead on the regular operations is low.

In another experiment, we measured Gateway’s overhead on network operations. Since Gateway isolates all drivers—including the networking driver—in another address space, we measured this effect. We connected two machines with a switch. We used Imbench’s network tests to measure network latency, TCP and UDP latencies, and throughput of TCP connections, and Iperf to measure UDP throughput. The results are shown in Table 5. Although Gateway’s overhead on network operations is low, it still affects TCP communication. We investigated the cost of TCP operations, and we found that some of the functions on the TCP code path were not

<i>Operation</i>	<i>Normal VM (ns)</i>	<i>Gateway VM (ns)</i>	<i>Overhead (%)</i>
Context-switch	2,400. (10)	2,560. (20)	6.67
Procedure call	3.6 (0)	3.6 (0)	0.
System call	80.6 (0)	80.7 (0)	0.12

Table 4. Execution time measured by Imbench without and with Gateway for context-switching, procedure calls, and system calls. Times reported in nanoseconds; smaller measurements are better. Values reported are medians, and values in parentheses show median absolute deviation.

<i>Operation</i>	<i>Normal VM</i>		<i>Gateway VM</i>		<i>Overhead (%)</i>
TCP latency (μ s)	431.6036	(5.1018)	470.9497	(13.9465)	9.12
UDP latency (μ s)	432.0758	(4.1355)	454.3676	(19.9881)	5.16
Connection latency (μ s)	908.0500	(15.0689)	966.9074	(32.4059)	6.48
TCP throughput (MB/sec)	8.08	(0.30)	7.98	(0.70)	1.23
UDP throughput (MB/sec)	1.06	(0.)	1.06	(0.)	0.

Table 5. Network latency and throughput measured by Imbench and Iperf without and with Gateway. Smaller measurements are better. Values reported are medians, and values in parentheses show median absolute deviation.

receiving fast path optimization because the driver was invoking kernel functionality via indirect calls. Since Gateway does not rewrite indirect instructions from drivers to the kernel, these control flows remain on the slow path.

To measure Gateway’s effect on CPU-bound execution, we tested it with computationally intensive work loads. We performed these experiments with BYTEmark, a benchmark that runs various CPU-intensive algorithms and measures the performance in iterations per second. We tested Gateway with all tests, and Table 6 shows our results. They indicate that Gateway’s overhead on CPU-bound applications are very low when compared with normal execution.

We next measured the effect of Gateway on file system performance. Since we isolated the file system drivers in the DPT, we measured the effect of this partitioning. We carried out this experiment with bonnie, a benchmark that measures the throughput of read and write operations performed in both character and block sizes. In this experiment, we created a file of size 2 GB, which exceeds the size of the memory allocated to guest VM to reduce caching effects. Our results, shown in Figure 13, indicate that Gateway’s read and write operations’ throughput remains close to the normal VM’s results.

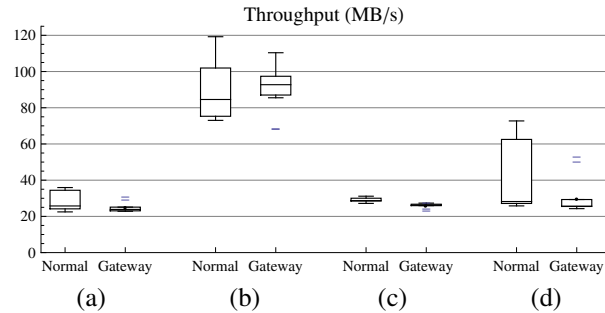


Figure 13. Gateway’s impact on the filesystem measured with Bonnie. All measurements show throughput in MB/s; higher measurements are better. Boxes show medians and first and third quartiles. Outliers appear as dashes. Groupings show performance of (a) character reads, (b) block reads, (c) character writes, and (d) block writes.

