# DriverGuard: A Fine-Grained Protection on I/O Flows

Yueqiang Cheng, Xuhua Ding, and Robert H. Deng

School of Information Systems,
Singapore Management University
{yqcheng.2008,xhding,robertdeng}@smu.edu.sg

**Abstract.** Most commodity peripheral devices and their drivers are geared to achieve high performance with security functions being opted out. The absence of security measures invites attacks on the I/O data and consequently threats those applications feeding on them, such as biometric authentication. In this paper, we present the design and implementation of DriverGuard, a hypervisor based protection mechanism which dynamically shields I/O flows such that I/O data are not exposed to the malicious kernel. Our design leverages a composite of cryptographic and virtualization techniques to achieve fine-grained protection. DriverGuard is lightweight as it only needs to protect around 2% of the driver code's execution. We have tested DriverGuard with three input devices and two output devices. The experiments show that DriverGuard induces negligible overhead to the applications.

## 1 Introduction

Device drivers are often blamed as the main cause for system failures and security breaches, mainly due to their enormous code size and the much higher bug rate than other kernel code [5]. Various schemes have been proposed to improve system reliability by isolating driver errors (e.g., Nook [25] and SafeDrive [32]), or to defend against device I/O misuses for illegal memory accesses (e.g., BitVisor [24] and schemes in [29]). In this paper, we study the other side of the coin: how to protect driver operations, which is motivated by attacks on sensitive I/O data, such as password keystrokes, fingerprint templates, sensor readings and confidential print-outs.

As compared to applications and other kernel components such as system call functions, driver operations or I/O flows are more attractive to malware targeted at sensitive data for the following reasons. Firstly, there exist more loopholes to exploit due to the complexity of I/O mechanisms and the abundance of driver bugs. For instance, IRQ number sharing allows a malicious interrupt handler to easily access another handler's data. Another reason is that more drivers handle raw data generated by or for hardware. In many applications, raw data are more favorable to attackers as compared to derived data. For instance, a user's fingerprint template is life-long valid whereas a secret key derived from the fingerprint template may remain valid only for a few hours. Furthermore,

V. Atluri and C. Diaz (Eds.): ESORICS 2011, LNCS 6879, pp. 227–244, 2011.

most commodity I/O devices nowadays are not encryption capable and raw data are exposed to any code accessing them.

We aim to protect application-device data flows against the untrusted kernel throughout the entire I/O lifecycle. In particular, we focus on those devices that render raw data, e.g., sound cards and printers, or generate raw data for applications, e.g., seismic sensors and fingerprint scanners. We are less concerned with disks and network adaptors, because these devices deal with derived data from applications. Therefore, a simple solution to protect disk I/O and network I/O is to encrypt the application data before and after I/O operations. In this work, we present DriverGuard, a holistic and compact I/O protection system making use of a combination of cryptographic and virtualization techniques. We have implemented DriverGuard with slight changes on the drivers and the hypervisor. Our experiments with several I/O devices demonstrate that DriverGuard imposes little overhead to the system and causes unnoticeable delays to user applications. DriverGuard is complementary to many user space protection schemes such as BIND [23], Overshadow [4], PRECIP [26] and Terra [8]. A composition of DriverGuard and a user-space protection scheme can protect the whole lifecycle of data processing.

Our work is remarkably different from secure I/O [16, 24, 29] and driver code security. Secure I/O copes with those attacks misusing the I/O mechanism (especially DMA operations) for illegal memory accesses. Driver code security tackles software attacks, such as return-address attacks [3] and code injection attacks [11], which gain the root privilege by attacking drivers. Although these attacks do not necessarily target on I/O data, they are one of the threats considered in our study.

Organization. The next section discusses the related work. We present the design of DriverGuard in Section 3, and the implementation details in Section 4. Section 5 shows the evaluation of DriverGuard through our experiments. We conclude the paper in Section 6.

## 2 Related Work

**Data Flow Security.** BIND [23] binds data and code and uses cryptographic techniques to guarantee the integrity of data. However BIND is limited to derived data. TERRA [8] builds an application specific domain with a trusted path from the hypervisor to an application specific kernel, then to the application. Trusted path schemes [10, 31] focus on providing a trusted GUI to user, protecting user inputs to the intended applications. These two schemes only address security issues at the driver-applications interface, whereas the battlefield of DriverGuard is the entire I/O path. Bumpy [14] proposes to protect user keyboard inputs by building a trust environment using Flicker [13]. It requires an encryption-capable keyboard and therefore is not applicable to generic devices.

**Secure I/O.** Most existing results on secure I/O deal with I/O misuses where an adversary attacks the system by exploiting the flexibility of I/O operations, especially DMA. The schemes described below serve a different purpose from ours

and are not applicable to I/O flow protection. *d*Anubis [16] is a system monitoring and analyzing device drivers using virtual machine introspection techniques. BitVisor [24] is a hypervisor dedicated to I/O management and supports only one VM. It uses a parapass-through mechanism whereby most I/O operations from the guest pass through the hypervisor with some of them being intercepted. The interception allows the hypervisor to protect itself and to perform security functions. DMA security receives more attention since DMA-capable devices can access memory without involving the CPU. In [30], a software based approach is proposed whereby the hypervisor validates all DMA descriptors before they are issued to the device. This approach is then extended to a hardware-based approach by utilizing I/O memory management units (IOMMUs) in [29], which deals with the bad-address fault, the invalid-use fault and the bad-device fault.

