Glider: A GPU Library Driver for Improved System Security

Technical Report 2014-11-14, Rice University

Ardalan Amiri Sani, Lin Zhong, Dan S. Wallach
Rice University

Abstract

Legacy device drivers implement both device resource management and isolation. This results in a large code base with a wide high-level interface making the driver vulnerable to security attacks. This is particularly problematic for increasingly popular accelerators like GPUs that have large, complex drivers. We solve this problem with library drivers*, a new driver architecture. A library driver implements resource management as an untrusted library in the application process address space, and implements isolation as a kernel module that is smaller and has a narrower lower-level interface (i.e., closer to hardware) than a legacy driver. We articulate a set of device and platform hardware properties that are required to retrofit a legacy driver into a library driver. To demonstrate the feasibility and superiority of library drivers, we present* Glider*, a library driver implementation for two GPUs of popular brands, Radeon and Intel. Glider reduces the TCB size and attack surface by about 35% and 84% respectively for a Radeon HD 6450 GPU and by about 38% and 90% respectively for an Intel Ivy Bridge GPU. Moreover, it incurs no performance cost. Indeed, Glider outperforms a legacy driver for applications requiring intensive interactions with the device driver, such as applications using the OpenGL immediate mode API.*

1. Introduction

Device drivers are the main sources of bugs in operating systems [29]. They are large, fast-changing, and developed by third parties. Since the drivers run in the kernel of modern monolithic operating systems and are fully shared by applications, their bugs are attractive targets for exploitation by attackers, and pose great risks to the security of the system.

This longstanding problem is particularly troubling for hardware accelerators such as GPUs because they tend to have large, complex device drivers. For example, GPU device drivers have tens of thousands of lines of code and have seen quite a few attacks recently [3, 8–10, 12]. This problem is increasingly critical as GPUs are used even by untrusted web applications (through the WebGL framework). Indeed, browser vendors are aware of the this security risk, disabling the WebGL framework in the presence of GPU drivers that they cannot trust [14]. Obviously, this is a rough solution and provides no guarantees even in the presence of other drivers.

Researchers have long attempted to protect the system from device drivers. For example, user space device drivers are one of the principles of microkernels [26]. Even for monolithic operating systems, there exist solutions that move the driver to user space [30], move it to a VM [42], or even sandbox it in-situ [59, 64]. Unfortunately, these solutions have yet to see any practical success, mainly due to their inferior performance.

In this work, we revisit this problem with a fresh insight: a large part of legacy drivers is devoted to device resource management. Our solution, inspired by library operating systems, such as Exokernel [27] and Drawbridge [51], is a *library driver* design, built on one fundamental principle: *untrusted resource management*.

The library driver design incorporates this principle in two steps. First, it separates device resource management code from resource isolation. In a library driver design, resource isolation is implemented in a trusted *device kernel* in the operating system kernel, and resource management is pushed out to the user space. Second, resource management is implemented as an untrusted library, i.e., a *device library*. That is, each application that intends to use the device loads and uses its own device library. Based on some scheduling policy, the device kernel exports the device hardware resources securely to applications, which manage and use the resources with their own device library.

The library driver design improves overall system security by reducing the size and attack surface of the Trusted Computing Base (TCB). With a legacy driver, the whole driver is part of the TCB. However, with a library driver, only the device kernel, which is smaller than a legacy driver, is part of the TCB. Moreover, compared to a legacy driver, the device kernel exposes a narrower lower-level interface to untrusted software, hence reducing the attack surface of the TCB. The security benefits of a library driver are two-fold: first, a library driver reduces the possibility of attacks on the operating system kernel through bugs in the driver. Second, it improves the isolation between applications using the device, as it reduces the amount of shared state between them. Importantly, a library driver improves the system security without hurting the performance. Indeed, a library driver can even outperform a legacy driver due to one fundamental reason: a library driver avoids the overhead of syscalls and user-kernel data copies since it is in the same address space and trust domain as the application. The performance improvement highly depends on the application; the more interactions there are between the application and the driver, the more significant the performance improvement will be.

Applying the principle of untrusted resource management to a device driver requires certain hardware properties on the device and platform. We articulate these properties into three requirements: *memory isolation primitives* such as an IOMMU, *innocuous device management interface*, and *attributable interrupts*. If a device and its platform meet these properties, then it is possible to retrofit the device resource management code in the form of an untrusted library. Every violation of these requirements, however, forces some of the resource management code to remain in the device kernel, resulting in weaker security guarantees.

We target library drivers mainly for accelerators, such as GPUs, for three reasons. First, they are an increasingly important subset of devices; we anticipate various accelerators to emerge in the near future. Second, they are sophisticated devices requiring large device drivers, in contrast to simpler devices such as a mouse. Third, they often meet the hardware requirements mentioned above, as we will discuss in §2.2.

Based on the aforementioned principle, we implement Glider, a Linux library driver implementation for two GPUs of popular brands, namely the Radeon HD 6450 and Intel Ivy Bridge GPUs. We implement Glider based on the original legacy Linux drivers. The library driver design allows us to implement both the device kernel and the device library by retrofitting the legacy driver code, which significantly reduces the engineering effort compared to developing them from scratch. We present a full implementation for the Radeon GPU and a proof-of-concept implementation for the Intel GPU.

Our evaluation shows that Glider improves the security of the system by reducing the size and attack surface of the TCB by about 35% and 84% respectively for the Radeon GPU and by about 38% and 90% respectively for the Intel GPU. We also show that Glider provides at least competitive performance with a legacy driver, while slightly outperforming it for applications requiring intensive interactions with the driver, such as GPGPU applications with a small compute kernel and graphics applications using OpenGL's immediate mode.

Beyond monolithic operating systems, library drivers can benefit other systems as well. For example, they can be integrated into a library operating system, such as Exokernel [27] or Drawbridge [51], or into sandboxing solutions such as Embassies [34], Xax [25], and Bromium micro-virtualization [11], to securely support multiplexed access to devices.

2. Library Driver: Design & Requirements

In this section, we discuss the library driver design, and elaborate on hardware properties necessary to implement it.

2.1. Design

The library driver design is based on an important principle from the library operating system research [27]: resource management shall be implemented as untrusted libraries in applications. Resource management refers to the code that is needed to program and use the device. In contrast, legacy device drivers implement resource management along with device resource isolation in the kernel, resulting in a large TCB with a wide high-level interface.

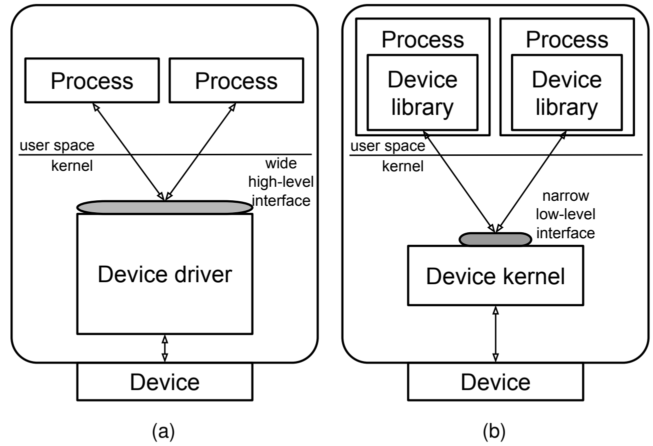

Figure 1: (a) Legacy driver design. (b) Library driver design.