Effect of Fast Path Optimization: Previous experiments showed that Gateway’s overhead on the fast path was low. In this set of experiments, we specifically compared the performance of the fast path with the slow path implementation of Gateway to show the effect of fast path design. Our test included a compilation

<i>Operations</i>	<i>Normal VM (iteration/sec)</i>		<i>Gateway VM (iteration/sec)</i>		<i>Overhead (%)</i>
Numeric sort	1095.80	(6.20)	1092.20	(3.50)	0.33
String sort	163.45	(0.32)	162.80	(0.24)	0.40
FP emulation	186.28	(0.20)	185.61	(0.09)	0.36
Fourier	30498.	(21.)	30390	(12.)	0.35
Assignment	37.63	(0.04)	37.37	(0.07)	0.69
Idea	5806.70	(4.)	5786.70	(4.)	0.34
Huffman	2358.10	(2.30)	2347.80	(0.90)	0.44
Neural net	45.52	(0.02)	45.45	(0.03)	0.15

Table 6. Gateway’s overhead on CPU-bound applications as measured with BYTEmark; higher measurements are better. Values reported are medians, and values in parentheses show median absolute deviation.

<i>Operations</i>	<i>Normal VM (sec)</i>		<i>Slow Path (sec)</i>		<i>Overhead (%)</i>	<i>Fast Path (sec)</i>		<i>Overhead (%)</i>
make	64.055	(0.119)	80.297	(1.452)	25.37	67.338	(0.628)	5.12
bzip2	41.847	(0.053)	51.777	(0.474)	23.73	43.287	(0.192)	3.44
tar	29.434	(0.423)	40.511	(0.427)	37.63	30.109	(0.199)	2.29

Table 7. Effects of Gateway’s fast path design; smaller measurements are better. Values reported are medians, and values in parentheses show median absolute deviation.

of the stripped-down version of the Linux kernel, file compression, and tarring of the Linux source directory. Our results, presented in Table 7, show that Gateway’s fast path design has improved the system’s performance substantially when compared to the overhead on the slow path. These results also confirm that our design of fast path is efficient, and the overhead of Gateway is acceptable on the system.

False Positive Evaluation: We tested Gateway’s proclivity to falsely block legitimate driver behavior by loading and using benign device drivers in the presence of our tools. A false positive occurs in Gateway if benign drivers bypass the kernel’s exported interfaces and execute control transfers to internal kernel code. We analyzed 36 benign drivers loaded into our test guest VM and found that none made invalid control transfers.

8. Discussion

Gateway monitors drivers’ interactions with the core kernel, but a malicious driver may attempt to invoke another driver’s functions to perform some of their malicious activities without involving the kernel. Though monitoring driver-to-driver operations are beyond the scope of this work, we could extend Gateway to monitor driver-to-driver operations.

Currently, Gateway isolates all drivers into a single address space separate from the kernel. An extended design could use provenance information associated with drivers to achieve more flexible isolation. For example, it could position drivers signed by trusted parties, such as Microsoft, together with the kernel code in the KPT. Only drivers whose provenance is either not known or not verified would be isolated in the DPT. This isolation strategy would further reduce the overhead of Gateway because operations involving trusted drivers in the KPT execute at full speed without interpositioning costs; only drivers in the DPT require binary rewriting and code generation.

The extended design would enable Gateway to monitor an untrusted driver’s interaction with both the core kernel and the KPT drivers. Given that users install drivers from different third party vendors without knowing their provenance, an inflexible preventive approach that outright blocks the loading of new drivers may not work in practice. Our flexible design, in contrast, could be adopted by commodity operating systems vendors such as Microsoft to restrict the operation of untrusted modules or drivers. The provenance based design further limits return-oriented programming [18], previously discussed in Section 3.2, as code from neither the core kernel nor any trusted driver could be used for gadget construction.

An attacker may attempt to bring the entire malicious functionality inside a single driver and execute without interacting with the core kernel or other drivers. Since the malicious driver would not interact with any other code in the KPT, Gateway monitors none of its behavior. Though this kind of attack is possible, it is difficult to launch in practice as it requires prediction of all possible configurations of hardware and file systems on the victim's system. Such attacks, if attempted, can be mitigated by monitoring the malicious driver's interaction with the hardware by incorporating techniques similar to BitVisor [38].