**Hypervisor-Based System Security.** Our scheme is also relevant to hypervisor-based security systems. SecVisor [21] utilizes a small hypervisor to prevent kernel code injection. Overshadow [4] protects device memory from untrusted software in user space. In HyperShield [17], a thin layer hypervisor is plugged into a running OS without rebooting, so that it prevents illegal code execution. Lares [19] is an architecture which establishes a secure environment for security tools to actively monitor a guest domain. HookSafe [28] uses a hypervisor to prevent kernel hooks from being hijacked. TrustVisor [12] is a tiny trusted computing base which protects code and data integrity by leveraging hardware features. Although these schemes take the rootkit as the adversary, they only provide a generic protection, not geared for I/O protection. Therefore, an attack on the I/O path might not be considered as adversarial in these schemes.

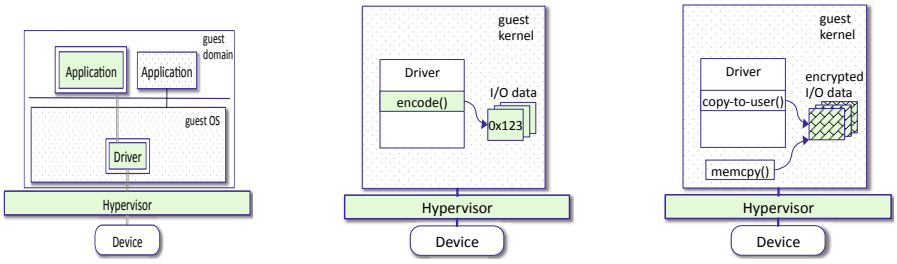
## 3 Design of DriverGuard

The objective of DriverGuard is to protect the confidentiality of a driver's I/O data from being attacked by a corrupted kernel. We remark that since I/O operations are heavily used, the low-overhead requirement is vital for the practicality of DriverGuard.

### 3.1 Trust Model

The bedrock of our scheme is a trusted hypervisor beneath the guest domain. Although there are known rootkit attacks on the hypervisor, we suppose that secure boot-up and load-time attestation with the support of TPM [9] can ensure the hypervisor's security in the bootstrapping phase. The hypervisor then loads itself into an isolated memory region with the highest privilege so as to block any illegal accesses from a guest domain [1]. Other techniques such as HyperSafe [27] can also be applied to ensure the hypervisor's security. We assume the presence of IOMMU protecting the hypervisor's memory territory from being invaded by DMA devices under the adversary's control.

We do not trust the guest kernel since it is vulnerable to various attacks such as return-oriented attacks [22, 2, 3] and code injection [11, 21]. In our attack




(a) An illustration of the trust model. The colored boxes represent trusted components.

(b) A PCB accesses the I/O data in plaintext.

(c) A non-PCB, e.g., the memcpy() function, accesses the encrypted I/O data.

**Fig. 1.** Illustrations of DriverGuard's trust model and the concept of PCB

model, we consider the subverted guest kernel as the adversary whose goal is to acquire I/O data transferred by a driver. The malicious kernel can read or write any memory region and any I/O port within the guest domain, but can not subvert the hypervisor. The driver is treated as a benign executable which actively demands I/O protection. We consider the scenarios that I/O requests are issued from an application well-protected in user space. Existing schemes such as Overshadow [4] can safeguard the application data against attacks from other applications and the guest kernel. Figure 1(a) depicts the trust model used in this paper. Note that hardware attacks such as bus sniffing are not investigated in this paper. Neither is the denial-of-service attack whereby the adversary attempts to sabotage I/O flows, instead of compromising the data.

*Attacks.* The attacks from the corrupted kernel are classified into two categories. The malicious kernel can attack the device I/O controls such that the device receives a manipulated I/O command and reads/writes data from/to regions to the adversary's advantage. This type of attacks requires the kernel to tamper with I/O control related regions such as I/O ports and DMA descriptors to inject commands or to modify the control region locations. The other type of attacks are directly targeted on the I/O data. The kernel can attempt to access data regions such as I/O data ports, MMIO data buffers, DMA buffers, the driver's memory regions and the application's data buffer. Alternatively, the kernel can attack the driver's execution flow or runtime states, such as the stack.

### 3.2 Design Rationale

A straightforward approach is that the hypervisor arbitrates whether a control flow can access the I/O data. It requires the hypervisor to introspect driver operations, which is difficult to implement due to the semantic gap (e.g., lack of details of driver operations) between the hypervisor and the driver. Considering the complexity of I/O operations, the workload on the hypervisor will significantly downgrade the whole system performance.

Isolation is a widely used method to protect program executions. To apply isolation on driver protection, one may propose location isolation or execution isolation. Location isolation is to place drivers and the kernel's I/O subsystem a separated domain, e.g., an I/O domain and Domain 0, or the hypervisor's space, such that malware in the guest kernel can not attack them directly. These approaches are efficient in terms of I/O performance. Nonetheless, the resulting protection is weak because the TCB size is increased significantly due to the drivers and the I/O subsystem.

In execution isolation, the drivers still reside in the untrusted guest kernel while their executions are in a secure environment established by the hypervisor, similar to TrustVisor [12] and Overshadow [4]. The generic execution isolation is not applicable for driver protection, because I/O operations are featured with frequent hardware interrupts and intensive driver-kernel interactions. Note that if the I/O subsystem is also enclosed in execution isolation, it suffers from the same drawback as in the location isolation approach.