Figure 1 compares a library driver against a legacy driver. The library driver design applies the aforementioned principle in two steps. First, it separates resource management from resource isolation. It enforces resource isolation in a trusted *device kernel* in the operating system kernel and pushes resource management into user space. Second, the library driver design retrofits the resource management code into an untrusted library, i.e., a *device library*, which is loaded and used by each application that needs to use the device.

The device kernel is responsible for securely multiplexing the device resources between untrusting device libraries. Based on some scheduling policy, it *binds* the device resources to a device library at the beginning of a scheduling epoch and *revokes* them at the end of the epoch. If possible, the device kernel preempts the execution upon revoke. When hardware preemption is impossible or difficult to implement, e.g., GPUs [46], the device kernel revokes access when allowed by the device[1]. Many devices, including the ones in our prototype, support only one hardware execution context. For them, the device kernel effectively time-multiplexes the device between device libraries. These devices are the main focus of this paper. Some devices, however, support multiple hardware execution contexts [5, 13, 24, 54, 63]. For those devices, the device kernel can dedicate a hardware execution context to a device library as will be discussed in §7.2.

The device library is responsible for managing the resources exported to it by the device kernel. It implements all the software abstractions and API calls needed by the application to use the device. For example, a GPU device library implements memory management API calls, such as `create_buffer`, `read_buffer`, `map_buffer`, and `move_buffer`, on top of the device memory exported to it by the device kernel. It is important to note that the device library implements the soft-

[1] It is up to the scheduling policy to ensure fairness, using techniques similar to the ones used in [33, 38, 47, 48]

ware abstraction and APIs in user space whilst legacy device drivers implement everything in the kernel. We also note that today's devices often employ user space libraries to create a unified interface to different drivers of the same device class, e.g., GPUs. Such user space libraries do not qualify as device libraries since they do not implement resource management.

Key Benefits

The library driver design improves system security because it reduces the size and attack surface of the TCB. In a library driver, only the device kernel is part of the TCB. Moreover, without any management responsibilities, the device kernel is able to expose a narrower lower-level interface to untrusted software. As a result, the library driver design improves two aspects of system security: it raises the bar for attacks that exploit device driver bugs, and it improves the isolation between applications by reducing the amount of shared state between them.

The library driver design improves the system security without hurting the performance. Indeed, a library driver can even outperform legacy drivers. This is because a library driver eliminates the overhead of the application interacting with the driver in the kernel, including costly syscalls [57] and user-kernel data copies. In a library driver, most of the interactions occur inside the process in the form of function calls rather than syscalls. Moreover, because the device library is in the same trust domain as the application, data can be passed by reference rather than being copied, which is commonly done in a legacy driver.

As we will demonstrate, the library driver design allows us to implement both the device kernel and the device library by retrofitting the legacy driver code. This significantly reduces the engineering effort to develop library drivers for existing devices compared to developing them from scratch.

In addition to security and performance, a library driver has other advantages and disadvantages. In short, the advantages include the possibility of driver customization for applications, easier user space development and debugging, and improved operating system memory usage accounting. The disadvantages include difficulty of multi-process programming, application launch time overhead, and coarser-grained device memory sharing for some devices. §7.1 discusses these issues in greater detail.

We find that library drivers are particularly useful for accelerators such as GPUs, which are increasingly important for a wide range of applications, from data centers to mobile devices. Accelerators are sophisticated hardware components and usually come with large, complex device drivers, particularly prone to the wide variety of driver-based security exploits. More importantly, we have found that accelerators like GPUs often have the necessary hardware properties to implement a library driver, elaborated next.

2.2. Hardware Requirements

Despite its benefits outlined above, library drivers cannot support all devices. The system must have three hardware properties for the device kernel to enforce resource isolation. The lack of any of these properties will leave certain resource management code in the trusted device kernel. (*i*) First, in order for each device library to securely use the memory exported to it by the device kernel, hardware primitives must be available to protect the part of the system and device memory allocated for a device library from access by other device libraries. (*ii*) Second, in order for a device library to safely program the device directly from user space, the registers and instruction set used to program and use the device must not be *sensitive*, i.e., they must not affect the isolation of resources. (*iii*) Finally, in order for the device kernel to properly forward the interrupts to the device libraries without complex software logic, the device interrupts must be easily attributable to different device libraries. Apparently not all devices have these properties; fortunately, accelerators of interest to us, e.g., GPUs, do have these properties, as elaborated below.

2.2.1. Memory Isolation Primitives

The library driver design requires hardware primitives to isolate the access to memory by different device libraries. There are two types of memory: system memory and device memory, the latter being memory on the device itself. Isolation needs to be enforced for two types of memory access: *system-side* access via the CPU's load and store instructions and *device-side* access via direct device programming. Modern processors readily provide protection for system-side access with their memory management units (MMU). As a result, we only need to be concerned with protecting device-side accesses.

Device-side access to system memory: The device library can program the device to access system memory via direct memory access (DMA). Therefore, the device kernel must guarantee that the device only has DMA access permission to system memory pages allocated (by the operating system kernel) for the device library currently using the device. The I/O memory management unit (IOMMU) readily provides this protection by translating DMA target addresses. The device kernel can program the IOMMU so that any DMA requests from a device library are restricted to memory previously allocated for it. We observe that IOMMUs are available on most modern systems based on common architectures, such as x86 and ARM [16, 45]. Moreover, GPUs typically have a built-in IOMMU, which can be used by the device kernel even if the platform did not have an IOMMU.

Device-side access to device memory: The device kernel allocates the device memory for different device libraries, and it must protect them against unauthorized access. A device library can program a device to access the device memory, and such an access does not go through the IOMMU. Therefore, the device must provide hardware primitives to protect memory allocated for one device library from access by another. There are different forms of memory protection that a device can adopt, e.g., segmentation and paging. Each form of memory protection has its pros and cons. For example, segmentation is less flexible than paging since the allocations of physical memory must be contiguous. On the other hand, paging is more expensive to implement on a device [49].

Isolating access to the device memory only applies to devices that come with their own memory. We note that accelerators packing their own memory, such as discrete GPUs and DSPs, often support some form of memory protection primitives. For example, NVIDIA GPUs (nv50 chipsets and later) support paging [39] and TI Keystone DSPs support segmentation [37].

A legacy driver does not require such memory protection primitives as it can implement software-based protection. A legacy driver implements the memory management code in the kernel and employs runtime software checks to ensure that untrusted applications never program the device to access parts of the memory that have not been allocated for them.

2.2.2. Innocuous Device Management Interface

The library driver design further requires the device management interface to be *innocuous* or not *sensitive*, as defined by the Popek-Goldberg theorem about virtualiziblity of an architecture [50]. According to Popek and Goldberg, sensitive instructions are those that can affect the isolation between virtual machines.

A device usually provides an interface for software to use it. This interface consists of registers and potentially an instruction set to be executed by the device. The device management interface is the part of the interface that is used for resource management. Examples include the GPU interface used to dispatch instructions for execution and the GPU interface used to set up DMA transfers. Other parts of the programming interface are used for initializing the device and also to enforce isolation between resources. Examples are the GPU interface used to load the firmware, the interface used to initialize the display connectors on the board, the interface used to prepare the device memory and memory protection primitives.

For **registers**, this requirement means that management registers cannot be sensitive. That is, registers needed for resource management cannot affect the resource isolation. For example on a GPU, software dispatches instructions to the GPU for execution by writing to a register. This register is part of the management interface, and therefore must not affect the isolation, e.g., change the memory partition bounds.

