Protecting Commodity Operating System Kernels from Vulnerable Device Drivers*

Rutgers University Technical Report DCS-TR-645, November 2008.

Shakeel Butt, Rutgers University
Vinod Ganapathy, Rutgers University
Michael M. Swift, University of Wisconsin-Madison
Chih-Cheng Chang, Rutgers University

Abstract

Device drivers on commodity operating systems execute with kernel privilege and have unfettered access to kernel data structures. Several recent attacks demonstrate that such poor isolation exposes kernel data to exploits against vulnerable device drivers, for example through buffer overruns in packet processing code. Prior architectures to isolate kernel data from driver code either sacrifice performance, execute too much driver code with kernel privilege, or are incompatible with commodity operating systems.

In this paper, we present the design, implementation and evaluation of a novel security architecture that better isolates kernel data from device drivers without sacrificing performance or compatibility. In this architecture, a device driver is partitioned into a small, trusted kernel-mode component and an untrusted user-mode component. The kernel-mode component contains privileged and performance-critical code. It communicates via RPC with the user-mode component which contains the rest of the driver code. A RPC monitor mediates all control and data transfers between the kernel- and user-mode components. In particular, it verifies that all data transfers from the untrusted user-mode component to the kernel-mode component preserve kernel data structure integrity. We also present a runtime technique to automatically infer such integrity specifications. Our experiments with a Linux implementation of this architecture show that it can prevent compromised device drivers from affecting the integrity of kernel data and do so without impacting common-case performance.

1. Introduction

Device drivers execute with kernel privilege in most commodity operating systems and have unrestricted access to kernel data structures. Because the kernel is part of the Trusted Computing Base (TCB) of the system, vulnerabilities in driver code can jeopardize the entire system.

Several studies indicate that device drivers are rife with exploitable security holes. A recent study of user/kernel bugs in the Linux kernel found that 9 out of 11 of these bugs were in device drivers [23]. An audit of the Linux kernel by Coverity also found that over 50% of bugs were in device drivers [12]. Our own analysis of vulnerability databases revealed several device drivers that are vulnerable to malformed input from untrusted user-space applications, allowing an attacker to execute arbitrary code with kernel privilege [4, 30]. Similarly, device drivers by their very nature copy untrusted data from devices to kernel memory. Because the kernel does not restrict the memory locations accessible to devices, a compromised driver can write arbitrary values to sensitive kernel data structures. For example, a compromised driver could overwrite the table of interrupt handlers in the operating system with pointers to attacker-defined code. As demonstrated by recently published exploits against wireless device drivers in Windows XP [7, 8] and Mac OS X [27], vulnerabilities in drivers are an increasingly attractive target for attackers.

Microkernels [26, 40, 42] offer one way to iso-

*Supported by NSF awards 0831268, 0915394 and 0931992.

late kernel data from vulnerable device drivers. They execute device drivers as user-mode processes and can prevent malicious modifications to kernel data by enforcing domain-specific rules, *e.g.,* as done in Nexus [40]. However, microkernels restructure the operating system, and the protection mechanisms that they offer are not applicable to commodity operating systems, which are structured as macrokernels. Moreover, enforcing security policies on device drivers may impose significant performance overhead. For example, Nexus reports CPU overheads of 2.5× on a CPU-intensive media streaming workload. User-mode driver frameworks [3, 10, 15, 24, 28, 38] allow commodity operating systems to execute device drivers in user mode. However, porting drivers to these frameworks often requires complete rewrites of device drivers and the resulting performance overheads are often significant [3, 38].

This paper extends prior work on Microdrivers [20] and proposes a security architecture that offers commodity operating systems the benefits of executing device drivers in user mode without affecting common-case performance. In this architecture, each device driver is composed of a trusted kernel-level component, called a *k-driver*, and an untrusted user-level component, called a *u-driver*. The k-driver contains code that requires kernel privilege (*e.g.,* interrupt processing functions) and performance-critical code (*e.g.,* functions on the I/O path). The rest of the code, which contains functions to initialize, shutdown, and configure the device, neither requires kernel privilege nor is on the critical path and executes as a user mode process. The combination of the u-driver and the k-driver is called a *microdriver*. A prior study with 297 Linux device drivers comprising network, sound and SCSI drivers showed that as much as 65% of driver code can execute in user mode without requiring kernel privilege or affecting common-case performance [20].

A u-driver and its corresponding k-driver communicate via an RPC-like interface. When the k-driver receives a request from the kernel to execute functionality implemented in the u-driver, such as initializing or configuring the device, it forwards this request to the u-driver. Similarly, the u-driver may also invoke the k-driver to perform privileged operations or to invoke functions that are implemented in the kernel. However, the u-driver is untrusted and all requests that it sends to the k-driver must be monitored. For example, a u-driver that has been compromised by exploiting a buffer overrun vulnerability may potentially send spurious updates to kernel data structure in its requests to the k-driver. Because the k-driver applies these updates to kernel data structures, the compromised u-driver may affect the security of the entire operating system.

We present a *RPC monitor* to interpose upon all communication between the u-driver and the k-driver, and to ensure that each message conforms to a security policy. The RPC monitor checks both *data values* and *function call targets* in these messages. Data values in messages may contain updates to data structures that the u-driver shares with the k-driver. The RPC monitor enforces integrity constraints on updates to kernel data structures initiated by the u-driver. In our implementation, these integrity constraints are specified as *data structure invariants*—constraints that must always be satisfied by the data structure. For example, one such invariant may state that the list of network devices must not change during an invocation of a u-driver function to obtain device configuration settings. We present an approach to automatically extract such data structure invariants using Daikon [18], a state-of-the-art invariant inference tool. Similarly, the RPC monitor also ensures that k-driver function calls that are invoked by the u-driver via RPC are allowed by a control transfer policy that is extracted using static analysis of the driver.

This paper makes two key contributions over prior work on Microdrivers [20]. First, it presents the design and implementation of the RPC monitor to mediate u-driver/k-driver communication. In prior work on Microdrivers, all communication between a u-driver and a k-driver was unchecked, thereby poorly isolating kernel data from untrusted u-drivers. Second, it presents a technique to automatically infer data structure integrity constraints to be enforced by the RPC monitor. The key property of these constraints is that they express invariants over heap data structures, thereby constricting the updates that a compromised u-driver can apply to kernel data structures.

The security architecture proposed in this paper offers several benefits over prior isolation architectures.

- **Reduction of kernel-mode driver code.** Isolation architectures such as Nooks [35], Mondrix [43] and SafeDrive [41] execute drivers in kernel mode and do not monitor driver-initiated updates to kernel data structures. Consequently, the kernel can be compromised by exploiting vulnerabilities that these architectures do not protect against, *e.g.,* race conditions and double-free bugs. In contrast, our architecture executes a large fraction of driver code in user space (as u-drivers) and monitors kernel data structure updates initiated by u-drivers.
- **Compatibility with commodity operating systems.** A k-driver interfaces with the kernel in much the same way as a traditional device driver. Kernel calls to functions implemented in the u-driver are transparently forwarded by the k-driver to the u-driver. Therefore, in contrast to prior work on microkernels, our security architecture is compatible with commodity operating systems.
- **Good common-case performance.** User-mode driver frameworks have often sacrificed performance for security [3, 38]. In contrast, our architecture executes performance-critical functionality in the kernel thereby imposing no runtime overhead for the common case.
- **Flexibility.** Rather than offering a rigid definition of trusted and untrusted components, our architecture offers the flexibility in choosing which portions of the driver execute with kernel privilege. Although performance-critical functions must preferably be executed in the k-driver, our architecture does not enforce such restrictions. Thus, for instance, the kernel can be protected from zero-day attacks by relegating code with newly-discovered vulnerabilities to the u-driver until the driver vendor issues a patch.

Despite these benfits, our security architecture is not a panacea and cannot completely prevent a compromised u-driver from hijacking the kernel. Nevertheless, our experiments show that it can prevent a significant fraction of attacks from propagating to and hijacking the kernel.