We adopt the idea of execution isolation, however, at a micro-level. It is well-known that most of the driver code is for housekeeping purposes, such as error handling, resource allocation and cleaning up [7], with only a small portion dealing with I/O data transferring. We further observe that among the code for data transferring, only a few code blocks, e.g., an encoding function, need to process the I/O data while the majority of them just move the data from one memory location to another without necessarily knowing the content. Based on these observations, we design DriverGuard as a fine-grained I/O protection mechanism which distinguishes those security-sensitive driver code from the rest. The hypervisor only needs to protect the executions of security-sensitive code blocks (around 1% of the driver code according to our experiments) because of the aforementioned driver code characteristics. The fine-grained protection is coupled with a hypervisor-based access control mechanism. Different from hypervisor introspection, access control does not impose comprehensive logics on the hypervisor. Hence, the overall cost of DriverGuard is remarkably low. Its performance is on par with the location isolation solution, however, the security strength is much stronger.

### 3.3 Overview

By and large, DriverGuard is constructed using three lightweight protection techniques as the building blocks: cryptography, access control and runtime protection. We use cryptographic techniques to protect all I/O data without interfering with most of the driver and the kernel executions. For regions holding data which cannot be protected by encryption, we resort to the hypervisor to enforce access control. These plaintext data can only be accessed by a few designated driver code blocks, whose executions are safeguarded by our runtime protection mechanism. We refer to these code blocks as *privileged code blocks* (PCBs) in the rest of the paper. By protecting the execution of PCBs, we successfully ensure the whole I/O data security with minimal overhead since PCBs only constitute a tiny fraction of the driver code.

*Privileged Code Block.* We consider three types of PCBs in a driver. One type of PCBs is the driver code blocks which make computation on the I/O data, e.g., an encoding function. We call them *computation-PCBs*. The second type of PCBs is the driver code blocks which issue I/O commands and parameters to a device. We call them *command-PCBs*. This type of code is security sensitive because their executions determine the locations of plaintext I/O data. The third type of PCBs is the driver code blocks which initialize the driver's cryptographic key. Each driver generates its own key in the driver initialization step, such as in `module_init`. We call them *key-PCBs*.

The critical property of PCBs is that they must access critical information in plaintext. It is desirable for a PCB's size to be small without making any function call to non-PCBs, because non-PCBs are unprotected and may compromise security. When a driver is loaded, the hypervisor is notified with the locations of the driver PCBs and sets them as read-only in order to protect the code integrity. A PCB is delimited by two hypercalls to request for and relinquish runtime protection. The runtime protection of a PCB means that when the PCB is scheduled off from the CPU, the hypervisor seals its context and cordons off all accesses to the data and states until it resumes its CPU control.

A high level view of DriverGuard's protection mechanism is as follows. A driver initially generates a secret encryption key in its key-PCB. The I/O data remains encrypted by this key whenever the guest domain's virtual CPU is not controlled by the driver's PCB. Within a computation-PCB, it may perform encryption, decryption or encoding functions on the data. If a computation-PCB's decrypts the data, it either re-encrypts it or requests the hypervisor's protection when it ends. For I/O controls, the hypervisor ensures that the device's I/O ports or MMIO regions can only be accessed by the driver's command-PCBs. With the assistance of the hypervisor, the command-PCB checks whether the I/O buffer address in use is legitimate before issuing the command to the hardware. Figures 1(b) and 1(c) illustrate the difference between a PCB's and a non-PCB's I/O data accesses. Next, we explain the design of three building blocks and leave the discussion of their integration in Section 4, since it involves the details of I/O operations.

### 3.4 Access Control over Critical Regions

Since we do not rely on encryption-capable devices, encryption is not applicable for data accessed by the hardware. To cordon off illicit accesses to those data, we utilize the hypervisor's access control mechanism. In general, there are two types of regions for access control: the data regions and the control regions. The former holds the raw data generated for or by the hardware while the latter holds the I/O parameters for the hardware. According to their address spaces, these regions are classified into memory regions and I/O ports, for which we apply different access control methods by leveraging the hardware features and the virtualization techniques available in the platform.

To intercept accesses to a protected memory region, DriverGuard sets the attribute bits in the corresponding page table entries (PTEs), while to intercept

accesses to an I/O port, it clears the corresponding IOPL bits and sets up the I/O bitmap. We use *checkpoints*[1] in the rest of the paper to refer to both the IOPL bits and the PTEs marked by the hypervisor for the purpose of access interception. Although the aforementioned protection techniques are used in many existing schemes, e.g., [4, 19], we are confronted with two new problems. First, given a memory buffer address, the hypervisor must make sure that the kernel can not bypass the checkpoint to access the region, which is challenging for memory regions allocated by the kernel. Secondly, the hypervisor must ensure that the sensitive I/O data is indeed placed in the region *with* a checkpoint. The first problem demands a careful page table walk checking while the second demands the I/O control integrity checking. We will present our solutions to both problems in the next section.

### 3.5 Cryptographic Components

We introduce to the guest kernel a symmetric-key encryption function and a decryption function, both of which can be called by any code. However, any write access to these function's code is denied by the hypervisor. We also add a key generation function to the driver as a PCB. The security of the I/O data relies on the secrecy of the driver's key, rather than the secrecy of the decryption function, which complies with the famous Kerckhoffs principle. The driver's secret key is securely generated based on a secret random seed supplied by the hypervisor. The secret key is securely stored in a kernel space buffer priorly appointed by the driver and can only be accessed by the driver's PCBs escorted by the hypervisor. This prevents any unauthorized code from decrypting the driver's data, even though the decryption function can be called arbitrarily.