For the **instruction set**, this requirement means that either the instruction set has no sensitive instructions, or the sensitive instructions fail when programmed on the device by untrusted code, such as a device library. The latter is more expensive to implement in the device as it requires support for different privilege levels, similar to x86 protection rings.

Commodity accelerators usually meet this requirement. This is because registers often have simple functionalities, hence management registers are not used for sensitive tasks. (§4.1.3 discusses one violation of this requirement). Also, the instruction set often does not contain sensitive instructions, and resource initialization and isolation is done through registers only.

A legacy driver does not need the device to meet the innocuous device management interface requirement. First, it does not allow untrusted software to directly access registers. Second, all the instructions generated by the application are first submitted to the driver, which can then perform software checks on them to guarantee that sensitive instructions, if any, are not dispatched to the device.

2.2.3. Attributable Interrupts

A device library using the device must receive the device interrupts in order to properly program and use the device. For example, certain GPU interrupts indicate that the GPU has finished executing a set of instructions. The interrupt is first delivered to the device kernel, and therefore, the device kernel must be able to redirect interrupts to the right device library without the need for any complex software logic.

This requirement is simply met for commodity accelerators with a single hardware execution context, since all interrupts belong to the device library using the device. On the other hand, we note that this requirement can be more difficult to meet for non-accelerator devices. One example is network interface cards with a single receive queue. Upon receiving a packet (and hence an interrupt), the device kernel cannot decide at a low level which device library the packet belongs to. This will force the device kernel to employ some management code, e.g., packet filters, to redirect the interrupts, similar to the solutions adopted by the Exokernel [27] and U-Net [61].

A legacy driver does not have this requirement because it incorporates all the management code that uses the interrupts. Device events are then delivered to the application using higher-level API.

3. GPU Background

Before presenting our library driver design for GPUs in §4, we provide background on the functions of a GPU device driver and GPU hardware as illustrated in Figure 2. A GPU driver has three functions: hardware initialization, resource management for applications, and resource isolation.

Initialization

GPU hardware initialization loads the firmware, sets up the GPU's memory controller or MMU in order to configure the address space used by the GPU, enables the interrupts, sets up the command processor (which is later used to dispatch instructions to the GPU), the device power management, and the display subsystem.

Management

Once the GPU hardware is initialized, the driver performs resource management for applications. It implements four important functionalities for this. First, it implements memory management API calls. An application can request three types

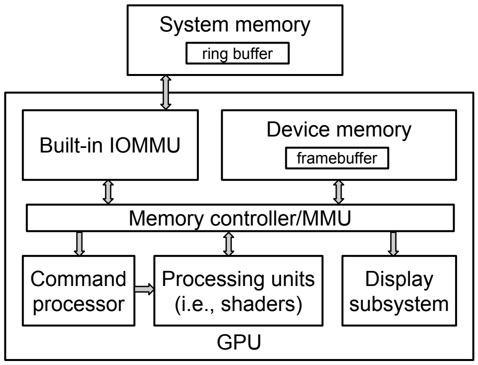

Figure 2: Simplified GPU hardware model.

of memory buffers: buffers on the device memory, buffers on the system memory, and buffers on the system memory accessible to the GPU for DMA. For the latter, the driver allocates a range of GPU addresses for the buffer and programs the GPU's built-in IOMMU to translate from these addresses to their actual physical addresses on the system memory. Moreover, the driver pins these pages so that they do not get swapped out by the operating system kernel.

Second, the driver accepts GPU instructions from applications and dispatches them to the GPU for execution. Dispatching the instructions in done through a ring buffer. The driver writes the instructions onto the ring buffer, and the command processor unit on the GPU reads them and forwards them to the processing units, i.e., shaders, in the GPU. Note that similar to CPUs, GPUs cache memory accesses and the built-in IOMMU has a TLB for its translations. Therefore, the driver flushes the caches and the TLB as needed.

Third, the driver handles interrupts from the GPU. Most importantly, it processes the interrupts that indicate the end of the execution of the instructions by the processing units. The driver then informs the application so that the application can send in new instructions to be executed.

Finally, the driver implements some API calls for the application to allocate and use a framebuffer and to set the display mode. The framebuffer is a memory buffer holding the content that will be displayed. Through programming the GPU or by directly accessing the memory, the application can write to the framebuffer. The display mode includes the resolution, color depth, aspect ratio, etc. Different applications require to set different modes for the display. Traditionally, the display mode was set through the X server in Linux. However, it was recently moved to the legacy Linux open source drivers and is referred to as Kernel Mode Setting, or KMS. The advantage of KMS is that it allows for better graphics at boot time (before X is even initialized) and during Linux virtual console switching.

Isolation

The driver also enforces isolation between applications by performing appropriate hardware configurations, such as setting up the MMU page tables if supported, or through software checks. For instance, a legacy GPU driver enforces software memory isolation as follows. When allocating memory buffers for an application, the driver only returns an ID of the buffer to the application, which then uses that ID in the instructions that it submits to the driver. The driver then replaces all the IDs with the actual addresses of the buffers and then writes them to the ring buffer.

Glider implements these three functionalities as follows. Initialization is performed in the device kernel. Once the GPU is initialized, the device kernel exports the resources to device libraries so that they can perform resource management in the user space. For isolation, the device kernel leverages hardware properties introduced in §2.2. This design reduces the size and attack surface of the TCB and hence improves the system security. The GPU hardware often meets the required hardware properties. It provides either a memory controller or an MMU that can be used to enforce isolation for device-side accesses to its memory. The management registers are often not sensitive and the instruction set has no sensitive instructions. Moreover, with GPUs with a single hardware execution context, interrupts can be simply forwarded to the device library using the GPU.

4. Glider: Library Driver for GPUs

In this section, we present Glider, library drivers for the Radeon HD 6450 and Intel Ivy Bridge GPUs based on retrofitting their corresponding legacy drivers. We provide a fully-functional implementation for the Radeon GPU and a proof-of-concept implementation for the Intel GPU.

4.1. Isolation of GPU Resources

We first identify the important GPU hardware resources and elaborate on how the device kernel securely exports them to device libraries.

4.1.1. Processing Units

The device kernel securely exports the GPU processing units, i.e., shaders, by allowing access to the GPU command processor. When binding the GPU to a device library, the device kernel allows the device library to update the command processor's registers that determine the ring buffer location. This enables the device library to allocate and use the ring buffer that it has allocated from its own memory. The device library then populates the ring buffer with instructions from the application and triggers the execution by writing to another register. At revoke time, the device kernel disallows further write to the trigger registers. It then waits for ongoing execution to finish on the GPU before binding the GPU to another device library. The device kernel also takes a snapshot of the registers updated by the device library so that it can write them back the next time it needs to bind the GPU to the same device library. Finally, it resets the command processor and flushes the GPU caches appropriately.

4.1.2. Memory

The operating system allocates system memory pages for a device library through standard syscalls, such as `mmap` and the device kernel allocates device memory for the device library through its API calls (§4.2). When using the device, the device kernel needs to guarantee that a device library can only access memory allocated for it, as discussed in §2.2.1. Here, we provide the implementation details of isolating device-side access to memory.