We have implemented our security architecture in the Linux-2.6.18.1 kernel and have applied it to four device drivers. Experiments show that our architecture can protect against compromised u-drivers and do so without affecting common-case performance.

2. Background and scope

Device drivers for commodity operating systems execute in the same protection domain as the rest of the kernel to achieve good performance and easy access to hardware. This architecture does not isolate kernel data from vulnerabilities in device drivers, which are written in C by third-party vendors. Such vulnerabilities, especially in packet-processing code and `ioctl` handlers, can be exploited by malicious user-space applications. For example, recent work [7, 8] shows that a remote attacker can hijack control of Windows machine by exploiting a buffer overflow in beacon and probe response processing code in an 802.11 device driver. Indeed, our study of vulnerability databases revealed several exploitable buffer overrun and memory allocation vulnerabilities in driver code [4, 30].

The threats posed to kernel data by compromised device drivers can broadly be classified into two categories.

- **Threats at the kernel/driver interface.** Kernel data structures are routinely updated by device drivers, and the kernel imposes no restrictions on the memory regions accessible to drivers or devices. This freedom can be misused by compromised drivers in a variety of ways. Compromised device drivers can corrupt kernel data structures, causing the kernel to crash. Similarly, drivers can update kernel hooks to point to attacker-defined code, leading to arbitrary code execution that cannot be detected by user-mode security tools.
- **Threats at the driver/device interface.** A compromised driver can maliciously modify the state of the device, *e.g.,* by writing arbitrary values to its registers or exhausting its resources. More seriously, a driver can harm kernel data structure integrity using DMA. The driver can initiate DMA transfers to an arbitrary physical memory address by simply writing this address to a device register. Because the kernel does not restrict the memory regions accessible to a device, a DMA transfer will overwrite these memory locations.

The architecture proposed in this paper helps detect and prevent several threats at the kernel/driver interface. By relegating a large portion of the device driver to a user-space u-driver and monitoring all data and control transfers at the user/kernel boundary, it restricts the amount of driver code that can directly access kernel memory. Our architecture can therefore protect against requests originating from a compromised u-driver. However, to ensure good performance, our architecture does *not* mediate the kernel/k-driver interface. Consequently, it cannot protect against malicious k-drivers and other kernel-resident malware. The k-driver is trusted in our architecture and can be protected using prior fault isolation techniques [17, 43], although we do not do so in our implementation.

We do not address threats at the driver/device interface in this paper. Monitoring data transfers from the device to kernel memory either requires the use of new hardware mechanisms for virtualized I/O (such as IOMMU [2] and VT-D [1]) as done in iKernel [36], or reference monitoring at the driver/device interface as done in Nexus [40]. These techniques are orthogonal to, and may possibly be used in conjunction with, the architecture proposed here.

We also assume the availability of driver source code. This is because our driver partitioning tool (discussed in Section 4) operates on source code. While this limitation precludes us from partitioning and protecting against device drivers that are only distributed in binary form, a partitioning tool that works at the binary level would allow even such drivers to be adapted to our architecture.

3. Design

Our security architecture aims to protect kernel data from vulnerable device drivers that can be compromised by malicious inputs from untrusted user-space applications and from hardware. We begin by outlining our design goals.

- **Kernel data structure integrity.** The architecture must monitor kernel data structure modifications initiated by device drivers and ensure that these updates comply with a security policy. Each device driver is associated with a security policy that specifies permissible updates to kernel data structures. These security policies specify kernel data structure integrity constraints, and may either be specified manually using domain-specific rules, *e.g.,* as in Nexus [40], or extracted automatically, as in our implementation.
- **Good common-case performance.** Device drivers are on the critical path that transfers data between user-space applications and external devices. Hence, to be practical, the architecture must not significantly impact I/O throughput.
- **Compatibility.** Modern operating systems support several thousand device drivers. The architecture must secure the kernel without requiring significant changes to either the operating system or requiring a rewrite of device drivers.

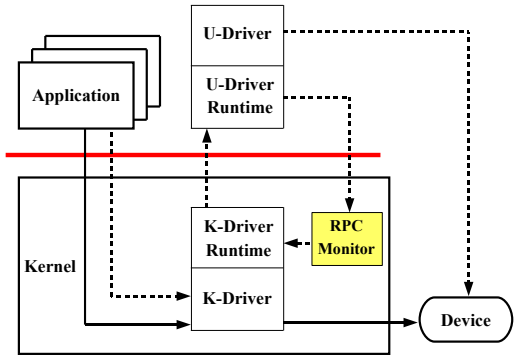


Figure 1. Design of our device driver security architecture. The solid lines show the performance-critical path while the dashed lines show the non-performance-critical path.

The above design goals are conflicting and are challenging to achieve simultaneously. Commodity operating systems often share several kernel data structures with device drivers that are updated on performance critical I/O paths making monitoring *all* updates impractical without restructuring the operating system.

Our security architecture, shown in Figure 1, therefore leverages prior work on the Microdrivers architecture to achieve the above goals on a large fraction of device driver code. The Microdrivers architecture offers mechanisms to *split* device drivers along performance and priority boundaries without

changing the kernel/driver interface. A microdriver consists of a small kernel-mode k-driver that contains performance-critical and high priority functions, and a u-driver that contains non-performance critical code, such as device initialization and configuration.

The kernel communicates with the k-driver via the standard driver/kernel interface to transfer data to/from the device. Some of these requests, such as those to initialize and configure the device, may invoke functionality that are implemented in the u-driver. When such requests arrive, the k-driver invokes the u-driver via the Microdrivers *runtime* (shown in Figure 1 as the k-driver runtime and the u-driver runtime). The runtime has two key responsibilities:

(1) Communication. It provides mechanisms to transfer control and data between the u-driver and the k-driver. It provides RPC stubs that implement *upcalls* (*i.e.,* the k-driver invoking the u-driver) and *downcalls* (*i.e.,* the u-driver invoking the k-driver) to enable transfer of control; it also implements marshaling/unmarshaling protocols to transfer data.

(2) Object tracking. Splitting a device driver into a k-driver and u-driver results in data structures being copied between address spaces. The runtime tracks and synchronizes the k-driver's and the u-driver's versions of driver data structures. Specifically, it is responsible for propagating the k-driver's changes to a driver data structure to a u-driver upon an upcall, and for propagating the u-driver's changes to the k-driver when the upcall returns or when the u-driver makes a downcall into the k-driver.

There are several key challenges that must be addressed by the runtime. For example, it must ensure that the u-driver and the k-driver can never simultaneously lock a data structure, and that when the lock is released, the copies of the data structure in the u-driver and the k-driver are synchronized. It must also correctly allocate and deallocate memory in user and kernel space in response to allocation/deallocation requests by the u-driver and the k-driver. We refer the interested reader to the Microdrivers paper [20], which describes mechanisms to deal with these challenges in detail.

3.1. RPC monitor

As discussed above, the runtime ensures that driver data structure changes made by the u-driver are propagated to the k-driver, either when an upcall returns or when the u-driver issues a downcall. Because the u-driver is untrusted, all *data* and *control* transfers initiated by the u-driver must be checked against a security policy. This is the task of the *RPC monitor*, shown in Figure 1, which mediates all RPC messages from the u-driver to the k-driver. Note that control and data transfers from the k-driver to the u-driver *need not* be mediated because the k-driver is trusted. Because our architecture seeks to protect the integrity of kernel data (rather than its secrecy), the RPC monitor need only monitor writes to kernel data structures. The RPC monitor is implemented as a kernel module that enforces security policies before control and data are transferred to the k-driver.

Monitoring data transfer. A compromised u-driver can maliciously modify kernel data structures by passing corrupt data. The RPC monitor must therefore detect and prevent malicious data transfers.

When a u-driver returns control to its k-driver following an upcall, or when the u-driver invokes functionality implemented in the kernel or the k-driver via a downcall, data structures in the k-driver are synchronized with their u-driver counterparts using the marshaling protocol. The RPC monitor ensures that each such update conforms to a driver-specific security policy. Intuitively, the goal of the security policy is to ensure that kernel data structures are not updated maliciously, *i.e.,* each update must preserve *kernel data structure integrity*. For instance, an update must not allow a compromised u-driver access to kernel/device memory regions that a benign u-driver does not normally access. Similarly, an update must not allow a compromised u-driver to execute arbitrary code with kernel privilege.