### 3.6 PCB Execution Escorting

The third building block in DriverGuard is the runtime protection mechanism that prevents a PCB's execution from deviating its expected behavior. The protection is requested at the PCB's entry and is relinquished at the exit via hypercalls. The hypervisor agrees to admit a control flow into the escorting only when the request is issued from the driver's PCB, and agrees to discharge a flow from escorting only when the request is issued from the PCB presently under escorting.

The PCB under the escorting is granted to access the critical data such as the driver's secret key and the I/O data, or to issue I/O commands. In our design, the hypervisor lifts the checkpoints on those regions accessed by the PCB, and restores them at the exit of escorting. Therefore, no duplicated exceptions or page faults will be raised despite that the PCB may access the same region multiple times within an escorted execution. An escorted PCB can be scheduled off from the CPU for various reasons. In that case, the hypervisor intercepts this event and restores all checkpoints. Meanwhile, it also securely saves the driver's

[1] Our definition of *checkpoint* has no relation with the *checkpoint* for rollback in distributed systems.

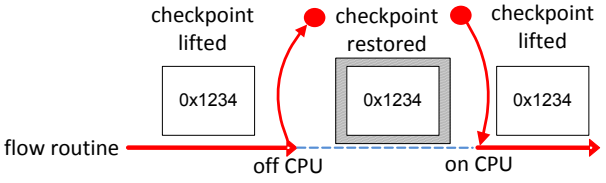



**Fig. 2.** An illustration of runtime protection, where $0x1234$ is an exemplary memory address with a PTE checkpoint

runtime stack and sets up a breakpoint for the PCB's upcoming CPU occupation. As a result, other code's accesses to the protected regions are denied. Figure 2 depicts a scenario of escorting. There are two methods for a PCB to restore the protection on the data. A computation-PCB encrypts the data before it exits from protection whereas a command-PCB requests the hypervisor to block all accesses on the region. In the end, a hypercall is invoked by a PCB to relinquish escorting.

## 4 Implementation

We build DriverGuard on top of the Xen hypervisor to protect the drivers running in a Linux guest domain. We systematically examine every step in I/O operations, from the device discovery to the application's (or device's) data fetching. In order to adaptively protect the driver operations, the hypervisor needs to store certain context information about the driver. We start with driver context initialization since it is performed by the hypervisor during the guest domain bootstrapping.

### 4.1 Driver Context Initialization

The context information of drivers is securely stored in three types of tables in the hypervisor space. A *device table* specifies the management relation between a driver and a device by paring their identifiers. For every protected driver, the hypervisor maintains a *PCB table* and a *region table*. The former stores the entry and exit addresses of all PCBs of the driver while the latter specifies the memory regions and the I/O ports to protect. There are five types of regions in the region table: 1) the application buffer; 2) the memory buffer allocated by the driver for data processes; 3) the I/O data buffer such as DMA buffers; 4) the control regions, including the I/O ports or MMIO regions for device control, DMA descriptor queues, the transfer descriptor queues; and 5) the buffer holding the driver's secret key.

**Device Table Initialization.** When a guest kernel image is uncompressed, the hypervisor inserts a hook function to the kernel to inform the hypervisor about

the device-driver association via a hypercall. The hypervisor then initializes the device table accordingly. The hypervisor also sets the checkpoints for the kernel structure maintaining the device-driver association. Whenever a driver takes the ownership of a device, the hypervisor intercepts the event and updates the device table properly.

**PCB Table Initialization.** The hypervisor also scans the driver code to obtain the locations of PCBs. It records the addresses of escorting-entry hypercalls and the addresses for escorting-relinquish hypercalls, and puts these pairs into the PCB table. The driver's initial code are considered as intact and the scanned PCBs are therefore legitimate.

**Region Table Initialization.** The control regions used by a driver can either be the default ones chosen by the manufacturer/the kernel or set by the driver. In the first case, the hypervisor updates them when the driver is loaded as in the device discovery step. In the latter case, the driver informs the hypervisor via a hypercall about the protected regions or I/O ports.

### 4.2 Checkpoint Deployment

Given a memory region or an I/O port, the hypervisor sets up the corresponding checkpoint to intercept and examine risky accesses. The detailed deployment method is dependent on the virtualization environment. Due to the length limit, we focus on deployment for paravirtualization domains and do not elaborate the techniques in a hardware virtualization domain (HVM).

*Memory Region Checkpoint.* For a memory page $P$, the hypervisor walks through the page tables according to CR3 register to locate the PTE pointing to $P$. The hypervisor can set the attribute bits on the PTE to specify different access rights. To set a page "read-only", the _PAGE_RW bit is cleared; and to set a page "non-access", the _PAGE_PRESENT bit is cleared.

Because all protected regions are in kernel space, PTE updates are propagated into the kernel portion of all other page tables in order to maintain consistency. Note that in the paravirtualization setting, only the hypervisor updates page tables. Therefore, those checkpoints will not be removed by the guest kernel. The hypervisor also checks page tables to remove double mappings pointing to protected memory regions using the method in HyperSafe [27].

*I/O Port Checkpoint.* I/O ports do not belong to the memory address space. During the launch of a paravirtualization domain, the hypervisor clears the IOPL bits of EFLAGS of the guest's virtual CPU. Namely, it sets the I/O privilege level to 0, such that the hardware always checks the I/O bitmap for PIO instructions because the guest kernel runs in Ring 1. Then, the hypervisor sets the bits corresponding to the protected I/O ports such that a PIO instruction will cause a general protection exception.