For system memory, we use the system IOMMU for isolation. The device kernel provides an API call for the device library to map a page into the IOMMU. More specifically, the device library can ask the device kernel to insert a mapping into the IOMMU in order to translate a DMA target address to a given physical page address. To enforce isolation, the device kernel only maps pages that have been allocated for the device library. The device kernel also pins the page into memory so that it does not get swapped out or deallocated by the operating system kernel. Upon revoke, the device kernel stores all the mappings in the IOMMU and replaces them with the mappings for the next device library. The mappings of an IOMMU is in the form of page tables [16], very similar to the page tables used by the CPU MMU. Therefore, similar to a context switch on the CPU, changing the IOMMU mappings can be done very efficiently by only changing the root of the IOMMU page tables.

Alternatively, the GPU's built-in IOMMU can be used for isolation, which is useful for systems without a system IOMMU. To demonstrate this, we used the built-in IOMMU for our Intel GPU library driver. In this case, the device kernel takes full control of the GPU's built-in IOMMU and updates its page tables through the same API call mentioned above.

It is up to the device library when and how much memory it maps in the IOMMU. In our current implementation, the device library allocates about 20 MB of memory at launch time and maps all the pages in the IOMMU. We empirically determined this number to be adequate for our benchmarks. Alternatively, a device library can allocate and map the pages in the IOMMU as needed. This alternative option speeds up the device library's launch process but may result in degraded performance at runtime.

For the Radeon device memory, we use the GPU memory controller for isolation. Isolating the device memory does not apply to the Intel GPU since it does not have its own memory. The memory controller on the Radeon GPU configures the physical address space seen by the GPU. It sets the range of physical addresses that are forwarded to the GPU memory and those that are forwarded to the system memory (after translation by the built-in IOMMU). By programming the memory controller, the device kernel can effectively segment the GPU memory, one segment per device library. Obviously, the memory segmentation is not flexible in that each device library can only use a single contiguous part of the GPU memory. Changing the segments can only be done using memory ballooning techniques used for memory management for VMs [62], although we have not implemented this yet.

4.1.3. Framebuffer and Displays

The hardware interface for the framebuffer can be securely exported to device libraries. The device kernel allows the device library to have access to the registers that determine the location of the framebuffer in memory. Therefore, the device library can allocate and use its own framebuffer. However, the hardware interface for display mode setting violates the requirement in §2.2.2, forcing us to keep the display mode setting code in the device kernel.

Every display supports a limited number of modes that it can support. However, instead of exposing the mode option with a simple interface, e.g., a single register, GPUs expect the software to configure voltages, frequencies, and clocks of different components on the GPU, e.g., display connectors and encoders, to set a mode, and it is not clear whether such a large hardware interface can be safely exported to unstrusted software without risking damaging the device. Exacerbating the problem, newer Radeon GPUs have adopted mode setting through GPU BIOS calls. However, BIOS calls can also be used for other sensitive GPU configurations, such as for power management and operating thermal controllers, and we cannot export the BIOS register interface to device libraries securely.

As a result, we keep the display mode setting code in the device kernel. The device kernel exposes this mode setting functionality with a higher level API call to device libraries. This results in a larger TCB size as is reported in §5.1. Despite its disadvantage of increasing the TCB size, keeping the mode setting in the device kernel has the advantage that it supports Kernel Mode Setting (§3).

4.2. The Device Kernel API

The device kernel API include seven calls for device libraries and two calls for the system scheduler. Except for the `set_mode` call, which is GPU-specific (§4.1.3), the rest are generic. Therefore, we expect them to be used for device libraries of other devices as well. They constitute the minimal set of API calls to support device library's secure access to system memory, device memory, and registers. The seven calls for device libraries are as follows:

`void *init_device_lib(void)`: This call is used when a device library is first loaded. The device kernel prepares some read-only memory pages containing the information that the device library needs, such as the display setup, maps them into the device library's process address space, and returns the address of the mapped pages to the device library.

`iommu_map_page(vaddr, iaddr)`, `iommu_unmap_page(vaddr)`: With these two calls, the device library asks the device kernel to map and unmap a memory page to and from the IOMMU. `vaddr` is the virtual address of the page in the process address space. The device kernel uses this address to find the physical address of the page. `iaddr` is the address that needs to to be translated by the IOMMU to the physical address. This will be the DMA target address issued by the device.

`alloc_device_memory(size)`, `release_device_memory(addr, size)`: With these two calls, the device library allocates and releases the device memory. These two calls are only implemented for GPUs with their own memory.

`int access_register(reg, value, is_write)`: With this call, a device library reads and writes from and to authorized registers. The implementation of this call in the device kernel is simple: it just checks whether the read or write is authorized or not, and if yes, it completes the request. The checking is done by maintaining a list of authorized registers. We implement read and write operations in one API call since their implementation is different only in a few lines of code.

Given that most registers on GPUs are memory-mapped (i.e., MMIO registers), One might wonder why the device kernel does not directly map these registers into the device library's process address space, further reducing the attack surface on the TCB. This is because with such an approach, protection of registers can be enforced at the granularity of a MMIO page, which contains hundreds of registers, not all of them are authorized for access by a device library.

`set_mode(display, mode)`: With this call, a device library asks the device kernel to set the mode on a given display.

We next present the two calls for the system scheduler. Schedulers for GPU resources, such as [33, 38, 47, 48], can be implemented on top of these two API calls.

`bind_device_lib(id)`, `revoke_device_lib(id)`: With these two calls, the scheduler asks the device kernel to bind and revoke the GPU resources to and from a device library with a given `id`. Since our GPUs do not support execution preemption (§2.1), the revoke call needs to block until the execution on the GPU terminates.

4.3. Reusing Legacy Driver Code for Glider

We reuse the Linux open source legacy driver code in Glider, both for the device kernel and the device library, rather than implementing them from the scratch, in order to reduce the engineering effort. Reusing the legacy driver code for device kernel is trivial since the device kernel runs in the kernel as well. However, reusing it for the device library is challenging since the device library is in the user space. We solve this problem by using the User-Mode Linux framework [23]. UML is originally designed to run a Linux operating system in the user space of another Linux operating system and on top of the Linux syscall interface. It therefore provides translations of kernel symbols to their equivalent syscalls, enabling us to compile the device library to run in the user space. We only use part of the UML code base that is needed to provide the translations for the kernel symbols used in our drivers.

Note that the UML normally links the compiled object files into an executable. We, however, link them into a shared library. Linking into a shared library may be challenging on some architectures. This is because assembly code, which is often used in the kernel, is not always position-independent, a requirement for a shared library. We did not experience any such problem for the x86 (32 bit) architecture. The solution to any such potential problem in other architectures is to either rewrite the assembly code to be position-independent or to replace it with equivalent non-assembly code.

It is interesting to understand how system memory allocation works in Glider's device library, which is compiled against the UML symbols. As mentioned in §4.1.2, we allocate about 20 MB of system memory at the device library's launch. This memory is then managed by the slab allocator of UML, similar to how physical memory is managed by the slab allocator in the kernel. The retrofitted driver code in the device library then allocates system memory from the UML's slab allocator by calling the Linux kernel memory allocation functions, i.e., `kmalloc()` and its derivatives.

4.4. Other Implementation Challenges

We solved two other challenges in Glider. First, we replace syscalls for the legacy driver with function calls into equivalent entry points in the device library. Fortunately in the case of GPU, we managed to achieve this by only changing about 20 instances of such syscalls in Linux GPU libraries including the `libdrm`, `xf86-video-ati`, and GalliumCompute [2] libraries. An alternative solution with less engineering effort is to use the `ptrace` utility to intercept and forward the syscalls to the device library. This solution, however, will have noticeable overhead.