Specifying such integrity constraints is challenging because of the quantity and heterogeneity of kernel data structures updated by device drivers. In addition, our security architecture splits device drivers to ensure good performance; consequently, several driver-specific data structures may be copied across the user/kernel boundary. For ex-

ample, Linux represents network devices using a per-driver `net_device` data structure. In a network microdriver, this data structure may be modified by the u-driver, for example, when the device is initialized or configured. It is also important to monitor updates to such driver-specific data structures because these updates propagate to the kernel. Specifying integrity constraints for driver data structures often requires domain-specific knowledge, therefore making manual specification of such integrity constraints cumbersome and error-prone.

To overcome these challenges, we present an approach that automatically infers integrity constraints by monitoring driver execution. In our architecture, these constraints are expressed as *data structure invariants*—properties that the data structure must always satisfy. For example, an invariant may state that a function pointer to the packet-send function (*e.g.,* the `hard_start_xmit` pointer in the `net_device` data structure in Linux) of a network driver must not change after being initialized. Our approach infers such invariants during *training*; these are checked during *enforcement*.

During the training phase, we execute the u-driver on several benign workloads, and use Daikon [18] to infer data structure invariants automatically. Daikon does so by observing the values of data structures that cross the user/kernel boundary and hypothesizing invariants. During the enforcement phase, the RPC monitor enforces these invariants on data structures received from a u-driver; it first copies these data structures to a *vault* area in the kernel, and checks that the invariants hold. If they do, it updates kernel data structures with values from the vault. The kernel itself never uses data structures directly from the vault before they are checked by the RPC monitor. By monitoring data transfers from the u-driver to the k-driver, the RPC monitor prevents compromised u-drivers from affecting kernel data integrity.

Monitoring control transfer. The RPC monitor checks u-driver to k-driver control transfers to prevent the u-driver from making unauthorized calls to kernel functions.

As the u-driver services an upcall, it may invoke the k-driver via a downcall, either to call a k-driver function or to execute a function implemented in the kernel. Downcalls are implemented using `ioctl` system calls that are handled in the k-driver. Because the u-driver is untrusted, these downcalls must be verified to be legitimate, *e.g.,* that a downcall is not initiated by a code injection attack on a compromised u-driver. Such unauthorized downcalls can be maliciously used by the u-driver, *e.g.,* to cause denial of service by invoking the kernel function to unregister a device. To avoid such attacks, we statically analyze the u-driver and extract the set of downcalls that a u-driver can issue in response to an upcall (static analysis is performed before the driver is loaded). The RPC monitor enforces this statically extracted policy when it receives a downcall from the u-driver.

Having checked both data and control integrity, the RPC monitor transfers control to the k-driver, which can now resume execution on newly-updated kernel data structures.

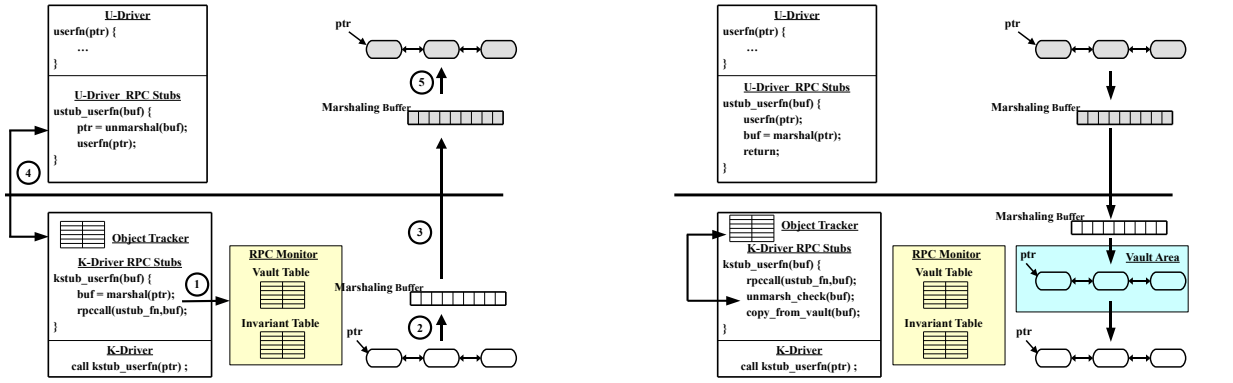
4. Implementation

We extended the implementation of the Microdrivers architecture on the Linux-2.6.18.1 kernel with support to monitor data and control transfers from the u-driver to the k-driver. In this implementation, the k-driver, the kernel runtime and the RPC monitor are implemented as a kernel module while the u-driver and the user runtime execute as a multi-threaded user-space process.

4.1. Background on Microdrivers

A microdriver begins operation when its kernel module is loaded and the user-space process is started. The main thread of the user-space process makes an `ioctl` call into the kernel module and blocks. The kernel module unblocks this thread when it needs to invoke functions in the u-driver.

The u-driver and k-driver exchange data and control using an RPC-like mechanism, shown in Figure 2. To invoke the u-driver using an an upcall (Figure 2(a)), the k-driver (**1**) registers the k-driver function that initiates the upcall with the RPC monitor; (**2**) marshals data structures that will be read/modified by the u-driver; and (**3**) unblocks the thread of the u-driver's user-space process. This transfers control to the u-driver, which in turn (**4**) consults the *object tracker* and unmarshals the data structures into its address space; and (**5**) invokes the appropriate u-driver function on the



(a) Data movement from a k-driver to a u-driver. **(b) Data movement from a u-driver to a k-driver.**

Figure 2. Data movement during upcalls and downcalls. During downcalls, data is first unmarshaled into the vault area to enforce invariants before updating kernel data structures.

unmarshaled data structure. The object tracker is a bi-directional table responsible for maintaining the correspondence between kernel- and user-mode pointers of data structures shared between the k-driver and the u-driver. As the u-driver runtime unmarshals data received from the k-driver into its address space, it uses the object tracker to identify u-driver objects that correspond to kernel-mode pointers received from the k-driver. If the runtime is unable to find such an object, *e.g.,* because the k-driver or the kernel created a new object that the u-driver is unaware of, the u-driver can allocate a new object and enter a new mapping into the object tracker.

When an upcall returns, or when the u-driver invokes functions in the k-driver via an `ioctl` system call (*i.e.,* a downcall), data is marshaled by the u-driver and unmarshaled in the kernel, as shown in Figure 2(b). The main difference in this case is that a RPC monitor interposes on these requests before they are forwarded to the k-driver. The RPC monitor has two key responsibilities—(i) to check control transfers; and (ii) to check data structure integrity. The RPC monitor uses a statically-extracted control flow policy to check control transfers—this policy statically determines the set of allowed downcalls for each upcall. For each downcall, the RPC monitor uses the k-driver function registered with it (in step (1) of Figure 2(a)) to ensure that the downcalls are allowed. If this downcall is allowed, the RPC monitor checks the integrity of data structures received from the u-driver. To do so, it unmarshals the data received from the u-driver into a *vault* area. This area is not accessed by the k-driver and is only used by the RPC monitor to check data structure integrity. The RPC monitor checks that each variable that was unmarshaled satisfies a set of invariants; if so, it uses the data from the vault area to update kernel data structures and frees any data structures the vault.

DriverSlicer. To allow existing device drivers on commodity operating systems to benefit from our architecture, we extended DriverSlicer, a device driver partitioning tool [20], to generate security enforcement code. DriverSlicer is implemented as a plugin to CIL [31], a source code transformation tool, and consists of about 11,000 lines of Ocaml code. Given a small number of annotations, DriverSlicer automatically partitions a device driver into a k-driver and a u-driver. It also generates code for the k-driver and u-driver runtimes, and the RPC monitor, including code to check control transfers from the u-driver to the k-driver and code to monitor data structure integrity.