9. Conclusions

Gateway monitors the interaction of drivers with the core kernel by creating a non-bypassable interface inside the kernel. It isolated all drivers from the kernel code by creating a separate address space for drivers. The address space isolation incurred performance overhead because each address space switch caused world switches to the hypervisor. Gateway solved this problem by establishing a fast path, which used on-demand runtime binary rewriting of guest kernel and code generation. With this design, Gateway allowed most control flow transfers between drivers and the core kernel to occur at native speed. The creation of an efficient, non-bypassable interface allowed Gateway to monitor kernel APIs invoked by driver through the interface. Our evaluation showed that Gateway's interface enforcement was effective, its monitoring was capable of logging kernel APIs, and its overhead on the system was low.

Acknowledgment of Support and Disclaimer

We thank our shepherd, Lujo Bauer, and our anonymous reviewers for their extremely helpful comments. We would also like to thank Neha Sood for her comments on the early drafts of the paper. This material is based upon work supported by National Science Foundation contract number CNS-0845309. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the NSF or the U.S. Government.

References

- [1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control flow integrity principles, implementations, and ap-

- plications. In *ACM CCS*, Alexandria, Virginia, Nov. 2005.
- [2] Alexander Tereshkin. Rootkits: Attacking personal firewalls. www.blackhat.com/presentations/bh-usa-06/BH-US-06-Tereshkin.pdf. Last accessed Aug. 05, 2010.
- [3] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structures invariants. In *ACSAC*, Anaheim, CA, Dec. 2008.
- [4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SOSP*, Bolton Landing, NY, Oct. 2003.
- [5] M. Becher, M. Dornseif, and C. Klein. Firewire all your memory are belong to us. In *CanSecWest*, 2005.
- [6] M. Castro, M. Costa, J. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *ACM SOSP*, Big Sky, Montana, Oct. 2009.
- [7] A. Chakrabarti. An introduction to Linux kernel backdoors. <http://www.infosecwriters.com/hhworld/hh9/lvtes.txt>. Last accessed Aug. 05, 2010.
- [8] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security*, Baltimore, MD, Aug. 2005.
- [9] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dworkin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, Seattle, WA, Mar. 2008.
- [10] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *ICDSC*, Mesa, AZ, Apr. 2001.
- [11] J. Criswell, N. Geoffray, and V. Adve. Memory safety for low-level software/hardware interactions. In *Usenix Security*, Montreal, Canada, Aug 2009.
- [12] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, Boston, MA, Dec. 2002.
- [13] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI*, Seattle, WA, Nov. 2006.
- [14] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX ATC*, Boston, MA, June 2008.
- [15] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *USENIX Security*, Washington, D.C., Aug. 2001.
- [16] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *ASPLOS*, Seattle, WA, Mar. 2008.
- [17] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, San Diego, CA, Feb. 2003.