Second, we implement fast interrupt delivery to device libraries for good performance. We experimented with three options for interrupt delivery. The first two, i.e., using the OS signals and syscall-based polling, resulted in performance degradation as these primitives proved to be expensive in Linux. The third option that we currently use is polling through shared memory. For example, in the case of the Radeon GPU, in addition to the interrupt, the GPU updates a memory page with the information about the interrupt that was triggered, and the device library can poll on this page. This approach provided fast interrupt delivery so that the interrupts do not become a performance bottleneck. However, as we will show in §5.2.2, it has the disadvantage of increased CPU usage.

We are considering two other options that we believe will provide fast interrupt delivery without the extra CPU usage. The first approach is using upcalls, which allow the kernel to directly invoke a function in user space. The second approach is to use the interrupt remapping hardware available from virtualization hardware extensions. This hardware unit can deliver the interrupts directly to the device library's process, similar to Dune [18].

5. Evaluation

We evaluate Glider and show that it improves the system security by reducing the size and attack surface of the TCB. We also show that Glider provides at least competitive performance with a legacy kernel driver, while slightly outperforming it for applications that require extensive interactions with the driver.

5.1. Security

We measure the size and attack surface of the TCB, i.e., the whole driver in the case of a legacy driver and the device kernel in the case of Glider. Unlike the legacy driver that supports various GPUs of the same brand, Glider only supports one specific GPU. Therefore, for a fair comparison, we remove the driver code for other GPUs as best as we manually can. This includes the code for other GPU chipsets, other display connectors and encoders, the code for the legacy user mode setting framework not usable on newer GPUs (§3), the code for audio support on GPUs, and the code for kernel features not supported in Glider, such as with Linux `debugfs`.

Our results, presented in Table 1, show that Glider reduces the TCB size by about 35% and 38% for the Radeon and Intel GPUs, respectively. In the same table, we also show the size of the code in C source files (and not header files). These numbers show that a large part of the code in Glider TCB are headers, which mainly include constants, such as register layouts. Not including the headers, Glider reduces the TCB size by about 47% and 43% for the Radeon and Intel GPUs.

As discussed in §4.1.3, the display subsystem hardware interface violates the hardware requirements for a library driver resulting in a larger TCB. To demonstrate this, we measure the code size in display-related files in the device kernel. For the Radeon and Intel GPUs, we measure this number to be 19 and 13 kLoC, which is 50% and 54% of the Glider TCB.

We also show the TCB attack surface in Table 1. Glider reduces the attack surface by 84% and 90% for the Radeon and Intel GPUs. Glider only exposes 9 and 7 API calls for these GPUs, as described in detail in §4.2 (Intel GPUs do not implement the two API calls for device memory). In contrast, the legacy driver exposes 56 and 68 API calls for the same GPUs. These large number of API calls are used for memory management, GPU execution, display mode setting, and inquiring information about the GPU. Glider supports the first two by securely giving a device library access to part of the GPU management interface. It supports mode setting with one API call and supports the information inquiring API either through the constants compiled into the device library or through the read-only information pages mapped into the device library (§4.1).

5.2. Performance

In this section, we evaluate the performance of Glider for the Radeon GPU using both compute and graphics benchmarks.

For the experiments, we run the drivers inside a 32-bit x86 (with Physical Address Extension) Ubuntu 12.04 operating system running Linux kernel 3.2.0. The machine has a 3rd generation core i7-3770 (with hyper-threading disabled). We configure the machine with 2 GBs of memory and 2 cores. In order to minimize the effect of the operating system scheduler on our experiments, we isolate one of the cores at boot time using the Linux `isolcpus` command-line boot option. With this option, Linux only schedules kernel threads on the isolated core, but it does schedule user application threads on it unless explicitly asked for. We then pin our benchmarks to the isolated core and set the highest scheduling priority for them. In order to use the system IOMMU for the Radeon GPU, we run the benchmarks inside a Xen VM with the same configurations mentioned above (2 GBs of memory and 2 cores). The Radeon GPU is assigned to the VM using the direct device assignment [16, 19, 32, 44].

		TCB (all files)	TCB (source files)	API calls
Radeon	Legacy	55	34	56
	Glider	36	18	9
	Reduction	**35%**	**47%**	**84%**
Intel	Legacy	39	30	68
	Glider	24	17	7
	Reduction	**38%**	**43%**	**90%**

Table 1: TCB size and attack surface for the legacy and Glider for the Radeon HD 6450 and Intel Ivy Bridge GPUs. The numbers for both TCB columns are in kLoC. The first TCB columns reports LoC in both source and header files. The second TCB column excludes the header files.

Figure 3: OpenCL matrix multiplication. The x-axis shows the (square) matrix order. The y-axis is the time to execute the multiplication on the GPU, normalized to the average time by the legacy driver.

5.2.1. Compute Benchmarks

We use a matrix multiplication benchmark running on top of the GalliumCompute [2] framework, an open source implementation of OpenCL. We evaluate the performance of multiplying square matrices of varying orders. For each experiment, we use a host program that populates input matrices and launches the compute kernel on the GPU. The programs repeats this for 1000 iterations in a loop and outputs the average time for a single iteration. We discard the first iteration to avoid the effect of Glider's launch time overhead (characterized in §5.3) and we do not include the time to compile the OpenCL kernel. We then repeat the experiment 5 times for each matrix size and report the average and standard deviation.

Figure 3 shows the results, normalized to the average performance by the legacy driver in each case. It shows that Glider outperforms the legacy driver for smaller matrix sizes, while providing competitive performance for all other sizes. This is because for large matrix sizes, the majority of the time is spent on transferring the data between the system and device memory and on the GPU executing the kernel. Consequently, the driver design does not impact the performance noticeably. For smaller sizes, on the other hand, the overhead of application's interaction with the driver becomes more significant.

5.2.2. Graphics Benchmarks

For graphics, we use two OpenGL benchmarks running on top of the MESA open source implementation of OpenGL [4]. Both benchmarks draw a teapot, and update the screen as fast as possible. One benchmark uses the Vertex Array API [7] (the immediate mode) and the other uses the Display List API [6]. In each experiment, we run the benchmark for 5 minutes, recording the framerate every second. We discard the first 5 frames to avoid the effect of Glider's launch time overhead (characterized in §5.3). We repeat each experiment three times.

The results, shown in Figure 4, demonstrates that Glider achieves similar performance as the legacy driver for the Display List benchmark but outperforms the legacy driver for the Vertex Array one. In order to explain these results, it is important to understand these two OpenGL API's. With Vertex Arrays, the program needs to send the vertices information to the device for every frame, which requires interacting with the driver quite significantly. On the other hand, the Display List API allows this information to be cached by the device in order to avoid resending from the application. Intuitively, Glider improves the performance for the benchmark with more driver interactions, or the Vertex Array.

We also measure the CPU usage for both drivers when running these benchmarks. Our results show that the legacy driver consumes 53.7% and 41.7% of the CPU time for the Vertex Array and Display List benchmarks, respectively, whereas Glider consumes 75.3% and 71.6% of the CPU time for the same benchmarks. The extra CPU usage of Glider is due to polling the memory for interrupts (§4.4). One might wonder whether the extra CPU usage is the source of the performance improvement. To investigate this, we attempted to employ similar polling methods in the legacy driver, but failed to improve the performance. Therefore, we are convinced that Glider's performance improvement is due to eliminating the application and driver interactions' overhead. We also report the performance when using syscall-based polling in the device library, which incurs delay in delivering the interrupt to the device library. With this method, for the Vertex Array and Display List benchmarks, respectively, Glider consumes 54.3% and 35.3% of the CPU time, and achieves 193.3 and 231.5 median frames per second, which are noticeably lower compared to when the device library polls the memory for interrupts.