DriverSlicer consists of two parts: a *splitter* and a *code generator*. The splitter analyzes a device driver and identifies functions that must execute in the kernel, *i.e.,* those that require kernel privilege to access the device or those that are performance critical. To do so, it uses programmer-supplied specifications in the form of *type signatures*, to identify such functions. For example, it identifies interrupt

handlers based upon their function prototypes; in Linux interrupt handlers always return a value of type `irqreturn_t`. Similarly, functions responsible for transmitting network packets typically have two parameters: a pointer to an `sk_buff` structure, and a pointer to a `net_device` structure. Such type signatures need only be supplied *once per family of drivers*, *e.g.,* one set of type signatures suffices to identify performance critical and privileged functions for most network drivers. The splitter uses a statically-extracted call-graph of the device driver to mark (**1**) all functions that match these type signatures; and (**2**) all functions potentially called (transitively) by such functions as those that must execute in the k-driver; the remaining functions are relegated to the u-driver.

DriverSlicer's code generator uses the output of the splitter to partition the driver into a k-driver and a u-driver, and generates RPC code to transfer control and data. In doing so, it may require programmer-supplied annotations to clarify the semantics of pointers. For example, to generate code to marshal an object referenced by a `void *` pointer, the code generator must be supplied with an annotation that determines the type of the object. Similarly, the code generator also requires annotations to determine whether a pointer refers to *one* instance of an object or to an *array* of instances. DriverSlicer currently uses eight kinds of annotations, details of which appear elsewhere [20]. DriverSlicer uses these annotations to generate RPC code that minimizes the amount of data copied between the u-driver and the k-driver; it does so by using static analysis to determine variables and data structure fields that are read/modified by the u-driver and only generating marshaling code to copy these variables and fields using RPC.

4.2. Monitoring kernel data structure updates

This section describes an anomaly detection-based approach to infer and enforce invariants on kernel data structures. The approach has two phases: a *training* phase, in which invariants are inferred by executing the driver on benign workloads, and an *enforcement* phase, in which the RPC monitor enforces these invariants. The training phase is a one-time activity that can possibly be completed during driver development. Section 4.2.1 presents an automated technique to infer invariants during training; Section 4.2.2 describes how these invariants are enforced.

4.2.1 Inferring data structure integrity constraints

To identify kernel data structure invariants, we adapted Daikon [18], an invariant inference tool. Daikon consists of two components, namely, a *front end* that records the values of variables during application execution and an *inference engine* that uses these values to hypothesize *likely invariants*. The front end records the values of global variables and formal parameters of functions at key program points, such as function entries and exits, as the application executes a set of test inputs. The inference engine uses these values to hypothesize invariants. For example, if the value of a variable is observed to be a constant across multiple executions of the program, Daikon hypothesizes that the variable is likely a constant. Daikon infers over 75 different kinds of invariants, including constancy of scalar-valued variables, arrays and pointers, bounds of scalars, and relationships between different variables.

Daikon currently only applies to user-space applications. However, our architecture executes the u-driver as a user-space process, which allows Daikon to be applied to driver code. We use Daikon's front end to monitor the execution of a u-driver as it runs several benign workloads for each driver, such as device initialization, configuration and shutdown, that exercise functionality implemented in the u-driver. The front end records a trace of values of all the global variables and formal parameters of functions that cross the user/kernel boundary. Daikon's inference engine processes this trace and hypothesizes invariants. Figure 3 presents several examples of invariants that Daikon identified for the 8139too network driver; the left column shows several functions that appear in the u-driver of the 8139too microdriver, and indicates whether the invariant was inferred at the entry or exit (or both program points) of the function. Invariants inferred at the exits of upcall functions are enforced by the RPC monitor (Section 4.2.2). We discuss below several classes of invariants that we found

Function	Invariant
rtl8139_init_module (entry)	rtl8139_intr_mask == C07F, rtl8139_norx_intr_mask == C02E
rtl8139_init_module (exit)	rlt8139_intr_mask == ORIG(rtl8139_intr_mask) rtl8139_norx_intr_mask == ORIG(rtl8139_norx_intr_mask) rtl8139_rx_config == ORIG(rtl8139_rx_config) rtl8139_tx_config == ORIG(rtl8139_tx_config)
rtl8139_get_ethtool_stats (exit)	rtl_chip_info has only one value
rtl8139_get_link (exit)	dev->hard_start_xmit has only one value
rtl8139_open (entry/exit)	dev->base_addr ∈ {0x0531C468, 0x06520468}
rtl8139_get_link (exit)	LENGTH(dev->mc_list) == ORIG(LENGTH(dev->mc_list))

Figure 3. Examples of invariants extracted from the 8139too driver.

useful in our experiments.

(1) Constancy of scalars and pointers. Daikon determines whether a scalar-valued variable (*i.e.,* a value of a base type, such as `int` or `char`) remains constant during driver execution. If so, it also records the constant value that the variable acquires. For example, consider the integer-valued global variable `rtl8139_intr_mask` of the 8139too driver, which represents a 16-bit mask. Daikon infers that this variable has a constant value of `C07F` when the function `rtl8139_init_module` is invoked (see Figure 3). Indeed, an analysis of the driver shows that this variable is always initialized in the k-driver to this value.

In addition to scalar variables, Daikon also determines whether a pointer always refers to the same object during program execution. For example, Daikon infers that the pointer-valued global variable `rtl_chip_info` is not modified by the `rtl8139_get_ethtool_stats` function. Similarly, it infers that the function pointer `dev->hard_start_xmit` is unmodified by a call to `rtl8139_get_link` (and most other functions in the u-driver). Inferring and enforcing such invariants on function pointers can prevent control hijacking attacks. In fact, a recent study of 25 Linux rootkits revealed 22 rootkits that hijacked kernel control flow by modifying function pointers to point to attacker-defined code [34]. Note that for pointers that refer to the same object, Daikon only reports that the pointer is a constant and does not report the actual value of the pointer (which would vary across reboots).

(2) Relationships between variables. Daikon correlates the values of variables and discovers relationships between them. For example, it can discover that two variables always acquire the same value at runtime. Importantly, Daikon can determine whether the value of a variable remains unchanged during a function call by observing its values at function entry (the ORIG value of the variable) and exit. For example, it determines that the value of `rtl8139_intr_mask` is unchanged by a call to `rtl8139_init_module`. Enforcing such an invariant constrains the values of `rtl8139_intr_mask` that can otherwise be modified by a compromised u-driver to initiate I/O to unauthorized ports.

(3) Ranges/sets of values. In several cases, a variable may not be a constant, but acquire one of a small set of values. As Figure 3 shows, Daikon infers such invariants as well; for example, it infers that the `dev->base_addr`, which represents the base address of I/O memory, can only acquire one of two values during driver execution. This constraint must be enforced when the k-driver's copy of `dev->base_addr` is synchronized with the u-driver's copy; for otherwise, a compromised u-driver could coerce the k-driver into writing to arbitrary I/O memory addresses belonging to other devices.

(4) Linked list invariants. The Linux kernel uses linked lists extensively to manage several critical data structures. Prior work demonstrates that kernel linked lists can be stealthily modified to achieve malicious goals [33]. Unfortunately, Daikon's C front end does not support inference of invariants on linked lists.

We therefore extended Daikon to infer invariants on linked lists. In particular, we augmented the marshaling protocol with code that records the contents of linked lists that cross the user/kernel boundary. Daikon then processes this trace of values and hypothesizes invariants. Our implementation cur-

rently supports inference of invariants that indicate that the length of a linked list is unmodified by a function call. Figure 3 presents one such invariant, which states that the linked list `dev->mc_list` is unmodified by a call to `rtl8139_get_link`.

A key feature of the above invariants is their ability to monitor the integrity of *both control and non-control data* in the kernel. For example, by inferring the constancy of function pointers, Daikon can detect attacks that hijack control flow by modifying function pointers to attacker-defined code. Similarly, Daikon can detect attacks that modify I/O memory addresses and allow a rogue driver to write to arbitrary memory locations, thereby preventing this non-control data attack. Daikon's dynamic analysis approach enables it to infer several kinds of invariants that would be difficult to discover using static analysis of the driver. For example, static analysis is ill-suited to infer invariants on lengths of linked lists. Similarly, in pointer-intensive code (as is common in device drivers), it is hard to statically infer whether a heap object is unmodified by a function call without access to precise aliasing information.