- [18] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Usenix Security*, Montreal, Canada, Aug 2009.
- [19] Intel. *System Programming Guide: Part 2*. Intel 64 and IA-32 Architectures Software Developer's Manual, 2004.
- [20] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *USENIX ATC*, Boston, MA, June 2006.
- [21] K. Kasslin. Evolution of kernel-mode malware. http://igloo.engineeringforfun.com/malwares/Kimmo_Kasslin_Evolution_of_kernel_mode_malware_v2.pdf. Last accessed Aug. 05, 2010.
- [22] J. Keniston, A. Mavinakayanahalli, P. Panchamukhi, and V. Prasad. Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps. In *Linux Symposium*, Ottawa, Canada, June 2007.
- [23] Kimmo Kasslin. Kernel malware: The attack from within. www.f-secure.com/weblog/archives/kasslin_AVAR2006_KernelMalware_paper.pdf. Last accessed Aug. 05, 2010.
- [24] S. T. King, P. M. Chen, Y. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.
- [25] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security*, San Francisco, CA, Aug 2002.
- [26] Larry McVoy and Carl Staelin. lmbench. <http://www.bitmover.com/lmbench/>. Last accessed Aug. 05, 2010.
- [27] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security*, San Jose, CA, Aug. 2008.
- [28] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu. Probing the guts of kprobes. In *Linux Symposium*, Ottawa, Canada, July 2006.
- [29] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.
- [30] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *USENIX Security*, Vancouver, BC, Canada, Aug. 2006.
- [31] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *ACM CCS*, Alexandria, VA, Nov. 2007.
- [32] A. Ramaswamy. Autopsy: Detecting pattern-searching rootkits via control flow tracing. In *Technical Report TR2009-644*, Dartmouth Computer Science, 2009.
- [33] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *RAID*, Boston, MA, Sept. 2008.
- [34] J. Rutkowska. Subverting Vista kernel for fun and profit. In *Black Hat USA*, 2006.
- [35] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *OSDI*, Seattle, WA, Oct 1996.
- [36] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM SOSP*, Stevenson, WA, Oct. 2007.
- [37] M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *ACM CCS*, Chicago, IL, Nov. 2009.
- [38] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: A thin hypervisor for enforcing I/O device security. In *ACM VEE*, Washington, DC, Mar. 2009.
- [39] Sourceforge. Iperf. <http://sourceforge.net/projects/iperf/>. Last accessed Aug. 05, 2010.
- [40] A. Srivastava, I. Erete, and J. Giffin. Kernel data integrity protection via memory access control. In *Technical Report GT-CS-09-05*, Georgia Institute of Technology, Atlanta, GA, 2009.
- [41] A. Srivastava and J. Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *RAID*, Boston, MA, Sept. 2008.
- [42] A. Srivastava and J. Giffin. Automatic discovery of parasitic malware. In *RAID*, Ottawa, Canada, Sept. 2010.
- [43] Sun Microsystem. Dtrace. <http://wikis.sun.com/display/DTrace/DTrace>. Last accessed Aug. 05, 2010.
- [44] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *ACM SOSP*, Bolton Landing, NY, Oct. 2003.
- [45] Symantec. Spam from the kernel: Full-kernel malware installed by mpack. <http://www.symantec.com/connect/blogs/spam-kernel-full-kernel-malware-installed-mpack>. Last accessed Aug. 05, 2010.
- [46] Tim Bray. Bonnie. <http://www.garloff.de/kurt/linux/bonnie>. Last accessed Aug. 05, 2010.
- [47] Uwe F. Mayer. BYTEmark. <http://www.tux.org/~mayer/linux/bmark.html>. Last accessed Aug. 05, 2010.
- [48] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM SOSP*, Asheville, NC, Dec. 1994.
- [49] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *ACM CCS*, Chicago, IL, Nov. 2009.
- [50] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *OSDI*, San Diego, CA, Dec. 2008.

- [51] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

A. List of isolated commodity drivers

Gateway isolated the following commodity Linux drivers in the DPT during system evaluation. No driver execution resulted in Gateway alerts or control flow failures.

<i>Module</i>	<i>Size</i>
ppdev	9220
autofs4	20100
hidp	16640
l2cap	25088
bluetooth	46308
sunrpc	141884
ip_contrack_netbios_ns	3328
ipt_REJECT	5632
xt_state	2432
ip_contrack	50860
nfnetlink	6808
xt_tcpudp	3456
iptables_filter	3328
ip_tables	12232
x_tables	13060
video	15876
button	7056
battery	9732
ac	5252
ipv6	235744
lp	12872
parport_pc	26276
parport	36040
floppy	60100
nvr	9096
i2c_piix4	8848
i2c_core	21120
8139too	26752
8139cp	21888
mii	5632
dm_snapshot	16428
dm_zero	2048
dm_mirror	20432
dm_mod	51992
ext3	121864
jbd	55700