Figure 4 also shows that Glider achieves a noticeably lower minimum framerate, although the minimum happens rarely. To further demonstrate this, we show a sample run of the Vertex Array benchmark in Figure 5. Our investigation shows that the performance drop is due to OS scheduling despite our attempts to minimize such effect.

5.3. Library Driver Overheads

We measure two important overheads of Glider: the device library's launch time overhead and the device kernel's switch time, i.e., the time it takes the core to revoke the GPU from one device library and bind it to another.

Figure 5 illustrates the effect of the device library's launch time overhead on the graphics benchmark reported before. It shows that the performance of Glider is inferior to that of the legacy driver in the first few seconds. There are two sources for this overhead: the device library's initialization overhead and the UML's slab allocator's initialization overhead. The former is the time it takes the device library to initialize itself so that it is ready to handle requests from the application. We measure this time to have an average/standard deviation of of 109 ms/7 ms and 66 ms/0.3 ms for the Radeon and Intel GPUs, respectively. The latter is because the slab allocator of UML (§4.3) takes some time to populate its own cache, very similar to the suboptimal performance of the slab allocator in the kernel at system boot time.

We also measure the switch time in the device kernel, as defined above. We measure this time to have an average and standard deviation of 42 μs and 5 μs for the Radeon GPU (we have not yet implemented this feature for the Intel GPU). The switch time consists of the device kernel taking a snapshot of GPU registers updated by the current device library, writing the register values for the new device library, changing the IOMMU mappings, resetting the command processor, and flushing the GPU caches and the built-in IOMMU TLB. These measurements show that changing the GPU binding can be done fairly quickly.

5.4. Engineering Effort

As mentioned, we build Glider by retrofitting the Linux open source drivers as a baseline. We added about 4 kLoC and 2 kLoC for the Radeon and Intel library drivers, respectively. These changes were applied to 49 and 30 files, and we added two new files in each case. We have implemented both the device kernel and the device library on the same driver, and the two are differentiated at compile time. While we believe that reusing the legacy driver code significantly reduced our engineering effort compared to developing from scratch, we note that implementing the library drivers still required notice-

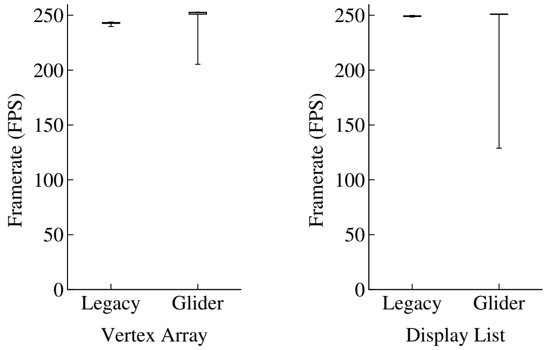

Figure 4: Graphics performance. The box plots show 90% percentile, 10% percentile, and median. The whiskerbars depicts maximum and minimum. The box plots are highly concentrated around the mean.

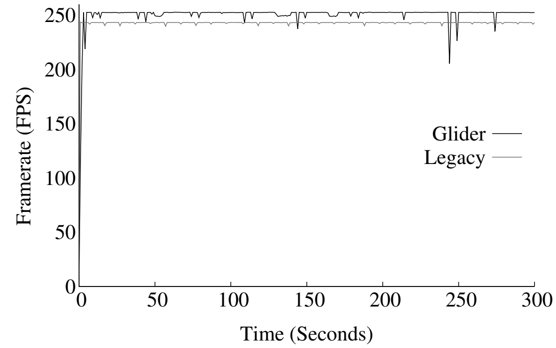

Figure 5: A sample run of the Vertex Array benchmark. The framerate is measured every second. The figure shows the rare drops of framerate and the slow start of Glider.

able effort. The main source of difficulty was gaining a deep understanding of the GPU internals and its device driver with 10s thousands of lines of code. Fortunately, our experience with the Radeon GPU library driver made it easier for us to prototype the Intel GPU library driver. We therefore believe that experienced driver developers can develop library drivers without prohibitive engineering effort.

6. Related Work

Library Operating Systems and Sandboxes

Library operating systems, such as Exokernel [27] and Drawbridge [51] improve the system security by executing the operating system management components as a library in the application's process address space. Library drivers are complementary; they can serve as a secure way for applications to use the devices.

Hardware sandboxing techniques, such as Embassies [34], Xax [25], and the Bromium micro-virtualization [11], improve the system security by reducing the TCB size. Library drivers can complement these sandboxes by providing them with secure access to devices.

Alternative Device Driver Designs

Many previous solutions have attempted to reduce the drivers' risk on the system security. Some solutions move the driver to user space or to a VM. For examples, in microkernels, drivers reside completely in user space in order to keep the kernel minimal [26, 28, 31, 41, 55]. SUD also encapsulates a Linux device driver in a user space process, and similar to Glider, uses the IOMMU and UML to achieve this [21]. Microdriver is another solution, which splits the driver between the kernel and user space [30]. It keeps the critical path code of the driver in the kernel, but redirects the execution to the user space elsewhere. Hunt [35] also presents a solution that moves the driver to the user space by employing a proxy in the kernel. LeVasseur et al. [42], on the other hand, execute the device driver in a VM. All these solutions improve the security of the operating system kernel by removing the driver. However, in contrast to a library driver, they cannot improve the isolation between processes using the device, since the driver is fully shared between them. Moreover, unlike a library driver, these solutions are reported to hurt the performance.

CuriOS [22] improves the isolation between applications that use a microkernel service, such as the file system, by providing the service with access to client states only when the service is processing a request. This allows for recovery from most of the errors in the service. In contrast, library drivers improve the isolation between application by reducing the TCB size.

Other solutions try to protect against errors in the drivers, either using runtime and hardware-enforced techniques such as Nooks [59] or using language-based techniques such as SafeDrive [64]. In contrast, a library driver improves the system security by reducing the TCB size through the principle of untrusted resource management.

Zhou et al. move the device driver to the user space in order to create a trusted path between the application and the device [65, 66]. However, they assume a single application using an I/O device and therefore they assign the device to the application. Unlike them, we tackle a more challenging problem where the I/O device is shared between untrusting applications.

Schaelicke [56] and Pratt [52, 53] propose hardware architectures to support secure user space access to devices. In this work, we show that adequate hardware primitives already exist on commodity accelerators, such as GPUs, platforms to run the device management code in the user space.

Exokernel [27], U-Net [61], and sv3 [58] move part or all of the network device driver to the user space for better security or performance. MyCloud SEP detangles the resource management functions for disks and make them untrusted in order to reduce the TCB of virtualization platforms [43]. We share the same goals with this line of work. However, in our work, we demonstrate that such an approach is applicable to a wider range of devices, especially accelerators such as GPUs, and present a framework to apply similar concepts to other devices as well.