One of the challenges that we faced during development was the sheer quantity of data recorded by Daikon's front end during the execution of a u-driver. This in turn resulted in two problems. First, Daikon's inference engine took longer to infer invariants, and sometimes even exhausted the memory available on the machine. Second, Daikon inferred several hundred invariants per function, which resulted in increased memory consumption during enforcement. For example, consider the 8139cp network microdriver: Daikon inferred an average of 878 invariants at the exit of each function in the u-driver. Worse, several of these invariants were *serendipitous*, *i.e.,* they were overly specific to the workloads used during inference and were not satisfied by other workloads, thereby resulting in false positives during enforcement.

To overcome these problems, we incorporated two key optimizations. First, we configured Daikon's front end to only record values transmitted to u-driver functions that communicate directly with the k-driver via upcalls and downcalls, and do not record values for functions internal to the u-driver. Second, we configured the front end so that only the values of variables that are accessed in the u-driver are recorded. For example, if a u-driver function only reads/modifies certain fields of an otherwise large C `struct`, we only record the values of the fields that are read/modified by that function. To implement this optimization, we employed a conservative static analysis of the u-driver to determine the fields read/modified by functions in the u-driver. Because DriverSlicer's code generator emits marshaling and unmarshaling code only for variables and fields of data structures that are read/modified by the u-driver, as discussed in Section 4.1, malicious modifications by the u-driver on other variables and data structure fields will *not* be synchronized with the k-driver; hence, they need not be monitored.

These optimizations drastically reduce the number of invariants that Daikon infers, which in turn reduces the memory consumption of the invariant table (described below) during enforcement. For example, in the 8139cp network microdriver, the average number of invariants at function exits drops over forty-fold.

We expect that inferring invariants would be a one-time activity, accomplished either during driver development (if the driver is developed as a microdriver), or when a legacy driver is split with DriverSlicer; these invariants can be distributed by vendors along with drivers. Note, however, that some invariants inferred by Daikon must be modified to be widely applicable across multiple installations and configurations. For example, the invariant for `dev->base_addr` in Figure 3 refers to specific I/O memory addresses, and is not applicable across multiple installations (the other invariants in Figure 3 are portable across multiple installations). To be portable, this invariant would have to be modified to generically state that `dev->base_addr` has only two values, rather than referring to specific I/O memory addresses, *e.g.,* as with the invariant for `dev->hard_start_xmit` in Figure 3.

4.2.2 Enforcing data structure integrity constraints

The invariants inferred by Daikon are enforced by the RPC monitor when the k-driver receives marshaled data from the u-driver. The RPC monitor unmarshals this data into a *vault* area in the kernel's

address space. Data structures in the vault area are only accessed by the RPC monitor and not by the kernel.

The RPC monitor itself is implemented as a kernel module that manages two tables: an *invariant table* and a *vault table*. The invariant table stores the set of invariants indexed by the u-driver variable(s) that it is associated with, and is initialized when the microdriver is loaded. The vault table stores pointers to data structures in the vault area and is filled by the RPC monitor when it populates the vault area.

The RPC monitor enforces invariants on data received from the u-driver by first unmarshaling this data into the vault area and inserting pointers to these resulting data structures in the vault table. This unmarshaling code is automatically generated by DriverSlicer's code generator. The marshaling code emitted by the code generator also makes a copy of the original values of variables before an upcall to support invariants that refer to the ORIG value of a variable. To enforce invariants, the RPC monitor retrieves the invariants associated with each variable in the vault table using the invariant table, and verifies that the invariant is satisfied. For invariants on variables that point to the head of a linked list, the RPC monitor traverses the list and ensures that the invariant is satisfied. Any failures raise an alert and can trigger recovery mechanisms, such as restarting the u-driver. If all invariants are satisfied, then the marshaling procotol synchronizes kernel data structures by overwriting them with their copies in the vault area.

4.3. Monitoring control transfers

This section describes the techniques used to extract and enforce policies on control transfers from the u-driver to the k-driver. A u-driver may issue downcalls as it serves an upcall from the k-driver. The RPC monitor enforces (Section 4.3.2) a statically extracted control transfer policy (Section 4.3.1) to ensure that the downcall is permitted. Extracting and enforcing such control transfer policies is necessary to prevent code injection attacks via a compromised u-driver; for example, an attacker with control over a u-driver can issue a downcall to a kernel function that unregisters a device, thereby causing denial of service.

4.3.1 Extracting control transfer policies

To extract a control transfer policy, we employ static analysis of the u-driver. We first use DriverSlicer to statically extract a call graph of the u-driver. This call graph contains one node for each function in the u-driver; an edge $f \rightarrow g$ indicates that f can potentially call g (possibly indirectly, via a function pointer). We resolve function pointers using a simple type-based pointer analysis: each function pointer can refer to any function whose address is taken, and whose type signature matches that of the function pointer. DriverSlicer's splitter identifies potential entrypoints into the u-driver; its code generator also includes an RPC stub in the k-driver for each such entrypoint via which upcalls are issued. For each entrypoint, we use the call graph to identify the set of downcalls that the entrypoint can potentially issue—this set of downcalls associated with each entrypoint serves as the control transfer policy.

Associating an upcall with a *set* of downcalls can result in a permissive policy that can potentially admit mimicry attacks [39]. However, we note that in order to compromise kernel data structures, a compromised u-driver issuing a downcall must also send appropriate data with the downcall. As discussed in Section 4.2, the RPC monitor checks the validity of this data in addition to monitoring control transfer, thereby constraining the attacker. Nevertheless, our architecture admits the enforcement of more complex control transfer policies, such as the *sequence* of downcalls that can follow an upcall. Prior work has developed techniques to extract such control transfer policies (*e.g.,* [21]); we plan to extend our architecture with such support in future work.

4.3.2 Enforcing control transfer policies

The RPC monitor enforces the control transfer policy extracted above. When a function in the k-driver makes an upcall into the u-driver, the k-driver registers the entrypoint invoked with the RPC monitor, which in turn pushes this entrypoint on a stack. When the u-driver issues a downcall, the RPC monitor interposes on this request and ensures that the downcall is allowed by the control transfer policy associated with the entrypoint at the head of the stack. The RPC monitor pops the stack when the

upcall returns.

It is important to use a stack to track the currently-active entrypoint because an upcall into the u-driver can possibly result in multiple control transfers between the user and the kernel. DriverSlicer's splitter ensures that there is at most one upcall along any path in the *static* call-graph of the driver. However, in response to an upcall, the u-driver may need to invoke a function that is implemented in the operating system kernel (*e.g.,* the `register_netdev` function to register a network device; note that this is a *kernel* function, not a *k-driver* function). In turn, the kernel function may call back into the driver and the relevant function may be implemented in the u-driver, thus resulting in multiple control transfers.

5. Evaluation

In this section, we report on experiments conducted on four drivers secured using our architecture. We ported the device drivers for the RealTek RTL-8139 (8139too) and 8139C+ (8139cp) network cards, the driver for the Ensoniq sound card (ens1371), and the driver for the Universal host controller (USB) interface (uhci-hcd) to our security architecture. We used QEMU 0.9.1 (for the network and USB drivers) and VMWare workstation 6 (for the sound driver) running an unmodified Linux-2.6.18.1 kernel as the testbeds for our experiments. Though the Linux kernel has several thousand drivers, we restricted ourselves to four drivers for two reasons. First, we only considered drivers available on our test platforms. Second, DriverSlicer is not yet completely automatic (neither are other RPC libraries, such as MSIDL [29]); porting drivers requires domain-specific understanding and is time-consuming. Nevertheless, the four drivers above represent three major driver families, with different kernel/driver interfaces.