### 4.3 PCB Execution Escorting

*PCB Admission.* A driver's PCB starts with the hypercall which takes as the parameter the buffer address it requests to access. To admit a PCB, the hypervisor checks whether the hypercall is issued from the instruction whose address is registered in the PCB table. If not, the hypervisor rejects the request.

For an admitted PCB, the hypervisor has two tasks. One is to set a *local* breakpoint at function `__switch_to`, which is the kernel function for a CPU context switch. The other task is for stack protection. The hypervisor allocates a dummy stack for the PCB. Therefore, an admitted PCB has two runtime stacks. A genuine stack is used for the PCB's execution while the dummy stack is used for untrusted code sharing the same execution flow due to interrupts. Figure 3 below describes the details of the PCB admission algorithm, where InEscorting is a flag bit indicating the current execution state.

**Admission Algorithm**

1) Fetch the EIP value stored at the top of the current guest kernel stack, which is the return address of the hypercall.
2) If EIP does not match any entry in the PCB table, return error.
3) If the address of requested buffer is legitimate, then
   a) set InEscorting to 1;
   b) set a breakpoint at the entry of *__switch_to* function.
   c) If the guest's kernel stack segment is not a dummy stack, then
      (i) allocate a dummy stack at the reserved space.
      (ii) save the machine addresses of the dummy stack and the present stack as $(MA'_{ss}, MA_{ss})$. Return 0.
   d) else, switch to the corresponding genuine stack. Return 0.
6) Return -1 as an error message for admission failure.

**Fig. 3.** Algorithm for PCB admission

*Escorting.* Once a PCB is admitted by the hypervisor, its execution is escorted and the checkpoints for the buffers it accesses are temporarily lifted. The essence of escorting is that the hypervisor intercedes whenever the PCB is scheduled off from the CPU, which takes place in two scenarios. One is that the PCB relinquishes the CPU and the other is due to hardware interrupts. Both cases open the door to attacks. We design a mechanism to intercept the CPU context switch and to use dummy stacks for untrusted control flows. The interception is via the interrupt handler and the exception handler as explained below.

**Interrupt.** To switch to a dummy stack, the hypervisor only replaces the content of the PTE for the present stack with the machine page number of the dummy stack allocated during admission. This change is transparent to any guest process, since the address in ESP register remains the same. Hence, the guest

kernel is not able to access the true stack while the subsequent execution can use the dummy stack without being affected. The algorithm for stack switching and checkpoint restore is shown in Figure 4.

**Interrupt Handler Algorithm**

(1) If $InEscorting = 0$, return.
(2) Restore the checkpoints that are removed during escorting.
(3) Switch to the dummy stack, by setting the PTE for the guest's stack base to point to $MA'_{ss}$.
(4) Set $InEscorting = 0$. Remove the breakpoint at `__switch_to` function.
(5) Set a local breakpoint at the instruction pointed by EIP. Save the address pair in EIP and ESP.
(6) Return and pass the control to the default interrupt handler.

**Fig. 4.** Interrupt handler algorithm for stack switching and checkpoint restore

**Debug Exception.** All breakpoints used by the hypervisor are *local* breakpoints. Therefore, they are triggered only for the present process. There are two types of local breakpoints used in DriverGuard. One is for the CPU context switch interception. For this breakpoint, the hypervisor exits from escorting and restores checkpoints, in a similar fashion to the interrupt handling.

The other type of breakpoints is to intercept the event of PCB resuming. For this type of breakpoint, the hypervisor enters into escorting only when both EIP and ESP values match the previously saved EIP and ESP pair. The hypervisor can distinguish these two types of debug exceptions easily by checking whether it is in escorting mode. The algorithm details are shown in Figure 5.

*PCB Exit.* To exit from the hypervisor escorting, the PCB issues another hypercall. The hypervisor checks if InEscorting is set. If not, it returns an error message; otherwise, it clears InEscorting flag. The PCB should also issue a hypercall to protect its data if the data are left in plaintext. The hypervisor sets no more breakpoints and will handle future interrupts and exceptions in the normal way.

### 4.4 Region Access Control

A risky access to a memory region with a checkpoint causes a page fault and an access to an I/O port with a checkpoint throws out a general protection exception. Therefore, we modify the hypervisor's page fault routine `do_page_fault` and the general protection exception handler `do_general_protection`. In the former, the hypervisor gets the address of the trapped instruction from EIP register and the address being checked from CR2 register, while in the exception handler, the I/O port number is enclosed in the instruction.

**Debug-handler Algorithm:** Breakpoint address stored in *EIP*, the stack address stored in *ESP*

/* Exit from Escorting */
(1) If $InEscorting = 1$ and *EIP* points to the entry of *__switch_to*, then
(a) execute step (2,3,4) of the IRQ-handler algorithm.
(c) Fetch *task_struct->thread ->ip*, which is the address of the next instruction for resuming the present flow. Denote it by $EIP_r$. Save $(EIP_r, ESP)$ tuple.
(d) set a local breakpoint on $EIP_r$ and return 0.
/* Enter into Escorting */
(2) If there exists a saved $(EIP', ESP')$ pair, s.t. $ESP' = ESP$ and $EIP' = EIP$, then
(a) remove the breakpoint at *EIP*;
(b) Restore to the genuine stack by replacing the stack PTE with $MA_{ss}$.
(c) set a local breakpoint at the entry of *__switch_to* function,
(d) Set $InEscorting = 1$, and return 0.
(3) Return -1 as an error message.