Some existing device drivers incorporate a user space components in addition to the kernel driver. Such drivers differ from library drivers in one of two ways. First, the user space component is in the TCB since it is shared by applications using the device. By compromising this user space component, a malicious application can attack other applications. This component is either a process (e.g., the driver host process in Windows User-Mode Driver Framework (UMDF) [15]) or a library loaded by a shared process (e.g., Mali GPU drivers for Android, where libraries are loaded by the surface_flinger process and libusb, where the loader process can control the device). Second, although the I/O stack includes untrusted libraries, the kernel driver implements high-level resource management with several APIs. For example, the in-kernel Display Miniport Driver of Windows Display Driver Model (WDDM) implements ~70 APIs including APIs for command submission [1]. Mali kernel drivers for Linux exports ~40 APIs.

Some devices provide per-process hardware support. Examples are some NVIDIA GPUs and Infiniband devices, which support per-process command queues. While such hardware support is added for performance, it improves the system security as well by reducing the TCB size, similar to library drivers. Our work shows that similar goals can be achieved for devices without such explicit hardware support as well.

Virtualization

gVirt [60] supports mediated passthrough of Intel GPUs. Privileged modules running in the Xen hypervisor and Dom0 give VMs direct access to performance-critical resources of the GPU while emulating their access to sensitive resources, allowing VMs to use the native device driver. In contrast, library drivers are designed to securely multiplex a device between processes in the same operating system. However, the hardware isolation techniques used in gVirt can be leveraged in library drivers for Intel GPUs as well.

The nonkernel [20] is proposed to give applications direct access to devices. In contrast, we demonstrated that a device can be shared between multiple applications if its resources are multiplexed at a finer granularity.

Dune [18] gives applications direct access to virtualization hardware extensions. Library drivers benefit from virtualization extensions as well. For example, the device kernel uses the IOMMU to isolate device DMA targets. However, in contrast to Dune, the hardware features are used by the device kernel running in the kernel.

Paradice [17] paravirtualizes I/O devices at the device file interface and hence allows virtual machines to directly talk to the device driver. For security, it uses a trusted hypervisor to provide fault and device data isolation between virtual machines assuming that the device driver is compromised. However, it cannot provide other guarantees such as functional correctness. Library drivers are complementary as they reduce the device driver TCB size and attack surface, reducing the possibility of the device driver getting compromised in the first place.

Other Accelerator Architectures

Heterogeneous System Architecture [40] is a standard for future accelerators, targeting both the accelerator hardware and software. One hardware features of the HSA is the support for user space dispatch, allowing applications to dispatch instructions to the accelerator without communicating with the driver in the kernel. In our work, we demonstrated that such a feature is feasible even with commodity accelerators. Moreover, we anticipate that other resource management tasks, such as memory management, still remain in the trusted device drivers for HSA-compliant devices, whereas a library driver makes all the management code untrusted.

7. Discussions

7.1. Pros and Cons of Library Drivers

Other than improved security and performance, library drivers have three other advantages and three disadvantages. The advantages are as follows. First, library drivers allow each application to customize its own device library. For example, applications can trade-off the initial cost of allocating a pool of memory buffers with the runtime cost of allocating buffers as needed. Second, library drivers greatly simplify driver development because the developer can use user space debugging frameworks or high-level languages, none of them are available in the kernel. Moreover, developers can use a unified debugging framework for both the application and the device library, which can greatly help with timing bugs and data races. Third, library drivers improve memory usage accounting by the operating system. Legacy drivers for some devices, such as GPUs, implement their own memory management, which gives applications a side channel to allocate parts of the system memory, invisible to the operating system kernel for accounting. In contrast, with a library driver, all the system memory allocated by a device library is through the standard operating system memory management interface, e.g., the `mmap` and `brk` syscalls in Linux.

The library driver design has the following disadvantages. First, library drivers complicate multi-process programming. For example, sharing memory buffers between processes is easily done in a legacy driver, but requires peer-to-peer communication between the device libraries in a library driver. Second, library drivers incurs launch time overheads to applications. We evaluated this overhead in §5.3 and showed that it is not significant. Third, depending on the device, a library driver may achieve coarser-grained device memory sharing for applications. A legacy driver can share the device memory at the granularity of buffers, whereas with devices without paging support for their own memory, such as the Radeon GPU in our setup, device memory can only be shared between device libraries using contiguous segments.

7.2. Devices with Multiple Hardware Execution Contexts

For devices with multiple execution contexts, the device kernel should bind and revoke the hardware contexts to device libraries independently. This puts new requirements on the device and platform hardware as we will explain next.

First, the hardware management interface for different contexts must be non-overlapping and isolated. That is, registers for each context should be separate and instructions should only affect the resources of the given context. Also, device DMA requests must be attributable to different contexts at a low level so that an IOMMU can be used to isolate them.

As an example, self-virtualized devices [5, 24, 54, 63] export multiple virtual devices, each of which can be separately assigned to a VM. As a result, they readily provide all the hardware primitives for the device kernel to bind different virtual devices to different device libraries.

7.3. Radeon Instruction Set

We allow a device library to directly dispatch instructions to our Radeon and Intel GPUs since the instructions are not sensitive. We noticed a curious case with the Radeon GPU's instruction set though. Using the instructions, an application can write to the registers of GPU processing units in order to program and use them. Fortunately, it is not possible to use the instructions to write to registers of components of the GPU that affect the isolation, such as the memory controller. However, we noticed that the Linux open source Radeon driver returns error when inspecting the instructions submitted by an applications and detecting accesses to large register numbers, which surprisingly correspond to no actual registers according to the AMD reference guide [36] and the register layout in the driver itself. Therefore, we believe that this is a simple correctness check by the driver and not a security concern. Therefore, we currently do not employ such a check in device kernel, although that is a possibility. Adding the check to the device kernel will increase the TCB size by about a few kLoC, but should not degrade the performance compared to what was reported in this paper since we already performed these checks in the device library in our benchmarks (although it was not necessary).

8. Conclusions

We presented library drivers, a driver design that improves system security by running device management code in untrusted libraries within application processes rather than in the kernel. We discussed the device and platform hardware properties needed for a library driver and showed that they are mostly available for commodity accelerators, such as GPUs, which are of interest to us. We presented Glider, a library driver implementation for two GPUs of popular brands, Radeon and Intel. Our evaluation showed that Glider reduced the size and attack surface of the TCB. Moreover, it improved the performance for benchmarks requiring intensive interactions with the driver. We believe that library drivers are a viable solution for accelerators, an increasing important subset of devices.

References

[1] Functions Registered by DriverEntry of Display Miniport Driver in Windows Display Driver Model (WDDM). http://msdn.microsoft.com/en-us/library/windows/hardware/ff566463(v=vs.85).aspx.

[2] GalliumCompute. http://dri.freedesktop.org/wiki/GalliumCompute/.

[3] Integer overflow in Linux DRM driver (CVE-2012-0044). https://bugzilla.redhat.com/show_bug.cgi?id=772894.

[4] Mesa. http://www.mesa3d.org/.

[5] NVIDIA GRID K1 and K2 Graphics-Accelerated Virtual Desktops and Applications. NVIDIA White Paper.

[6] OpenGL Microbenchmark: Display List. http://www.songho.ca/opengl/gl_displaylist.html.

[7] OpenGL Microbenchmark: Vertex Array. http://www.songho.ca/opengl/gl_vertexarray.html.

[8] Privilege escalation using an exploit in Linux NVIDIA binary driver. http://seclists.org/fulldisclosure/2012/Aug/4.