5.1. Privilege separation

We used DriverSlicer to partition the drivers that we considered into a k-driver and a u-driver. The k-driver of each driver contains performance-critical code and code that requires kernel privilege to execute. Figure 4 compares the size of the k-driver and the u-driver. As this figure shows, our architecture reduces the amount of hand-written driver code running with kernel privilege and was able to remove several non-critical functions to user space. As discussed in Section 4.1, to split a driver into a microdriver, DriverSlicer requires that the driver be annotated to clarify semantics of pointers that cross the user/kernel boundary. Figure 4 also presents the number of annotations needed, classified as annotations on kernel headers, which have to be provided just once per version of the kernel, and driver-specific annotations. As this Figure shows, device drivers can be ported into our architecture with only a small number of annotations.

In addition to the k-driver, the kernel runtime and the RPC monitor also execute with kernel privilege and contain RPC code for control and data transfer. Though DriverSlicer currently emits several thousand lines of RPC code, we note that this code is highly stylized and is automatically generated by DriverSlicer. The correctness of this code can be ensured by verifying DriverSlicer.

5.2. Ability to prevent attacks

We evaluated the ability of our architecture to prevent attacks by simulating common attacks on driver code. As indicated by recent vulnerability reports device drivers, buffer overflows, especially in packet processing code and `ioctl` handlers are the most exploited class of vulnerabilities. Because u-drivers contain the bulk of non-performance-critical functionality, including parsing of control packets and `ioctl` handling, we tested the ability of our security architecture at preventing simulated buffer overflow attacks on u-drivers.

To do so, we first obtained a set of invariants for each driver using a benign workload in a controlled training phase. This workload exercised functions implemented in the u-driver of each driver, such as initializing and closing the device and configuring device parameters. Specifically, for the network drivers, we configured several device parameters using the `ethtool` utility, for the sound driver, we played music files in several formats and adjusted parameters using the `alsamixer` utility, while for the USB driver, we inserted and removed several USB devices. In addition, we also initialized and closed the devices repeatedly. We configured the training workload to maximize the number of func-

	Size of K-driver		Size of U-driver		Number of Annotations	
Driver	**SLOC**	**# Functions**	**SLOC**	**# Functions**	**Kernel header**	**Driver specific**
8139too	545 (33.7%)	11 (21.6%)	1070 (66.2%)	40 (78.4%)	34	8
8139cp	735 (44.7%)	21 (36.8%)	908 (55.3%)	36 (63.1%)	18	16
ens1371	890 (59.7%)	28 (43.7%)	599 (40.3%)	36 (56.3%)	7	7
uhci-hcd	2060 (81.8%)	60 (87.0%)	457 (18.2%)	9 (13.0%)	27	146

Figure 4. Sizes of the k-driver and the u-driver, and the number of annotations needed by DriverSlicer.

Driver	# Funcs. in u-driver	# Funcs. covered
8139too	40	35
8139cp	36	33
ens1371	36	14
uhci-hcd	9	7

Figure 5. Function coverage (in the u-driver) obtained by the training workload.

Driver	# Invariants	Inv. tab.	Vault tab.
8139too	2607	247,661	65,180
8139cp	212	17,217	14,817
ens1371	750	70,218	3,918
uhci-hcd	163	12,888	7,455

Figure 6. Memory consumption (in bytes) of the invariant and vault tables.

tions invoked in the u-driver. Figure 5 presents the coverage obtained by our training workload. Although we could not achieve 100% coverage using our workload, [1] we expect that such coverage can be achieved by vendors during driver development using regression test suites. Figure 6 shows the total number of invariants inferred for each driver.

In the testing phase, we used the RPC monitor to enforce these invariants on u-driver to k-driver communication. During this phase, we simulated a compromised u-driver by considering three classes of attacks, as discussed below. Figure 6 presents the memory consumption of the RPC monitor.

• ***Control hijacking via injected downcalls.*** We simulated code injection attacks on the u-driver by injecting `ioctl` system calls that would result in a downcall to the k-driver. For example, we injected a downcall to the kernel function `netif_wake_queue` in one of the u-driver functions. The purpose of this function is to allow upper layers to call the driver's function to transmit packets and for flow control when transmit resources are available. This code injection attack may result in data loss or block the wait queue.

Because the RPC monitor verifies the set of downcalls that each upcall is allowed to issue, using a control transfer policy, it was successfully able to detect such injection attacks.

• ***Control hijacking via modified function pointers.*** An attacker with control over a u-driver can find function pointers that are communicated from the u-driver to the k-driver, and set them to point to arbitrary code (either injected code or to existing kernel functions), thus resulting in arbitrary code execution within the kernel. We simulated such an attack within a u-driver function (`rtl8139_get_link` in the 8139too driver) by modifying the `dev->hard_start_xmit` function pointer to point to attacker-injected code. The `dev->hard_start_xmit` function pointer typically refers to the function that transmits packets. When the upcall to `rtl8139_get_link` returns, the kernel's copy of the `hard_start_xmit` function pointer will be updated, thereby resulting in attacker-defined code executing with kernel privilege each time the driver attempts to send a packet. The RPC monitor was able to prevent this attack by enforcing the invariant that `dev->hard_start_xmit` is not modified by a call to the `rtl8139_get_link` function (see Figure 3).

Although we only executed the above attack in our experiments, we verified that Daikon had

[1] We either did not know how to invoke the functions that were not covered by the training workload or were unable to use the applications needed to invoke these functions on our testbed. For instance, we were unable to call several MIDI-related functions in the audio driver because our test application (Realplayer 11) refused to play MIDI files on the Linux distribution we used (Fedora 5).

		Original driver		Driver in our architecture	
Driver	**Workload**	**Throughput**	**CPU (%)**	**Throughput**	**CPU (%)**
8139too	TCP-send	63.39Mbps	99.76%	61.20Mbps (-3.45%)	99.86% (0%)
8139too	TCP-receive	91.96Mbps	34.84%	90.35Mbps (-1.8%)	34.96% (0%)
8139cp	TCP-send	64.02Mbps	99.88%	64.51Mbps (+0.7%)	99.94% (0%)
8139cp	TCP-receive	90.88Mbps	31.82%	91.66Mbps (+0.8%)	29.94% (-5.9%)
uhci-hcd	Copy	585.84Kbps	4.92%	578.95Kbps (-1.1%)	7.01% (+42%)

Figure 7. Performance of unmodified network and USB drivers and drivers in our security architecture.

inferred an invariant for each function pointer that crossed the u-driver/k-driver boundary in all four drivers (most of them of the form `fptr` = ORIG(`fptr`)). These invariants will detect unauthorized function pointer modifications within the u-driver and prevent control hijacking attacks.

• ***Non-control data attacks.*** Sensitive scalar values, such as I/O memory addresses, interrupt masks and configuration parameters, that are marshaled between the u-driver and the k-driver can be maliciously modified by a compromised u-driver. For example, scalars that represent I/O memory address ranges, *e.g.,* `dev->base_addr`, which represents the base address of the driver's I/O memory region, are set by the kernel when the driver is loaded. These values must not normally be modified by the driver because it will allow the driver write access to memory regions that it does not own, *e.g.,* to the I/O memory regions of other devices. Yet, a compromised u-driver can maliciously modify such sensitive scalar values; when these values are marshaled into the k-driver, they will maliciously update kernel data as well.

We simulated such an attack by modifying several non-control data values. For instance, we modified the value of `rtl8139_intr_mask` within the u-driver. This variable represents a 16-bit mask; copying this value unchecked into the k-driver will allow the driver to write to an undesired I/O port. We were able to successfully detect this attack using invariants that expressed relationships between the value of the scalar before an upcall and the value after the upcall, *e.g.,* `rtl8139_intr_mask` = ORIG(`rtl8139_intr_mask`). We also implemented an attack that modified a kernel linked list (`dev->mc_list`) within the u-driver, and were successfully able to detect this attack using linked list invariants.