**Fig. 5.** Exception handler for escorting

If the access is granted by the hypervisor, the event will not be forwarded to the guest kernel. In that case, The legitimate flow continues to execute the intercepted instruction without being re-scheduled as the guest kernel does not observe this exception. For unauthorized accesses, the page fault or exception is passed to the guest kernel. DriverGuard is compatible with memory mapping for page sharing because the checkpoints are deployed at the PTEs. A buffer mapped to two addresses has two PTE checkpoints. In the following, we elaborate the details of region access control according to all types of regions except the control region.

**Application Buffer.** The application data buffer is the starting or ending point of an I/O flow. We use the system call interception applied in [18] to get the buffer address. The technique used in [18] is to replace the first byte of the system call handler with instruction *HLT*, which causes a protection exception and passes the control to the hypervisor.

**I/O Buffer.** The addresses of I/O buffers are obtained within an escorted command-PCB. Since the I/O buffer contains the data to/from the device, they are not protected by encryption. The hypervisor blocks all accesses not from an escorted PCB. For an input buffer containing the data from the device, the driver always encrypts the data before moving them to other locations, whereas for an output buffer the driver must decrypt the data after copying them to the output buffer.

**Driver Buffer.** Driver buffers are for the driver to temporarily hold data for processing. When the data in those buffers are encrypted, the hypervisor does not set up checkpoints for them. Only when the escorted PCB is temporarily scheduled off from the CPU, the hypervisor sets up the checkpoints against all

accesses as the data are in plaintext. In this case, the PCB notifies the hypervisor about the buffer address.

**Key Buffer.** The key buffer holds the secret key used by the driver. The hypervisor allows the key to be read only from the instructions from the encryption/decryption functions (i.e. key-PCBs) and is currently in escorting mode. Thus, other code can not access the secret key.

### 4.5 Device Control Protection

The hypervisor denies all write accesses to the region not from an escorted PCB, and maintains the consistency between the I/O buffer address specified in an I/O command and the buffer addresses requested by the device driver. This is because the kernel may manipulate the I/O command such that the device uses an unprotected I/O buffer for transferring. To defeat such attacks, the driver's command-PCB informs the hypervisor the locations of the I/O buffers in use, such as the DMA buffer and the DMA descriptor queue. The hypervisor inserts them in the region table and sets up the checkpoints accordingly. Therefore, it ensures that the I/O buffer in use is always protected.

## 5 Evaluation

We have implemented DriverGuard and run experiments on five char devices to evaluate the security and performance. We tested three input devices (a USB keyboard, a web camera and a fingerprint reader) and two output devices (a sound card and a printer). In principle, DriverGuard is applicable to network and disk I/O as well. Nonetheless, as argued earlier, this type of I/O can be protected using application level data encryption. Therefore, we do not experiment with them. To demonstrate the effectiveness of DriverGuard, we ran it against three kernel-level keyloggers [6, 15, 20]. None of the keyloggers is able to steal the keystroke information.

### 5.1 Usage of PCB

In our experiments, we manually identify all PCBs on the source code of device drivers and the drivers in the kernel's I/O subsystems, e.g., a host controller driver. It is straightforward to identify command-PCBs and key-PCBs, because key-PCBs are introduced by DriverGuard while command-PCBs are the code accessing port I/O, MMIO or structures used by devices (e.g., frame list of UHCI). Identifying computation-PCB requires the semantic knowledge of the code. We trace the I/O data to spot code segments computing on the I/O data. Note that code segments for copying or moving data are not PCBs.

Table 1 lists all the involved drivers used in our experiments and the number of PCBs in each of them. We found that a driver typically has only around ten PCBs and each PCB has approximately 15 lines of code without making function calls (except the encryption and decryption functions). The total PCB code only

**Table 1.** The number of PCBs and the average size for each driver used in our experiments. The drivers labeled with stars are those within the kernel's I/O subsystem. The PCB size includes the hypercalls and the calls to the encryption and decryption functions.

| **Driver** | **Size** (LOC) | **# of PCBs** | **Avg. PCB Size** (LOC) | **Device** |
|---|---|---|---|---|
| keyboard driver | 4964 | 11 | 17 | keyboard |
| HID* | 12771 | 13 | 10 | keyboard |
| UVC driver | 7838 | 7 | 11 | camera |
| EHCI* | 10011 | 6 | 15 | camera |
| HDA-Intel | 47825 | 8 | 6 | sound card |
| Sound-core* | 18722 | 5 | 4 | sound card |
| devio | 1628 | 7 | 12 | printer, fingerprint reader |
| UHCI* | 7600 | 5 | 14 | printer, fingerprint reader |

account for $1 \backsim 3\%$ of the driver code. The tiny size of PCB and its simple logic allow for high security assurance, as compared to protecting the execution of thousands of lines of driver code.

### 5.2 Performance Evaluation

We experiment DriverGuard on a PC with Intel(R) Core(TM)2 Duo CPU E7200 @2.53GHz, 2GB main memory, running Xen 4.0.0 and a PV guest domain with Linux kernel 2.6.31.13. DriverGuard adds only 1727 SLOC to the Xen hypervisor. Our performance evaluation includes a cost measurement of DriverGuard's component functions and a set of application tests with five devices. We remark that the I/O characteristic is favorable to our scheme as data generation/rendering devices are usually much slower than the CPU. Therefore, DriverGuard does not affect the driver performance since the device speed is the performance bottleneck.