[9] Privilege escalation using NVIDIA GPU driver bug (CVE-2012-4225). http://www.securelist.com/en/advisories/50085.

[10] Stack buffer overflow in NVIDIA display driver service in Windows 7. https://www.securityweek.com/researcher-unwraps-dangerous-nvidia-driver-exploit-christmas-day.

[11] Understanding Bromium Micro-virtualization for Security Architects. Bromium White Paper.

[12] Unprivileged GPU access vulnerability using NVIDIA driver bug (CVE-2013-5987). http://nvidia.custhelp.com/app/answers/detail/a_id/3377/~/unprivileged-gpu-access-vulnerability---cve-2013-5987.

[13] VMDq. http://www.intel.com/content/www/us/en/network-adapters/gigabit-network-adapters/io-acceleration-technology-vmdq.html.

[14] WebGL Security. http://www.khronos.org/webgl/security/.

[15] Windows User-Mode Driver Framework (UMDF). http://msdn.microsoft.com/en-us/library/windows/hardware/ff560442(v=vs.85).aspx.

[16] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 2006.

[17] A. Amiri Sani, K. Boos, S. Qin, and L. Zhong. I/O Paravirtualization at the Device File Boundary. In *Proc. ACM ASPLOS*, 2014.

[18] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazieres, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proc. USENIX OSDI*, 2012.

[19] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B. A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. USENIX OSDI*, 2010.

[20] M. Ben-Yehuda, O. Peleg, O. Agmon Ben-Yehuda, I. Smolyar, and D. Tsafrir. The nonkernel: A Kernel Designed for the Cloud. In *Proc. ACM APSYS*, 2013.

[21] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in Linux. In *Proc. USENIX ATC*, 2010.

[22] F. M. David, E. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving Reliability through Operating System Structure. In *Proc. USENIX OSDI*, 2008.

[23] J. Dike. User-Mode Linux. In *Proc. Ottawa Linux Symposium*, 2001.

[24] Y. Dong, Z. Yu, and G. Rose. SR-IOV Networking in Xen: Architecture, Design and Implementation. In *Proc. USENIX Workshop on I/O Virtualization (WIOV)*, 2008.

[25] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *Proc. USENIX OSDI*, 2008.

[26] K. Elphinstone and G. Heiser. From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels? In *Proc. ACM SOSP*, 2013.

[27] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an Operating System Architecture for Application-Level Resource Management. In *Proc. ACM SOSP*, 1995.

[28] A. Forin, D. Golub, and B. N. Bershad. An I/O System for Mach 3.0. In *Proc. USENIX Mach Symposium*, 1991.

[29] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP Kernel Crash Analysis. In *Proc. USENIX LISA*, 2006.

[30] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The Design and Implementation of Microdrivers. In *Proc. ACM ASPLOS*, 2008.

[31] David B. Golub, Guy G. Sotomayor, and Freeman L. Rawson, III. An Architecture for Device Drivers Executing As User-Level Tasks. In *Proc. USENIX MACH III Symposium*, 1993.

[32] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, D. Tsafrir, and A. Schuster. ELI: Bare-Metal Performance for I/O Virtualization. In *Proc. ACM ASPLOS*, 2012.

[33] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: Coordinated Scheduling for Virtualized Accelerator-Based Systems. In *USENIX ATC*, 2011.

[34] J. Howell, B. Parno, and J. Douceur. Embassies: Radically Refactoring the Web. In *Proc. USENIX NSDI*, 2013.

[35] G. C. Hunt. Creating User-Mode Device Drivers with a Proxy. In *Proc. USENIX Windows NT Workshop*, 1997.

[36] Advanced Micro Devices Inc. Radeon Evergreen 3D Register Reference Guide, Revision 1.0. http://www.x.org/docs/AMD/old/evergreen_3D_registers_v2.pdf, 2011.

[37] Texas Instruments. KeyStone Architecture Multicore Shared Memory Controller (MSMC) - User Guide. Literature Number: SPRUGW7A, 2011.

[38] S. Kato, K. Lakshmanan, R. R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU Scheduling for Real-time Multi-tasking Environments. In *Proc. USENIX ATC*, 2011.

[39] M. Kościelnicki. NVIDIA Hardware Documentation. https://media.readthedocs.org/pdf/envytools/latest/envytools.pdf.

[40] George Kyriazis. Heterogeneous System Architecture: A Technical Review, AMD White Paper. 2012.

[41] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-Level Device Drivers: Achieved Performance. *Journal of Computer Science and Technology*, 20(5), 2005.

[42] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proc. USENIX OSDI*, 2004.

[43] M. Li, Z. Zha, W. Zang, M. Yu, P. Liu, and K. Bai. Detangling Resource Management Functions from the TCB in Privacy-Preserving Virtualization. In *Proc. European Symposium on Research in Computer Security (ESORICS)*, 2014.

[44] J. Liu, W. Huang, B. Abali, and D. K. Panda. High Performance VMM-Bypass I/O in Virtual Machines. In *Proc. USENIX ATC*, 2006.

[45] ARM Ltd. ARM Cortex-A15 Technical Reference Manual. *ARM DDI*, 0438C, 2011.

[46] J. Menon, M. De Kruijf, and K. Sankaralingam. iGPU: Exception Support and Speculative Execution on GPUs. In *Proc. IEEE ISCA*, 2012.

[47] K. Menychtas, K. Shen, and M. L. Scott. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *Proc. ACM ASPLOS*, 2014.

[48] Konstantinos Menychtas, Kai Shen, and Michael L Scott. Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack. In *Proc. USENIX ATC*, 2013.

[49] B. Pichai, L. Hsu, and A. Bhattacharjee. Architectural Support for Address Translation on GPUs. In *Proc. ACM ASPLOS*, 2014.

[50] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 1974.

[51] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *Proc. ACM ASPLOS*, 2011.

[52] I. Pratt and K. Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *Proc. IEEE INFOCOM*, 2001.

[53] Ian A. Pratt. The User-Safe Device I/O Architecture. *Doctoral thesis, University of Cambridge*, 1997.

[54] H. Raj and K. Schwan. High Performance and Scalable I/O Virtualization via Self-Virtualized Devices. In *Proc. ACM HPDC*, 2007.

[55] D. S. Ritchie and G. W. Neufeld. User Level IPC and Device Management in the Raven Kernel. In *USENIX Microkernels and Other Kernel Architectures Symposium*, 1993.

[56] L. Schaelicke. Architectural Support for User-Level I/O. *Doctoral thesis, University of Utah*, 2001.

[57] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proc. USENIX OSDI*, 2010.

[58] J. Stecklina. Shrinking the Hypervisor One Subsystem at a Time: A Userspace Packet Switch for Virtual Machines. In *Proc. ACM VEE*, 2014.

[59] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proc. ACM SOSP*, 2003.

[60] K. Tian, Y. Dong, and D. Cowperthwaite. A Full GPU Virtualization Solution with Mediated Pass-Through. In *Proc. USENIX ATC*, 2014.

[61] T. Von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. ACM SOSP*, 1995.

[62] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. *ACM SIGOPS Operating Systems Review*, 2002.

[63] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In *Proc. IEEE High Performance Computer Architecture (HPCA)*, 2007.

[64] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proc. USENIX OSDI*, 2006.

[65] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2012.

[66] Z. Zhou, M. Yu, and V. D. Gligor. Dancing with Giants: Wimpy Kernels for On-demand Isolated I/O. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2014.