• ***False positives and negatives.*** It is well-known that Daikon can possibly infer *serendipitous* invariants, *i.e.,* those that are overly specific to the training workload. To determine whether such invariants result in false positives during enforcement, we ran the drivers with several benign test workloads that called functions in the u-driver (the training workload used to infer invariants was the same as the one in Section 5.2). *We did not observe any false positives during this experiment.* While it is unclear whether the same result will hold for other drivers as well, we note that in a real deployment, false positives could be eliminated by manually inspecting and refining the invariants.

To evaluate false negatives, *i.e.,* cases where invariants fail to detect a compromised u-driver, we conducted fault-injection experiments using the 8139too and 8139cp drivers. (We could not conduct these experiments on the ens1371 and uhci-hcd drivers because of limitations of our prototype infrastructure.) We used an off-the-shelf fault injector [43] to inject 400 random faults in the u-driver of each microdriver. We measured the number of faults that propagated to the kernel (via RPC) and the number of these faults that were detected by our invariants. Note that our prototype currently lacks a recovery subsystem. Therefore, faults that propagate to the kernel crash the system, *i.e.,* the RPC monitor can *detect* data corruption, but cannot *prevent* or *recover* from a system crash. Our experimental methodology was therefore to inspect system logs following each system crash to determine whether the RPC monitor detected the crash.

Figure 8 presents the results of this study. As this figure shows, there were several cases in which the system did not crash and in which the faults

were contained within the u-driver (the #NoCrash and #UD columns, respectively). The remaining faults, which constituted the majority, propagated to the kernel, thereby showing the need for an RPC monitor to inspect kernel data structure updates initiated by the u-driver. As discussed above, we used system logs to determine whether the RPC monitor detected a crash. In several cases (shown in the #Clear column), we observed that the system log had been cleared following the crash. In these cases, we could not determine whether the RPC monitor would have detected the crash. Nevertheless, there were several cases in which we observed a crash for which we could inspect our logs to determine the effectiveness of invariants (shown in the #InLog column). The #Detect column shows the number of #InLog crashes that were detected by the RPC monitor. As these results indicate, the RPC monitor could detect 84% of the injected faults in the 8139too driver and 61% of the faults in the 8139cp driver. These results also show that the RPC monitor can effectively thwart a significant fraction of attacks enabled by a compromised u-driver.

5.3. Performance

We measured both the throughput and CPU utilization of the two network drivers and the USB driver using our QEMU testbed. While QEMU does not provide an accurate representation of performance on real hardware, it allows us to measure differences in performance. If the driver has lower performance, it will be reflected either as higher CPU utilization or low throughput. If neither changes, the performance on real hardware should be unchanged.

We measured throughput and CPU utilization of the network drivers using netperf [14]. We transmitted packets between our QEMU test environment and a client machine. The netperf tests used TCP receive and send buffer sizes of 87KB and 16KB, respectively. To test the USB driver, we copied a 140MB file into a USB disk. All our measurements are averaged over 10 runs, and are presented in Figure 7. As this Figure shows, our security architecture minimally impacts common-case performance (the minor speedups that we observed are within the margin of experimental error). This is because the code to transmit packets is in the k-driver; sending a packet does not involve any user/kernel transitions. For the sound driver, we compared the CPU utilization of both the original driver and the split driver as they played a 256-Kbps MP3; CPU utilization in both cases was zero.

However, uncommon functionality, such as device initialization, shutdown and configuration, resulted in several user/kernel transitions and took almost thrice as long. During the training phase of the experiments reported in Section 5.2, we used several benign workloads that exercised such functionality implemented in the u-driver of each device driver. Figure 9 presents the number of user/kernel transitions and the amount of data transferred in upcalls and downcalls during this training phase.

6. Related Work

Hardware-based isolation techniques, such as Nooks [35] and Mondrix [41], rely on memory protection at the page level (Nooks) or with fine-grained segments (Mondrix) to isolate device driver failures. There are two main differences between Nooks/Mondrix and our work. First, both Nooks and Mondrix execute device drivers in kernel mode. Second, they do not enforce integrity specifications on kernel data structure updates, because doing so is likely to impose significant performance overheads. The consequence of these differences is that while Nooks and Mondrix can improve reliability with benign but vulnerable drivers, they cannot protect against compromised drivers that attempt to subvert the kernel. For example, they cannot protect against buffer overflow exploits that maliciously modify kernel data structures.

Virtual machine-based techniques isolate device drivers by running a set of device drivers within their own virtual machine *e.g.,* [16, 19, 25]. In principle, this approach offers all the benefits of our architecture. However, in practice, there are two key difficulties. First, these techniques require the use of a VMM. Although VMMs have seen wide deployment for server-class systems, they are still not in wide use on personal desktops—platforms that support a wide variety of devices and hence, drivers. Second, VM-based techniques must provide a front-end driver within the guest VM that communicates requests between the device driver (running on a separate VM) and I/O requests from

Driver	Faults	NoCrash	UD	Clear	InLog	Detect
8139too	400	49	26	212	113	95 (84%)
8139cp	400	134	14	147	105	64 (61%)

Figure 8. Results from fault injection.

Driver	KBytes sent/received	# upcalls	# downcalls
8139too	813	200	160
8139cp	9.5	206	124
ens1371	15.6	395	777
uhci-hcd	25.1	36	126

Figure 9. Data movement in up and downcalls.

applications in the guest. Although such front-ends can be developed easily for standard classes of drivers (*e.g.,* network, sound, SCSI), developing front-ends for other one-of-a-kind drivers, *e.g.,* those that support non-standard `ioctl` interfaces, is cumbersome. Thus, while the VMM-based approach has several benefits, it is not applicable to a wide variety of devices and drivers.

SafeDrive [43] and XFI [17] are ***language-based mechanisms*** to isolate device drivers. SafeDrive is an adaptation of CCured [32] to protect against type-safety violations in device drivers. While SafeDrive offers low-performance overhead and compatibility, device drivers protected with SafeDrive still execute with kernel privilege. Moreover, SafeDrive only protects against type-safety violations; in contrast, our RPC monitor can protect against violations that transcend type-safety, such as requests by the u-driver to allocate large amounts of memory, which may lead to memory exhaustion. Similarly XFI ensures control-flow integrity for device drivers. Our security architecture allows the use of any user-space security mechanism to be applied to a large fraction of device driver code without investing the effort needed to adapt these mechanisms to kernel code.

Microkernels [26, 40, 42] provide new operating system abstractions that allow device drivers to execute in user mode. Nexus [40] is one such microkernel OS that enforces domain-specific rules on driver/device communication using a kernel-resident reference monitor. Supplied with appropriate rules, Nexus can prevent attacks at the driver/device interface that our architecture cannot prevent. The effort required to port Linux drivers to Nexus is also comparable to the effort required to port them to our architecture. However, Nexus is a microkernel; consequently, its security mechanisms are largely inapplicable to commodity operating systems, which are structured as macrokernels. In addition, Nexus reports high CPU utilization for CPU-intensive workloads and lower throughputs with a network driver. In contrast, our architecture imposes minimal overheads in the common-case because performance-critical code executes in kernel mode.

User-mode driver frameworks [10, 15, 24, 28, 37] also attempt to execute drivers without kernel privilege. However, these techniques either offer poor performance [3, 38] because they transmit large amounts of data frequently across the user/kernel boundary, or are incompatible with commodity operating systems, often requiring complete rewrites of drivers and modifications to the kernel [10, 24, 28, 37].

Program partitioning techniques have previously been for privilege separation [6] and to create secure web applications [9]. In contrast to prior work, our architecture applies partitioning to device driver code, which enables user-mode drivers and the use of user-mode tools such as Daikon to infer invariants. Prior work has also investigated the use of program invariants for bug detection [22], data structure repair [13], rootkit detection [5] and improving the security of web applications [11]. Again, our contribution is to apply these techniques to improve the security of device drivers.

7 Summary

Device drivers bloat the size of the TCB in commodity operating systems because kernel data is isolated poorly from vulnerabilities in driver code. The security architecture proposed in this paper offers a practical way to better isolate kernel data from device drivers without sacrificing performance and in a manner that is compatible with commodity operating systems.

References

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Weigert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3), 2006.

[2] AMD. AMD I/O virtualization technology (IOMMU) specification, Feb 2007.