We choose 128-bit RC4 as the encryption cipher in our implementation, because RC4's compact code is easier to protect and does not significantly expand the PCB size. We instrument the DriverGuard code to measure the CPU cycles consumed by its main components including the escort hypercalls, the interrupt handler `do_IRQ`, the debug handler `do_debug`, the page fault handler `do_page_fault` and the general protection exception handler `do_general_protection`. Note that the encryption cost comprises the overhead of loading the secret key which incurs one page fault and the hypervisor's checkpoint removal. The results are shown in Table 2.

For each device we have experimented with, we measure the overhead and evaluate whether DriverGuard causes significant delay to the driver operation and the applications. Table 3 shows all the measured results.

**Keyboard.** In our experiment, we measure the time cost of the interrupt handler which moves the data from the keyboard to the tty buffer. Although the

**Table 2.** Cost of DriverGuard components

| **Components** | `do_IRQ` | `do_debug` | `do_page_fault` | `do_general_protection` | *Encryption 1KB* |
|---|---|---|---|---|---|
| **CPU cycles** | 844 | 739 | 961 | 1813 | 23355 |

**Table 3.** Performance overhead of protected keyboard, camera, fingerprint, printer and sound card I/O

| | **keyboard code transfer** | **camera waiting** | **fingerprint collection** | **1 page print** | **sound card open** |
|---|---|---|---|---|---|
| without DriverGuard | 0.053ms | 33.24ms | 2.61s | 15.74s | 7.8$\mu s$ |
| with DriverGuard | 0.138ms | 33.38ms | 2.63s | 16.19s | 12.3$\mu s$ |
| percentage | 160.40% | 0.42% | 0.77% | 2.86% | 57.7% |

protected keyboard I/O is more than 2 times slower than the unprotected one, it does not affect the application because it is still negligible as compared to the speed of human keystrokes.

**Camera.** The web camera in our experiment is managed by the default Linux UVC driver. When the camera is opened by an application, it continuously collects video data and sends them to the application. The UVC driver's interrupt handler moves and decodes the data stream from the camera into a video frame, which resides in the driver's buffer mapped to the user space. The user application can directly use the frame data like normal user-space data.We measure the time overhead of the application's waiting time for getting new data, which is a key factor to the quality of the generated video stream. Although the interrupt handler in protection is much slower due to the encryption of four pages data, the drivers spends much more time in waiting for the camera's data generation. Thus, the cost of the interrupt handler does not cause the overall performance degradation. We have also tested video chatting using Empathy 2.30.2, which is a graphic instant messenger. We do not perceive delays in the experiments.

**Fingerprint-Reader.** Our fingerprint reader is the Upek Touchchip fingerprint sensor. In our evaluation experiment, we choose *Fingerprint GUI* [2] as the application which uses the default Linux driver *devio* to communicate with the fingerprint reader. When the fingerprint reader is active, the driver's interrupt handler continuously loads the collected fingerprint data into its buffers, which are then fetched by *Fingerprint GUI* by calling the *ioctl* function. In our experiments, we measure the whole I/O session of fingerprint collection.

**Printer.** The printer in our experiments is HP Officejet 7210 and the device driver in use is *devio*. The print-process opens the printer and issues *ioctl* to send data to the printer. After sending out the data, it waits for a signal sent

[2] `http://www.n-view.net/Appliance/fingerprint/index.php`

back by the printer to close the printer. In our experiments, we measure the turnaround time between the printer open and the printer close. Note that the relative overhead drops if more pages are printed out.

**Sound Card.** The sound card in our test is Intel Corporation 82801I (ICH9 Family) HD Audio and the driver in use is *HDA Intel*. We run the application *Totem* which plays MP3 files. Totem places its sound data into a user space buffer, which is mapped into the DMA buffer specified by the driver. When the music is playing, Totem directly sends data into mapped DMA region in user space, and issues *ioctl* to synchronize and update information. The hardware fetches the data from the DMA buffer directly without the driver's involvement. Hence, DriverGuard is only involved in protecting the control region so that the kernel can not change the location of the DMA buffer in use. There is no cost for DriverGuard during music playing, though the cost in opening the sound card is high.

## 6 Conclusion

We have proposed DriverGuard which is a hypervisor-based system protecting I/O flows between devices and applications, especially for devices generating data or rendering data. DriverGuard protects I/O device control, I/O data transfer and a driver's data processing, against attacks from an untrusted guest kernel. It is featured with fine granularity protection with strong security assurance and low overhead. It only introduces 1727 SLOC to the hypervisor and a few lines to the driver code. DriverGuard can work jointly with user-space data protection schemes to safeguard the entire data lifecycle.

The main drawback of our scheme is the need for manually discover PCBs from a driver, a process which requires the domain knowledge of the I/O data flow. In our future work, we will investigate techniques to automate PCB discovery. We will also consider extending our work to the multi-core platform.

**Acknowledgement.** We are grateful to Virgil Gligor and Adrian Perrig for their constructive suggestions. We also thank the anonymous reviewers for their helpful comments.

## References

1. Bhargava, R., Serebrin, B., Spadini, F., Manne, S.: Accelerating two-dimensional page walks for virtualized systems. In: ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 26–35. ACM, New York (2008)
2. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: Generalizing return-oriented programming to RISC. In: Syverson, P., Jha, S. (eds.) Proceedings of CCS 2008, pp. 27–38. ACM Press, New York (2008)
3. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Keromytis, A., Shmatikov, V. (eds.) Proceedings of CCS 2010, pp. 559–572. ACM Press, New York (2010)

4. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.K.: Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008), Seattle, WA, USA (March 2008)
5. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. In: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, SOSP 2001, pp. 73–88. ACM, New York (2001), http://doi.acm.org/10.1145/502034.502042
6. Gadgetweb.de: How to: Building your own kernel space keylogger (2010), http://www.gadgetweb.de/programming/39-how-to-building-your-own-kernel-space-keylogger.html
7. Ganapathy, V., Renzelmann, M.J., Balakrishnan, A., Swift, M.M., Jha, S.: The design and implementation of microdrivers. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pp. 168–178. ACM, New York (2008), http://doi.acm.org/10.1145/1346281.1346303
8. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: a virtual machine-based platform for trusted computing. In: Proceedings of the 9th ACM Symposium on Operating Systems Principles, pp. 193–206. ACM, New York (2003)
9. Trusted Computing Group: TPM main specification. Main Specification Version 1.2 rev. 85 (February 2005)
10. Langweg, H.: Building a trusted path for applications using cots components. In: In Proceedings of NATO RTO IST Panel Symposium on Adaptive Defence in Unclassified Networks (2004)
11. Lineberry, A.: Malicious code injection via /dev/mem. In: Black Hat (March 2009)
12. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: Trustvisor: Efficient tcb reduction and attestation. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP 2010, pp. 143–158. IEEE Computer Society, Washington, DC, USA (2010), http://dx.doi.org/10.1109/SP.2010.17
13. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An execution infrastructure for TCB minimization. In: Proceedings of the ACM European Conference in Computer Systems (EuroSys) (April 2008)
14. McCune, J.M., Perrig, A., Reiter, M.K.: Safe passage for passwords and other sensitive data. In: Proceedings of the Symposium on Network and Distributed Systems Security (NDSS) (February 2009)
15. Mercenary: Kernel based keylogger (2002), http://goo.gl/7qwmr
16. Neugschwandtner, M., Platzer, C., Comparetti, P.M., Bayer, U.: *d*anuis - dynamic device driver analysis based on virtual machine introspection. In: Proceedings of the 7th Detection of Intrusions and Malware & Vulnerability Assessment (2010)
17. Nomoto, T., Oyama, Y., Eiraku, H., Shingawa, T., Kato, K.: Using a hypervisor to migrate running operating systems to secure virtual machines. In: Proceedings of the 34th Annual IEEE Computer Software and Application Conference (2010)
18. Onoue, K., Oyama, Y., Yonezawa, A.: Control of system calls from outside of virtual machines. In: Proceedings of Symposium of Applied Computing (2008)
19. Payne, B.D., Carbone, M., Sharif, M., Lee, W.: Lares: An architecture for secure active monitoring using virtualization. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy, pp. 233–247. IEEE Computer Society, Washington, DC, USA (2008), http://portal.acm.org/citation.cfm?id=1397759.1398072

20. Phrack: Writing linux kernel keylogger (2002), http://www.phrack.org/issues.html?issue=59
21. Seshadri, A., Luk, M., Qu, N., Perrig, A.: Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007, pp. 335–350. ACM, New York (2007), http://doi.acm.org/10.1145/1294261.1294294
22. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: De Capitani di Vimercati, S., Syverson, P. (eds.) Proceedings of CCS 2007, pp. 552–561. ACM Press, New York (2007)
23. Shi, E., Perrig, A., Doorn, L.V.: Bind: A fine-grained attestation service for secure distributed systems. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 154–168 (2005)
24. Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y., Kato, K.: Bitvisor: a thin hypervisor for enforcing i/o device security. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2009, pp. 121–130. ACM, New York (2009), http://doi.acm.org/10.1145/1508293.1508311
25. Swift, M.M., Bershad, B.N., Levy, H.M.: Improving the reliability of commodity operating systems. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP 2003, pp. 207–222. ACM, New York (2003), http://doi.acm.org/10.1145/945445.945466
26. Wang, X., Li, Z., Li, N., Choi, J.Y.: PRECIP: Towards practical and retrofittable confidential information protection. In: Proceedings of NDSS (2008)
27. Wang, Z., Jiang, X.: Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In: Proceedings of IEEE Symposium on Security and Privacy (2010)
28. Wang, Z., Jiang, X., Cui, W., Ning, P.: Countering kernel rootkits with lightweight hook protection. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 545–554 (2009)
29. Willmann, P., Rixner, S., Cox, A.L.: Protection strategies for direct access to virtualized i/o devices. In: Proceedings of USENIX Annual Technical Conference (2008)
30. Willmann, P., Shafer, J., Carr, D., Menon, A., Rixner, S., Cox, A.L., Zwaenepoel, W.: Concurrent direct network access for virtual machine monitors. In: Proceedings of the 13th International Symposium on High Performance Computer Architecture (2007)
31. Ye, Z.E., Smith, S., Anthony, D.: Trusted paths for browsers. ACM Trans. Inf. Syst. Secur. 8(2), 153–186 (2005)
32. Zhou, F., Condit, J., Anderson, Z., Bagrak, I., Ennals, R., Harren, M., Necula, G., Brewer, E.: Safedrive: safe and recoverable extensions using language-based techniques. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI 2006, pp. 45–60. USENIX Association, Berkeley (2006), http://portal.acm.org/citation.cfm?id=1298455.1298461