[3] Francois Armand. Give a process to your drivers! In *EurOpen Autumn 1991*, 1991.

[4] AusCERT. Esb-2006.0896: Intel network adapter driver local privilege escalation, 2006. `http://www.vul.org.au/render.html?it=7058`.

[5] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC*, 2008.

[6] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security*, 2004.

[7] Yuriy Bulygin. Remote and local exploitation of network drivers. In *Blackhat-USA*, 2007.

[8] J. Cache, H. D. Moore, and Skape. Exploiting 802.11 wireless driver vulnerabilities on windows. `http://www.uninformed.org/?v=6\&a=2\&t=sumry`.

[9] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *ACM SOSP*, 2007.

[10] P. Chubb. Get more device drivers out of the kernel! In *Ottawa Linux Symp.*, pages 149–161, 2004.

[11] M. Cova, D. Balzarotti, V. Felmgester, and G. Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *RAID*, 2007.

[12] Coverity. Anaylsis of the Linux kernel, 2004. Available at `http://www.coverity.com`.

[13] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA*, 2006.

[14] Information Networks Division. Netperf: A network performance benchmark. `http://www.netperf.org`.

[15] J. Elson. FUSD: A Linux framework for user-space devices, 2004. User manual for FUSD 1.0.

[16] Ú. Erlingsson, T. Roeder, and T. Wobber. Virtual environments for unreliable extensions. Technical Report MSR-TR-05-82, Microsoft Research, 2005.

[17] Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George Necula. XFI: Software guards for system address spaces. In *OSDI*, 2006.

[18] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamanat, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3), 2007.

[19] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.

[20] V. Ganapathy, M. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and

implementation of microdrivers. In *ACM ASPLOS*, 2008.

[21] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *NDSS*, 2004.

[22] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

[23] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, 2004.

[24] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Jour. Comp. Sci. and Tech.*, 20(5), 2005.

[25] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI*, 2004.

[26] J. Liedtke. On μ-kernel construction. In *ACM SOSP*, 1995.

[27] D. Maynor. Os X kernel-mode exploitation in a weekend. `http://uninformed.org/index.cgi?v=8\&a=4`.

[28] Microsoft. Architecture of the user-mode driver framework, 2006.

[29] Microsoft Inc. Microsoft interface definition language.

[30] Linux device driver vulnerabilities from the MITRE database. CVEs 2007-4571, 2007-05, 2007-4308, 2008-0007, 2005-0504, 2006-2935, 2006-2936, 2005-3180, 2004-1017, 2007-4997, 2006-1368.

[31] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate languages and tools for analysis and transformation. In *Compiler Construction*, 2002.

[32] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Symposium Principles of Programming Languages*, 2002.

[33] N. L. Petroni, T. Fraser, A. Walters, and W. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *USENIX Security Symposium*, 2006.

[34] N. L. Petroni and M. W. Hicks. Automated detection of persistent kernel control-flow attacks. In *ACM CCS*, 2007.

[35] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1), 2005.

[36] L. Tan, E. M. Chan, R. Farivar, N. Mallick, J. C. Carlyle, F. M. David, and R. C. Campbell. iKernel: Isolating buggy and malicious device drivers using hardware virtualization support. In *IEEE Intl. Symp. on Dependable, Autonomic and Secure Computing*, 2007.

[37] L. Torvalds. UIO: Linux patch for user-mode I/O, 2007.

[38] K. T. Van Maren. The Fluke device driver framework. Master's thesis, Dept. of Computer Science, Univ. of Utah, 1999.

[39] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM CCS*, 2002.

[40] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gun Sirer, and Fred B. Schneider. Device driver safety through a reference validation mechanism. In *OSDI*, 2008.

[41] E. Witchel, J. Rhee, and K. Asanovic. Mondrix: Memory isolation for linux. In *ACM SOSP*, 2005.

[42] M. Young, M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, and A. Tevanian. Mach: A new kernel foundation for UNIX development. In *Summer USENIX Conference*, 1986.

[43] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and

```
struct cp_private {
       char * IOMem regs;
       struct cp_desc * Array(64) rx_ring;
       ...
}
struct net_device {
       void * Opaque(struct cp_private) priv;
       struct net_device * Sentinel(next≠0) next;
       ...
}
```

Figure 10. Structure definition from the 8139cp driver, illustrating the IOMem, Array, Opaque and Sentinel annotations.

E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI*, 2006.

A Examples of annotations used by DriverSlicer

Figure 10 presents four kinds of annotations used by DriverSlicer using an example from the 8139cp network driver. These annotations are applied to structure definitions and formal parameters of functions. DriverSlicer supports eight kinds of annotations in total; these are described in detail in prior work on Microdrivers [20].

- The IOMem annotation, applied to the `regs` field of the `cp_private` structure, informs DriverSlicer that `regs` points to device I/O memory. Pointers to I/O memory must be handled differently than pointers to kernel/device memory by the object tracker. DriverSlicer uses this annotation to generate appropriate marshaling code for pointers to I/O memory pointers.
- The Array annotation informs DriverSlicer that the pointer-valued `rx_ring` field of the `cp_private` structure points to an array of 64 `cp_desc` objects. DriverSlicer uses this annotation to marshal the entire array of objects instead of simply marshaling the object instance that `rx_ring` points to.
- The Opaque annotation, which is applied to the `void *` pointers, helps DriverSlicer identify the type of the object that the `priv` of the `net_device` data structure points to in the 8139cp driver. DriverSlicer uses this annotation to generate marshaling code for the `cp_private` data structure when it marshals the `priv` field of the `net_device` structure.
- The Sentinel annotation is applied to recursive data structures such as the `next` field of a linked list. The annotation shown in Figure 10 also contains the predicate used to end linked list traversal (checking that the `next` field is non-NULL). DriverSlicer uses this annotation to generate code to marshal the entire linked list of elements, using `next≠0` as the condition to terminate traversal.

B Marshaling protocol

Figure 11 shows an example of the marshaling protocol augmented to check data structure invariants. As this Figure shows, the marshaling protocol is augmented to record the original values of variables in the vault table; this is required to enforce invariants of the form `var=Orig(var)`. The unmarshaling protocol (implemented in `checkinv_rtl8139_init_one`) copies values received from the u-driver into the vault area and verifies that invariants are satisfied. If so, kernel data structures are updated with values from the vault using the `copy_from_vault` function, which copies the value of a data structure/field from the vault area to the kernel.

```
// RPC stub in the k-driver containing code for invariant enforcement.
int rtl8139_init_one (struct pci_dev *dev, ...)  {
       void *mbuf, *dmbuf;
       ...
       // Marshal values into mbuf.
       marshal (mbuf, "pdev->hdr_type", pdev->hdr_type);
       add_vault_tab ("pdev->hdr_type", &pdev->hdr_type, ORIGVAL);
       marshal (mbuf, "pdev->devfn", pdev->devfn);
       add_vault_tab ("pdev->devfn", &pdev->devfn, ORIGVAL);
       ...
       // Call the u-driver with marshaled data.
       dmbuf = do_upcall ("rtl8139_init_one", mbuf);
       // Demarshaling:  copy from vault.
       if (checkinv_rtl8139_init_one(dmbuf)) {
              copy_from_vault ("pdev->hdr_type", &pdev->hdr_type);
              copy_from_vault ("pdev->devfn", &pdev->devfn);
              ...
       }
}

// RPC monitor function that unmarshals data into the vault
// and checks invariants
int checkinv_rtl8139_init_one(void *unmarshbuf) {
       void *ptr;
       ptr = unmarsh_to_vault (unmarshbuf, "pdev->hdr_type");
       add_vault_tab ("pdev->hdr_type", ptr, MODIFVAL);
       ptr = unmarsh_to_vault (unmarshbuf, "pdev->devfn");
       add_vault_tab ("pdev->devfn", ptr, MODIFVAL);
       ...
       if (check_invariants()) return 1;
       else { //trigger recovery }
}
```

Figure 11. Code snippet from the 8139too microdriver showing marshaling protocol modified to check for data structure invariants.